Master's thesis

2019

Joakim Fauskrud

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

Joakim Fauskrud

Hybrid analysis for Android malware family classification in a time-aware setting

June 2019

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of Science and Technology

NTNU

Norwegian University of
Science and Technology

Hybrid analysis for Android malware family classification in a time-aware setting

Information Security
Submission date: June 2019
Supervisor: Prof. Katrin Franke
Co-supervisor: Post doc. Andrii Shalaginov

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Preface

This master thesis was carried out for the faculty of Information Technology and Electrical Engineering at the Norwegian University of Science and Technology (NTNU). The master thesis was written in the spring semester of 2019, and marks the completion of my Master in Information Security.

01-06-2019

Acknowledgment

I would like to thank my supervisors, Andrii Shalaginov and Katrin Franke for excellent guidance and feedback throughout the project. I would also like to thank the contributors of AndroZoo for sharing access to their collection of Android applications.

J.F.

Abstract

Millions of malicious Android applications are detected every year. These applications are disguised as goodware applications in Android marketplaces in order to trick users. When the malicious application (malware) is installed on an Android device it can display advertisement, steal banking credentials, register to premium SMS services, encrypt and hold files for ransom or install additional applications. While it is critical to detect and remove malicious applications from the marketplaces before these applications are installed by users, it is also important to categorize these threats. Categorizing these threats can aid a security analyst in threat assessment, identifying appropriate mitigation strategies and removal techniques.

It is common practise in the industry to group malware into families based on similarity in code, behavior and author attribution. The malware threat is continuously evolving to avoid detection, exploit new vulnerabilities and adapt different monetization strategies in order to generate revenue. As a result new variants will be added to a family as the threat is evolving over time. Given the increasing number of malware variants, manual analysis is not practical and automated measures are required to assists security analysts. Machine learning based classifiers in the literature have shown great performance in both malware detection and malware family identification (categorization). However, the timeline of the malware samples, and thereby the evolution of malware, is often neglected when these classifiers are evaluated.

In this study we investigate the performance implications of the malware evolution on a machine learning based Android malware family classifier. We use a dataset of 14582 malware samples from the 54 most common malware families found in the markets during 2014-2016. Static and dynamic (hybrid) malware analysis methods are used to a extract features that have shown promise in the literature for distinguishing between malware families. We compare the results of classifiers evaluated in settings where samples in the training set are dated before samples in testing set (time-aware) with classifiers evaluated using k-fold cross validation (time-unaware). A 15.45% decrease in accuracy was found for the best classifier in the time-unaware setting. We therefore conclude that the performance of classifiers evaluated in a time-unaware setting introduces significant bias to the result. The set of features that performed best in all settings includes feature related to: Android API calls, permissions, intents, receivers, services, opcodes, system commands, native code, and finally strings that are longer than 5000 characters. We also compared the results that could be produced with different feature subsets, static, dynamic and hybrid features. However, due to a considerable amount of applications crashing during the dynamic analysis, the results of this comparison is inconclusive. Finally we measured the time required to extract the features used in the study to shed some light on the cost affiliated with the feature extraction process and how this affects identification of malware samples.

Sammendrag

Flere millioner ondsinnede Android applikasjoner (skadevare) detekteres hvert år. Disse applikasjonene skjuler seg i Android-markedsplasser ved utgi seg for å være godsinnede applikasjoner. Når en bruker installerer skadavaren på enheten sin kan den vise reklame, stjele bankopplysninger, registrere seg til Premium SMS-tjenester, kryptere og holde filer for løsepenger eller installere flere applikasjoner. Det er kritisk at skadevaren blir detektere og fjernet fra markedsplassene før de blir installert av brukere. Men det er også viktig å kategorisere truslene skadevaren utgjør. Kategorisering av trusler kan hjelpe sikkerhetsanalytikere med å gjøre trusselvurdering, velge riktige mitigering strategier og framgangsmåter for fjerning.

Det er vanlig praksis i bransjen å gruppere skadevare inn i skadevare-familier basert på likheter i kode, oppførsel og attribusjon til trussel aktør. Skadevare trusselen utvikler seg kontinuerlig for å unngå deteksjon, utnytte nye sårbarheter og for endre inntektsstrategier. Som et resultat blir nye varianter lagt til i skadevare familiene ettersom de utvikler seg. På grunn av den økende mengden med skadevare varianter er ikke manuell analyse praktisk mulig, og automatiserte løsninger er derfor nødvendig. Maskinlærings-baserte klassifiserere presentert i forsknings-litteraturen har oppnådd gode resultater for både detektering og familie-klassifisering av Android skadevare. En svakhet ved resultatene til disse klassifiserererene er at tidslinjen til skadevaren blir ignorert under evaluering, og dermed blir ikke skadevare-utviklingen tatt med i betraktningen.

I dette prosjektet undersøker vi resultat implikasjonene for en maskinlærings-basert klassifiserer når skadevare-utviklingen blir tatt med i betraktningen. Vi bruker et datasett med 14582 skadevare applikasjoner fra de 54 mest vanlige skadevare-familiene funnet i markedene mellom 2014-2016. Statiske og dynamiske (hybride) skadevare analyse metoder blir benyttet til å hente ut egenskaper som har produsert gode resultater for skille mellom skadevare-familier. Vi sammenligner resultater produsert av klassifiserere der skadevare i trening settet er datert tidligere enn skadevare i testing settet (tids-bevisst), med klassifiserere evaluert med *k-fold cross validation* (tids-ubevisst). Vi fant en 15.45% reduksjon i *accuracy* for den beste klassifisereren i den tids-bevisste situasjonen. Vi konkluderer derfor med at resultatene av en klassifiserer evaluert i en tids-ubevisst situasjon introduserer betydelig bias. Egenskapene som produserte best resultater for klassifiserererene våre inkluderer egenskaper relatert til: Android API kall, *permissions*, *intents*, *receivers*, *services*, system kommandoer, *native code*, og strenger som er lengre enn 5000 karakterer. Vi sammenlignet også hvilke resultater som kunne produseres med egenskap-sett som inkluderte statiske, dynamiske og hybride egenskaper. Men på grunn av at en betydelig mengde av skadavaren krasjet under dynamiske analyse, kan vi ikke konkludere basert på resultatene. Til slutt målte vi tiden som kreves for å hente ut egenskapene som ble brukt i prosjektet for å gi et grunnlag for kostnaden av å hente ut egenskaper og hvordan dette kan påvirke klassifisering av Android skadevare-familier.

Contents

List of Figures

List of Tables

Abbreviations

ABI	Application Binary Interface
AMD	Android Malware Dataset
API	Application Programming Interface
APK	Android Package
AVD	Android Virtual Device
C2	Command and Control (server)
CART	Classification And Regression Tree
CPU	Computer Processing Unit
D	Dynamic (used as a feature prefix)
DEX	Dalvik Executable
DT	Decision Tree
ELF	Executable and Linkable Format
ExtraTrees	Extremely Randomized Trees
GUI	Graphical User Interface
H	Hidden (used as a feature prefix)
HC	Hidden Code
HTTP	Hypertext Transfer Protocol
IMEI	International Mobile Equipment Identity
IMSI	International Mobile Subscriber Identity
JAR	Java ARchive
JNI	Java Native Interface
JSON	JavaScript Object Notation
K-NN	k-Nearest Neighbour
LR	Logistic Regression
M	Metadata (used as feature prefix)
MLP	Multi-Layer Perceptron
NB	Naive Bayes
NC	Native Code (used as feature prefix)
OS	Operating System
PC	Personal Computer
PE32	Portable Executable 32-bit
PID	ProcessID

PIN	Personal Identification Number
PLT	Procedure Linkage Table
PNG	Portable Network Graphics
PUA	Possibly Unwanted Application
RF	Random Forest
S	Static (used as feature prefix)
SDK	Software Development Kit
SMS	Short Message Service
SSD	Solid State Drive
SVM	Support-Vector Machine
URI	Uniform Resource Identifier
VM	Virtual Machine
VT	Virus Total
XML	Extensible Markup Language

1 Introduction

1.1 Topics covered by project

Anti-virus companies group similar malware into malware families based on author attribution, similarities in source-code and behavior[1]. Machine learning methods can be utilized to automate the process of classifying new (unseen) malware samples into malware families that are known beforehand. A machine learning model can learn to recognize which family a new sample belongs to based on characteristics from malware samples in the same family, such as Android API usage, requested permissions, network traffic, file-system usage, system calls, etc. Static and dynamic (hybrid) malware analysis methods can be used to extract these characteristics from malware sample. Static analysis involves techniques that can be used to examine a sample without running it. While in dynamic analysis the sample is executed and monitored in a controlled environment.

This project will investigate to what extent a machine learning based classification system can classify new (unseen) Android malware samples into known malware families in a time-aware setting.

1.2 Keywords

Android malware, Machine learning, classification, identification, malware family, security, static analysis, dynamic analysis

1.3 Problem description

The number of mobile malware attacks has increased rapidly over the years, and in 2018 Kaspersky[2] detected 5,3 million mobile malware installation packages. Android phones have the vast majority of the mobile market share[3], and is therefore the most lucrative mobile OS to target for malware authors. Android malware are known to steal banking credentials, send premium SMS-messages, click on advertisement, encrypt and hold files for ransom. These malicious applications are mostly distributed on third-party markets where the security of users is not a priority. While Google has implemented security measures for the official Android market place, malware is still found in Google play [4]. Malware is therefore posing a serious threat to any Android user.

Existing machine learning based classifiers has mostly been focused towards binary classification of malware and goodware. Simply detecting that an application is malicious and removing it does not address the capabilities of the malware sample, and the damages that might have occurred. Furthermore, these approaches often does not take the timeline of the malware samples into account when evaluating their proposed classifiers. In a real world setting a classifier relies on knowledge obtained from malware samples in the past to classify samples that are detected in the future. Malware samples in the training set should therefore be dated prior to the samples in the validation

set. Having malware samples from "the future" in the training set will lead to considerably biased results [5]. Allix et al [5] states that state-of-the-art malware detection systems from research that does not take history constraint[1] intro consideration will not be powerful in a real-world setting.

1.4 Justification, motivation and benefits

The sheer amount of Android malware detected today is far too large to be manually analyzed. A machine learning based classification system can help the analyst by classifying new (unseen) malware into malware families that are known beforehand. Knowing which malware family a sample belongs to can have several benefits for a security analyst. Information about the capability malware sample can quickly be identified by looking up the malware family. Removal techniques can be reused. If manual analysis is to be performed, having a basic idea about the malware can speed up the the process [6].

1.5 Research Questions

- ***RQ1:*** *What level of accuracy can be achieved for Android malware family classification in a time-aware and time-unaware setting, and how do the results of these settings compare?*
- ***RQ2:*** *Can the performance of the classifiers be improved by combining features extracted with static analysis and features extracted with dynamic analysis?*
- ***RQ3:*** *What set of Android malware characteristics/features can produce the best results for a machine learning classifier?*
- ***RQ4:*** *How much time is required to extract the optimal set of features and classify a new malware sample?*

1.6 Contributions

The main contribution of this master thesis will be to answer the previously stated research questions. Providing new knowledge to the Android malware research community, and thereby aiding further research into the topic. To the best of the authors knowledge the impact of the history constraint have not been studied for a multinomial classifier that classifies malware samples into malware families. It is important to determine the capabilities of such a system if it were to be deployed in a real-world setting [5]

1.7 Thesis outline

- **Chapter 2** will provide necessary background information to give the reader a better understanding of the problem area and topics covered by the project. The following topics will be described: Android OS fundamentals, Android malware and naming of malware, finally an introduction to static and dynamic Android malware analysis will be given.
- **Chapter 3** discusses literature related to the research questions, and gives a description of the state-of-the-art for machine learning based Android malware family classification.

[1] **History constraint**: In a real-word setting a classifier cannot be trained on knowledge from the future.

- **Chapter 4** provides and overview over the methodology used to investigate the research questions. It includes detailed descriptions of malware sample collection, dataset construction, feature extraction, feature extraction time, the machine learning methods that was used, and finally how we evaluated the classifiers.
- **Chapter 5** describes the experimental setup, machine specification, tools and work flow used to conduct the experiment. Finally the thresholds and parameters used for the machine learning algorithms will be discussed.
- **Chapter 6** discusses the validity of the study, present the results and discuss the important findings. First threats to validity of the study is discussed, followed by results and discussions for each research question.
- **Chapter 7** provides a summary of the thesis and findings related to the research questions. Followed by a discussion of theoretical implications and Reproducibility of the study. Finally we discuss future work.

2 Background

2.1 Android OS fundamentals

Android applications are written in the Kotlin, Java and/or C++ programming languages. The Android Application Programming Interface (API) is utilized to access the functionality of the Android Operating System (OS). The source-code, resources and data of the application is compiled into an Android Package (APK) file using the Android-SDK. APK files are archive files used to distribute and install Android applications [7].

The Android operating system (OS) is a multi-user linux system, where each application is a user. The Android OS sets permissions on all files of an application, such that the files can only be accessed by that particular application. Each process has it's own Virtual machine, and application code is isolated from other applications when executed. The principle of least privilege is implemented in the Android OS. An application can only access system resources that are absolutely essential to run. The application must request specific permissions from the user of the Android device to gain access to additional system resources [7].

Android applications are built on four following components: Activities, Services, Broadcast Receivers and Content providers. An **activity** is the window of an application providing a user interface, and it's the component that the user interacts with. **Services** are for any kind of work an application can accomplish in the background (e.g. downloading updates). There are two sub-categories of services: Started services and bound services. Started services are used when an application has some unfinished work and needs to keep running until it is completed. Bound services provides an API for other processes, and is run when the system or another application needs it. **Broadcast Receivers** are the component that enables an application to receive to system wide broadcasts. Broadcasts are received by the Broadcast receiver even if the application is not currently running, which enables the application to start up and perform some action. Broadcasts can be initiated by applications and the system, and is mostly used as a gateway to communicate with other components. **Content Providers** manages a shared set of application data that is stored in a location accessible by the application. The application data can for example be stored in the file system, a database or in the cloud. The content provider enables an application to publish data items named using an URI scheme. The application maps data items to the URI namespace, and other entities can access the data items using these URIs. As an example the Android OS manages a content provider to share access to the device contact information. Application can access the contact information through the content provider given that it has the proper permission to do so. Content providers can also be used to read/write data that is private to the application [7].

Activities, services and broadcast receivers are activated by sending an **Intent** (an asynchronous message). Because applications run in isolated environments, they must send an **Intent** to inform

the system to start a particular component. An application can send an intent to start it's own component, or the component of another app. Content providers are activated by something called content resolvers [7].

As mentioned previously Android applications are installed using APK files. The APK files contains the compiled source code, resources and data. The typical structure of an APK file looks something like this:

```
Application.APK/
    assets/
    lib/
        armeabi/
            libfoo.so
        x86/
            libfoo.so
    META-INF/
    res/
    AndroidManifest.xml
    classes.dex
    resources.arsc
```

The ***assets*** folder contains files that the application can retrieve using the AssetManager API [8]. The ***lib*** directory contains native code files, that is c/c++ code compiled for specific ABIs[1] (armeabi and x86 in the example above). The Java Native Interface (JNI) is used to call functions in the native code files [10]. The ***META-INF*** directory contains application certificates and SHA1-digests for all files in the APK. The ***res*** folder holds application resources (e.g. images) that are not compiled into ***resources.arsc*** [8]. ***AndroidManifest.xml*** declares all components of the application, any permission that the application requires, SDK versions[2], Android API libraries the app needs to be linked against and finally hardware and software features required by the app[7]. ***Classes.dex*** contains the Dalvik bytecode, and is used to execute the application using Android runtime (ART) [11]. A list of the bytecodes can be found here[3].

[1] **ABIs:** Android devices use different CPUs, which support different instruction sets. All combinations of CPUs and instruction sets has an Application Binary Interface (ABI) that defines how the machine code is supposed to interact with the system at runtime. An application must select which ABI it wishes to support [9].

[2] **SDK versions:** Different Android devices runs different Android platform versions (SDK version or API levels). An application must specify the targeted SDK version, as well as the minimum SDK version where it can be run.

[3] https://source.android.com/devices/tech/dalvik/dalvik-bytecode

2.2 Android malware

2.2.1 Construction

Three different methods are commonly used to create Android malware [12]:

- **Standalone:** The malware is written from scratch.
- **Repacking or piggybacking:** A goodware application is decompiled, and a malicious payload is inserted before the application is recompiled. Malware authors are known to use two different forms of repackaging, **Isolated repackaging** and **integrated repacking**. In isolated repacking the malicious code is included in to the application, but is no way connected with the applications original functionality. The malicious code will have its own event handler as activation component. In **Integrated repackaging** the malware author modifies the original code to insert the malicious payload, making it more stealthy and less likely to be detected.
- **Library:** The malicious code is contained within a library of an otherwise benign application. The library is included by the original author of the application, who may be unaware of the malicious code. This method is common for advertising malware (adware).

In a study of 24,650 malicious applications collected from 2010-2016, Wei et al [12] found that 35% were created from scratch, 7% were repacked and, 58% of the applications contained a malicious library.

2.2.2 Distribution

Android applications are commonly distributed in marketplaces such as Google Play[4] and Appchina[5]. These marketplaces are also used by malware authors to distribute malware disguised as legitimate applications. The malicious payload may be hidden within the malware, or downloaded by the malware at a later time.

Malware are also distributed through different websites, using a method called drive-by-download. A **Drive-by-download** can be defined as "(1) Downloads which a person authorized but without understanding the consequences; and (2) Any download that happens without a person's knowledge."[13]. For instance the user may be prompted to download Flash player while visiting a website [12].

2.2.3 Activation

The malicious payload may not be triggered when the application is started. The most common activation methods are described in the following list:

- **By-host-app:** The malicious payload is activated alongside the code of the host application. This activation method is used for malware that is constructed by integrated repackaging [12].
- **Time-based:** The malicious payload is activated after a certain time after the application is started [12]

[4] https://play.google.com/store
[5] http://www.appchina.com/

- **Event-based:** Payload is activated based on an Android related events, such as when the user comes into presence, device connectivity changes or when an application is installed or deleted [12].

2.2.4 Command and Control (C2) server

Malware communicates with C2 servers in order to increase functionality and to better adapt to the environment. By communicating with a C2 server, the malware can receive instructions, updates and send information to the threat actor. Android malware are known to use both SMS and HTTP for communication with the C2 [12].

2.2.5 Information Stealing

Wei et al [12] observed that more than 87% of the malware samples in their study collected various information about the infected device. Information such as IMEI[6], IMSI[7], installed applications, OS version and language. The IMEI and IMSI are unique to the device, and can be used as an identifier with the C2. While the other information may be used by the C2 to decide further actions on the device.

2.2.6 Persistence

Persistence techniques are utilized by malware in order to remain on the infected device after installation. More time on the device equals more revenue. Persistence can be achieved by [12]:

- **Being stealthy:** The malware can hide it's presence on the device by cleaning up logs, running in the background, and by hiding SMS and call notifications.
- **Preventing removal:** Hiding itself from the device admin list, killing Anti virus software and locking the device.

2.2.7 Privilege escalation

By obtaining admin privileges, the malware can achieve persistence, and access privileged functionality (e.g. changing lockscreen PIN, locking the device and deleting data). Malware must trick the user into granting admin privileges. Rooting exploits has become less popular due to the increased security of the Android OS [12].

2.2.8 Types

The most common Android malware types are listed below [2]. Keep in mind that these types are not mutually exclusive, and that malware may be categorized into one or more of these types.

- **Trojan:** Malware that seems to be legitimate, but contains a malicious payload. Usually require user interaction for installation.
- **Premium Service Subscription** The malware subscribes to premium SMS services in order to generate revenue, while hiding this activity from the user [12].
- **Banking Trojan:** Will detect if there are a banking application installed on the device. Some

[6] **IMEI:** international mobile station equipment identity.
[7] **IMSI:** international mobile subscriber identity

banking trojans will replace the original banking application with itself. Other banking trojans are known to create screen overlays while the real bank application is being used, tricking the user into thinking that he/she is still using the banking application [12].

- **Dropper:** Droppers are used as a means to hide a malicious payload, while evading detection. The payload can be packed[8] or hidden in the APK-file.
- **Downloader:** Similar to the dropper type, but the payload is downloaded from a server.
- **Ransomware:** Locks the users device by making it unresponsive or encrypts files. Demands a ransom to unlock the device or to decrypt files [12].
- **Adware:** steals personal data, displays unwanted advertisement in an aggressive manner, and tempts the used to download potentially harmful applications [12].
- **Spyware:** Spyware monitors the activity of the user to collect information such as location, usernames and passwords [14].
- **RiskTool:** Programs that includes functionalists such as hiding files in the system, hiding the window of running applications, or terminating active processes. These programs are not necessarily malicious by themselves. One example is cryptocurrency miners[9].
- **Backdoor:** Programs that allows undetected and unauthorized access to the device.
- **Worm:** Programs that makes copies of itself and spreads to other devices.

2.3 Malware naming

The common practise in the industry is the categorize malware into malware families. A malware family name can indicate author attribution, malware campaign, or other characteristics, such as similarities in the source-code or assets [1, 12, 15]. Malware families can be further categorized into variants. An example is the Zen family which is a grouping of malware based on author attribution. The Zen authors utilized different monetization strategies in order to generate revenue. The simplest variant of the malware family inserts an advertising library into a trojan, while another variant escalated the tactics to click fraud[10] [15].

Antivirus engines commonly assigns a label to each malware sample. Labels given to a malware sample by 8 different Antivirus engines are shown in table 1. From the labels we can extract information about the platform that the malware is targeting, as well as malware type, family and variant. As shown in the table, there are several inconsistencies in the malware labels: Antivitus enginies tends to disagrees on which family the sample belongs to, no common naming scheme is used (e.g. PLATFORM.TYPE.FAMILY.VARIANT) and different names are assigned to the same family (aliases). There exists effort towards creating common naming scheme for malware labels, CARO[11] and CME[12], but unfortunately these are not widely used [17].

In research on malware, samples are often collected from various sources, and the malware samples may not already have assigned labels. A common approach in the literature for labeling

[8] **Packed:** encrypted or compressed so that the payload cannot be identified.
[9] https://encyclopedia.kaspersky.com/knowledge/risktool/
[10] **Click fraud:** automated means are used to click on pay-per-click online advertisement [16]
[11] http://www.caro.org/articles/naming.html
[12] https://cme.mitre.org/about/docs.html

AntiVirus	Label	Type	Platform	Family	Variant
Sophos AV	Android Dowgin (PUA)	pua	android	dowgin	n/a
Tencent	a.gray.floatgame.t	n/a	n/a	n/a	n/a
F-Secure	Android.Adware.GingerMaster	adware	android	gingermaster	n/a
BitDefender	Android.Adware.GingerMaster.MP	adware	android	gingermaster	MP
ESET-NOD32	Android/AdDisplay.Kuguo.AA potentially unwanted	adware	android	kuguo	AA
GData	Android.Adware.GingerMaster.MP	adware	android	gingermaster	MP
Avira	ADWARE/ANDR.Kuguo.AF.Gen	adware	android	kuguo	AF
Fortinet	Android/Generic.Z.2ECE44!tr	genric	android	n/a	n/a

Table 1: Malware family label example

malware samples is to use VirusTotal[13] reports [12, 18, 6]. These reports contains decisions and labels given by 55 or more anti-virus engines for each of the samples. A report can be retrieved through the VirusTotal API[14] by uploaded a sample, or query the API with the hash of a sample (given that the sample has previously been analyzed in VirusTotal). Reliably selecting a malware family name from a report is a challenge. Wei et al [12] used a dominant keyword algorithm to select a malware family name for each sample based on VirusTotal reports. Efforts have been made to develop tools that can automate the labeling process based VirusTotal reports in a reproducible manner. Euphony [19] uses clustering to infer malware family names based on VirusTotal reports. AVClass [17] implements plurality voting to select a malware family name. The Antivirus labels are normalized, generic tokens are removed (e.g. android, adware etc.), and aliases are detected. To do so, AVClass relies on a list of previously known generic tokens and aliases derived from a large reference set. The tool also includes functionality to generate these lists based on a large set of malware samples.

During their study, Sebastian et al [17] observed that the labels given by Antivirus engines are not fine-grained enough to separate the families into variants. Wei et al [12] used clustering within each family to detect variants after the labels had been assigned.

[13] https://www.virustotal.com/#/file/ffed6ab3b997d28fa008fbf6f1b264c47b3c7c2a63a435194ca5f19bf04475e4/detection

[14] https://developers.virustotal.com/reference

2.4 Android Malware analysis

There generally exists three types of malware analysis techniques, static analysis, dynamic analysis and hybrid analysis. **Static analysis** consists of all techniques that can be used to analyze a malware sample without having to execute it. **Dynamic analysis** are techniques that can be used to analyze the behavior of a malware sample during execution. **Hybrid analysis** is a combination of the two previous techniques.

An extensive list of Android analysis tools and resources can be found in this[15] Github repository. In the three following subsection we will give a brief introduction to the techniques and tools that can be used in static, dynamic and hybrid Android malware analysis:

Static analysis

As mentioned in the Android OS fundamentals section, the main artifacts for static analysis are classes.dex, AndroidManifext.xml and native code files.

The classes.dex file can be disassembled into smali files using smali/backsmali [20], which is a disassembler/assembler for the DEX format used by dalvik, the (discontinued) Android java VM implementation. The smali files are human readable text files containing dalvik opcodes [21]. These files can be parsed to extract information such as API calls, opcodes and strings. Alternatively a DEX files can be converted to a JAR file using the dex-2-jar [22] tool. JAR files are achieves that contains the java source-code. However, parsing java-source code to extract information using automated tools is more complicated, and smali files are more commonly used for this purpose. APKTool[16] is a reverse engineering tool that is capable of disassembling the classes.dex, resources.arsc, and inflating AndroidManifest.xml into a human readable format. Dex-2-jar [22] also include a standalone tool, d2j-samli, that can be used to disassemble DEX files. AndroGuard [23] is a python library for reversing engineering an APK-file.

FlowDroid [24] does taint analysis of an APK-file. The tool performs data-flow analysis that identifies multiple source and sinks in the source code.

Native code files are ELF-files compiled for different architectures. These files can be disassembled into assembly code using any disassembler that supports the respective architecture (e.g. x86 or arm).

Dynamic analysis

Dynamic analysis can be used to extract information about the malware's behaviour during execution. Interesting artifacts in dynamic analysis are network activity, sms/phone activity, cpu usage, memory and file system operations. It is imported that dynamic analysis is conducted in a safe environment to ensure that the malware is properly isolated. Malware are known to use propagation tactics in order to infect additional devices.

DroidBox [25] a dynamic analysis tool that monitors an application using an Android Emulator. DroidBox retrieves information about file operations, network traffic, sms/phone activity, crypto

[15] https://github.com/ashishb/android-security-awesome
[16] https://ibotpeaches.github.io/Apktool/

usage, started services and dynamically loaded files.

Hybrid analysis

AndroPyTool [26] is a hybrid analysis tool that incorporates different tools in a modular fashion. AndroPyTool consists of the following analysis modules: Dynamic analysis modul based on DroidBox and Strace[17], Taint analysis module using FlowDroid, Static analysis module that extracts information about API calls, permissions, Android components and intents, strings, system commands and opcodes.

CuckooDroid [27], an automated Android malware analysis tool based on the Cuckoo sandbox[18]. CuckooDroid does both static analysis and dynamic analysis. AndroGuard[23] is used for the static analysis. Dynamic analysis is based on dynamic API inspection by hooking API calls.

MobSF [28] is a mobile pentesting framework that performs both static and dynamic analysis. Both MobSF and CuckooDroid are capable of evading certain VM detection techniques. DroidBox[25] and CuckooDroid [27] both supports API level 16, while MobSF [28] offers different analysis VMs up to API level 19 (released in 2013).

2.4.1 Anti-analysis techniques

The most common anti-analysis techniques specifically targets static analysis, and dynamic analysis might therefore have an advantage over static analysis. However, there are also anti-analysis techniques that targets dynamic analysis. Using a combination of the two can make malware analysis very difficult. This section will discuss the most common anti-analysis techniques found in Android malware.

Renaming

Renaming is one of the most used obfuscation techniques. Packages, classes, methods, fields and parameters are renamed into meaningless words. Making manual analysis[19] significantly more difficult. However, API calls cannot be renamed, and this method does not affect automated static analysis [12].

Reflection

According to Garcia et al [18] malware authors are increasingly utilizing java reflection as an anti-analysis technique to hide the malicious behavior of malware. Reflection provides an application with the capability to inspect and modify itself during run-time. Benign applications use reflection to apply updates and bug-fixes without having to re-install the application. Malware can use reflection to obfuscate sensitive API calls and libraries [29].

String encoding/encryption

Strings such as C2 domain, intent actions, JSON/XML key values, components names and java reflection strings can help anti-virus identify malware. Malware encrypt/encodes strings using methods such as base64, DES/AES to make analysis more difficult [12].

[17] **Strace:** Linux tool that tracks low level system calls
[18] https://cuckoosandbox.org/
[19] **Manual analysis:** manually inspecting the malware

Native code

Native code libraries are used by games, and other applications for optimization purposes. These files are often overlooked during static analysis, and malware are known to hide malicious payloads and sensitive strings in native code files [18, 12].

Dynamic loading

Dynamically loading DEX files is becoming increasingly popular. Additional DEX files may be stored among the assets, or downloaded from a C2 server. These DEX files can also be encrypted[12].

Code hiding

Malware are known to hide malicious code within the APK file. For instance Root exploits hidden in innocent looking files like *install.png*. Some malware are known to go further and use steganography to hide malicious scripts within real image files. [29] Additional APK-files hidden in the original APK can be installed at run-time using the package manager. Requires confirmation from user [6].

Evading dynamic analysis

Evading dynamic analysis is achieved by verifying that the malware is not being run in an analysis environment. The malware will compare certain device information (e.g. IMEI, MODEL, FINGERPRINT, MANUFACTURER, BRAND and DEVICE) with known values of emulators and analysis environments. If the malware detects that it is being analyzed it will not conduct any malicious behavior, and stop running [12].

3 Related Work

This chapter will discuss literature related to the research questions which guides the efforts of this study. In order to build a classifier, a set of Android malware characteristics (features) that can be used to distinguish malware families from each other is required. As discussed in the background chapter 2.4, different analysis techniques can be used to extract features that describe the malware. We will therefore start off by discussing which features has performed well in other studies, and which machine learning algorithms were used to build the classifiers. Followed by a discussion of what has been previously done to evaluate Android malware family classifiers in a time-aware setting. Finally we will discuss the time used for extraction of features and classification in these studies.

3.1 Features, analysis and machine learning classifiers

As described in the 2.4, static and dynamic analysis are two different techniques that can be used to extract features from Android malware samples. Static analysis are the techniques that can be used to examine a sample without running it. In dynamic analysis the sample is executed and monitored in a controlled environment. Hybrid analysis is an alternative that combines these two methods. In this section we will discuss state-of-the-art classifiers built on features that was extracted with each of these analysis methods. A comparison of the proposed classifiers is shown in table 2.

Classifier	Static	dynamic	Machine learning algorithm	k-fold cross val.
RevealDroid[18]	yes	no	CART, SVM	10
FalDroid[30]	yes	no	SVM, DT, k-NN, RF	10
Kang et al[31]	yes	no	NB, SVM, PART, RF	10
DroidSieve[29]	yes	no	ExtraTrees	hold-out validation
Massrelli et al[32]	no	yes	SVM	20
DroidScribe[33]	no	yes	SVM	20
EC2[34]	yes	yes	Unsupervised: DBSCAN, k-means, Affinity clustering, Hierarchical clustering and MeanShift. Supervised: RF, DT, k-NN, SVM, NB, LR	2, 5
UpDroid[6]	yes	yes	k-NN, RF, DT	20

Table 2: Classifier comparison

3.1.1 Static analysis

Garcia et al [18] presents RevealDroid, a machine learning based approach that achieves promising results for both Android malware detection and family identification. Classification and Regression Trees (CART), a batch-learning based machine learning algorithm was used for family identification. RevealDroid leverages the four following types of static features for the classifier: package-level Android-API usage, method-level Android-API usage, APIs invoked using reflection and function calls of native binaries within the Android applications.

Fan et al [30] developed FalDroid a multinomial familial classifier that uses fregraphs, a novel graph based feature. The fregraphs are extracted from function call graphs, and sensitive API calls are weighted according to the within family frequency.

Kang et al [31] evaluated the performance of n-gram opcodes for binary and multinomial classification of Android malware. They found that n-gram opcodes can achieve satisfactory performance by itself. n-grams up to a size of 10 was tested, and they found that the performance started to stabilize at a size of 4. Two types of n-gram opcode features was tested, frequency n-grams and binary n-grams. Binary n-grams is a boolean feature that is set to 0 or 1 based on the presence of the n-gram in an application. While frequency n-gram counts the number of occurrences of the n-gram in the application. Binary n-grams were observed to be more accurate. The advantage of this feature is that no expert knowledge is required to specify the feature set beforehand [31].

Suarez-Tangil et al [29] presented DroidSieve an Extra trees classifier that relies on a wide range of static features. DroidSieve uses features related to API calls, Android components, intents, permissions, anti-analysis techniques, certificates, native code and more. Including a set of novel features. For instance, a feature that checks the difference in time between the date when the certificate was issued and the date when the app was signed. If the difference is less than a day, it is likely the app was signed when the malware repackaged with automated tools. A complete overview of the features can be found in figure 1 in their paper [29].

3.1.2 Dynamic analysis

Massarelli et al [32] extracted dynamic features related to resource consumption over time, and built a multinomial classifier using SVM. Samples were run in an emulated environment, and a time-series of 26 different metrics were collected from the proc file. System-wide and application specific metrics such as CPU, memory and network usage was monitored. The time-series of metrics were processed into a feature vector that was used for classification. The advantage of their apporach is that it does not require any modification to the android emulated environemnt, and can be used on real devices.

Dash et al [33] developed DroidScribe, the first approach to multinomial classification of Android malware that relies exclusively on dynamic features. They used CopperDroid [35], a dynamic analysis tool that extract high-level behavior features from samples. These features are related to network access, file access, binder methods[1], and file execution.

[1] **Binder methods** are used for inter-process/inter-component communication, and are the interface to Android system services and app-to-app interaction [33].

Dash et al [33] highlights two challenges in dynamic Android analysis: Firstly only limited information towards classification can be learned from tracking low-level events (system calls). The second challenge is that dynamic analysis has imperfect coverage. A single path is shown for each execution, and one path only covers a limited portion of the source code. Android applications are highly interactive, and simulation can be used to improve code coverage [8]. Additionally command and control servers that the malware attempt to communicate with might be inactive during the analysis [33].

3.1.3 Hybrid analysis

Chakraborty el al [34] presented EC2, Ensemble Clustering and Classification, a novel algorithm for discovering malware families. EC2 is an early warning system that achieves good classification performance for both seen (known) and unseen (unknown) malware families. EC2 is also capable of classifying malware families with very few samples (samples < 10). They found that the following feature set was most important for malware family classification: re-using signatures for signing malware, requested permissions related to network and SMS, use of encryption. EC2 relies on DroidBox[25] for dynamic feature extraction. Chakraborty el al [34] found that some malware families can be better described with static features, and others with dynamic analysis. For instance, dynamic features performs better for malware families that decrypts strings at runtime.

Aktas and Sen [6] extracted features using both dynamic and static analysis. Features related to permissions, Android components, APK size, and a broad range of dynamic features (extracted and constructed based on the analysis output of DroidBox[25]) was used. Android malware family classifiers were built and evaluated using three different machine learning algorithms, k-NN, Random Forest and J48. In their experiments k-NN procured the best results. Aktas and Sen [6] compared their results with other state-of-the-art to Android malware family classification on the Android Malware Genome Project[2] dataset. The best accuracies was achieved by DroidSieve[29] (97.79%), UpDroid[6] (97.32%), FalDroid[30] (97.2%), RevealDroid[18] (95%), in that order.

3.2 Time-aware setting

Allix et al [5] highlights the importance of considering the timeline for malware detection systems. No paper that investigates the performance implications of Android family classification in a time-aware setting was identified in the literature study. As shown in figure 2, k-fold cross validation is the most common method used to evaluate classifiers. k-fold validation randomly splits the dataset into k groups. k-1 group are used for training and 1 group is used for testing. A classifier is trained k times so that each group is used for testing. The final evaluation is the average of the k tests.

The performance of RevealDroid [18] for malware detection (goodware vs malware) was tested in a time-aware and time-unaware setting. High accuracy was achieved for both settings, but there was a significant decrease for the time-aware setting. The malware family classification in RevealDroid was evaluated in a time-unaware setting, and the performance impact of taking the timeline into consideration is unclear.

[2] http://www.malgenomeproject.org/

3.3 Feature extraction and classification time

Symantec detected 24,000 mobile malware samples every day in 2017 [36]. Meaning that a malware classification system should potentially be capable of extracting features and classifying a large number of samples each day.

Garcia et al [18] developed their own tool for feature extraction. Requiring 90 seconds on average to extract features from an application. FalDroid the tool developed by Fan et al [30], uses on average 4,6 seconds to classify a new malware sample into a family. DroidSieve [29] has an impressive median feature extraction time of only 2,53 seconds.

In dynamic analysis malware samples are usually run for a fixed duration in a sandbox in order to monitor behavior. Aktas and Sen [6] used 15 minutes to analyze the behavior of a sample in DroidBox. While Chakraborty used [34] 120 seconds to monitor the behavior of a sample.

4 Methodology

This chapter will discuss the methods that was used to investigate the research questions:

- ***RQ1:*** *What level of accuracy can be achieved for Android malware family classification in a time-aware and time-unaware setting, and how do the results of these settings compare?*
- ***RQ2:*** *Can the performance of the classifiers be improved by combining features extracted with static analysis and features extracted with dynamic analysis?*
- ***RQ3:*** *What set of Android malware characteristics/features can produce the best results for a machine learning classifier?*
- ***RQ4:*** *How much time is required to extract the optimal set of features and classify a new malware sample?*

We used Quantitative research methods[1] to answer these request questions. A wide range of machine learning classifiers are evaluated in different settings and the results are compared. We evaluate the classifiers in time-unaware and two time-unaware settings to answer **research question 1**. The classifiers in each of these settings are evaluated with different feature sets (dynamic, static and hybrid), and the results are compared to investigate **research question 2**. We consider **Research question 3** based on the results of all the different classifiers. While **Research question 4** is based on the time required to perform hybrid analysis, extract features and classify a samples. An overview of the methodology is shown in figure 1. The green field represents all the steps required to prepare features for classification, the blue field describes how the features was split to create different evaluation sets, and the red field represents the classification step.

The remainder of this chapter is structured as follows: Section 1 describes the collection of malware samples, section 2 discusses dataset construction by selecting a representative sample, section 3 discusses extraction of features and dataset analysis, sections 4 discusses how we measured the time required to extract features, section 5 described the different machine learning methods used for feature selection and classification, lastly, section 6 discusses how we evaluated classifiers in time-unaware and time-aware settings.

[1] **Quantitative research methods** revolves around quantification of observations within a representative sample, using statistical methods and mainly deductive reasoning to draw logical conclusions about the population of interest[37].

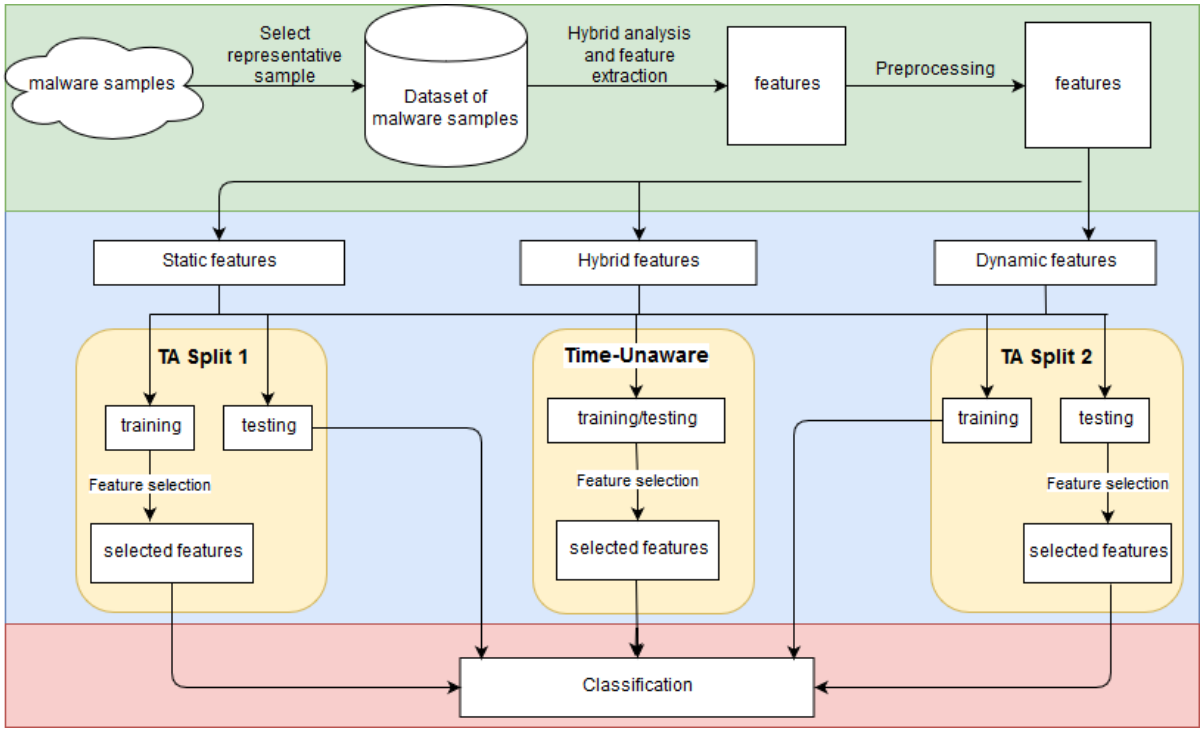

Figure 1: Methodology overview

4.1 Malware sample collection

To be able to investigate the research questions stated in 1.5, a dataset of malicious Android applications that meet certain criteria is required: **(1)** First of all the dataset should include wide verity of malware families, and must at least cover the most common malware families; **(2)** Secondly, the samples must be dated over at least two years, and should be as recent as possible. The malware population is continuously evolving and the android malware threat landscape has changed over time. The results are likely to be more relevant for dealing with the threat landscape we are facing today if the dataset is recent.

Different datasets targeting a variety of problem areas have been proposed and shared with the research community. These datasets can be categorized into two main types. Raw datasets containing Android APK files, and datasets that provides analysis logs or already extracted features. The latter type is good for benchmarking different machine learning approaches, but the feature space is limited to only the features provided in the dataset. One such dataset is the **OmniDroid** dataset [38] created with a wide verity features extracted using the AndroPyTool[26]. Unlike this study, the **OmniDroid** targets the malware detection problem of distinguishing malware from goodware applications.

Because we did not want to restrict the feature extraction process to a predefined set of features,

a dataset of raw Android APK files were preferable. Out of these, the most notable dataset in the literature are the **Android Malware Genome Project**[2] and **Drebin** [39]. **The Android Malware Genome Project** contains 1260 samples from 49 different families. The project was discontinued in the end of 2015, and the dataset is no longer shared. **Drebin** contains 5560 samples from 179 malware families, that was collected from 2010 to 2012[3]. **The Android Malware Genome Project** and **Drebin** are often used as a benchmark-datasets to compare state of the art approaches to Android malware classification.

A more recent dataset is the **Android Malware Dataset (AMD)**, that contains 24,650 samples from 71 families dated between 2010-2016 [12]. Another recent dataset is the **UpDroid** dataset containing 2479 samples from 21 families, dated from 2015 and later [6]. The **UpDroid** dataset targets the update attack[4] problem area, and includes malware families known to conduct such attacks.

There also exists various online repositories of Android applications. Goodware applications can be downloaded from Android marketplaces such as **Google Play**[5] and **Appchina** [6]. Android malware can be downloaded with limited downloads or special permissions from **VirusTotal**[7], **Hybrid-analysis** [8] or **Koodous** [9]. **VirusShare** [10] is a repository of all kinds of malware shared in zip archives to aid the research community. However the VirusShare repository provides little to no information about the contents of the zip archives.

In [4] Allix et al presents **AndroZoo**, a large repository of Android malware and benign applications. At the time of writing the repository contains more than 8,5 million Android applications collected from marketplaces and other sources. A large portion of the applications have been scanned using VirusTotal. Out of these, more than 1,8 million applications was flagged as malicious by one or more anti-virus engines.

The **AndroZoo** repository was selected for this project because it provides the freedom to construct a dataset that is tailored for the problem area. **AndroZoo** shares a csv file containing DEX date[11], VirusTotal detects and other metadata for each application in the dataset. A json file containing malware family labels derived using the Euphony tool[19] is also provided (see Euphony 2). Having prior knowledge of the dataset time distribution and the malware family labels made it possible to verify that the dataset satisfies the first and second criterion.

[2] http://www.malgenomeproject.org/

[3] https://www.sec.cs.tu-bs.de/~danarp/drebin/

[4] **Update attacks** Evasion techniques that updates the application at run-time by loading a malicious payload. The payload may be stored in the application as data (not code) or downloaded from a C2 server [6]. Malware in this category typically belongs to the dropper and downloaded type discussed in the background chapter 2.1.

[5] https://play.google.com/store

[6] http://www.appchina.com/

[7] https://www.virustotal.com/

[8] https://www.hybrid-analysis.com/

[9] https://koodous.com/

[10] https://virusshare.com/

[11] **DEX date:** Android applications are installed and distributed using APK-files. APK-files are zip-archives that contains the binary executable classes.dex file, and various other resources. Zip-archives holds a last-modified date for each file contained in the archive. Based on literature review the last-modified date of the classes.dex file is commonly used to assign a date to a samples [4, 18].

4.2 Dataset construction

To complete the project in a timely manner a representative subset had to be selected from the AndroZoo repository [4]. In order to answer *research question 1* 1.5, families in the dataset should have samples dated over a longer time period. Families that had fewer than 50 samples were therefore removed from the dataset. To keep the dataset as recent as possible, only samples dated after the first three quarters of 2014 was included. Based on the distribution of the samples in AndroZoo, there were a sufficient amount of samples from the third quarter of 2014 until the third quarter of 2016. Giving us a dataset with samples dated over at least two years.

Because we are investigating the multinomial classification problem of Android malware families, only the malicious samples were of interest. A common method of determining if a sample is malicious is to set a threshold of the number of anti-virus engines flagging a sample as malicious in the VirusTotal report. wei et al [12] required that at least 50% of the anti-virus engines flagged a sample as malicious when they constructed the AMD dataset. Aktas et al [6] set the threshold to 20 when the UpDroid dataset was constructed. To put the problem into perspective, the impact of such a threshold for the AndroZoo dataset is shown in table 3. There is a significant reduction of the number of samples and families in the dataset as the threshold increases. The threshold can be considered as a trade-off between having more certainty in whether the samples are malicious or not and the bias of removing the samples that are possibly more difficult to detect. In order to keep the experiment more realistic while having some level of certainty a threshold of 5 was selected.

VirusTotal detections	Samples	Families
1	435,163	287
5	194,285	113
10	99,079	79
15	44,077	51
20	14,236	29
25	2,889	11

Table 3: The impact of VT detection rate threshold on the dataset size

The following table 4 displays the number of samples in the 10 most populated families after time interval and VT detection rate threshold was set.

As mentioned earlier, Euphony was used to give malware family labels to the AndroZoo dataset. Euphony is capable of labeling samples that is only detected by a single anti-virus engine, but does not exclude generic family labels [19]. Generic labels was filtered from the dataset manually. We searched for each family name on Google (e.g. "FAMILY android malware"). If no information could be found, the family would be removed from the dataset. For instance "artemis", rank 4 in table 4 is used by MacAfee for any sample that is put in quarantine or blocked [12]. The label "genpua", rank 10 is likely the short name for "generic PUA". To identify aliases the list[13] constructed by AVClass

[12] https://service.mcafee.com/webcenter/portal/cp/home/articleview?articleId=TS100414
[13] https://github.com/malicialab/avclass/blob/master/data/default.aliases

[17] was used.

Rank	Family	Samples
1	dowgin	39,702
2	kuguo	34,074
3	revmob	17,592
4	artemis	16,685
5	airpush	15,257
6	smspay	6,666
7	feiwo	6,393
8	jiagu	6,388
9	eldorado	4,650
10	genpua	3,741

Table 4: Top 10 families

In table 4 we can see that there is a clear class imbalance. The largest family has 10 times more samples than the 10th largest family, and the smallest families only has around 50 samples. In order to balance the dataset a fixed subset of the larger families were selected. It was also necessary to reduce the total number of samples in the dataset to finish the project in a timely manner. The

following algorithm describes how subsets was selected from the families:

Algorithm 1: Select samples from families

Result: Selects representative samples from each family with respect to the date distribution of samples within the family. Prioritize samples with a higher VirusTotal detect rate.
Input: *families:* list of lists containing malware family metadata.
Output: *dataset:* list of selected samples.

```
dataset←empty list
foreach family ∈ families do
    len ←Length of family
    if len ≥ 20,000 then
        select ←2000
    else if len ≥ 10,000 then
        select ←1000
    else if len ≥ 3,000 then
        select ←500
    else if len ≥ 1,000 then
        select ←300
    else if len ≥ 150 then
        select ←150
    else
        select ← len
    end
    family ←sort family on date
    chunks ←split family into 50 chunks
    select ← select/50
    foreach chunk ∈ chunks do
        chunk ← sort chunk on VirusTotal detects, descending
        for i = 0; i < select; i++ do
            dataset ← append chunk[i] to dataset
        end
    end
end
return dataset
```

To increase the quality of the malware family labels, the dataset was re-labeled with new VT reports. AndroZoo is labeled using VT reports dating up to several years back, and some of the reports might contain outdated information. The *vt_report.py* A.2 script was used to fetch updated reports from the VirusTotal API. One must register a free user on VirusTotal to obtain an API key, and the key is limited to only 4 requests per minute. Which is also the reason for why dataset was relabeled after and not before a subset was selected. AVClass [17] was used parse the new VT reports and to assign new labels to the samples. The manual family filtering process was repeated for any new family names occurring.

Finally some samples had to be excluded during the feature extraction process. 13 samples was considered as invalid by AndroGuard[14]. The emulator used in dynamic analysis only support

[14] https://github.com/androguard/androguard

applications up to Android API level 16, and 216 samples with a minimum SDK level of higher than 16 was removed[15]. Lastly, 175 samples could not run in the emulator.

The final dataset consists of 14582 samples from 54 families. A histogram of the date distribution of samples for each quarter is shown in figure 2. The number of samples in each family is displayed in table 5. The dataset construction process is mostly automated, and can be recreated using *construct_dataset.py* found in appendix A.6. *get_dataset.py* A.7 was used to query the AndroZoo API[16] and download the dataset.

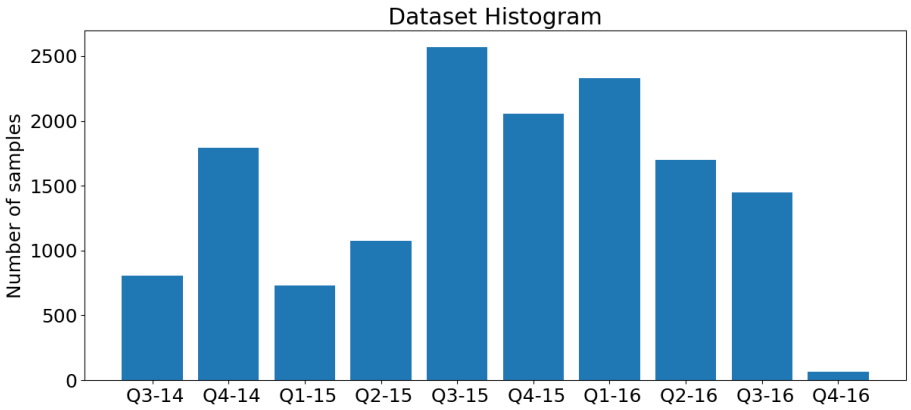

Figure 2: Dataset Histogram

4.3 Feature Extraction and dataset analysis

A broad range of features was extracted in order to find an optimal set of features (research question 3 1.5). Most of the features were chosen to be extracted based on their success in the literature (see related work chapter 3). While other features was chosen in a more or less experimental fashion. The efforts required to extract additional features are minimal, and features with no merit will be removed in the feature selection step.

Some of the feature extraction tools used was designed to count various statistical properties in the dataset while extracting features. All the statistical data is parsed and analyzed by the *apk_statistics.py* script to output tables and figures used in the remainder of this section.

The feature extraction section is structured as follows: The first subsection will discuss statistics related to file types and extensions in the dataset; the remainder of the subsections will describe the different features that was extracted. Note that the native and hidden code features belong to the static feature category. Hidden and native code features are discussed in individual subsections because different tools were used to extract these features.

[15] The *get_min_sdk.py* A.5 python script was used to find the min sdk level for each sample.
[16] https://androzoo.uni.lu/api_doc

Family	Samples	Family	Samples	Family	Samples
kuguo	1,933	mecor	223	hypay	100
dowgin	1,190	kyview	211	adcolony	97
revmob	1,004	fictus	174	ramnit	95
airpush	989	admogo	172	mobidash	93
secapk	775	xinhua	156	xynyin	82
jiagu	645	domob	149	ginmaster	76
smsreg	615	inmobi	147	morepaks	75
baiduprotect	519	igexin	144	silverpush	73
feiwo	497	cimsci	144	systemmonitor	71
leadbolt	488	dianjin	129	cauly	69
gappusin	289	kirko	129	xinyinhe	67
smspay	284	skymobi	127	pircob	64
plankton	283	tachi	123	appflood	59
anydown	260	adflex	121	nineap	52
youmi	257	pandaad	117	wiyun	51
adwo	243	autoins	106	glooken	51
ewind	236	minimob	104	clevernet	50
tencentprotect	225	mobeleader	103	wateh	46

Table 5: Families in the final dataset

A tables enumerating all features of each type can be found in the respective subsections: dynamic features in table 8, static features in table 9, native code features in table 10, hidden code features in table 12 and metadata features in table 14. Features are named to keep track of which category they belong to, starting with D for dynamic, S for static or M for metadata. The second word indicates what sub-category the feature belongs to. Some of the features have a wildcard as part of the feature name, this indicates that there are multiple features of this specific type. For instance the D_Strace_<"sys call"> feature (sys call being the wildcard) represents many different features (e.g. write, read, ioctl, etc). The rightmost column "N", is also used to indicate that there is one or many such features.

4.3.1 APK statistics

APK files are zip archives, and can potentially contain all sorts of files. When extracting features from a dataset of APK files it can be helpful to have an overview over what sorts of file-types that will be encountered during the analysis. If there is a large number of a certain file-type, it might be interesting to included this file-type in the analysis and feature extraction process. Moreover, it was highlighted by suarez-Tangil et al [29] that inconsistencies in file type and file extensions may indicate malicious intent. Statistics about file-types and extensions in the dataset was therefore extracted.

Table 6 shows a ranking of the number of files per file type. File types with a file count that is less than 1500 is excluded. While traversing the dataset, zip archives are unzipped to look for files

Filetype short	Count	Filetype
PNG	4,599,408	PNG image data
Android	1,873,378	Android binary XML
data	798,519	data
ASCII	361,703	ASCII text
compiled	249,611	compiled Java class data
JPEG	210,084	JPEG image data
XML	108,146	XML 1.0 document
ELF	77,954	ELF 32-bit LSB shared object
Ogg	76,259	Ogg data
HTML	66,457	HTML document
UTF-8	58,143	UTF-8 Unicode text
MPEG	44,483	MPEG ADTS
Audio	39,397	Audio file with ID3 version 2.3.0
RIFF	30,787	RIFF
Targa	28,512	Targa image data - RLE 208 x 65536 x 10 +1 +28 ""
exported	27,334	exported SGML document
Dalvik	24,113	Dalvik dex file version 035
Java	24,046	Java serialization data
PE32	23,376	PE32 executable
Standard	21,666	Standard MIDI data
ISO-8859	18,802	ISO-8859 text
C	15,688	C source
GIF	15,221	GIF image data
Zip	14,487	Zip archive data
TrueType	8,243	TrueType Font data
Lua	6,852	Lua bytecode
gzip	6,542	gzip compressed data
ISO	6,411	ISO-8859 text
AppleDouble	5,909	AppleDouble encoded Macintosh file
empty	5,103	empty
SQLite	4,263	SQLite 3.x database
Macromedia	3,878	Macromedia Flash data
SVG	3,339	SVG Scalable Vector Graphics image
PGP\011Secret	3,277	PGP\011Secret Key -
Apple	2,399	AppleDouble encoded Macintosh file
LZMA	2,042	LZMA compressed data
DOS	1,983	DOS executable
Microsoft	1,933	Microsoft OOXML
Non-ISO	1,670	Non-ISO extended-ASCII text

Table 6: File type statistics

within the archives as well. The Python library, "Python-magic"[17] was used to identify the file type of each file. "Filetype short" is the first word in the filetype, and "Filetype" is the largest count of a specific file type starting with the "Filetype short" word. Multiple file types may start with the same word, especially considering that file type information can contain detailed information.

From the file type ranking we can see that the APK files in the dataset contains a wide range of different file types. Commonly the focus of Android malware feature extraction is limited to the AndroidManifest.XML and Dalvik executable files. Additionally features extracted from ELF files, or "Native code" files have shown promising results in[29, 18]. From the ranking we can see several executable binary files where malicious code potentially could be hidden (e.g. Lua, DOS or PE32).

The number of file extensions for some of the more curious extensions is shown in 7. We can see that ".png" is the most common extension. However, if we compare the two tables we can see there are about 24000 more ".png" extensions than there are PNG files. We can also see that there are more ELF files that there are .so (Shared Object) extensions. This extension is commonly used for Native code libraries.

Extension	Count
png	4,623,367
zip	10,582
dex	24,384
bin	15,871
dll	24,474
so	70,619
jar	6,205
lua	23,752
js	38,952
apk	2,437

Table 7: File extension statistics

The data about file types and extensions was extracted using the *extract_native_code.py* A.10 tool.

4.3.2 Dynamic features

The dockerized version of AndroPyTool [26] was used to dynamically analyze the dataset and monitor the behavior of each sample. AndroPyTool has integrated a slightly modified version of DroidBox, which is used for the dynamic analysis part.

Droidbox [25] is a sandbox that performs dynamic analysis of Android applications. DroidBox runs applications in an Android emulator to monitor events such as file access, network traffic, SMS/phone activity, crypto usage, started services and dynamically loaded dex files. DroidBox is implemented for an Android Virtual Device (AVD) targeting Android 4.1.2. Meaning that only applications with a minimum SDK level of 16 or lower can be run in DroidBox[18]. DroidBox uses the

[17] https://github.com/ahupp/python-magic

[18] 216 samples has a minimum SDK level higher than 16, and had to be removed during this step

Monkey tool[19] to generate pseudo random user events at run-time such as clicks and touches. These events can be used to stress test an application, or to trigger the malicious payload. The modifications made to DroidBox by Garcia et al [38] includes: scaling up the number of user events, allowing samples to be run in non GUI mode (enabling parallization) and including the Strace tool inside the AVD used by DroidBox. Strace enables tracking of system calls performed during run-time at the Linux level [38].

Id	AndroPyTool (Strace)	Example	Type	N
1	D_Strace_<"sys call">	write	Numeric	*
2	D_Strace_Pid36_<"sys call">)	ioctl	Numeric	*
Id	AndroPyTool (DroidBox)	Example	Type	N
3	D_Network_UniqueOpenedConnections		Numeric	1
4	D_Network_UniqueClosedConnections		Numeric	1
5	D_Network_SizePackets		Numeric	1
6	D_Network_SizePackets_recv		Numeric	1
7	D_Network_SizePackets_sent		Numeric	1
8	D_Network_OpenedConnections		Numeric	1
9	D_Network_ClosedConnections		Numeric	1
10	D_Network_SentPackets		Numeric	1
11	D_Network_ReceivedPackets		Numeric	1
12	D_Crypto_Uasage		Numeric	1
13	D_CryptoOperation_<"operation">	encryption	Numeric	*
14	D_Cryptoalgorithm_<"algorithm">	AES	Numeric	*
15	D_Dataleak_Type_<"type">	netwrite	Numeric	*
16	D_Dataleak_Size		Numeric	1
17	D_Dataleak		Numeric	1
18	D_Receivers		Numeric	1
19	D_Filesystem_AccessedFiles		Numeric	1
20	D_Filesystem_<"operation">	read	Numeric	*
21	D_Filesystem_Fileaccess_<"path">*(numbers are excluded from path because of different PIDs)*	/proc/version	Numeric	*
22	D_SMS_sent		Numeric	1
23	D_Phonecalls		Numeric	1
24	D_DexClassUsage_<"type">	dexload	Numeric	*
25	D_DexClassUsage		Numeric	1
26	D_StartedServices		Numeric	1
27	D_EnforcedPermissions		Numeric	1
28	D_EnforcedPermission_<"permission">	FLASHLIGHT	Boolean	*

Table 8: List of dynamic features

AndroidPyTool has hard-coded the run-time for applications in DroidBox to 300 seconds. Con-

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

sidering the large amount of samples in the dataset, the run-time was reduces to 120 seconds to finish the feature extraction in a timely manner[20]. The cost of reducing the analysis time is that the malicious payload might not be triggered for families using a time-based triggering mechanism. AndroPyTool outputs formatted analysis logs for each sample. *preprocessing.py*A.1 was used to parse these logs and construct dynamic features.

The complete list of dynamic features extracted is shown in table 8. The occurrence of each unique system call was counted for all process IDs (PIDs) and used as features. Additionally, system calls made from PID 36 is counted individually (the Strace tool is run for the PID 36, and parameter to include forks are set). The DroidBox related features are mostly constructed based on the count of various events and their type/operation. UpDroid [6] also uses DroidBox for dynamic feature extraction, and several of the features was constructed based on their success in UpDroid. The list of features used by Aktas and Sen for UpDroid can be found in table 2 in [6].

4.3.3 Static features

The AndroPyTool [26] docker image was used to extract the static features that are most commonly used in the literature. These features include Permissions, API calls, Opcodes, Intents, Receivers, Services, Activities, Strings and System commands.

AndroPyTool uses the APKtool[21] to decompress the AndroidManifest.xml file and to disassemble the Dalvik executable files into Smali files. Smali files are text files containing Dalvik Opcodes. A complete list of the Opcodes and their descriptions can be found here [21]. AndroPyTool parses smali files to count the occurrence of API calls, opcodes and Strings. The system command feature are the count of any string that is equal to a system command (e.g. ls, chmod or su). The complete list of system calls can be found in the AndroPyTool Github repository[22]. Permissions are obtained using the Androguard[23] python tool. The remaining features are extracted from the decompressed AndroidManifest.xml file [26].

For each sample, AndroPyTool outputs a json file with the previously described features. *preprocessing.py*A.1 was used to construct some additional features based on the json files. The total number of strings, and strings of certain lengths were counted. The string lengths were selected in an experimental fashion. The use of strings with a certain length may indicate use of encrypted strings. The total number of permissions defined in the Android permissions and other permissions were counted and used as features. Each individual Permission was used as a boolean feature. Finally the number of activities, services and receivers were simply counted. A complete list of these features are shown in 9.

[20] This value can be altered by the following steps: **(1)** Run the AndroPyTool docker image with "/bin/bash" as entry-point (the default entry-point is the androPyTool.py script). **(2)** Replace the value "300" in androPyTool.py with the desired run-time (e.g. sed -i 's/300/120/g' androPyTool.py)". **(3)** Run androPyTool.py with the desired parameters

[21] https://ibotpeaches.github.io/Apktool/

[22] https://github.com/alexMyG/AndroPyTool/blob/master/info/system_commands.txt

[23] https://github.com/androguard/androguard

Id	AndroPyTool (Static)	Example	Type	N
1	S_Permission_<"permission">	android.permission.SEND_SMS	Boolean	*
2	S_Permission_NumAndroidPermissions		Numeric	1
3	S_Permission_NumCustomPermissions		Numeric	1
4	S_Opcode_<"opcode">	xor-long	Numeric	*
5	S_ApiCall_<"apicall">	android.telephony.-TelephonyManager.getImei	Numeric	*
6	S_ApiPackage_<"api package">	android.telephony	Numeric	*
7	S_String_len_<"len"> (100,200,300,400,500,100,2500,5000, 10000,15000,20000)	5000	Numeric	*
8	S_Strings		Numeric	1
9	S_SystemCmd_<"cmd">	su	Numeric	*
10	S_Intent_<"intent">	com.manager.msg	Numeric	*
11	S_Intents		Numeric	1
12	S_Activities		Numeric	1
13	S_Services		Numeric	1
14	S_Receivers		Numeric	1

Table 9: Static features extracted with AndroPyTool

4.3.4 Native code features

Native code is used in Android application for optimization purposes, but can also be used by malware authors to hide malicious code[18] (See description in 2.1). Native code files are written in c++ and compiled into Executable and Linkable Format (ELF) files for the Android Linux environment. Features extracted from native code files has shown merit in both RevealDroid [18] and DroidSieve [29], and was therefore chosen to be included in the feature-set for this project. *extract_native_code.py* was used to extract native code features and statistics about the native code usage in the dataset.

Suarez-Tangil et al [29] used features extracted from the ELF header and individual sections. These features includes: number of entries in the program header, program header size, number and size of section headers and boolean features based on which flags were set for individual sections (e.g. Read, Write and execute). Suarez-Tangil et al [29] also constructed features based on inconsistencies between file type and file extension. In addition to the features used in DroidSieve, the size of each section was extracted.

Garcia et al [18] used the number of calls made to individual functions in the Procedure Linkage Table (PLT) as features. The PLT is used to determine the address of an external functions that is unknown at linking time, and is therefore more difficult to obfuscate[18]. The external call features used by RevealDroid was included in the feature set.

Other feature that was used is the number of files found in the APK for each architecture, and the combined size of the native code files compiled for ARM (ARM is the most used architecture 11, discussion will follow). The complete list of features can be found in table 10

Id	NativeCode	Example	Type	N
1	S_NC_<"arch">	ARM	Numeric	*
2	S_NC_ProgramHeaders		Numeric	1
3	S_NC_ProgramHeader_Size		Numeric	1
4	S_NC_Sections		Numeric	1
5	S_NC_SectionHeader_Size		Numeric	1
6	S_NC_Section_<"name">_size	.text	Numeric	*
7	S_NC_Section_<"name">_Flag_<"flag"> (Flags: [R,W,A,X etc])	.text	Boolean	*
8	S_NC_<"arch">_pltCall_<"call">	fputs	Numeric	1
9	S_NC_IncorrectExtensions		Numeric	1
10	S_NC_IncorrectExtension_<"ext">	png	Boolean	*
11	S_NC_ARM_Size		Numeric	1

Table 10: List of native code features

In order to find all native code files, the *extract_native_code.py* A.10 tool traverses the APK-file, and any nested APK-files or other archives within the APK-file. The python-magic library [40] was used to identify file types.

Native code features were only extracted from ELF files compiled for the ARM architecture. The vast majority of the ELF files in the dataset was compiled for ARM, and most of the ELF files that weren't compiled for ARM was simply the same code compiled to support additional architectures (See table 11).

The ELF files was disassembled using the objdump Linux tool compiled for ARM. The dissemble was parsed, and external calls referencing the PLT section were counted. In order to extract header and section features from the ELF files, the Pyelftools library was used. Pyelftools [41] is a python tool for parsing and analyzing ELF files.

Statistics about the native code usage in the dataset were extracted alongside the features. Figure 3 displays the usage of native code over time. From the figure we can see that the use of native code increased in the start of 2015.

Native code architecture related information were counted, as shown in table 11. Approximately 2/3 of the samples in the dataset contains native code files. The average a sample contains several native code files. The majority of native code files are compiled for ARM, and more than 3/4 of the samples that are compiled for another architecture has the same name as the ones compiled for ARM. This is common for applications that support multiple architectures (see example in 2.1).

4.3.5 Hidden code features

Features relating to API calls, system commands, strings, file size and extensions were extracted from "hidden code files". In this project "hidden code files" is used as a term to describe any JAR, APK or none-standard DEX files found within the APK file being analyzed. As described in the background chapter 2.1, APK files must contain a dex file named *classes.dex*. In case the source code of the application is too large to be contained within a single dex file, an additional file named

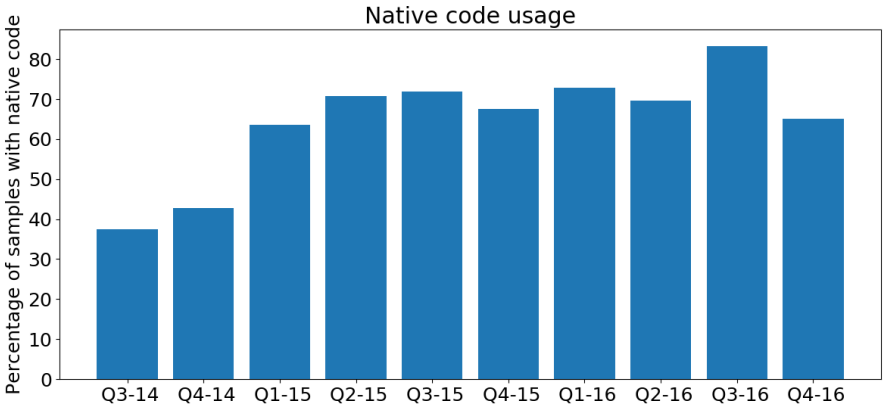

Figure 3: Native Code usage

Samples	14582
Samples with NC	9668
NC files	76751
Architecture ARM	55414
Failed to analyse ARM	1675
Not Arm same name	16400
Not Arm different name	4937

Table 11: Native code architecture stats

classes2.dex may be used. Any other dex file is considered a hidden code file, including *classes.dex* files found in nested APK files[24]. The tool used by AndroPyTool to disassemble DEX files (APKTool) only disassembles the standard dex files. Any none-standard DEX files are therefore not included in the static analysis of AndroPyTool. The complete list of hidden code features are displayed in table 12.

These features were inspired by DroidSieve, where Suarez-Tangil et al [29] extracted features from what they called "incognito apps". Incognito apps are any APK or additional DEX files found within the original APK file. Furthermore Suarez-Tangil et al [29] highlights that malicious payloads can be disguised within the assets of an APK file.

The *extract_hidden_code.py* A.9 script was used for feature extraction. Similar to the native code extraction tool described in the previous section, a function is used to recursively traverse an APK file to look for hidden code files. The dex2jar tools [22] are used to convert JAR files to DEX files, and to disassemble DEX files into smali. The smali files are then parsed to count API calls, system commands, the total number of strings, and strings of certain lengths (the parsing algorithm is inspired by AndroPyTool[26]). The same list of known system commands used by AndroPyTool are

[24]Nested APK files: any APK file found within the APK file being analyzed

Id	Hidden Dex/Jar files	Example	Type	N
1	S_H_File_Apk_size		Numeric	1
2	S_H_File_Jar_size		Numeric	1
3	S_H_File_Dex_size		Numeric	1
4	S_H_incorrectExtension		Numeric	1
5	S_H_incorrectExtension_<"ext">	png	Boolean	*
6	S_H_NumAndroidApiCalls		Numeric	1
7	S_H_NumOtherApiCalls		Numeric	1
8	S_H_Strings		Numeric	1
8	S_H_String_len_<"len">	5000	Numeric	*
9	S_H_String_<"string"> (Only strings containing the word "invoke-")		Numeric	*
10	S_H_SystemCmd_<"cmd">	su	Numeric	*
11	S_H_Dex_ApiCalls_<"api call">	android.telephony.- TelephonyManager.getImei	Numeric	*
12	S_H_Jar_ApiCalls_<"api call">	android.telephony	Numeric	*

Table 12: List of hidden code features

loaded from file, and strings that match any entry in the list are counted. API calls can be identified in the smali code by looking for the "invoke-" opcode. The API calls are compared to a list[25] of known Android API calls loaded from file. The total number of Android and not Android API calls were counted. If the API call match an entry in the list, a feature for each part of the API call is incremented. For instance if the API call is "android.telephony.TelephonyManager.getImei", the 4 following features will be incremented:

- "S_H_Dex_ApiCalls_android"
- "S_H_Dex_ApiCalls_android.telephony"
- "S_H_Dex_ApiCalls_android.telephony.TelephonyManager"
- "S_H_Dex_ApiCalls_android.telephony.TelephonyManager.getImei"

Android API calls extracted from DEX and JAR files were counted individually.

During the implementation and testing of the *extract_hidden_code.py* tool, some strings containing "invoke-" (the opcode used to invoke API calls) were identified. These strings were counted and added to the feature set. Finally, features related to incorrect extensions and total size of APK, JAR and DEX files were added.

Hidden code statistics were extracted alongside the features. The usage of hidden code has increased over the timeline in the dataset as shown in figure 4.

The different hidden code files and their extensions were counted as shown in 13. Only about 1/5 of the samples contains hidden code files. Most of these files are DEX files, and only a very small amount of JAR files was found. The extensions used for DEX and JAR files were mostly consistent, but only about 1/3 of the APK files had the ".apk" extension.

[25]The list of known Android API calls are scraped from the Android API reference[42] using the *scrape_android_api.py* A.3

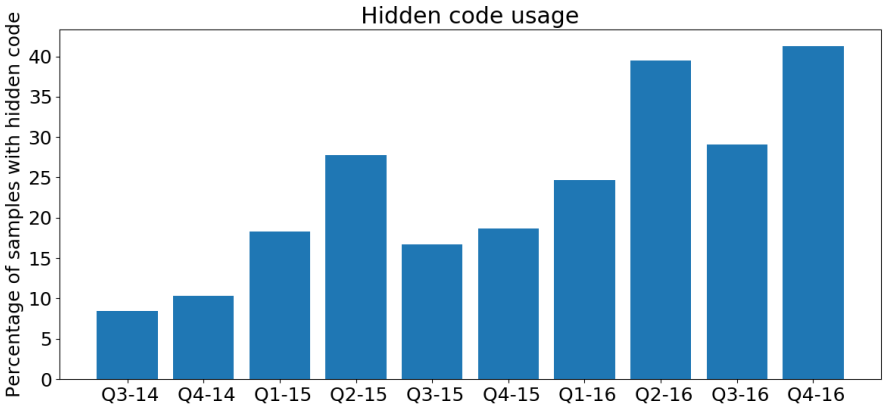

Figure 4: Hidden Code usage

Samples	14582
Samples with HC	3191
Dex files	7351
Dex correct extensions	7345
Dex cannot disassemble	0
APK files	4304
APK correct extensions	1510
Jar files	319
Jar correct extensions	289
Jar cannot disassemble	0

Table 13: Hidden code file and extension stats

4.3.6 Metadata features

The APK and DEX size from the AndroZoo metadata was used as features 14. Out of these, APK size has already shown merit for familial classification in UpDroid [6].

Id	Metadata	Type	N
1	M_Metadata_APKSize	Numeric	1
2	M_Metadata_DEXSize	Numeric	1

Table 14: List of metadata features

4.4 Feature extraction time

For real-world implementations it is important to understand the complexity affiliated with feature extraction and how this affects identification of malware samples. In order to answer *research question 4* 1.5, the feature extraction time for each of the methods described in the previous section 4.3

must be measured.

Due to the size of the dataset, the feature extraction process had to be run in parallel on different machine with different resources. The dataset was split in an arbitrary manner to distribute the load. Resulting in splits that are not likely to be representative for the dataset. The results from the feature extraction process is therefore not suitable to answer the research question.

Instead a subset of 100 samples was selected to measure the mean feature extraction time for a sample. Samples was selected based on the assumption that APK file size is correlated with feature extraction time. The larger the APK file the more time is required to extract features. The subset should therefore have a similar APK size distribution to the dataset. The same methodology and assumption is made in [18], where Garcia et al measures the feature extraction time of RevealDroid.

The following algorithm is used to extract a representative subset from the dataset:

Algorithm 2: Select samples from families

Result: Selects a representative subset of 100 samples with respect to the APK size distribution in the dataset.
Input: *dataset:* list of dataset metadata
Output: *subset:* list of hashes

```
subset←empty list
dataset← sort dataset on APK size
chunks← split dataset into 100 chunks
foreach chunk ∈ chunks do
    len← Length of chunk
    i←random int in range [0, len]
    subset← append chunk[i] to subset
end
return subset
```

To make the results comparable, all the feature extraction methods were run in the same system, with the same resources available.

4.5 Machine learning methods

The Weka data mining software [43] was used for all machine learning methods in the project. Weka offers a large collection machine learning algorithms, and is a very popular tool in the literature.

The remained of the section will discuss the machine learning algorithms we used for classification, and feature selection.

4.5.1 Classification

We cannot assume that one machine learning algorithm will perform best for an untested dataset. As show in table 2, different machine learning classifiers have shown promise in the literature. SVM and CART had similar accuracy in RevealDroid, although Garcia et al [18] found that CART had a performance advantage over SVM for family classification. J48, Random Forest and k-NN with standard Weka parameters was used in UpDroid [6]. DroidSieve [29] used the ExtraTrees algorithm. We therefore chose to evaluate a wide range of classifiers on the dataset. Also, having a basis for

what level of accuracies can be achieved with the different algorithms can lay the groundwork in case of future work.

The remainder of the section will give a brief description of each machine learning classifier used in the project.

Classification And Regression Trees (CART)

CART is a decision tree algorithm that can construct trees for regression or classification. Each node in the tree corresponds to an attribute, and the branches corresponds to the results of a condition for splitting (e.g. $x > 5$). The leaf nodes corresponds to class predictions. Gini Index is used to measure the purity of leaf nodes, and determine which splits to take in the learning phase.

Random Forest

Random forest generates an ensemble of decision trees. The votes of each tree is used to classify a new sample [44].

Extra Trees

Extra-trees is a tree based ensemble method similar to Random Forest. Using a top-down approach, the algorithm generates unpruned decision or regression trees. The choice of attribute and cut-point is strongly randomized during the splitting of tree nodes. One of the big advantage of the ExtraTrees algorithm is the computational efficiency [45].

Extra Tree

A single extra tree.

Rotation Forest

Rotation Forest is method used to build an ensemble classifier based feature extraction. The feature set is randomly split into K subsets, and Principle Component Analysis (PCA) is used on each subset. The principle components are arranged in a rotation matrix. K axis rotations are made to create a new training set for a tree based classifier [46].

Support Vector Machine (SVM)

Support Vector Machine is a discriminant classifier that uses hyper-planes in the attribute space to separate the classes. The learning examples nearest the hyper-plane is called support vectors. The optimal hyper-plane has an equal (and thereby maximal) distance to support vectors of the two classes it is separating. For non linear problems the attribute space is transformed into a higher dimension with the help of kernel functions [44]. The Weka implementation of SVM is a wrapper for LIBSVM, a well known library for Support Vector Machines [47].

Multi-Layer Perceptron (MLP)

Multi-layer Perceptron is an artificial neural network algorithm used to mimic a biological neural network with the help of abstraction. In the abstraction neurons are nodes that can summarize input and normalize output. The neurons are interconnected to create a neural network. MLP consists of multiple layers: An input layer of nodes representing the attributes, one or more hidden layers, and a output layer representing the classes. The attribute values are sent to the input nodes, and

the weight on the connections between the nodes determine the output value, and thereby the class prediction. In the learning phase the weights on the connections are updated according to classification error [44].

K-nearest neighbors (k-NN)

K-nearest neighbours is considered a lazy classifier because it simply uses the learning set as it's knowledge base. In order to classify a new example, k-NN finds k learning examples that are most similar to the new example based on some distant metric. The new example is labeled with the class that is prevalent among it's neighbors [44].

Bayesian Network

A Bayesian classifier calculates the conditional probabilities for all the classes. The Bayesian network uses directed acyclic graphs to model the dependencies between attributes and class. Conditional independence is assumed between nodes that are not directly connected [44].

Naive Bayesian

The Naive Bayesian classifier assumes independence between attributes, and is therefore considered to be "naive" or simple.

4.5.2 Feature selection

Feature selection are methods used to reduce the feature space. Features with good qualities for distinguishing classes are selected, and irrelevant and redundant features are removed. Reducing the feature space (also known as dimensionality reduction) can have several benefits such as: giving a better understanding of the Android malware domain by finding features useful for identifying malware families, more efficient time and space complexity for the classification algorithm, can lead to more accurate classification results, but can also be considered as a trade-off between accuracy and model complexity. Feature selection may also help avoid the curse of dimensionality[26] and overfitting[27] the model.

Feature selection methods can generally be categorized as filter[28] and wrapper[29] methods. Filter methods are fast but does not always fit the classifier, while wrapper methods are slow but always fit the classifier. Because we are using a wide range of different classifier algorithms as discussed in the previous section, we relied exclusively on filter method to finish the experiment in a timely manner.

The selected feature subsets that produces the best results between the classifiers will be used to answer **research question 3**.

[26] **Curse of dimensionality** Phenomena that is used to describe problems that can arise when having to deal with a high-dimensional feature space (data points times number of features)[44]

[27] **Overfitting:** Overfitting occurs when the classifier fits the training data too closely. The classifier will have good performance for the training data, and bad performance for future observations (testing data). Overfitting is likely in cases where the training set is too small and the model is adjusted to random features that have no correlation with the target classes.

[28] **Filter methods:** Filter methods that ranks features independently of the machine learning algorithm that will be used for classification

[29] **Wrapper methods** Wrapper methods evaluates a subset of the features using the same machine learning algorithm that will be used for classification.

The following subsections will give a brief description of the feature selection methods that was used in the experiment.

Information Gain

Information Gain measures the amount of information an individual attribute provides about a class [44]. Information gain uses entropy as an impurity measure. A decrease in impurity results in information gain.

ReliefF

ReliefF is an extension of the RELIEF algorithm, designed to handle multi-class problems. RELIEF takes the context of other attributes into account when measuring the merit of an attribute. For each learning example in a random subset, RELIEF finds the nearest examples of the same class and the nearest example of the opposite class. The quality of the Attributes are then updated by their undesired and desired properties. The undesired property is an attributes ability to differentiate the two examples of the same class, and desired property is the attributes ability to differentiate the two examples from opposite classes. Thereby evaluating an attributes local ability to differentiate between classes, and by using locality, the context is implicitly taken into account. In order to evaluate attributes in multi-class problems reliefF searches for the k nearest examples from each class. Prior probabilities are used to weight the contributions of the different classes [44].

Correlation-based Feature Subset Selection (CFSSubsetEval)

CFSSubsetEval selects a subset of attributes that have high correlation with the different classes, while having low intercorrelation. The quality of the subset is evaluated by the attributes individual predictive ability, and their degree of redundancy [48].

Search methods

The ranker method in Weka was used to select attributes evaluated by information gain and reliefF. The ranker method ranks features bases on their individual merit, and enables subset selection by setting a threshold or selecting the top n features [43]. A subset was selected by setting a lower threshold for merit of each attribute. Different thresholds was tested with different classifiers in order to find an optimal threshold.

CFSSubsetEval does not evaluate attributes individually, and a search method must be used to identify an optimal subset of attributes. The BestFirst search method in Weka with recommended parameters was used to find a optimal subset. The BestFirst search uses greedy hillclimbing augmented with a backtracking facility to search the attribute space for a subset [43].

4.6 Evaluation

To investigate research question (RQ) 1 and 2, the classifiers must be evaluated in different settings. RQ1 requires that classifiers are evaluated in a time-aware and time-unaware setting, while RQ2 requires that these classifiers are evaluated using different feature subsets (dynamic[30], static and

[30]Note: The two metadata features were not part of the static and dynamic analysis process, and was used in the dynamic feature set (see 6.1).

hybrid).

To evaluate the performance difference between a classifier in a time-aware and time-unaware setting, the dataset should be split into training and testing sets in a similar manner.

A common approach to evaluate the performance of a classifier in a time-unaware setting is to use k-fold cross validation (k is usually 10) [44]. In k-fold cross validation the dataset is split into k chunks. k-1 chunks are used for training and 1 chunk are used for testing. A classifier is trained k times so that each chunk is used for testing. The final evaluation is the average of the k tests. For classification the splits for k-fold cross validation is usually made in a stratified manner. Meaning that the original class distribution is preserved in the training and testing set [44].

If 10-fold cross validation is used for the time-unaware setting, the training/testing distribution should be the same for the time-aware setting. Meaning that 9/10 of the dataset should be used for training, and 1/10 of the dataset should be used for testing. In order to have a more substantial testing set that increases the impact of time, 5-fold cross validation was selected.

Splitting the dataset in a stratified manner for the time-aware setting will create an unrealistic scenario, but will be more comparable to the time-unaware setting. Two different time-aware splits were therefore made. **Time-aware split 1**: the dataset is split using hold-out validation in a stratified manner. **Time-aware split 2**: the dataset is split on a certain date on the timeline. The *preprocessing.py* script A.1 was used to construct training and testing sets for all the different settings. Details about the time-aware splits are described in the following subsections.

4.6.1 Time-aware split 1

The following algorithm was used for time-aware split 1:

Algorithm 3: Time-aware Split 1

Result: Splits the dataset into training and testing sets with properties similar to stratified 5-fold cross validation. The split is only time-aware within the families.
Input: *families:* list of lists containing malware family metadata.
Output: *trainingset; testingset*

```
trainingset←empty list
testingsett←empty list
foreach family ∈ families do
    family ←sort family on date
    split ←split family into 5 chunks
    trainingset ←first four chunks in split
    testingset ←last chunk in split
end
return trainingset, testingset
```

The properties of the split made by the algorithm is similar to the properties of the splits made in stratified 5-fold cross validation. A stratified distribution is maintained in the training and testing set, and the size of the training and testing set is equal to that of 5-fold cross validation. The earliest, latest and split date for each family is displayed in figure 6. As shown in the figure the split is only "time-aware" within the families, and not for the dataset as a whole. An abstract version of the split

is shown in figure 5

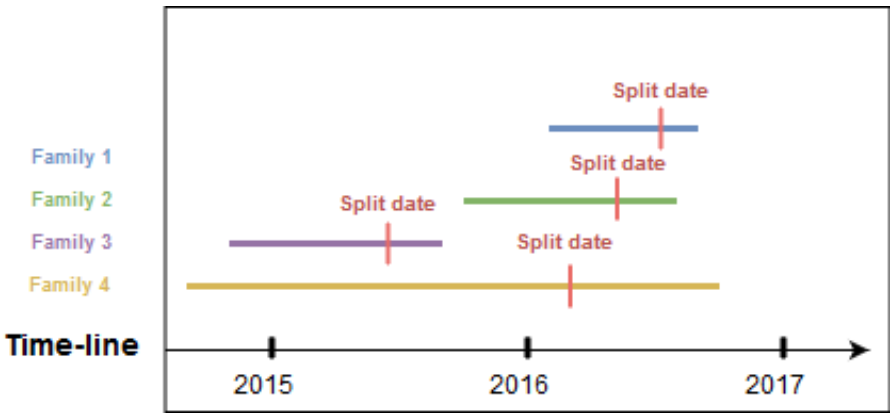

Figure 5: Time-aware split 1

4.6.2 Time-aware split 2

A more realistic setting would be to split the entire dataset on a certain date, lets call it the "split date". All samples that are dated earlier than the split date will be used for training, and all samples dated after will be used for testing. An illustrative example is shown in figure 7. Splitting the dataset in such a manner results in some families only having samples in the training set (family 3), and other families only having samples in the testing set (family 1). These scenarios are also true in the real world. Some families will go inactive or die out, and new families will be introduced.

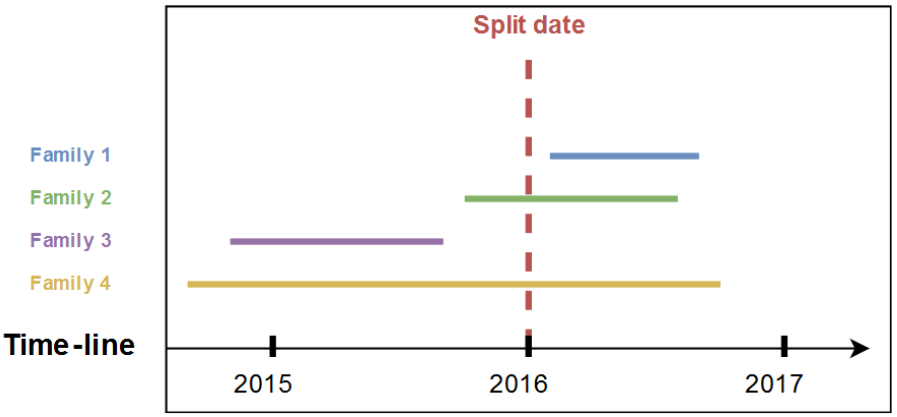

Figure 7: Time-line split 2

The classifiers used in this project have no way of handling samples from unknown families, and any such sample will directly attribute to decrease in accuracy. Families with no samples in the

training set were therefore removed.

The first day of 2016 was selected as the split date, resulting in the split shown in table 15. 62% of the samples in the dataset are used to training and, the remaining 38% is used for testing. Only 2 families and 325 samples had to be removed from the testing set (the tencentprotect and hpay family).

Family	training	testing	Family	training	testing	Family	training	testing
admogo	92	80	mecor	222	1	morepaks	25	50
jiagu	82	563	gappusin	253	36	xynyin	81	1
secapk	506	269	ginmaster	52	24	smspay	84	200
feiwo	401	96	mobidash	28	65	autoins	63	43
revmob	894	110	baiduprotect	189	330	adflex	94	27
kuguo	978	955	ramnit	28	67	inmobi	14	133
cimsci	45	99	systemmonitor	47	24	silverpush	62	11
leadbolt	414	74	hypay	0	100	domob	129	20
cauly	63	6	clevernet	30	20	tachi	100	23
youmi	195	62	kyview	72	139	kirko	124	5
dowgin	1015	175	glooken	51	0	pircob	50	14
minimob	88	16	skymobi	99	28	dianjin	52	77
adwo	129	114	xinhua	95	61	appflood	56	3
ewind	212	24	igexin	97	47	adcolony	97	0
plankton	232	51	anydown	242	18	wiyun	35	16
smsreg	369	246	wateh	37	9	mobeleader	103	0
airpush	386	603	fictus	51	123	pandaad	63	54
tencentprotect	0	225	nineap	46	6	xinyinhe	67	0
Total	9039	5543						

Table 15: Time-aware split 2

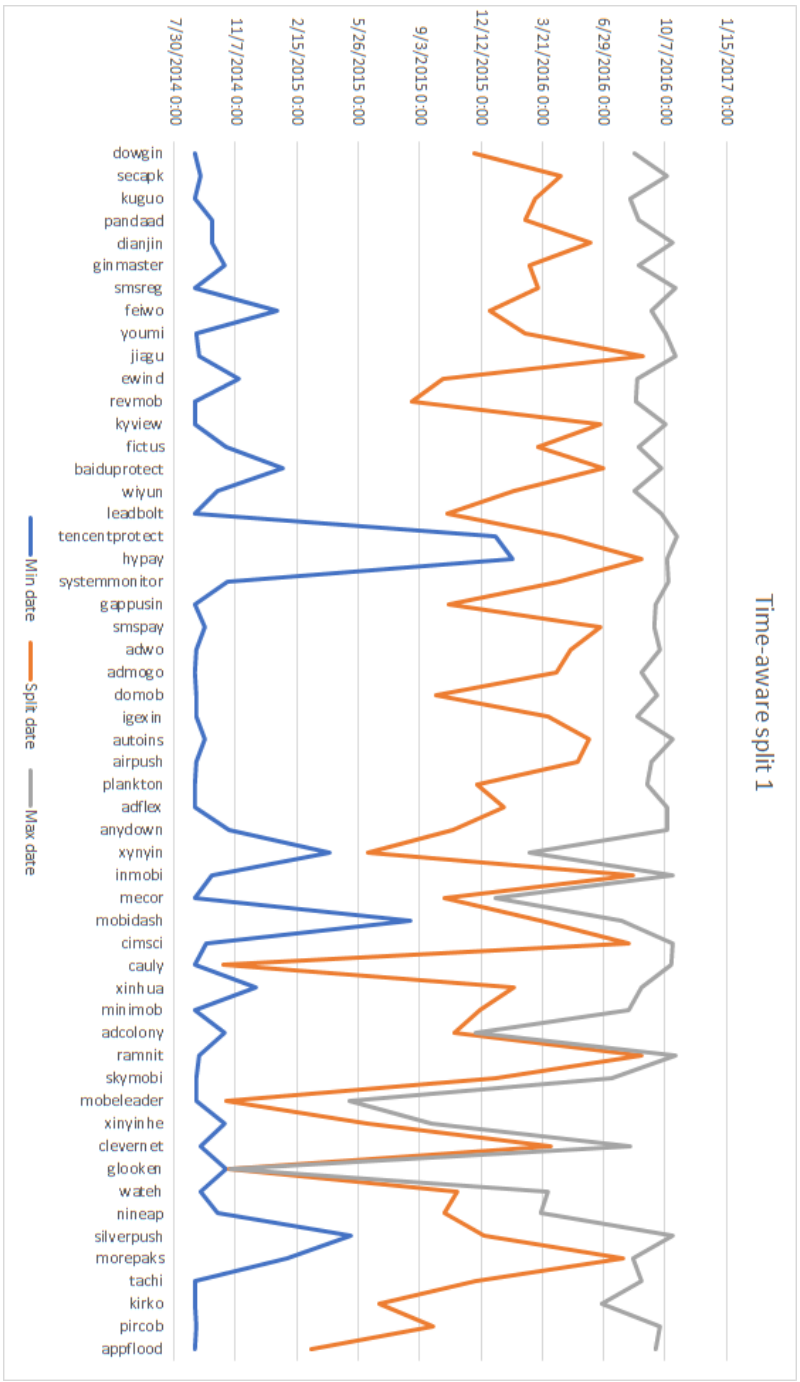

Figure 6: Time-aware split 1

5 Experimental Setup

This chapter will discuss the experimental setup and work flow used to accomplish the practical part of the project. First the computational resources will be described. Next the dynamic analysis environment will be explained. Followed by a description of the work-flow and tools that was used. Finally the parameters and thresholds used for the machine learning algorithms will be discussed.

Many of the tasks in this project requires a considerable amount of computer processing time due to the large size of the dataset. Based on initial estimations, the feature extraction alone would have to run for more than 38 days[1] straight, even without taking errors and crashes into consideration. Fortunately most of the tasks can be parallelized to the extent of CPU cores and memory in the project environment. Running the feature extraction in two processes would then only require 19 days, and 9.5 days for 4 processes, etc. Parallelization was therefore used as much as possible to complete the project in a timely manner. The authors personal computer and a VM provided by the university was used for the experiment. The machine specifications are shown in table 16.

	PC specs	PC-VM specs
OS	Windows 10 educational 64-bit	Ubuntu 18.04 LTS 64-bit
CPU	Ryzen 2700X, 8-Core, 16-Thread, 4.35GHz, 20MB cache	8-Cores
RAM	HyperX Fury DDR4 2666MHZ, 16GB	12GB
SSD	Samsung 970 EVO 500GB, 3500/3200 MB/s read/write	200GB
	Uni-VM specs	**Nested VM**
OS	Ubuntu 18.04 LTS 64-bit	Ubuntu 18.04 LTS 64-bit
CPU	CPU E5-2630 v3 @ 2.40GHz, 16-Cores	8-Cores
RAM	32GB	12GB
HDD	3TB	500GB

Table 16: Machine specifications

5.1 Dynamic analysis environment

In order to avoid any unnecessary risks, it is important that malware analysis is conducted in a safe environment. Running the samples in an isolated virtual machine (VM) is consider as best practise to prevent infecting the host [49]. The virtual machines used for this purpose is show in the rightmost column of table 16. The VM should not be connected to the internet to ensure that it is not used for malicious activity, such network propagation, spamming or being a node in a

[1] **Estimation based on pilot run of feature extraction tools**: 200 seconds per sample for dynamic analysis, and 20 seconds per sample for static analysis. Resulting in about (15000*220/60/60/24) 38 days. In dynamic analysis 120 seconds was used for analysis, and 80 seconds were used to reset the environment

distributed denial of service attack. Moreover, malware often communicate with a Command and Control (C2) server to receive updates and instructions. Having the VM connected to the internet may therefore inform threat actors that you are analyzing their malware, given that the C2 server are still active [49]. Because of the previously stated reasons, network analysis is a very import aspect of malware analysis, and should be included in the process. To be able to conduct network analysis, the analysis VM was placed in a host-only network. A host-only network is a private LAN isolated from the host machine and the internet [49]. An additional VM running Remnux[2] was placed inside the host-only network to act as the default gateway for the analysis VM. Remnux is a lightweight Linux distribution that incorporates malware analysis tools to make it easier forensic investigators to set up an analysis environment. One of these tools is INetSim[3], a tool designed to simulate the most common internet services. The dynamic analysis setup is shown in figure 8. The dataset was split into 7 parts, 4 parts was analyzed on the Uni-VM, and 3 parts was analyzed on the authors PC.

As discussed in the methodology chapter 4.3, the AndroPyTool [26] running inside a docker container was used to conduct the dynamic analysis. Using docker had several advantages: the dynamic analysis processing running inside the same VM was isolated from each other; docker provides another level of isolation; and running docker in parallel is almost effortless.

5.2 Tools and workflow

The overall work-flow is shown in figure 9. Python scrips was created to automate several of the steps. The following subsections will a give a brief description for each of the steps in the work-flow.

5.2.1 Dataset construction

All the malware samples used in the project was collected from AndroZoo [4]. ***construct_dataset.py*** was used to select a dataset targeting the problem area of the project, and write the dataset metadata to file. The Metadata was collected from AndroZoo. ***get_dataset.py*** queries the AndroZoo API to download the selected dataset. ***vt_report.py*** queries the VirusTotal API[4] for each sample in the dataset to obtain updated reports. AVClass [17] uses these reports to assign new malware family label to each sample. ***re_label.py*** uses the output from AVClass to update the dataset metadata file. ***get_min_sdk.py*** gets the min SDK level of each application. As discussed in the methodology chapter 4.3, DroidBox[25], the dynamic analysis tool integrated in AndroPyTool, only supports up to SDK level 16. Samples with a higher SDK level are removed from the dataset.

5.2.2 Feature extraction/preprocessing

scrape_android_api.py was used to get a list of all Android API calls (the list is used in *extract_hidden_code.py*). ***extract_hidden_code.py*** is responsible for extracting the hidden code features. ***extract_native_code.py*** extracts features related to native code files. AndroPyTool [26] extracts all the dynamic features used in this project, as well as a wide range of static features. All

[2] https://remnux.org/
[3] https://www.inetsim.org/
[4] https://www.virustotal.com/nb/documentation/public-api/

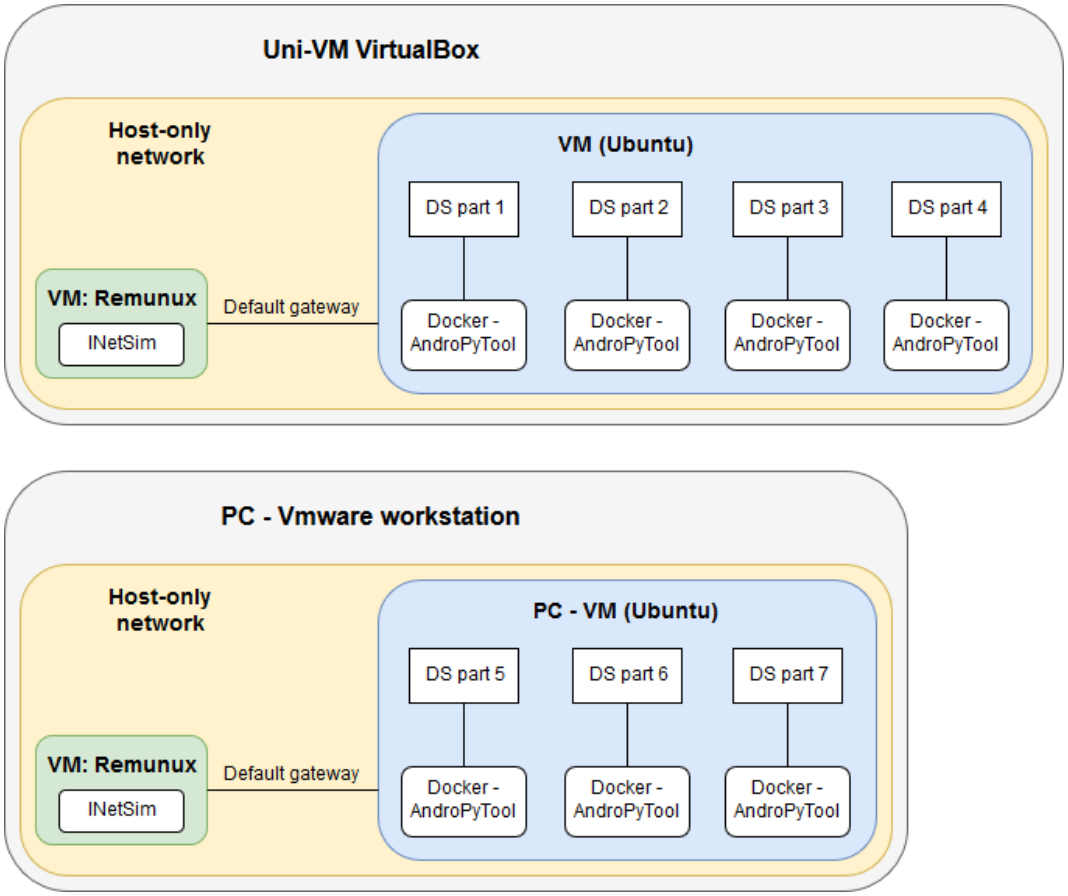

Figure 8: Dynamic analysis setup

the above feature extraction tools extracts features from a single sample at a time. Resulting in 5 feature files per sample in the dataset (droidbox, strace, andropytool-static, native code and hidden code). The benefit of keeping everything separate is that if a feature extraction tool crashes the progress will not be lost.

The ***preprocessing.py*** tool takes the features files as input, and outputs all the different datasets (training and testing sets) in ARFF[5] format for Weka. The tool also screens irrelevant features to reduce the size of the datasets. if a feature is present in less than 3% of the samples within a family it is removed. For instance, if hidden code files are present in 3 out of 100 samples within a family, hidden code features are not really useful for describing this family.

The size of the raw feature files were more than 35GB, which is more than the memory in the VMs 16. ***preprocessing.py*** was therefore designed to keep only the necessary information in

[5] https://www.cs.waikato.ac.nz/~ml/weka/arff.html

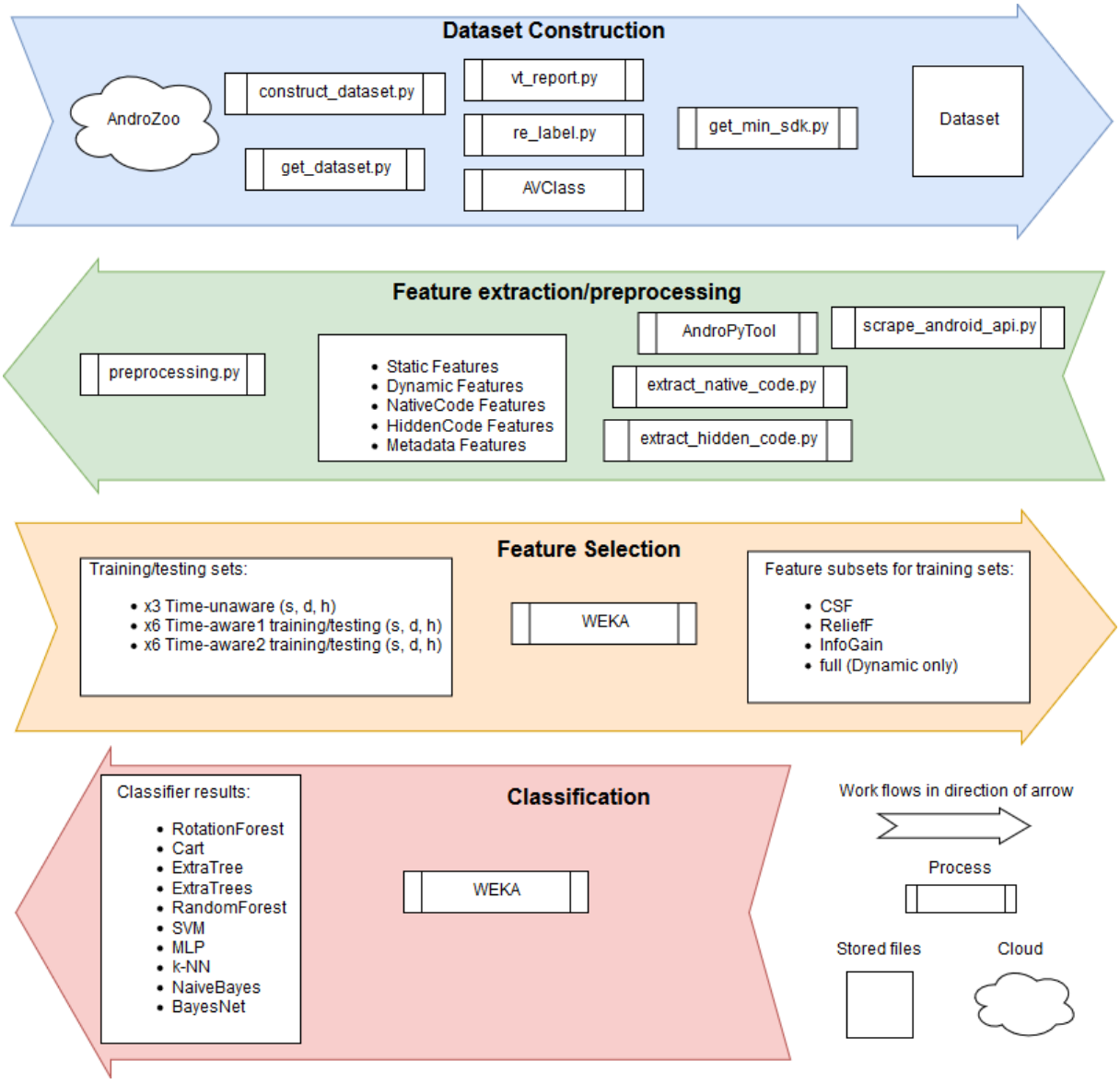

Figure 9: Workflow for the experiment

memory at any time. First all the feature files are parsed to determine the feature set. Feature occurrence within each family is counted, and features with less than 3% occurrence is removed. Next the scripts start to incrementally write the dataset to file, one sample at a time. The features files of a sample will be parsed again. The features that are in the feature set will be written to file, and the value of missing features are set to 0. This approach is very scalable with respect to memory

restrictions, but is slower due to more I/O. The SSD of the PC is very fast, and the time required to create the datasets for Weka was negligible.

5.2.3 Feature selection

All the different training sets created in the previous step was manually run through feature selection methods in Weka. Resulting in three new training sets for each of the original training sets.

5.2.4 Classification

Different classifiers were trained and evaluated in the Weka. This process was done manually.

5.3 Feature selection and classification parameters and thresholds

This section will cover the parameters and thresholds used for machine learning methods in Weka [43].

5.3.1 Feature selection parameters and thresholds

As explained in the methodology section 4, the ReliefF and Information gain method ranks features based on individual merit towards predicting the class. To select a feature subset a lower threshold must be selected, and testing is required to find a optimal threshold. Different thresholds was tested in a time-unaware setting, using 5-fold cross validation to evaluate the classifiers. The Random Forest and SVM methods were chosen based on success in the literature[18, 6], and due to the fact that these methods are fundamentally different. Relying on a single classifier type (e.g. decision trees) may result in a threshold that has a bias towards this type.

Figure 10 shows classifier accuracy for feature subsets selected based on different information gain thresholds. We can see SVM has the best performance for the 0.4 and 0.2 threshold, with slightly better performance for 0.2. Random Forest performed best for thresholds in between 0.3 and 0.8, with a peak at 0.4. The 0.4 threshold was selected as a compromise to avoid having bias towards a single method. A higher threshold also reduces the size of the feature space, in this case from 6665 (0.2 threshold) to 3017 (0.4 threshold). Another information gain threshold was used for the feature subsets consisting of only dynamic features. The dynamic features were ranked with very low individual merit, and a threshold of 0.05 was used.

ReliefF threshold	RandomForest	SVM
ReliefF 0.05	80.78%	81.65%
ReliefF 0.1	79.61%	67.73%

Table 17: Accuracy for ReliefF thresholds

Table 17 displays the results for the different ReliefF thresholds that was tested. Due to ReliefF requiring significantly more computational time, only two thresholds were tested (about 3 days per run). The 0.05 threshold performed best for both methods and were therefore selected.

The parameters for the feature selection methods in Weka are shown in figure 11.

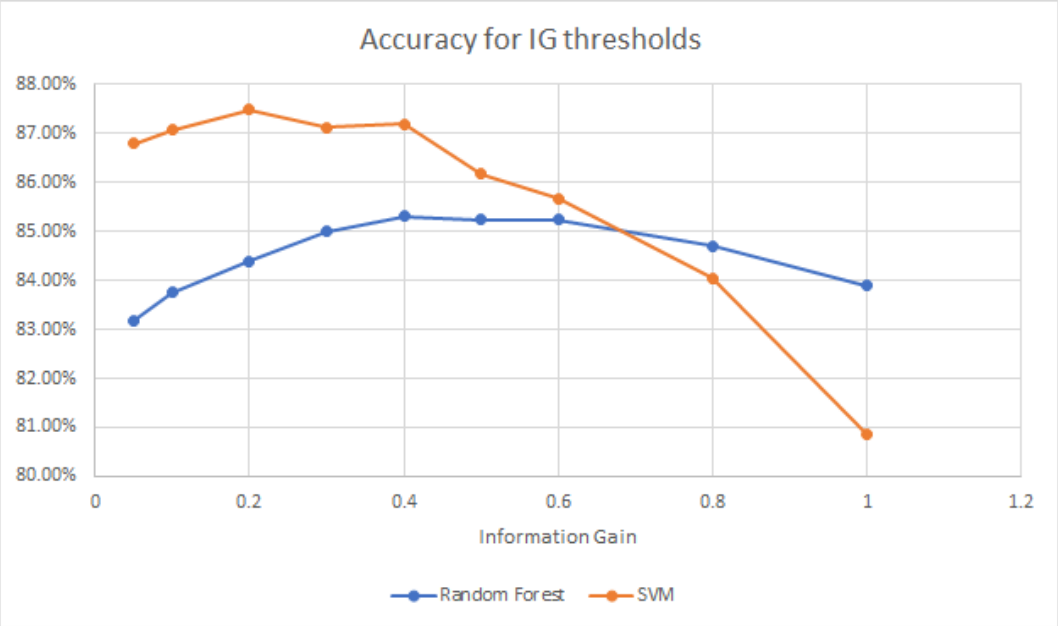

Figure 10: Classification accuracy for different Information Gain threshold

Evaluator settings	Search setting
CfsSubsetEval -P 1 -E 1	BestFirst -D 1 -N 5
ReliefFAttributeEval	ranker -N -1 -T 0.05
InfoGainAttributeEval	ranker -N -1 -T (depends)

Figure 11: Feature selection parameters

5.3.2 Classifier parameters

The parameters used for the different classifiers in Weka are shown in figure 12. The default parameters was used where possible, and no efforts were made to optimize the classifier parameters. 10 different classifiers was tested for all settings and feature subsets, and there was simply no time for optimization within the time-frame of the project. Nevertheless, using default parameters makes it easier to compare the results to other studies. For instance, default Weka parameters was used by Aktas and Sen in UpDroid [6].

Non-default parameters was used for SVM, MLP and ExtraTrees. SVM uses the "redial basis function" as the default kernel function. Kernel function was changed to "linear" because of a significant performance increase during initial testing. MLP uses a default wildcard parameter "a" for the number of hidden layers. The wildcard parameter is calculated based on the number of features and classes in dataset (features + classes)/2. Using the wildcard parameter required too much time to finish the practical part in a timely manner. A single test could take up to several days using the

ML method	Default Weka settings
SimpleCart	SimpleCart -M 2.0 -N 5 -C 1.0 -S 1
ExtraTree	ExtraTree -K -1 -N -1 -S 1
BayesNet	BayesNet -D -Q K2 -- -P 1 -S BAYES -E SimpleEstimator -- -A 0.5
NaiveBayes	NaiveBayes
RandomForest	RandomForest -P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1
RotationForest	RotationForest -G 3 -H 3 -P 50 -F "PrincipalComponents -R 1.0 -A 5 -M -1" -S 1 -num-slots 1 -I 10 -W J48 -- -C 0.25 -M 2
K-nn	IBk -K 1 -W 0 -A "LinearNNSearch -A \"EuclideanDistance -R first-last\""
ML method	**Modified Weka settings**
SVM	LibSVM -S 0 -K 0 -D 3 -G 0.0 -R 0.0 -N 0.5 -M 40.0 -C 1.0 -E 0.001 -P 0.1 -seed 1
MLP	MultilayerPerceptron -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H 3
ExtraTrees	RandomCommittee -S 1 -num-slots 1 -I 100 -W ExtraTree -K -1 -N -1 -S 1

Figure 12: Classifier parameters

wildcard parameter, as opposed to other classifiers that were able to finish the test within seconds or minutes. The parameter was reduced to "3", in order to run MLP with a computational time similar to that of the other classifiers. The RandomCommittee implementation was used to build an ensemble of ExtraTrees (as was suggested in the notes of the ExtraTree implementation). RandomCommittee builds an ensemble of 10 trees by default, and the number of trees was changed to 100 [43].

Normalization

SVM, MLP and k-NN requires that numeric features are normalized within a fixed range in order to work properly. For simplicity all feature subsets were therefore normalized after the feature selection step. The Weka Normalize implementation was used for this purpose [43].

6 Results and Discussion

This chapter will discuss the validity of the study and present the results and discuss the important findings.

6.1 Threats to validity

This section will discuss different threats to validity of the study and the reliability of the measurement instruments.

6.1.1 Internal validity

Leedy and Ormrod defines internal validity of a research study as "the extent to which its design and the data it yields allow the researcher to draw accurate conclusions about the cause-and-effect and other relationship in the data"[37].

With respect to internal validity we would like to highlight that the two metadata features was placed in the dynamic feature set during the generation of subsets. Although the metadata features was not part of the malware analysis process, they belong to the static feature category as no execution is required for extraction. Unfortunately there was no time to redo the classification process when this issue was discovered. Which resulted in a dynamic feature set that does not entirely consists of dynamic features. Thereby some unnecessary bias was introduced for research question 2, where we investigate if the performance of the classifier can be increased by combining static and dynamic features, as opposed to using either or.

6.1.2 External validity

External validity is the extent to which the results can be generalized to other contexts such as the entire population and a real-life setting [37]. In this study we collected a subset from the AndroZoo repository consisting of more than 8.5 million applications. Samples in the subset was collected from the end of 2014 to the end of 2016. Due to massive size of the AndroZoo repository, we consider the subset used in this experiment to be representative for the most common malware threats that could be found in the markets during that period. During the selection we excluded malware families that had not been flagged by at least 5 anti-virus engines (VirusTotal detect threshold), and families that had less than 50 samples. As discussed in the methodology chapter 4.2, the threshold can be considered as a trade-off between having more certainty in whether the samples are malicious or not and the bias of removing the samples that are possibly more difficult to detect. By doing so we created a lab setting that does not entirely reflect a real-life setting. A decrease in performance is therefore expected for a real-life setting.

6.1.3 Reliability of measurement instruments

Leedy and Ormrod defines reliability of a measurement instruments as "the consistency with which a measurement instrument yields a certain, consistent result when the entity being measured hasn't changed"[37]. The measurement instruments in this context refers to the malware analysis tools used in the experiment. In essence the question is then, if we repeat the feature extraction process, will we obtain the same results?

The tools used for static analysis are deterministic, and will yield the exact same results on each run. DroidBox[25] on the other hand, relies on the Monkey tool[1] to generate a random sequence of simulated user input. Choudhary et al [50] found that random exploration strategies, such as the one implemented in monkey tool, achieves higher code coverage than more sophisticated strategies in other tools. However, the code execution is not deterministic, and the result may not be identical for each run. A seed value could therefore be used to ensure that the same sequence of random generated events are used, and that the code execution is deterministic [34].

We also experienced that a considerable amount of the samples would crash and/or stop running at some point during the dynamic analysis. The exact cause is unknown, but it was highlighted by Chakraborty et al [34] that malware applications crash often during analysis based on input simulators. Alternatively some malware may stop running if it detects a sandbox.

6.2 Classification

6.2.1 Results

All the classification results are shown in table 13. The table is divided into three sections. The first section show the results for the time-unaware setting, the second and third section shows the results for the time-aware setting. Section two for time-aware split 1, and section three for time-aware split 2. The first column shows what feature selection method and threshold that was used. The second column displays the number of features that remains after feature selection (class excluded). The remaining columns shows the result for the different machine learning classifiers. The background colors are used to describe the type of features that was used: **green** is for dynamic features (the first four columns in each section), **blue** is for static features (the three next columns in each section), **orange** is for hybrid features (the three last columns in each section). **Gray** background color indicates that the results were obtained without the use of feature normalization. The best results in each section, for each machine learning algorithm, for each feature type is highlighted with a darker color shade. The best results in each section, for each feature type is highlighted with bold text, and an even darker color shade.

A malware family F-Measure[2] comparison between the best result in the time-unaware setting, and in time-aware split 1 is shown in 14. Malware family labels are plotted on the y-axis. The blue bars represent F-measures in the time-aware setting, and orange bars represents F-measures in

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

[2] The **F-Measure**, F-Score or F1 score is defined as the harmonic mean of precision and recall. Precision is measured as (True Positives/ True positives + False positives) for individual classes. Recall is measures as (True positives/ True positives + False negatives) for individual classes [51].

time-aware split 1. The best results in the time-unaware setting is for ExtraTrees with a subset of static features selected using CsfSubsetEval. The best result in time-aware split 1 is for ExtraTrees with a subset of hybrid features selected with CsfSubsetEval.

Time-unaware setting											
Feature sel	Features	RotationF	CART	Extratree	ExtraTrees	RandomF	SVM	MLP	k-NN	NaiveB	BayesN
InfoGain 0.05	168	72.96%	65.09%	60.28%	73.11%	72.38%	52.77%	20.02%	64.54%	35.24%	45.11%
relieff	6	52.07%	51.15%	51.86%	52.39%	52.43%	27.47%	27.62%	52.46%	29.19%	38.99%
CfsSubsetEval	22	73.40%	65.88%	62.91%	**74.68%**	74.67%	36.57%	29.17%	62.62%	38.14%	57.28%
full	661	73.60%	65.46%	61.50%	73.89%	70.64%	54.33%	13.26%	65.48%	36.92%	45.44%
InfoGain 0.4	2931	87.99%	82.15%	75.96%	85.83%	85.08%	86.52%	12.23%	77.64%	55.85%	65.55%
relieff	714	84.32%	78.84%	74.15%	83.36%	80.48%	80.15%	24.18%	73.12%	42.81%	51.79%
CfsSubsetEval	111	88.26%	82.42%	78.75%	**89.10%**	88.84%	65.69%	17.36%	81.18%	65.69%	79.66%
InfoGain 0.4	3017	87.79%	82.61%	75.27%	86.46%	85.32%	87.19%	12.23%	77.39%	60.12%	66.63%
relieff	723	84.72%	79.17%	73.95%	83.66%	80.78%	81.65%	23.73%	74.03%	44.37%	52.05%
CfsSubsetEval	138	88.45%	81.20%	77.92%	**89.02%**	88.71%	83.07%	25.71%	79.81%	66.86%	79.65%
Time-aware setting 1											
Feature sel	Features	RotationF	CART	Extratree	ExtraTrees	RandomF	SVM	MLP	k-NN	NaiveB	BayesN
InfoGain 0.05	169	45.98%	38.04%	19.20%	50.80%	49.47%	43.52%	15.91%	40.80%	21.40%	19.07%
relieff	6	24.52%	19.40%	14.35%	18.94%	15.55%	23.19%	22.49%	15.61%	23.95%	28.77%
CfsSubsetEval	21	41.79%	31.43%	28.14%	46.18%	44.65%	34.12%	30.40%	35.12%	17.34%	20.76%
full	661	46.21%	41.20%	23.95%	**52.92%**	48.14%	46.05%	12.92%	43.59%	22.79%	19.60%
InfoGain 0.4	3167	61.13%	47.21%	38.44%	70.33%	67.71%	**72.39%**	12.92%	60.50%	29.60%	48.07%
relieff	939	60.30%	56.28%	36.35%	66.68%	61.89%	64.45%	19.44%	54.09%	32.99%	36.68%
CfsSubsetEval	99	66.08%	47.51%	44.78%	72.13%	71.59%	68.14%	19.47%	62.76%	35.28%	47.71%
InfoGain 0.4	3255	61.76%	45.35%	41.06%	68.04%	66.41%	72.19%	12.92%	62.06%	31.20%	50.20%
relieff	948	58.01%	54.19%	40.96%	66.08%	61.50%	65.28%	23.92%	55.32%	33.49%	36.84%
CfsSubsetEval	135	67.14%	45.45%	48.94%	**73.65%**	72.49%	68.27%	19.53%	62.62%	31.83%	51.50%
Time-aware setting 2											
Feature sel	Features	RotationF	CART	Extratree	ExtraTrees	RandomF	SVM	MLP	k-NN	NaiveB	BayesN
InfoGain 0.05	167	36.26%	32.73%	14.57%	42.39%	41.51%	29.32%	3.60%	40.11%	23.73%	27.16%
relieff	5	25.32%	21.96%	19.82%	18.67%	16.62%	19.84%	18.34%	16.63%	23.11%	23.76%
CfsSubsetEval	19	35.22%	37.60%	23.54%	48.58%	**50.23%**	22.67%	27.88%	30.70%	22.12%	32.23%
full	661	40.46%	33.31%	18.39%	44.38%	40.07%	31.74%	16.83%	42.18%	23.74%	27.41%
InfoGain 0.4	2751	45.78%	34.11%	41.24%	70.10%	67.86%	69.28%	3.35%	54.73%	36.93%	48.16%
relieff	1120	42.35%	39.88%	36.95%	66.04%	62.71%	56.46%	3.35%	46.24%	32.10%	34.76%
CfsSubsetEval	119	62.19%	33.25%	40.97%	**71.94%**	71.85%	64.26%	3.35%	57.21%	42.89%	59.75%
InfoGain 0.4	2826	46.55%	32.98%	29.00%	69.22%	68.05%	71.45%	3.35%	55.73%	40.30%	48.70%
relieff	1131	43.31%	39.88%	33.84%	65.71%	62.30%	58.47%	3.35%	46.44%	34.06%	35.32%
CfsSubsetEval	141	63.47%	46.76%	45.65%	**74.19%**	71.56%	65.77%	3.35%	55.37%	44.75%	60.56%

Figure 13: Classification results. (Green=dynamic features, Blue=static features, Orange=hybrid features)

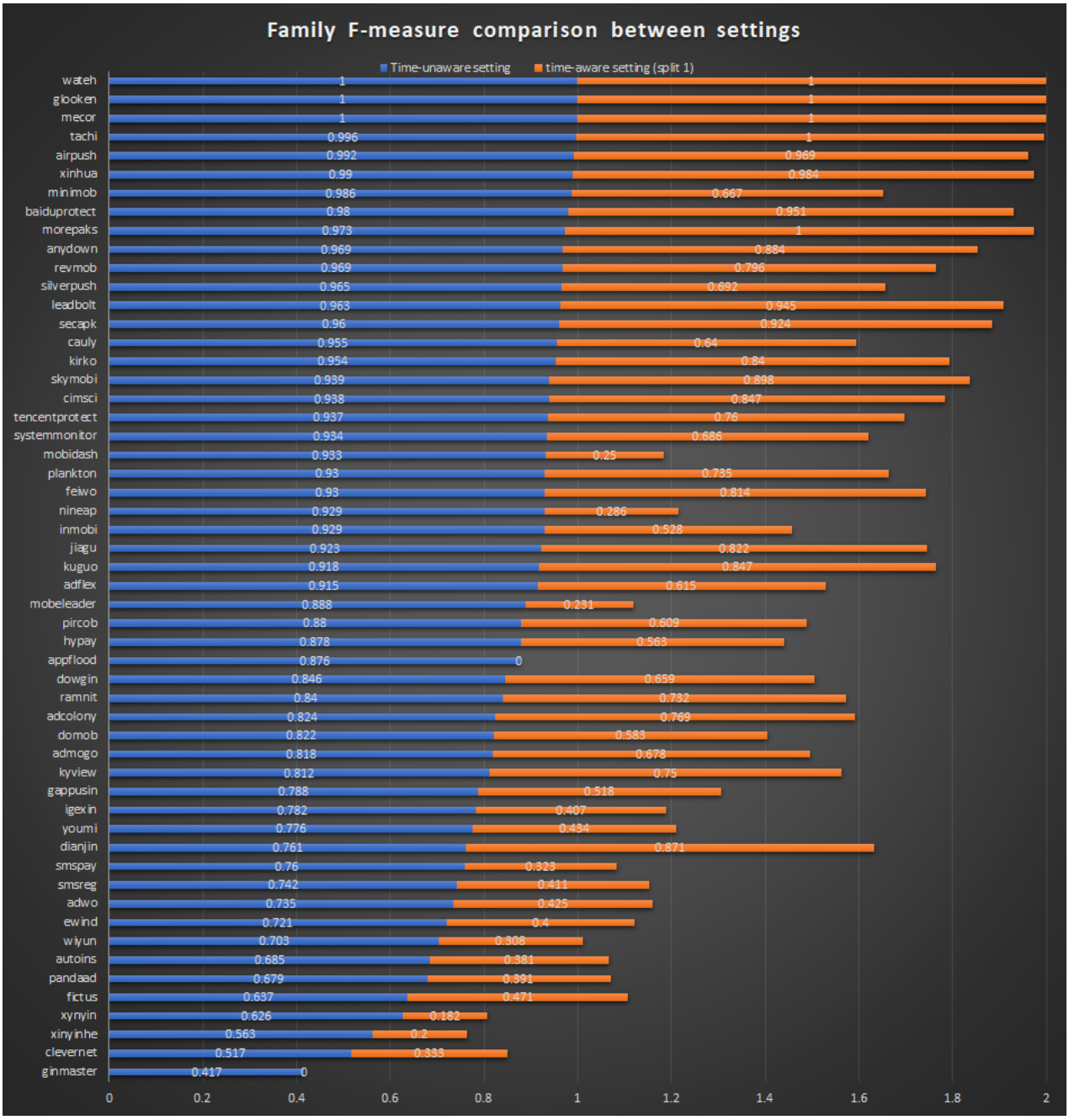

Figure 14: Malware family F-Measure comparison between the best result in the time-unaware setting, and in the time-aware split 1

6.2.2 Discussion

A lot of information can be extracted from the classification results in figure 13. We can see that the Extra Trees method provided the best results for all settings. Closely followed by Random Forest and SVM. The best results were obtained using feature subsets selected with the CsfSubsetEval feature selection method. Hybrid features performed better in the time-aware setting, while static features performed slightly better in the time-unaware setting. Although the results between static and hybrid features sets were very close. Dynamic features performed significantly worse than static and hybrid features for all settings. Comparing the results between the time-unaware setting and time-aware split 1, we can see a significant performance decrease when time is taken into consideration. While the results for time-aware split 2 were surprisingly similar to that of time-aware split 1. A summary of the best results is displayed in figure 15

Time-unaware setting				
F.type	F.sel.	F.num.	ML alg.	Acc.
dynamic	CfsSubsetEval	22	ExtraTrees	74.68%
static	CfsSubsetEval	111	ExtraTrees	**89.10%**
hybrid	CfsSubsetEval	138	ExtraTrees	89.02%
Time-aware setting 1				
dynamic	full	661	ExtraTrees	52.92%
static	InfoGain 0.4	3,167	SVM	72.39%
hybrid	CfsSubsetEval	136	ExtraTrees	**73.65%**
Time-aware setting 2				
dynamic	CfsSubsetEval	19	RandomForest	50.23%
static	CfsSubsetEval	119	ExtraTrees	71.94%
hybrid	CfsSubsetEval	141	ExtraTrees	**74.19%**

Figure 15: Summary of best classification results

The F-Measure comparison in figure 14 shows how the classifier performs with respect to the individual families. We can see that 28 families has a F-measure score higher than 0.9 in the time-unaware setting. The time-aware setting (split 1) has a decrease in F-measure for all families except for 5. The Wateh, Gloken and Mecor family has a F-measure of 1 for both settings, while Morepaks and Tachi has a slight increase in F-measure. No samples from the Ginmaster and Appflood families were correctly classified in the time-aware setting. As shown in the methodology section 4, Appflood has 56 samples in the training set and 3 in the testing set for time-aware split 1. Meaning that the low F-measure for Appflood is the result of only 3 misclassifications. Ginmaster on the other hand has 52 samples in the training set and 24 in the testing set. Further analysis is required to determine what exactly is causing the classifier to struggle with the Ginmaster family. Some likely causes may be obfuscation or different variants within the family (see discussion in background 2).

RQ1: *What level of accuracy can be achieved for Android malware family classification in a time-aware and time-unaware setting, and how do the results of these settings compare?*
Findings: *For a dataset of 14582 samples from 54 different families, a classification accuracy of 89.1% was achieved in a time-unaware setting. In a time-aware setting constructed to be comparable to the time-unaware setting, a classification accuracy of 73.65% was achieved. The best result for a time-aware setting constructed to be more realistic was 74.19% classification accuracy. Taking time into consideration significantly decrease the accuracy for a classifier, in this case a decrease 15.45% classification accuracy (time-unaware vs time-aware setting 1). An overall decrease in F-measure was found for individual families, some families were impacted considerably more than others.*

RQ2: *Can the performance of the classifiers be improved by combining features extracted with static analysis and features extracted with dynamic analysis?*
Findings: *A small increase in classification accuracy could be achieved by using a combination of static and dynamic features in the time-aware setting, as opposed to using only static features. The best results for the time-unaware setting was obtained using static features. Dynamic features performed considerably worse than static features for all three settings, with a 15-20% decrease in classification accuracy. Considering the feature extraction time discussed in 6, it is hard to justify the use of dynamic features with respect to scalability. Although no conclusions can be made regarding this research question due to the issues discussed in 6.1*

6.3 Optimal feature set

6.3.1 Results

In order to find an optimal feature set, we must consider the best results from each of the classifier settings. The best results for each of the settings are shown in figure 13:

- **Time-unaware setting:** Static feature subset selected with CfsSubsetEval.
- **Time-aware split 1:** Hybrid feature subset selected with CfsSubsetEval.
- **Time-aware split 2:** Hybrid feature subset selected with CfsSubsetEval.

The following subsections will list all of the features used in the feature subsets that produced the best results. The first subsection will list the intersection of the best features from each setting. Features that were not present in all three settings will be listed in their own subsection. Notice that a prefix is used for all features to dictate feature category (**S**=Static, **D**=Dynamic, **M**=Metadata), followed by a word to describe the sub-category that the feature belongs to (e.g. Opcode or ApiCall).

Intersection between settings

Android API related features are listed in table 18, and the remaining features are listed in table 19.

	Android API related features
1	S_ApiCall_android.app.Activity.onConfigurationChanged
2	S_ApiCall_android.app.Activity.onKeyDown
3	S_ApiCall_android.app.Activity.onUserLeaveHint
4	S_ApiCall_android.app.AlarmManager
5	S_ApiCall_android.app.Application
6	S_ApiCall_android.app.Service.onStart
7	S_ApiCall_android.content.Context.getDir
8	S_ApiCall_android.content.Context.getFileStreamPath
9	S_ApiCall_android.content.res.AssetManager
10	S_ApiCall_android.hardware.Sensor
11	S_ApiCall_android.hardware.SensorManager
12	S_ApiCall_android.hardware.SensorManager.unregisterListener
13	S_ApiCall_android.media.AudioManager
14	S_ApiCall_android.net.NetworkInfo.getState
15	S_ApiCall_android.net.NetworkInfo.isAvailable
16	S_ApiCall_android.telephony.SmsManager
17	S_ApiCall_android.util.Base64
18	S_ApiCall_android.view.ViewGroup.invalidate
19	S_ApiCall_android.view.animation.AnimationSet.setInterpolator
20	S_ApiCall_android.webkit.WebChromeClient.onJsConfirm
21	S_ApiCall_android.webkit.WebSettings.getUserAgentString
22	S_ApiCall_android.webkit.WebSettings.setJavaScriptCanOpenWindowsAutomatically
23	S_ApiCall_android.webkit.WebView.setId
24	S_ApiCall_android.webkit.WebView.setOnTouchListener
25	S_ApiCall_android.webkit.WebView.setScrollBarStyle
26	S_ApiCall_android.widget.Button.setPadding
27	S_ApiCall_android.widget.LinearLayout.clearAnimation
28	S_ApiCall_android.widget.RelativeLayout.removeView
29	S_ApiCall_android.widget.RemoteViews
30	S_ApiCall_android.widget.VideoView
31	S_ApiCall_java.io.BufferedInputStream
32	S_ApiCall_java.io.BufferedInputStream.close
33	S_ApiCall_java.io.BufferedReader
34	S_ApiCall_java.io.BufferedWriter
35	S_ApiCall_java.io.File.mkdir
36	S_ApiCall_java.io.FileReader
37	S_ApiCall_java.lang.Boolean
38	S_ApiCall_java.lang.Exception
39	S_ApiCall_java.lang.Process
40	S_ApiCall_java.lang.System.exit
41	S_ApiCall_java.util.zip.ZipEntry.getTime
42	S_ApiCall_javax.crypto.SecretKeyFactory
43	S_ApiCall_org.apache.http.params.HttpParams
44	S_ApiCall_org.json.JSONObject.isNull
45	S_ApiCall_org.xmlpull.v1.XmlPullParserFactory.newInstance
46	S_ApiPackage_android.os

Table 18: Intersection of API features between settings

Other Features	
1	S_Intent_android.intent.action.PACKAGE_ADDED
2	S_Intent_android.intent.action.PACKAGE_REMOVED
3	S_Intent_android.intent.action.USER_PRESENT
4	S_Intent_android.net.conn.CONNECTIVITY_CHANGE
5	S_Intent_android.net.wifi.WIFI_STATE_CHANGED
6	S_Intents
7	S_Receivers
8	S_Services
9	S_SystemCmd_chmod
10	S_SystemCmd_getprop
11	S_Opcode_rem-int/2addr
12	S_Opcode_aget-byte
13	S_Permission_NumAndroidPermissions
14	S_Permission_NumCustomPermissions
15	S_Permission_android.permission.ACCESS_COARSE_LOCATION
16	S_Permission_android.permission.GET_ACCOUNTS
17	S_Permission_android.permission.GET_TASKS
18	S_Permission_android.permission.SEND_SMS
19	S_Permission_android.permission.SYSTEM_ALERT_WINDOW
20	S_Permission_com.android.launcher.permission.INSTALL_SHORTCUT
21	S_NC_arm_ProgramHeaders
22	S_NC_arm_Section_.interp_Flag_A
23	S_String_len5000

Table 19: Intersection of features between settings

Time-unaware setting

Features are listed in table 20

	Features unique to Time-unaware setting
1	S_ApiCall_android.app.Activity.finishActivity
2	S_ApiCall_android.app.Application.attachBaseContext
3	S_ApiCall_android.app.Application.onCreate
4	S_ApiCall_android.app.ProgressDialog.dismiss
5	S_ApiCall_android.app.Service.onStartCommand
6	S_ApiCall_android.content.ContentResolver.insert
7	S_ApiCall_android.content.Intent.setFlags
8	S_ApiCall_android.graphics.Matrix.preRotate
9	S_ApiCall_android.telephony.TelephonyManager.getDeviceSoftwareVersion
10	S_ApiCall_android.telephony.TelephonyManager.getSubscriberId
11	S_ApiCall_android.util.FloatMath
12	S_ApiCall_android.view.animation.AnimationSet
13	S_ApiCall_android.webkit.WebView.removeAllViews
14	S_ApiCall_android.widget.Button
15	S_ApiCall_android.widget.CheckBox.setId
16	S_ApiCall_android.widget.ViewFlipper.showPrevious
17	S_ApiCall_dalvik.system.DexClassLoader
18	S_ApiCall_java.io.BufferedReader.readLine
19	S_ApiCall_java.io.DataInputStream.close
20	S_ApiCall_java.io.DataInputStream.readByte
21	S_ApiCall_java.io.IOException
22	S_ApiCall_java.lang.ClassLoader
23	S_ApiCall_java.lang.Object.equals
24	S_ApiCall_java.lang.reflect.AccessibleObject.isAccessible
25	S_ApiCall_java.lang.reflect.Method
26	S_ApiCall_java.util.Properties.load
27	S_ApiCall_java.util.zip.ZipFile
28	S_ApiCall_java.util.zip.ZipInputStream
29	S_ApiCall_java.util.zip.ZipInputStream.read
30	S_ApiCall_org.xmlpull.v1.XmlPullParser
31	S_H_Dex_ApiCalls_android.content.Intent.setClassName
32	S_H_Dex_ApiCalls_android.content.pm.ApplicationInfo
33	S_H_Dex_ApiCalls_android.telephony.TelephonyManager.getSubscriberId
34	S_NC_ARM_Size
35	S_NC_arm_Sections
36	S_Opcode_aput-byte
37	S_Opcode_move-result
38	S_Opcode_move/from16
39	S_Opcode_neg-int
40	S_Opcode_or-int/lit16
41	S_Permission_android.permission.ACCESS_GPS
42	S_Permission_android.permission.MOUNT_UNMOUNT_FILESYSTEMS

Table 20: Time-unaware setting - features

Time-aware setting 1

Android API related features are listed in table 21, and the remaining features are listed in table 22.

Android API related features - Time-aware setting 1	
1	S_ApiCall_android.app.Application.attachBaseContext
2	S_ApiCall_android.app.Application.onCreate
3	S_ApiCall_android.app.ProgressDialog
4	S_ApiCall_android.app.Service
5	S_ApiCall_android.content.ActivityNotFoundException
6	S_ApiCall_android.content.Context.checkCallingOrSelfPermission
7	S_ApiCall_android.content.Context.getAssets
8	S_ApiCall_android.content.Intent.setFlags
9	S_ApiCall_android.content.pm.ApplicationInfo
10	S_ApiCall_android.location.LocationManager
11	S_ApiCall_android.media.MediaPlayer
12	S_ApiCall_android.os.Handler.postDelayed
13	S_ApiCall_android.telephony.TelephonyManager.getDeviceSoftwareVersion
14	S_ApiCall_android.telephony.TelephonyManager.getSimState
15	S_ApiCall_android.telephony.TelephonyManager.getSubscriberId
16	S_ApiCall_android.webkit.WebSettings.setPluginsEnabled
17	S_ApiCall_android.webkit.WebView.removeAllViews
18	S_ApiCall_android.widget.CheckBox.setId
19	S_ApiCall_android.widget.LinearLayout.setLayoutParams
20	S_ApiCall_android.widget.Scroller.getCurrVelocity
21	S_ApiCall_java.io.BufferedReader.ready
22	S_ApiCall_java.io.DataInputStream.close
23	S_ApiCall_java.io.FileOutputStream.flush
24	S_ApiCall_java.io.IOException
25	S_ApiCall_java.io.RandomAccessFile
26	S_ApiCall_java.lang.ClassLoader
27	S_ApiCall_java.lang.reflect.Method
28	S_ApiCall_java.nio.channels.SelectionKey.selector
29	S_ApiCall_java.util.StringTokenizer
30	S_ApiCall_java.util.zip.ZipFile
31	S_ApiCall_java.util.zip.ZipInputStream.read
32	S_ApiCall_org.xmlpull.v1.XmlPullParser.nextText
33	S_ApiPackage_android.view
34	S_ApiPackage_dalvik.system

Table 21: Time-aware split 1 - API features

Features - Time-aware setting 1	
1	D_CryptoOperation_keyalgo
2	D_Cryptoalgorithm_DES
3	D_Filesystem_AccessedFiles
4	D_Filesystem_write
5	D_Strace_epoll_wait
6	D_Strace_execve
7	D_Strace_fgetxattr
8	D_Strace_fstat64
9	D_Strace_lseek
10	D_Strace_mkdir
11	D_Strace_pipe
12	D_Strace_pread
13	D_Strace_pwrite
14	D_Strace_readlink
15	D_Strace_stat64
16	D_Strace_unlink
17	D_Strace_write
18	S_H_Dex_ApiCalls_android.content.pm.ApplicationInfo
19	S_H_Dex_ApiCalls_android.telephony.TelephonyManager.getSubscriberId
20	S_H_incorrectExtension
21	M_Metadata_APKSize
22	S_NC_arm_
23	S_Opcode_int-to-char
24	S_Opcode_iput-short
25	S_Opcode_move/from16
26	S_Opcode_neg-int
27	S_Opcode_or-int/lit16
28	S_Permission_android.permission.ACCESS_FINE_LOCATION
29	S_Permission_android.permission.CALL_PHONE
30	S_Permission_android.permission.MOUNT_UNMOUNT_FILESYSTEMS
31	S_Permission_android.permission.READ_PHONE_STATE
32	S_Permission_android.permission.WRITE_SETTINGS

Table 22: Time-aware split 1 - features

Time-aware setting 2

Android API related features are listed in table 23, and the remaining features are listed in table 24.

Android API related features - Time-aware setting 2	
1	S_ApiCall_android.app.Activity.finish
2	S_ApiCall_android.app.Activity.runOnUiThread
3	S_ApiCall_android.app.ProgressDialog
4	S_ApiCall_android.app.Service
5	S_ApiCall_android.app.Service.onStartCommand
6	S_ApiCall_android.content.ContentResolver.insert
7	S_ApiCall_android.content.Context.checkCallingOrSelfPermission
8	S_ApiCall_android.content.Context.getApplicationInfo
9	S_ApiCall_android.content.Context.getAssets
10	S_ApiCall_android.content.Context.getSharedPreferences
11	S_ApiCall_android.content.IntentFilter.setPriority
12	S_ApiCall_android.content.SharedPreferences.getLong
13	S_ApiCall_android.content.pm.ApplicationInfo
14	S_ApiCall_android.content.pm.PackageManager.getInstalledApplications
15	S_ApiCall_android.net.NetworkInfo.getExtraInfo
16	S_ApiCall_android.os.Vibrator
17	S_ApiCall_android.preference.PreferenceManager
18	S_ApiCall_android.telephony.TelephonyManager.getLine1Number
19	S_ApiCall_android.telephony.TelephonyManager.getSimOperatorName
20	S_ApiCall_android.telephony.TelephonyManager.getSimState
21	S_ApiCall_android.view.animation.AnimationSet
22	S_ApiCall_android.webkit.WebSettings.setLoadsImagesAutomatically
23	S_ApiCall_android.webkit.WebView.loadUrl
24	S_ApiCall_android.webkit.WebView.setClickable
25	S_ApiCall_android.widget.Button.setTypeface
26	S_ApiCall_android.widget.ImageButton.setLayoutParams
27	S_ApiCall_android.widget.LinearLayout.setLayoutParams
28	S_ApiCall_android.widget.RelativeLayout.onAttachedToWindow
29	S_ApiCall_android.widget.RelativeLayout.removeAllViews
30	S_ApiCall_android.widget.RelativeLayout.setGravity
31	S_ApiCall_android.widget.RelativeLayout.setPadding
32	S_ApiCall_android.widget.Scroller.getCurrVelocity
33	S_ApiCall_android.widget.Toast
34	S_ApiCall_android.widget.ViewFlipper.setInAnimation
35	S_ApiCall_java.io.DataInputStream.readShort
36	S_ApiCall_java.io.FileOutputStream.flush
37	S_ApiCall_java.lang.ClassLoader.getSystemClassLoader
38	S_ApiCall_java.lang.reflect.AccessibleObject
39	S_ApiCall_java.net.InetAddress.getHostAddress
40	S_ApiCall_java.security.NoSuchAlgorithmException
41	S_ApiCall_java.util.Properties
42	S_ApiCall_java.util.regex.Matcher.group
43	S_ApiCall_java.util.zip.ZipInputStream
44	S_ApiCall_org.xmlpull.v1.XmlPullParser
45	S_ApiPackage_dalvik.system
46	S_ApiPackage_javax.crypto
47	S_ApiPackage_javax.net.ssl

Table 23: Time-aware split 2 - API features

Features - Time-aware setting 2	
1	D_CryptoOperation_keyalgo
2	D_Filesystem_write
3	D_Strace__llseek
4	D_Strace_execve
5	D_Strace_fstat64
6	D_Strace_lseek
7	D_Strace_pipe
8	D_Strace_pread
9	D_Strace_pwrite
10	D_Strace_readlink
11	D_Strace_stat64
12	D_Strace_unlink
13	D_Strace_write
14	S_H_Dex_ApiCalls_android.app.ActivityManager.getRunningAppProcesses
15	S_H_Dex_ApiCalls_java.util.Collection
16	S_Intent_com.google.android.c2dm.intent.REGISTRATION
17	M_Metadata_APKSize
18	S_NC_arm_pltCall_scandir
19	S_Opcode_aput-byte
20	S_Permission_android.permission.ACCESS_FINE_LOCATION
21	S_Permission_android.permission.CALL_PHONE
22	S_Permission_android.permission.READ_PHONE_STATE
23	S_Permission_android.permission.SET_WALLPAPER
24	S_Permission_android.permission.WAKE_LOCK
25	S_Permission_android.permission.WRITE_SETTINGS

Table 24: Time-aware split 2 - features

6.3.2 Discussion

Features from a wide variety of categories were considered and evaluated in different classifier settings to find optimal feature subsets. An overview of the number features found in each feature category for the optimal subsets are shown in Table 25. The "ALL" column shows the total number of features from each category that was included in the feature selection step. The overview shows that features from all categories except for Activities were part of the subsets. A clear majority of these features are static features related to the Android API. An interesting observation is that the feature subsets that consisted of fewer number of features produced the best results. The number of features in the optimal feature sets ranged from only 111-141 features. Dynamic features were only selected as part of the subsets used for the time-aware setting. The majority of the dynamic features was related to the occurrences of certain system calls in the strace category.

Feature	TU	TA1	TA2	ALL
S_ApiCall	75	77	89	19231
S_ApiPackage	1	3	4	159
S_Permission	10	13	14	254
S_Opcode	7	7	3	216
S_Intent	6	6	7	734
S_Receivers	1	1	1	1
S_Services	1	1	1	1
S_Activities	0	0	0	1
S_SystemCmd	2	2	2	52
S_String	1	1	1	12
S_H (Hidden Code)	3	3	2	20322
S_NC (Native Code)	4	3	3	8606
M_Metadata	0	1	1	2
D_(DroidBox)	0	6	4	528
D_Strace	0	13	11	131
TOTAL	111	135	141	50250

Table 25: Summary of the optimal feature sets

It is important to note that these features were only tested for the most common families found in the AndroZoo [4] repository. Most of these families belong to the adware and riskware types, and can be categorized as possible unwanted applications (PUA).

RQ3: *What set of Android malware characteristics/features can produce the best results for a machine learning classifier?*
Findings: *The sets of features that produced the best results for classifying the most common Android families consisted of 111-141 features. These feature were from a wide verity of feature categories, with the majority being from the Android API category. Features that was part of the subsets for both the time-aware and the time-unaware setting includes: Android API related features; the number of certain intents; the number of different intents, receivers and services; The number of certain Opcodes and system commands; certain permissions, and the count of Android permissions and other permissions; the number of native code program headers, and section flags; and finally the number of strings that are longer than 5000 characters.*

6.4 Feature extraction time

6.4.1 Results

The APK size distribution for the entire dataset is shown in figure 16, and the APK size distribution for the subset used in the experiment is shown in figure 17. A comparison between the native code usage in subset and dataset is shown in table 26. A similar comparison for the hidden code usage is shown in figure 27. Finally the time to extract each time of feature is shown in table 28. The feature extraction process of AndroPyTool is broken down into several tasks. The time required by

each task is displayed in the table.

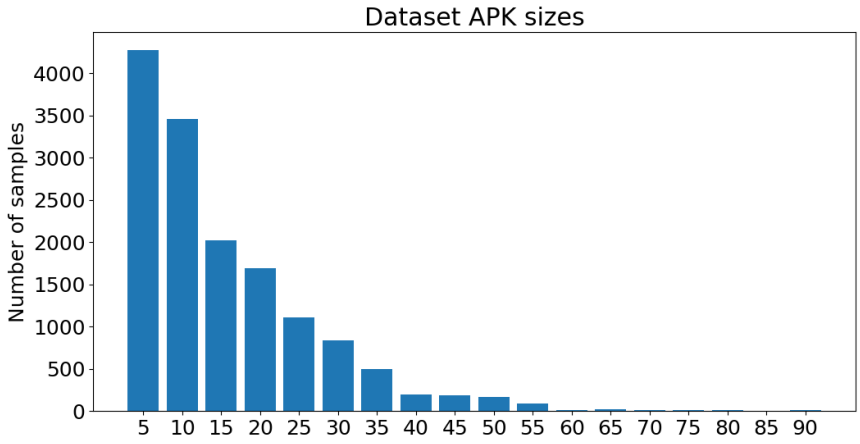

Figure 16: Dataset APK size histogram

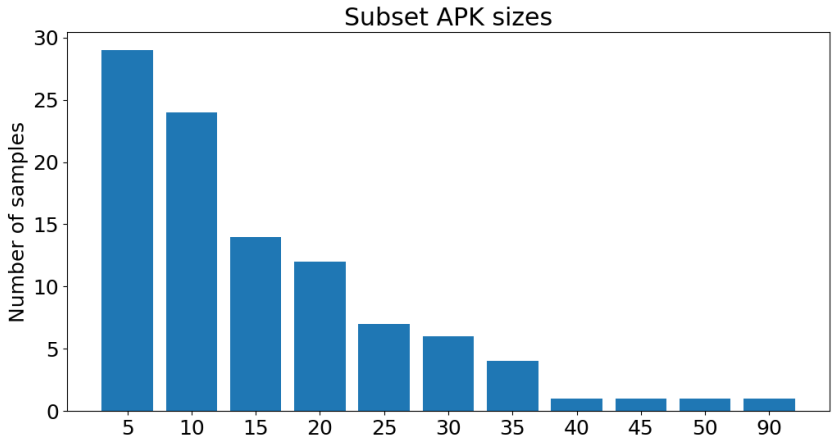

Figure 17: Subset APK size histogram

6.4.2 Discussion

A subset of the dataset was used to measure the feature extraction time for each tool. The subset was selected based on the assumption that feature extraction time is highly correlated with APK size. As shown in the result section, the APK size distribution for the subset is representative for the APK size distribution found in the dataset. As discussed in the methodology chapter 4.3, native code

Samples	100	14582
Samples with NC	60	9668
Native code ARM files	334	55414
Failed to analyze	8	1575

Table 26: Subset native code file comparison

Samples	100	14582
Samples with HC	18	3191
Dex files	30	7351
Dex cannot disassemble	0	0
APK files	21	4304
Jar files	2	319
Jar cannot disassemble	0	0

Table 27: Subset hidden code file comparison

Feature extraction tool	Features	Task	Time per sample	Time total
AndroPyTool	Static and dynamic	All	3m 33.1s	5h 55m 12.7s
		Filter invalid apk files	0.127s	12.7s
	Dynamic	All	201.5s	5h 35m 5s
	Dynamic	Analysis	201.37s	5h 35m 37s
	Dynamic	Parse logcat logs	0.13s	13s
	Static	All	11.46s	19m 6s
extract_native_code.py	Native code	All	2.98s	4m 58s
extract_hidden_code.py	Hidden code	All	0.33s	33s

Table 28: Time to extract features for each tool

features are only extracted from the files that are compiled for the ARM architecture. On average there are 3.8 such files per sample in the dataset, and 3.3 files per sample in the subset. The failure rate for native code feature extraction is 0.02 for the subset, and 0.03 for the dataset. The hidden code feature extraction tool analyses DEX and JAR files found in the original and nested APK files. The tool therefore had to analyze 0.3 files per sample in the subset, and 0.5 files per sample in the dataset. Without taking the file sizes of hidden and native code files into consideration, the feature extraction time for these tool will be slightly less for the subset.

The feature extraction tools used in the project can easily be run in parallel. Feature extraction time per sample can therefore be considered as the time required by the tool taking the most time to run. Each of the tools are designed to extract a large set of features, such that an optimal feature subset can be found. Once the optimal feature subset is known, the tools that extracts static features can be modified to extract only the relevant features, reducing the feature extraction time. Contrar-

ily, the dynamic analysis tool runs for a fixed amount of time, and the time required to parse the analysis logs is negligible compared with the time required to analyze a sample. As discussed in the methodology section 4.3, each sample is analyzed for 120 seconds. Meaning that approximately 80 seconds of the time used to extract dynamic features from a sample is used to extract longs, reset and prepare the analysis environment. DroidBox [25] is an old tool that has not been updated in several years, and there is likely room for optimization or improvements.

RQ4: *How much time is required to extract the optimal set of features and classify a new malware sample?*

Finding: *The time required to extract the optimal set of features are: 12 seconds per sample for static features and 203 seconds per sample for dynamic features. Because feature extraction can easily be run in parallel, the feature extraction time for hybrid features are 203 seconds per sample. The time required to classify a new sample is negligible after the classifier has been trained.*

7 Conclusion

This chapter will give a brief summary of the thesis, and discuss the theoretical implications of the results and the reproducibility of study.

In this thesis we have investigated several aspects of a machine learning based Android malware family classification. We have shown that evaluating such a system in a time-unaware setting will yield significantly biased results. In a real-world setting, a classification system have to rely on knowledge learned from the past (training set) in order to classify malware samples in the future (testing set). In the experiment we found that our best time-unaware classifier **(89.10%)** outperformed the best time-aware (setting 1) classifier **(73.65%)** with **15.45%** classification accuracy.

A dataset of **14,582** malware samples dated from the last quarter of 2014 to the last quarter of 2016 was used in the experiment. Samples from the **54** most common malware families during this period were collected from AndroZoo [4], a repository of more than 8 million Android malware and goodware applications that were primarily collected by crawling Android market places.

Classifiers were evaluated in the time-unaware setting by using 5-fold cross validation. Two different time-aware settings were created and evaluated: **Time-aware setting 1** was constructed with properties similar to that of 5-fold cross validation so that the results could be compared to the time-unaware setting. Training and testing sets were split in a stratified manner maintaining the malware family distribution in both sets. Like in 5-fold cross validation, the training set consists of 4/5th of the dataset, while the testing set consists of the remaining 1/5th. Time-aware setting 1 was only time-aware within each family, such that all samples from individual families in the training set are dated earlier than in the testing set. **Time-aware setting 2** were constructed to include a more realistic setting, and the dataset was split into training and testing set on a certain date.

A broad range of features were extracted from the samples and evaluated in order to find the feature sets that produced the best results for the classifiers. We extracted features from both static and dynamic analysis, and compared the results that could be produced with only static features, (almost) only dynamic features and a combination of the two (hybrid features). In our experiment, we found that features extracted with static analysis significantly outperformed the features extracted from dynamic analysis, with an approximate **15-20%** increase in classification accuracy. By using the hybrid features we were able to obtain around **2%** increase in classification accuracy for the time-aware setting. Considering the runtime of the feature extraction tools, it is difficult to justify the use of dynamic features. However, due to some issues during the dynamic analysis process, where a considerable amount of the applications would stop running, the results relating to dynamic features is inconclusive (see 6.1).

The mean feature extraction time per sample were **12** seconds for static features and **203** seconds for dynamic features. In the dynamic analysis, samples were run in an isolated environment

for **120** seconds to cover different execution paths. The remainder of the 203 seconds were used to install and start the application, extract analysis logs and reset the environment. These are feature extraction time calculated based on extracting the full feature set, which leaves room for optimization by only extracting the features that produces the best results for the classifiers. However, this essentially only applies to static feature extraction, as dynamic features extraction runs for a fixed amount of time.

The sets of features that produced the best results for classifying the most common Android malware families consisted of **111-141** features. These feature were from a wide verity of feature categories, with the majority being from the Android API category. Features that was part of optimal subsets for both the time-aware and the time-unaware setting includes: Android API related features; the number of certain intents; the number of different intents, receivers and services; The number of certain Opcodes and system commands; certain permissions, and the count of Android permissions and other permissions; the number of native code program headers, and section flags; and finally the number of strings that are longer than 5000 characters.

7.1 Theoretical implications

We have shown the importance of taking the timeline into consideration when evaluating an Android malware family classifier. A classifier evaluated in a time-unaware setting will produce considerably biased results. The same can be said for machine learning based Android malware detection systems (binary classifiers) [5]. k-fold cross validation is the common approach to classifier evaluation in the literature. The evaluation method is considered to reduce bias and generalize the results by running k different tests. However, by design, this method cannot be used to evaluate a classifier in a time-aware setting. Other evaluation methods for machine-learning based Android malware classifiers should therefore be considered.

Based on the classification results in our experiment, we observed that the ExtraTree algorithm produced the best results for the all classifier settings. A considerable advantage of the ExtraTrees algorithm is the computational efficiency [45]. However, similar to other machine learning algorithms (e.g. SVM, MLP and other tree-based ensemble classifiers) it is difficult to explain the predictions made by the classifier. Decision trees can be converted to decision rules that explains any prediction. But it becomes significantly more difficult to interpret predictions that are a result of 100 different decision rules.

As mentioned in the methodology chapter 4.6, the classifiers have no way of dealing with samples from malware families that were not part of the training set. Any such samples will directly result in a misclassification. Efforts are therefore required to update the model as new malware families are detected. Batch-learning algorithms are by far the most common methods used for binary and multinomial malware classification in the literature, and is also exclusively used in our experiments. These methods assumes a stationary population and can not by themselves adapt to population and concept drift[1]. A solution is therefore to continuously re-train the model using a

[1]Population drift is a phenomena that occurs when a population changes over time. This can be said for the malware population, because it is continuously evolving for various reasons. For instance, malware is known to utilize new techniques

updated training set. Alternatively one can utilize on-line machine learning methods. These are methods that are designed to be continuously updated without having the re-train the entire model [52].

We also do not know which of the samples are classified correctly and which are not. Based on the F-Measures 14 for individual families we can say that predictions for some families are much more certain than for other families.

7.2 Reproducibility of study

Apart from the challenges of dynamic analysis discussed in 6.1, there are no considerable challenges in reproducing the experiment. All tools used in the experiment are both free and publicly available. Most of the practical part can also be automated with the python scripts found in the appendix A. The experiment can however be time consuming if only restricted resources are available. The feature extraction and selection process would require almost two months of computational time without running multiple processes in parallel. These processes can be run in parallel to the extent of cores and memory available. For instance, Running the feature extraction and selection in 6 process would only require about 10 days.

The malware samples used in the experiment was collected from the AndroZoo repository [4]. Access to the dataset must be applied for, and access conditions can be found here [2].

to avoid detection, and to exploit new vulnerabilities[52]. The population drift will in time lead to statistical changes in the features used to describe the concept of malware. New features may emerge, and others may become less significant. This phenomena is named concept drift [5, 52].

[2] https://androzoo.uni.lu/access

8 Future work

This chapter will discuss potential future work and possible improvements.

8.0.1 Updating model

As discussed in the conclusion chapter 7, the classifier needs to be continuously updated to account for new malware families appearing and concept drift within the families that are already in the training set.

Generally there are two approaches to this problem, on-line learning or re-training with a batch-learning algorithm. In case of a batch-learning classifier, one must determine how long samples should be kept in the training set. As new samples are added to the training set, the size will quickly grow and the time required to train the model will increase. Moreover as samples get older they might become less representative for the current threat landscape. Narayanan et al [52] found that batch-learning systems should be re-trained daily, with a batch size of at least a year for the binary malware detection problem. They argue that this process is computationally expensive with respect to time and resources, and that on-line learning is better suited for the Android malware detection problem.

Allix et al [5] states that the training set must be historically close to the testing set. The training set should be recent to represent the current malware threat landscape. A challenge in a practical setting is then to obtain new samples that already has a correct malware family label. Further allix et al [5] states that the performance of a classifier can't be maintained by simply updating the training set with samples labeled by the classifier itself. Further work is required to determine how the training set of a classifier should be updated in a sustainable manner.

8.0.2 Certainty of classifier predictions

As mentioned in the conclusion we can use the overall accuracy for individual families to say something about the certainty of the classifier predictions. For instance the Wateh, Gloken and mecor families achieved a perfect F-Measure of 1 as show in 14. If a new samples is classified into one of these families we can be more certain the prediction is correct, than if the sample is classified into Ginmaster, which has the lowest F-measure in both the time-aware and time-unaware settings.

It would be interesting to investigate if the certainty for individual predictions can be used to increase the performance of a classifier. The ExtraTrees algorithm uses the decision of 100 decision trees in order to make a prediction. If we set a lower threshold for the number of trees that needs to agree on a family, and classify samples with uncertain predictions as "uncertain". Will the performance of the classifier increase significantly compared to the number of samples labeled as "uncertain"? It may be beneficial to have more certainty in the prediction at the cost of being unable to classify some percentage of the samples.

8.0.3 File-type analysis

During the experiment, we extracted statistics related to file-type occurrences within all the APK-files in the dataset as a whole. We have shown that a wide range of file-types are found within the APK-files in the dataset. The file-types that are most commonly included in the analysis are the Android manifest, DEX files and in some studies native code files. Which means that a large portion of the file-types are left out during analysis.

Malware authors will come up with creative ways to evade detection, and it is not unlikely that files not considered by common malware detection systems will be utilized to hide malicious payloads. Doing an extensive within malware family file-type analysis may help to identify new trends and evasion techniques. As a result new features that can be used to describe malware family behavior can be identified.

8.0.4 Dynamic analysis tool

In the background chapter 2.4 we discussed existing dynamic analysis tools. As highlighted by Garcia et al [38], the weakness of these tools are that they were developed for older versions of the Android platform. DroidBox[25] and CuckooDroid [27] both supports API level 16, while MobSF [28] offers different analysis VMs up to API level 19 (released in 2013). The newest Android API level at the time of writing is Android 9 Pie (API 28). An updated tool designed for large-scale dynamic analysis and feature extraction for machine learning could greatly benefit the Android malware research community.

Bibliography

[1] Levinec & Simpson, D. 2018. Malware names. https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/malware-naming. Latest accessed: 14.12.2018.

[2] Chebyshev, V. 2019. Mobile malware evolution 2018. Latest accessed 05.25.2019.

[3] Wikipedia contributors. 2018. Mobile operating system — Wikipedia, the free encyclopedia. [Online; accessed 14-December-2018]. URL: https://en.wikipedia.org/w/index.php?title=Mobile_operating_system&oldid=873291540.

[4] Allix, K., Bissyandé, T. F., Klein, J., & Le Traon, Y. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, 468–471, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/2901739.2903508, doi:10.1145/2901739.2903508.

[5] Allix, K., Bissyandé, T. F., Klein, J., & Le Traon, Y. 2015. Are your training datasets yet relevant? In *Engineering Secure Software and Systems*, Piessens, F., Caballero, J., & Bielova, N., eds, 51–67, Cham. Springer International Publishing.

[6] Aktas, K. & Sen, S. 2018. Updroid: Updated android malware and its familial classification. In *Secure IT Systems*, Gruschka, N., ed, 352–368, Cham. Springer International Publishing.

[7] Android. 2019. Application fundamentals. https://developer.android.com/guide/components/fundamentals. Latest access: 03.24.2019.

[8] Tam, K., Feizollah, A., Anuar, N., Salleh, R., & Cavallaro, L. 01 2017. The evolution of android malware and android analysis techniques. *ACM Computing Surveys*, 49, 1–41. doi:10.1145/3017427.

[9] Android. 2019. Abi management. https://developer.android.com/ndk/guides/abis.html. Latest access: 05.05.2019.

[10] Android. 2019. Add c and c++ code to your project. https://developer.android.com/studio/projects/add-native-code. Latest access: 05.05.2019.

[11] Android. 2019. Art and dalvik. https://source.android.com/devices/tech/dalvik/. Latest access: 03.24.2019.

[12] Wei, F., Li, Y., Roy, S., Ou, X., & Zhou, W. 2017. Deep ground truth analysis of current android malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Polychronakis, M. & Meier, M., eds, 252–276, Cham. Springer International Publishing.

[13] Wikipedia contributors. 2019. Drive-by download — Wikipedia, the free encyclopedia. [Online; accessed 24-May-2019]. URL: https://en.wikipedia.org/w/index.php?title=Drive-by_download&oldid=885330291.

[14] Kaspersky. 2019. Android mobile security threats. Latest accessed 05.25.2019.

[15] Lukasz Siewierski, A. S. . P. T. 2019. Pha family highlights: Zen and its cousins. Latest accessed 05.25.2019.

[16] Wikipedia contributors. 2019. Click fraud — Wikipedia, the free encyclopedia. [Online; accessed 25-May-2019]. URL: https://en.wikipedia.org/w/index.php?title=Click_fraud&oldid=890943556.

[17] Sebastián, M., Rivera, R., Kotzias, P., & Caballero, J. 2016. Avclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses*, Monrose, F., Dacier, M., Blanc, G., & Garcia-Alfaro, J., eds, 230–253, Cham. Springer International Publishing.

[18] Garcia, J., Hammad, M., & Malek, S. January 2018. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Trans. Softw. Eng. Methodol.*, 26(3), 11:1–11:29. URL: http://doi.acm.org/10.1145/3162625, doi:10.1145/3162625.

[19] Hurier, M., Suarez-Tangil, G., Dash, S. K., Bissyandé, T. F., Le Traon, Y., Klein, J., & Cavallaro, L. May 2017. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 425–435. doi:10.1109/MSR.2017.57.

[20] Gruver, B. 2019. smali/baksmali. https://github.com/JesusFreke/smali. Latest access: 05.25.2019.

[21] Android. 2019. Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Latest access: 05.05.2019.

[22] Pan, B. 2019. dex2jar: Tools to work with android .dex and java .class files. https://github.com/pxb1988/dex2jar. Latest access: 05.05.2019.

[23] androguard. 2019. Androguard. https://github.com/androguard/androguard. Latest access: 05.25.2019.

[24] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., & McDaniel, P. June 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6), 259–269. URL: http://doi.acm.org/10.1145/2666356.2594299, doi:10.1145/2666356.2594299.

[25] Lantz, P. 2019. Droidbox - dynamic analysis of android apps. https://github.com/pjlantz/droidbox. Latest access: 05.05.2019.

[26] Martín García, A., Lara-Cabrera, R., & Camacho, D. 09 2018. A new tool for static and dynamic android malware analysis. 509–516. doi:10.1142/9789813273238_0066.

[27] Revivo, I. & Caspi, O. 2019. Cuckoodroid - automated android malware analysis. https://github.com/idanr1986/cuckoo-droid. Latest access: 15.05.2019.

[28] Ajin Abraham india, Dominik Schlecht germany, M. c. M. D. i. & france, V. N. 2019. Mobile security framework (mobsf). https://github.com/MobSF/Mobile-Security-Framework-MobSF. Latest access: 15.05.2019.

[29] Suarez-Tangil, G., Dash, S. K., Ahmadi, M., Kinder, J., Giacinto, G., & Cavallaro, L. 2017. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, 309–320, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/3029806.3029825, doi:10.1145/3029806.3029825.

[30] Fan, M., Liu, J., Luo, X., Chen, K., Tian, Z., Zheng, Q., & Liu, T. Aug 2018. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 13(8), 1890–1905. doi:10.1109/TIFS.2018.2806891.

[31] Kang, B., Yerima, S. Y., Mclaughlin, K., & Sezer, S. June 2016. N-opcode analysis for android malware classification and categorization. In *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, 1–7. doi:10.1109/CyberSecPODS.2016.7502343.

[32] Massarelli, L., Aniello, L., Ciccotelli, C., Querzoni, L., Ucci, D., & Baldoni, R. Oct 2017. Android malware family classification based on resource consumption over time. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, 31–38. doi:10.1109/MALWARE.2017.8323954.

[33] Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., & Cavallaro, L. May 2016. Droidscribe: Classifying android malware based on runtime behavior. In *2016 IEEE Security and Privacy Workshops (SPW)*, 252–261. doi:10.1109/SPW.2016.25.

[34] Chakraborty, T., Pierazzi, F., & Subrahmanian, V. S. 2018. Ec2: Ensemble clustering and classification for predicting android malware families. *IEEE Transactions on Dependable and Secure Computing*, 1–1. doi:10.1109/TDSC.2017.2739145.

[35] Tam, K., Khan, S. J., Fattori, A., & Cavallaro, L. 2015. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*.

[36] Symantec. 2018. Symantec 2018 internet security threat report.

[37] Leedy, P. D. 2015. Practical research : planning and design.

[38] Martín García, A., Lara-Cabrera, R., & Camacho, D. 12 2018. Android malware detection through hybrid features fusion and ensemble classifiers: The andropytool framework and the omnidroid dataset. *Information Fusion*, 52. doi:10.1016/j.inffus.2018.12.006.

[39] Arp, D., Spreitzenbarth, M., Gascon, H., & Rieck, K. 2014. Drebin: Effective and explainable detection of android malware in your pocket.

[40] Hupp, A. 2019. python-magic. https://github.com/ahupp/python-magic. Latest access: 05.05.2019.

[41] Bendersky, E. 2019. Pyelftools: Parsing elf and dwarf in python. https://github.com/eliben/pyelftools. Latest access: 05.05.2019.

[42] Android. 2019. Api reference. https://developer.android.com/reference. Latest access: 05.05.2019.

[43] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. November 2009. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1), 10–18. URL: http://doi.acm.org/10.1145/1656274.1656278, doi:10.1145/1656274.1656278.

[44] Kononenko, I. & Kukar, M. 2007. *Machine Learning and Data Mining: Introduction to Principles and Algorithms*. Horwood Publishing Limited.

[45] Geurts, P., Ernst, D., & Wehenkel, L. Apr 2006. Extremely randomized trees. *Machine Learning*, 63(1), 3–42. URL: https://doi.org/10.1007/s10994-006-6226-1, doi:10.1007/s10994-006-6226-1.

[46] Rodriguez, J. J., Kuncheva, L. I., & Alonso, C. J. Oct 2006. Rotation forest: A new classifier ensemble method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(10), 1619–1630. doi:10.1109/TPAMI.2006.211.

[47] Chang, C.-C. & Lin, C.-J. May 2011. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3), 27:1–27:27. URL: http://doi.acm.org/10.1145/1961189.1961199, doi:10.1145/1961189.1961199.

[48] Hall, M. A. 1999. Correlation-based feature selection for machine learning.

[49] Sikorski, M. & Honig, A. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition.

[50] Choudhary, S. R., Gorla, A., & Orso, A. 2015. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, 429–440, Washington, DC, USA. IEEE Computer Society. URL: https://doi.org/10.1109/ASE.2015.89, doi:10.1109/ASE.2015.89.

[51] Wikipedia contributors. 2018. F1 score — Wikipedia, the free encyclopedia. [Online; accessed 21-May-2019]. URL: https://en.wikipedia.org/w/index.php?title=F1_score&oldid=874064435.

[52] Narayanan, A., Yang, L., Chen, L., & Jinliang, L. July 2016. Adaptive and scalable android malware detection through online learning. In *2016 International Joint Conference on Neural Networks (IJCNN)*, 2484–2491. doi:10.1109/IJCNN.2016.7727508.

A Scripts

This chapter lists all the essential python script used in the project.

A.1 preprocessing.py

```
import os
import json
import numpy as np
from datetime import datetime

#Paths
PATH_CWD = os.getcwd() + "/"
PATH_METADATA = PATH_CWD + "new_dataset.csv"
PATH_DATASET = PATH_CWD + "datasets/"
PATH_STATIC_FEATURES = PATH_CWD + "static/"
PATH_STATIC_NATIVECODE = PATH_CWD + "nativecode/"
PATH_STATIC_HIDDENCODE = PATH_CWD + "hidden_dex/"
PATH_DYNAMC_DROIDBOX = PATH_CWD + "dynamic/droidbox/"
PATH_DYNAMC_STRACE = PATH_CWD + "dynamic/Strace/"
PATH_SPLIT1_DATA = PATH_CWD + "split1.csv"
PATH_SPLIT2_DATA = PATH_CWD + "split2.csv"

#Metadata indexes
FAMILY = 0
TYPE = 1
DEX_DATE = 2
APK_SIZE = 3
DEX_SIZE = 4

#globals
metadata = {}
family_hashes = {}  # {"family": [hash1..n], ...}
families = []       # list of families in the dataset
errors = []         # list of hashes for samples that failed at some point
feature_set = []       # List of all features

#file extenstions
DBOX_EXT = ".json"
STATIC_EXT = "-analysis.json"
STRACE_EXT = ".csv"
NC_EXT = "-nc.json"
```

```
HIDDEN_EXT = "-hidden.json"

#Combine api calls extracted from jar and dex files
COMBINE_JAR_DEX = False

#feature selection - features that occur in less
# than 3% of samples within family are removed
MIN_FEATURE_OCCURANCE = 3

"""
Modify globals under this comment to modify
    -features to be included in dataset
        -change title of dataset file accoring
            to features included
    -if the dataset should be split into
    training and testing
"""

#features to include in dataset

#number of unique strings (too many to be loaded in Weka)
STRINGS = False

#Static
HIDDEN = True
STATIC = True
NATIVE = True

#Dynamic
DBOX = False
STRACE = False

#Metadata
METADATA = False

#Dataset filename: hybrid, dynamic, static
FEATURE_TITLE = "dynamic"

#How to split into training/testing (0,1,2). Split 0 is no split
SPLIT = 1
#For split 2
TIMELINE_SPLIT_DATE = datetime(2016, 1, 1, 0, 0, 0)


def dump_featureset(title):
    with open(PATH_DATASET + title, "w+") as f:
```

```
        out = ""
        for i in feature_set:
            out += i + ","
        f.write(out[:-1])

def load_featureset(title):
    with open(PATH_DATASET + title, "r") as f:
        return f.read().rsplit(",")

def read_metadata(selected_dataset):
    """
    Reads metadata from a csv file.
    Expected format of csvfile:
    hash,family,type,date,apksize,vtdetect,dexsize,markets+"\n"

    loads following globals:
    metadata, families, family_hashes, errors
    """
    with open(PATH_METADATA, "r") as f:
        for row in f:
            s = row.replace("\n", "").rsplit(",")
            hash = s[0]
            if hash in selected_dataset:
                try:
                    family = s[1]
                    type = s[2]
                    dexdate = datetime.strptime(s[3], "%Y-%m-%d %H:%M:%S")
                    apksize = int(s[4])
                    dexsize = int(s[6])

                    metadata[hash] = [family, type, dexdate, apksize, dexsize]
                    if family in families:
                        family_hashes[family].append(hash)
                    else:
                        family_hashes[family] = [hash]
                        families.append(family)
                except:
                    errors.append(hash)
        print("errors: ", errors)

def normalize_path(str):
    """remove all numbers from path
        e.g. /proc/23/cmdline = /proc//cmdline
    """
    return ''.join(i for i in str if not i.isdigit())

```

```
128 def normalize_syscall(str):
129     '''
130     -Remove syscall parameters
131     -Returns "_excluded_" for system info that is not a syscall:
132         e.g. <... futex resumed>
133     '''
134     if str.startswith("<") or str.startswith("-") or str.startswith("+"):
135         return "_excluded_"
136     if "(" in str:
137         str = str.rsplit("(")[0]
138     return str
139 
140 def preprocess_strings(hash):
141     filepath = PATH_STATIC_FEATURES + hash + STATIC_EXT
142     feature_dict = {}
143 
144     with open(filepath, "r") as f:
145         json_data = json.load(f)
146 
147     analysis = json_data["Static_analysis"]
148     strings = analysis["Strings"]
149 
150     for k, v in strings.items():
151         feature_dict["S_String_" + k ] = v
152 
153     return feature_dict
154 
155 def preprocess_static(hash):
156     filepath = PATH_STATIC_FEATURES + hash + STATIC_EXT
157     feature_dict = {}
158 
159     with open(filepath, "r") as f:
160         json_data = json.load(f)
161     analysis = json_data["Static_analysis"]
162     packagename = analysis["Package name"]
163     permissions = analysis["Permissions"]
164     opcodes = analysis["Opcodes"]
165     mainactivity = analysis["Main activity"]
166     apicalls = analysis["API calls"]
167     strings = analysis["Strings"]
168     apipackages = analysis["API packages"]
169     systemcmds = analysis["System commands"]
170     intents = analysis["Intents"]
171     activities = analysis["Activities"]
172     services = analysis["Services"]
173     receivers = analysis["Receivers"]
```

```
174
175     #package name - Excluded
176     # feature = "S_PackageName_" + packagename
177     # feature_dict[feature] = 1
178
179     #Permissions
180     android_permissions = 0
181     custom_permissions = 0
182     for p in permissions:
183         if p.startswith("android"):
184             android_permissions += 1
185         else:
186             custom_permissions += 1
187         feature = "S_Permission_" + p
188         feature_dict["S_Permission_" + p ] = 1
189     feature_dict["S_Permission_NumAndroidPermissions"] = android_permissions
190     feature_dict["S_Permission_NumCustomPermissions"] = custom_permissions
191
192     #opcodes
193     for k, v in opcodes.items():
194         feature_dict["S_Opcode_" + k] = v
195
196     #mainactivity name - Exluded
197     # try:
198         # feature = "S_MainActivity_" + mainactivity
199         # feature_dict[feature] = 1
200     # except:
201         # pass
202     #api calls
203     for k, v in apicalls.items():
204         feature_dict["S_ApiCall_" + k] = v
205
206     #strings
207     len_100 = 0
208     len_200 = 0
209     len_300 = 0
210     len_400 = 0
211     len_500 = 0
212     len_1000 = 0
213     len_2500 = 0
214     len_5000 = 0
215     len_10000 = 0
216     len_15000 = 0
217     len_20000 = 0
218     num_strings = 0
219     for k, v in strings.items():
```

```
            num_strings += 1
            if len(k) > 100:
                len_100 += 1
            if len(k) > 200:
                len_200 += 1
            if len(k) > 300:
                len_300 += 1
            if len(k) > 400:
                len_400 += 1
            if len(k) > 500:
                len_500 += 1
            if len(k) > 1000:
                len_1000 += 1
            if len(k) > 2500:
                len_2500 += 1
            if len(k) > 5000:
                len_5000 += 1
            if len(k) > 10000:
                len_10000 += 1
            if len(k) > 15000:
                len_15000 += 1
            if len(k) > 20000:
                len_20000 += 1
        feature_dict["S_String_len100"] = len_100
        feature_dict["S_String_len200"] = len_200
        feature_dict["S_String_len300"] = len_300
        feature_dict["S_String_len400"] = len_400
        feature_dict["S_String_len500"] = len_500
        feature_dict["S_String_len1000"] = len_1000
        feature_dict["S_String_len2500"] = len_2500
        feature_dict["S_String_len5000"] = len_5000
        feature_dict["S_String_len10000"] = len_10000
        feature_dict["S_String_len15000"] = len_15000
        feature_dict["S_String_len20000"] = len_20000
        feature_dict["S_Strings"] = num_strings

        #api packages
        for k, v in apipackages.items():
            feature_dict["S_ApiPackage_" + k] = v

        #system commands
        for k, v in systemcmds.items():
            feature_dict["S_SystemCmd_" + k] = v

        #intents
        count = 0
```

```
    for k, v in intents.items():
        feature_dict["S_Intent_" + k] = v
        count += 1
    feature_dict["S_Intents"] = count


    #activities
    count = 0
    for k, v in activities.items():
        count += 1
    feature_dict["S_Activities"] = count

    #services
    count = 0
    for k, v in services.items():
        count += 1
    feature_dict["S_Services"] = count

    #receivers
    count = 0
    for k, v in receivers.items():
        count += 1
    feature_dict["S_Receivers"] = count

    return feature_dict

def preprocess_dynamic_dbox(hash):
    feature_dict = {}
    filepath = PATH_DYNAMC_DROIDBOX + hash + DBOX_EXT

    with open(filepath, "r") as f:
        json_data = json.load(f)
    accessedfiles = json_data["accessedfiles"]
    closenet = json_data["closenet"]
    cryptousage = json_data["cryptousage"]
    dataleaks = json_data["dataleaks"]
    dexclass = json_data["dexclass"]
    enfperm = json_data["enfperm"]
    fdaccess = json_data["fdaccess"]
    opennet = json_data["opennet"]
    phonecalls = json_data["phonecalls"]
    recvnet = json_data["recvnet"]
    recvsaction = json_data["recvsaction"]
    sendnet = json_data["sendnet"]
    sendsms = json_data["sendsms"]
    servicestart = json_data["servicestart"]
```

```
312
313     #Network
314     uniq_open_cons = []
315     for k, v in opennet.items():
316         con = v["desthost"] + ":" + v["destport"]
317         if con not in uniq_open_cons:
318             uniq_open_cons.append(con)
319     feature_dict["D_Network_UniqueOpenedConnections"] = len(uniq_open_cons)
320     uniq_closed_cons = []
321     for k, v in closenet.items():
322         """TODO VERIFY THAT THESE EXIST IN CLOSENET"""
323         con = v["desthost"] + ":" + v["destport"]
324         if con not in uniq_closed_cons:
325             uniq_closed_cons.append(con)
326     feature_dict["D_Network_UniqueClosedConnections"] = len(uniq_closed_cons)
327
328     size_pcts = 0
329     size_recv_pcts = 0
330     size_sent_pcts = 0
331     for k, v in recvnet.items():
332         len_data = len(v["data"])
333         size_pcts += len_data
334         size_recv_pcts += len_data
335     for k, v in sendnet.items():
336         len_data = len(v["data"])
337         size_pcts += len_data
338         size_sent_pcts += len_data
339     feature_dict["D_Network_SizePackets"] = size_pcts
340     feature_dict["D_Network_SizePackets_recv"] = size_recv_pcts
341     feature_dict["D_Network_SizePackets_sent"] = size_sent_pcts
342     feature_dict["D_Network_OpenedConnections"] = len(opennet)
343     feature_dict["D_Network_ClosedConnections"] = len(closenet)
344     feature_dict["D_Network_SentPackets"] = len(sendnet)
345     feature_dict["D_Network_ReceivedPackets"] = len(recvnet)
346
347     #Crypto
348     feature_dict["D_Crypto_Uasage"] = len(cryptousage)
349     cryptalg = []
350     cryptoop = {}
351     for k, v in cryptousage.items():
352         try:
353             alg = v["algorithm"]
354             if alg not in cryptalg:
355                 cryptalg.append(alg)
356         except:
357             #algorithm is not always present for cryptooperations
```

```
            pass
        op = v["operation"]
        if op in cryptoop:
            cryptoop[op] += 1
        else:
            cryptoop[op] = 1
    for k,v in cryptoop.items():
        feature_dict["D_CryptoOperation_" + k] = v
    for alg in cryptalg:
        feature_dict["D_Cryptoalgorithm_" + alg] = 1

    #Dataleakage
    types = []
    size = 0
    for k, v in dataleaks.items():
        type = v["type"].replace(" ", "")
        if type not in types:
            size += len(v["data"])
            types.append(type)
    for t in types:
        feature_dict["D_Dataleak_Type_" + t] = 1
    feature_dict["D_Dataleak_Size"] = size
    # Number of data leakage for each way numeric 3
    feature_dict["D_Dataleak"] = len(dataleaks)

    #Receivers
    feature_dict["D_Receivers"] = len(recvsaction)

    #Filesystem
    feature_dict["D_Filesystem_AccessedFiles"] = len(accessedfiles)
    operation = {}
    paths = {}
    for k, v in fdaccess.items():
        op = v["operation"]
        path = v["path"]
        path = normalize_path(path)
        if path in paths:
            paths[path] += 1
        else:
            paths[path] = 1
        if op in operation:
            operation[op] += 1
        else:
            operation[op] = 1
    for k, v in operation.items():
        feature_dict["D_Filesystem_" + k ] = v
```

```
404     for k, v in paths.items():
405         feature_dict["D_Filesystem_Fileaccess_" + k ] = v
406 
407     #SMS/Phone
408     feature_dict["D_SMS_sent"] = len(sendsms)
409     feature_dict["D_Phonecalls"] = len(phonecalls)
410 
411     #Dexclass usage
412     dexclassusage = {}
413     for k,v in dexclass.items():
414         if v["type"] in dexclassusage:
415             dexclassusage[v["type"]] +=1
416         else:
417             dexclassusage[v["type"]] = 1
418     for k, v in dexclassusage.items():
419         feature_dict["D_DexClassUsage_" + k] = v
420     feature_dict["D_DexClassUsage"] = len(dexclass)
421 
422     #Services
423     feature_dict["D_StartedServices"] = len(servicestart)
424 
425     #EnfPerm
426     feature_dict["D_EnforcedPermissions"] = len(enfperm)
427     for p in enfperm:
428         feature_dict["D_EnforcedPermission_" + p] = 1
429         p = p.rsplit(".")[-1]
430         feature_dict["D_EnforcedPermission_" + p] = 1
431 
432     return feature_dict
433 
434 def preprocess_dynamic_strace(hash):
435     feature_dict = {}
436     filepath = PATH_DYNAMC_STRACE + hash + STRACE_EXT
437 
438     with open(filepath, "r") as csv:
439         for row in csv:
440             row = row.rsplit(",")
441             pid = row[1]
442             syscall = row[2]
443             syscall = normalize_syscall(syscall)
444             feature = "D_Strace_" + syscall
445             if feature in feature_dict:
446                 feature_dict[feature] += 1
447             else:
448                 feature_dict[feature] = 1
449             #Strace is run for pid 36, but forks are included
```

```
450             if pid == "36":
451                 syscall = "Pid36" + "_" + syscall
452                 feature = "D_Strace_" + syscall
453                 if feature in feature_dict:
454                     feature_dict[feature] += 1
455                 else:
456                     feature_dict[feature] = 1
457         if "_excluded_" in feature_dict:
458             # returned by normalize_syscall() if row contains sysinfo and not
                ↪ syscall
459             feature_dict.pop("_excluded_")
460     return feature_dict
461
462 def preprocess_nativecode(hash):
463     feature_dict = {}
464     filepath = PATH_STATIC_NATIVECODE + hash + NC_EXT
465     with open(filepath, "r") as f:
466         feature_dict = json.load(f)
467     return feature_dict
468
469 def preprocess_hiddencode(hash):
470     feature_dict = {}
471     filepath = PATH_STATIC_HIDDENCODE + hash + HIDDEN_EXT
472     with open(filepath, "r") as f:
473         tmp_dict = json.load(f)
474
475     if COMBINE_JAR_DEX:
476         for k, v in tmp_dict.items():
477             if "S_H_Jar_ApiCalls_" in k:
478                 k = k.replace("S_H_Jar_ApiCalls_", "S_H_Dex_ApiCalls_")
479
480             if k in feature_dict:
481                 feature_dict[k] += v
482             else:
483                 feature_dict[k] = v
484     else:
485         feature_dict = tmp_dict
486
487     return feature_dict
488
489 def preprocess_feature(hash):
490
491     sample_features = {}
492
493     if METADATA:
494         sample_features["M_Metadata_DEXSize"] = metadata[hash][DEX_SIZE]
```

```
        sample_features["M_Metadata_APKSize"] = metadata[hash][APK_SIZE]

    if STATIC:
        static = preprocess_static(hash)
        sample_features.update(static)
    if NATIVE:
        nativecode = preprocess_nativecode(hash)
        sample_features.update(nativecode)
    if STRINGS:
        strings = preprocess_strings(hash)
        sample_features.update(strings)
    if DBOX:
        droidbox = preprocess_dynamic_dbox(hash)
        sample_features.update(droidbox)
    if STRACE:
        strace = preprocess_dynamic_strace(hash)
        sample_features.update(strace)

    if HIDDEN:
        hidden = preprocess_hiddencode(hash)
        sample_features.update(hidden)

    return sample_features

def select_features():
    counter = 0
    print(len(feature_set))
    for family, hashes in family_hashes.items():
        feature_counter = {}
        percent_cut = round(len(hashes)/100*MIN_FEATURE_OCCURANCE)
        for hash in hashes:
            counter += 1
            sample_features = preprocess_feature(hash)
            for k, v in sample_features.items():
                if k in feature_counter:
                    feature_counter[k] += 1
                else:
                    feature_counter[k] = 1
            if counter % 100 == 0:
                print(counter)
        for k, v in feature_counter.items():
            if v > percent_cut:
                if k not in feature_set:
                    feature_set.append(k)
        print("len features: ", len(feature_set))
        print(family)
```

```
541
542 def fix_sample_features(sample_features):
543     #Only use selected features and fill missing with 0
544     updated_features = {}
545     for feature in feature_set:
546         if feature in sample_features:
547             updated_features[feature] = sample_features[feature]
548         else:
549             updated_features[feature] = 0
550     return updated_features
551
552 def write_arff_header(title):
553     classes = ""
554     for fam in families:
555         classes += fam + ","
556     classes = classes[:-1]
557     with open(PATH_DATASET + title, "w") as f:
558         f.write("@RELATION \"" + "dataset" +  "\"\n\n")
559         for feature in feature_set:
560             f.write("@ATTRIBUTE \"" + feature + "\" " + "NUMERIC" + "\n")
561         # f.write("@ATTRIBUTE \"" + "date" + "\" DATE \"yyyyMMdd\" \n")
562         f.write("@ATTRIBUTE class {" + classes + "}\n\n")
563         f.write("@DATA\n")
564
565 def write_dataset(title, dataset):
566     write_arff_header(title)
567     with open(PATH_DATASET + title, "a") as f:
568         for hash in dataset:
569             sample_features = preprocess_feature(hash)
570             sample_features = fix_sample_features(sample_features)
571             output = ""
572             for feature in feature_set:
573                 output += str(sample_features[feature]) + ","
574             output += metadata[hash][FAMILY] + "\n"
575             f.write(output)
576
577 def split1():
578     training_set = []
579     testing_set = []
580     family_split_date = {}
581
582     for k, v in family_hashes.items():
583         hash_date = []
584         for hash in v:
585             date = metadata[hash][DEX_DATE]
586             hash_date.append([hash, date])
```

```
        hash_date = sorted(hash_date,key=lambda l:l[1])

        num_samples = len(v)
        part = int(num_samples/5)
        training = hash_date[:part*4]
        testing = hash_date[part*4:]

        # statistics
        first_date = training[0][1]
        last_date = testing[-1][1]
        split_date = training[-1][1]
        family_split_date[k] = [first_date, split_date, last_date]

        for i in training:
            training_set.append(i[0])
        for i in testing:
            testing_set.append(i[0])

    # Write split statistics
    with open(PATH_SPLIT1_DATA, "w+") as f:
        f.write("Family,Split date\n")
        for k, v in family_split_date.items():
            out = k
            for d in v:
                out += "," + str(d)
            out += "\n"
            f.write(out)

    return training_set, testing_set

def split2(dataset):
    training = []
    testing = []
    family_split_data = {}

    #Split based on date
    for hash in dataset:
        date = metadata[hash][DEX_DATE]
        family = metadata[hash][FAMILY]

        if family not in family_split_data:
            family_split_data[family] = [0,0]

        if date > TIMELINE_SPLIT_DATE:
            testing.append(hash)
            family_split_data[family][1] += 1
```

```
        else:
            training.append(hash)
            family_split_data[family][0] += 1

    # Write split statistics
    with open(PATH_SPLIT2_DATA, "w+") as f:
        f.write("Family,training,testing\n")
        for k, v in family_split_data.items():
            out = k + "," + str(v[0]) + "," + str(v[1]) + "\n"
            f.write(out)

    #Filter families with zero samples in the trainingset
    for k, v in family_split_data.items():
        remove = []
        if v[0] == 0:
            for hash in family_hashes[k]:
                testing.remove(hash)

    return training, testing

def main():
    dataset = os.listdir(PATH_CWD + "download/done/")
    for i in range(len(dataset)):
        dataset[i] = dataset[i].replace(".apk", "")

    read_metadata(dataset)
    select_features()

    if SPLIT == 1:
        training, testing = split1()
        write_dataset("ta1_training_" + FEATURE_TITLE + ".arff", training)
        write_dataset("ta1_testing_" + FEATURE_TITLE + ".arff", testing)
    elif SPLIT == 2:
        training, testing = split2(dataset)
        write_dataset("ta2_training_" + FEATURE_TITLE + ".arff", training)
        write_dataset("ta2_testing_" + FEATURE_TITLE + ".arff", testing)
    else:
        write_dataset("tu_" + FEATURE_TITLE + ".arff", dataset)

main()
```

A.2 vt_report.py

```
#vt_report.py
'''
Requires a VirusTotal api key.
```

```
4
5   Sends hashes to the VT API and recieves json reports containing
6   anti-virus decisions for each hash.
7   Write the reports to files named with the hash of the samples.
8   '''
9
10  import requests
11  import os
12  import numpy as np
13  import time
14
15  api_key = "INSERT_KEY_HERE"
16  url="https://www.virustotal.com/vtapi/v2/file/report"
17
18  #File containing hashes of all samples in dataset
19  PATH_DATASET = os.getcwd() + "/hashes"
20  PATH_LABELS = os.getcwd() + "/labels/"
21
22  def make_label_dir():
23      if not os.path.exists(PATH_LABELS):
24          os.makedirs(PATH_LABELS)
25
26  def get_dataset():
27      dataset = []
28      with open(PATH_DATASET, "r") as f:
29          for hash in f:
30              dataset.append(hash.replace("\n", ""))
31      #remove already downloaded
32      alrdy_found = os.listdir(PATH_LABELS)
33      for h in alrdy_found:
34          if h in dataset:
35              dataset.remove(h)
36      return dataset
37
38  def get_vt_reports(hashes, api_key, path):
39      print("[+] Started making requests to VirusTotal API")
40      counter = 1
41      for hash in hashes:
42          if counter%4 == 0:
43              time.sleep(60)
44
45          params = {'apikey': api_key, 'resource': hash}
46
47          try:
48              response = requests.get(url, params=params)
49              with open(path + hash, "w") as f:
```

```
                f.write(str(response.json()))
        except:
            print(hash)
        counter += 1

def main():
    print("[+] starting")
    make_label_dir()
    dataset = get_dataset()
    get_vt_reports(dataset, api_key, PATH_LABELS)

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

A.3 scrape_android_api.py

```
#scrape_android_api.py
import requests
import os
import re

PATH_LIB = os.getcwd() + "/lib/"
PATH_CLASS_LIST = PATH_LIB + "class_list.txt"
PATH_PACKAGE_LIST = PATH_LIB + "package_list.txt"


page = requests.get('https://developer.android.com/reference/packages')
html = str(page.text.encode(encoding='UTF-8'))

## All packages can be found in this format:
# <a href="/reference/android/package-summary" class="devsite-nav-title">
raw_packages =
 ↪  re.findall(r"/reference/\w+/*\w*/*\w*/*\w*/*\w*/*\w*/package-summary", html)
for i, p in enumerate(raw_packages):
    raw_packages[i] = p.replace("/reference", "").replace("package-summary", "")

package_dict = {}
for p in raw_packages:
    p = p.replace("/", ".")[1:-1]
    # p = p.rsplit(".")
    if p in package_dict.items():
        package_dict[p] += 1
    else:
        package_dict[p] = 1
print(len(package_dict))

```

```
fin_pack = []
for k, v in package_dict.items():
    words = k.rsplit(".")
    part = ""
    for w in words:
        part += w + "."
        if part[:-1] not in fin_pack:
            fin_pack.append(part[:-1])

print(len(fin_pack))

with open(PATH_PACKAGE_LIST, "w+") as f:
    for p in fin_pack:
        f.write(p + "\n")

raw_classes = []
regex1 = r'reference'
regex2 = r'\w+\.*\w*\.*\w*\.*\w*\.*\w*'
for package in raw_packages:
    raw_classes += re.findall(regex1 + re.escape(package) + regex2, html)


classes = []
for cl in raw_classes:
    cl = cl.rsplit("/")
    for c in cl:
        if c[0].isupper():
            classes.append(c)

fin_classes = []
class_dict = {}
for c in classes:
    if c in class_dict:
        class_dict[c] += 1
    else:
        class_dict[c] = 1


for k, v in class_dict.items():
    words = k.rsplit(".")
    part = ""
    for w in words:
        part += w + "."
        if part[:-1] not in fin_classes:
            fin_classes.append(part[:-1])

```

```
with open(PATH_CLASS_LIST, "w+") as f:
    for k, v in class_dict.items():
        f.write(k + "\n")
```

A.4 select_samples_runtime_experiment.py

```
#select_samples_runtime_experiment.py
import os
import operator
import numpy as np
import random
import matplotlib.pyplot as plt

random.seed(23)

#Paths
PATH_CWD = os.getcwd() + "/"
PATH_HISTOGRAMS = PATH_CWD + "histograms/"
PATH_METADATA = PATH_CWD + "final_dataset.csv"
PATH_SELECTED_SAMPLES = PATH_CWD + "runtime_experiment_hashes.txt"

def write_selected_samples(selected_samples):
    with open(PATH_SELECTED_SAMPLES, "w+") as f:
        out = ""
        for hash in selected_samples:
            out += hash + "\n"
        f.write(out)

def to_mb(bytes):
    mb = bytes/(10**6)
    div = int(mb/5)
    return div*5 + 5

def create_histogram(data, title):
    sorted_data = sorted(data.items(), key=operator.itemgetter(0))
    index = []
    labels = []
    values = []

    for i, d in enumerate(sorted_data):
        index.append(i)
        labels.append(str(d[0]))
        values.append(d[1])

    plt.close(1)
    font = {'size'   : 18}
```

```
41      plt.rc('font', **font)
42      plt.figure(figsize=(12,6))
43      plt.bar(index, values)
44      plt.xticks(index, labels)
45      plt.ylabel("Number of samples")
46      plt.title(title)
47      plt.savefig(PATH_HISTOGRAMS + title + ".png")
48
49  def read_sample_sizes():
50      """
51      Reads apk size for hash from csv file.
52      Expected format of csvfile:
    ↪   hash,family,type,date,apksize,vtdetect,dexsize,markets+"\n"
53      """
54
55      sample_size = []
56      sizes = {}
57      with open(PATH_METADATA, "r") as f:
58          for row in f:
59              s = row.replace("\n", "").rsplit(",")
60              hash = s[0]
61              apksize = to_mb(int(s[4]))
62              if apksize in sizes:
63                  sizes[apksize] += 1
64              else:
65                  sizes[apksize] = 1
66
67              sample_size.append([hash, apksize])
68      return sample_size, sizes
69
70  def select_samples(sample_size):
71      selected_sizes = {}
72      selected_samples = []
73      sample_size = sorted(sample_size, key=operator.itemgetter(1))
74      split = np.array_split(sample_size, 100)
75      for array in split:
76          i = random.randint(0, len(array)-1)
77          selected_samples.append(array[i][0])
78          apksize = array[i][1]
79          apksize = int(apksize)
80          if apksize in selected_sizes:
81              selected_sizes[apksize] += 1
82          else:
83              selected_sizes[apksize] = 1
84      return selected_samples, selected_sizes
85
```

```
sample_size, sizes = read_sample_sizes()
selected_samples, selected_sizes = select_samples(sample_size)
write_selected_samples(selected_samples)
create_histogram(sizes, "Dataset APK sizes")
create_histogram(selected_sizes, "Subset APK sizes")
```

A.5 get_min_sdk.py

```
from zipfile import ZipFile
import os
import androguard.core.bytecodes.apk as apk

PATH_SAMPLES = os.getcwd() + "/download/"

def get_minsdk(apk_path):
    min_sdk = "999"
    try:
        app = apk.APK(apk_path)
        min_sdk = app.get_min_sdk_version()
    except:
        pass
    if min_sdk == None:
        min_sdk = "999"
    return min_sdk

def write_midsdk_for_samples():
    samples = {}
    counter = {}
    for fn in os.listdir(PATH_SAMPLES):
        fp = PATH_SAMPLES + fn
        min_sdk = get_apk_info(fp)
        samples[fn] = min_sdk
        if min_sdk in counter:
            counter[min_sdk] += 1
        else:
            counter[min_sdk] = 1

    with open(os.getcwd() + "/minsdk", "w+") as f:
        for k, v in samples.items():
            if int(v) > 16:
                f.write(k + "," + v + "\n")
    print(counter)

def unzip(fn):
    with ZipFile(PATH_SAMPLES + fn, "r") as zf:
        hash = fn.replace(".apk", "")
```

```
        dir = PATH_SAMPLES + hash
        zf.extractall(dir)

write_midsdk_for_samples()
```

A.6 construct_dataset.py

```
#construct_dataset.py
import json
import os
from datetime import datetime
import numpy as np
import matplotlib.pyplot as plt
from operator import itemgetter

start_time = datetime.now()

#Parameters
START_DATE = datetime(2014, 9, 1, 0, 0, 0)
END_DATE = datetime(2017, 1, 1, 0, 0, 0)
MIN_NUM_OF_SAMPLES = 50
MIN_DETECTED = 5

#Path to data
PATH_TO_LABELS = os.getcwd() + "/labels/names/proposed.json"
PATH_TO_TYPES = os.getcwd() + "/labels/types/proposed.json"
PATH_TO_RANKING = os.getcwd() + "/ranking_fams_" + str(MIN_DETECTED) + ".csv"
PATH_TO_ALIAS = os.getcwd() + "/alias"
PATH_TO_METADATA = os.getcwd() + "/latest.csv"
PATH_TO_EXTRACTED_METADATA = os.getcwd() + "/metadata.csv"
PATH_TO_DATASET = os.getcwd() + "/dataset.csv"
PATH_TO_HISTOGRAMS = os.getcwd() + "/histograms/"
PATH_TO_WHITELIST = os.getcwd() + "/whitelist"

#lastest.csv indexes (will be updated to metadata indexes after loaded)
SHA256 = 0
DEX_DATE = 3
APK_SIZE = 4
VT_DETECTION = 7
DEX_SIZE = 9
MARKETS = 10
FAMILY = None
TYPE = None

#Globals
metadata = {}    # metadata pre-selection of samples
```

```
families = {}    #[hash, date, vt_detect]
selected = {}    # {fam:[hash1...n]}
dataset = {}     # metadata post-selection of samples

def update_indexes():
    #update index variables
    global FAMILY
    global TYPE
    global DEX_DATE
    global APK_SIZE
    global VT_DETECTION
    global DEX_SIZE
    global MARKETS
    FAMILY = 0
    TYPE = 1
    DEX_DATE = 2
    APK_SIZE = 3
    VT_DETECTION = 4
    DEX_SIZE = 5
    MARKETS = 6

def remove_families_with_few_samples():
    rem = []
    for k, v in families.items():
        if len(v) < MIN_NUM_OF_SAMPLES:
            rem.append(k)
    for fam in rem:
        del families[fam]
    remove = []
    for k, v in metadata.items():
        if v[FAMILY] in rem:
            remove.append(k)
    for hash in remove:
        del metadata[hash]

def remove_aliases():
    num_aliases = 0
    aliases = []
    with open(PATH_TO_ALIAS, "r") as f:
        for line in f:
            line = line.rsplit(" ")
            aliases.append(line[0])

    for alias in aliases:
        if alias in families:
            print(alias)
```

```
            del families[alias]
            num_aliases += 1
    remove = []
    for k, v in metadata.items():
        if v[FAMILY] in aliases:
            remove.append(k)
    for hash in remove:
        del metadata[hash]
    print("Number of aliases: ", num_aliases)

def load_androzoo_metadata():
    num_missing_dates = 0
    #load labels
    with open(PATH_TO_LABELS, "r") as f:
        labels = json.load(f)
    #load types
    with open(PATH_TO_TYPES, "r") as f:
        types = json.load(f)
    #read AndroZoo metadata
    with open(PATH_TO_METADATA, "r") as f:
        for row in f:
            row = row.rsplit(",")
            hash =  row[SHA256].lower()
            try:
                detect = int(row[VT_DETECTION])
            except:
                detect = 0
            # if date format invalid or missing the following if statement is
            ↪ false
            try:
                date = datetime.strptime(row[DEX_DATE], "%Y-%m-%d %H:%M:%S")
            except:
                num_missing_dates += 1
                date = END_DATE
            if hash in labels and date > START_DATE and date < END_DATE and
            ↪ detect >= MIN_DETECTED:
                family = labels[hash]
                try:
                    type = types[hash]
                except:
                    type = ""
                mdata = [family,
                         type,
                         date,
                         row[APK_SIZE],
                         row[VT_DETECTION],
```

```
                        row[DEX_SIZE],
                        row[MARKETS]]
                metadata[row[SHA256].lower()] = mdata
                #load families dict
                fdata = [hash, date, row[VT_DETECTION]]
                if family in families:
                    families[family].append(fdata)
                else:
                    families[family] = [fdata]
    print("Number of missing dates:", num_missing_dates)

def extract_metadata():
    load_androzoo_metadata()
    update_indexes()
    remove_families_with_few_samples()
    #list of aliases from AVClass github repository
    remove_aliases()

def select_samples():
    global selected
    global dataset
    t1 = None
    t2 = None

    #List of manually confirmed malware families
    whitelist = get_whitelist()

    for k, v in families.items():
        if k in whitelist:
            num_to_select = get_num_to_select(len(v))
            selected[k] = []
            if num_to_select >= 150:
                v = sorted(v, key=itemgetter(1))
                v = np.array_split(v, MIN_NUM_OF_SAMPLES)
                select = int(num_to_select/MIN_NUM_OF_SAMPLES)
                for samples in v:
                    samples = sorted(samples, key=itemgetter(2), reverse=True)
                    for i in range(select):
                        selected[k].append(samples[i][0])
            else:
                for sample in v:
                    selected[k].append(sample[0])
    #generate metadata for selected dataset
    for k, v in selected.items():
        print(k, len(v))
        for hash in v:
```

```
176             dataset[hash] = metadata[hash]
177
178 def write_metadata(metadata, path):
179     with open(path, "w+") as f:
180         for k, v in metadata.items():
181             out = str(k)
182             for i in v:
183                 out += "," + str(i)
184             f.write(out)
185
186 def write_ranking(families, filepath):
187     ranking = []
188     for k, v in families.items():
189         ranking.append([k, len(v)])
190     ranking = sorted(ranking, key=itemgetter(1), reverse=True)
191     with open(filepath, "w+") as f:
192         for r in ranking:
193             f.write(str(r[0]) + "," + str(r[1]) + "\n")
194
195 def get_num_to_select(n):
196     if n >= 20000:
197         return 2000
198     elif n >= 10000:
199         return 1000
200     elif n >= 3000:
201         return 500
202     elif n >= 1000:
203         return 300
204     elif n >= 150:
205         return 150
206     else:
207         return n
208
209 def get_whitelist():
210     whitelist = []
211     with open(PATH_TO_WHITELIST, "r") as f:
212         whitelist = f.read().rsplit(",")
213     return whitelist
214
215 extract_metadata()
216 select_samples()
217 write_metadata(dataset, PATH_TO_DATASET)
218 write_ranking(families, PATH_TO_RANKING)
219
220 #print runtime
```

```
print(datetime.now() - start_time)
```

A.7 get_dataset.py

```
import urllib.request as ur
import os
import numpy as np
import argparse

PATH_TO_DATASET = os.getcwd() + "/dataset.csv"
PATH_TO_DOWNLOADS = os.getcwd() + "/download/"
PATH_TO_EXCLUDE = os.getcwd() + "/downloaded"
PATH_TO_SPLIT = os.getcwd() + "/split"
API_KEY = "INSERT_API_KEY_HERE"
url = "https://androzoo.uni.lu/api/download?apikey=" + API_KEY + "&sha256="

download = {}

def get_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument("-p", "--part", default=0, type=int, help="Part of
    ↪  dataset to download 0,1,2, 4")
    return parser.parse_args()

def make_down_dir():
    if not os.path.exists("download"):
        os.makedirs("download")

def get_dataset_hashes():
    with open(PATH_TO_DATASET, "r") as f:
        for l in f:
            hash = l.rsplit(",")[0]
            download[hash] = None

def filter_already_downloaded():
    already_downloaded = os.listdir(PATH_TO_DOWNLOADS)
    for apk in already_downloaded:
        hash = apk.replace(".apk", "")
        if hash in download:
            del download[hash]

def split_download(n = 4):
    make_down_dir()
    get_dataset_hashes()
    filter_already_downloaded()
    hashes = []
```

```
    for hash, v in download.items():
        hashes.append(hash)
    div = np.array_split(hashes, n)
    for i, d in enumerate(div):
        with open(PATH_TO_SPLIT + str(i), "w+") as f:
            for h in d:
                f.write(h + "\n")

def download_part(part):
    make_down_dir()
    with open(PATH_TO_SPLIT + part, "r") as f:
        for i, hash in enumerate(f):
            hash = hash.replace("\n", "")
            try:
                ur.urlretrieve(url + hash, PATH_TO_DOWNLOADS + hash)
            except:
                print(hash)

args = get_arguments()
download_part(str(args.part))
```

A.8 re_label.py

```
#re_label.py
import os
import json

#Path labels
PATH_CWD = os.getcwd() + "/"
# PATH_OLD_LABELS = PATH_CWD + "/labels/names/proposed.json"
# PATH_NEW_EUPHONY_LABELS = PATH_CWD + "/output/proposed.json"
PATH_NEW_AVCLASS_LABELS = PATH_CWD + "/output/avclass.verbose"
PATH_DATASET = PATH_CWD + "/dataset.csv"
PATH_NEW_DATASET = PATH_CWD + "/final_dataset.csv"
PATH_RANKING = PATH_CWD + "/final_ranking.csv"
PATH_FILTER_MINSDK = PATH_CWD + "/minsdk.filter"
PATH_FILTER_INVALID = PATH_CWD + "/invalid.filter"
PATH_FILTER_CANNOTRUN = PATH_CWD + "/cannotrun.filter"

#Metadata indexes
HASH = 0
FAMILY = 1
TYPE = 2
DATE = 3
APK_SIZE = 4
DETECT = 5
```

```
DEX_SIZE = 6
MARKET = 7

MIN_NUM_OF_SAMPLES = 50

avclass = {}
avclass_fams = {}
dataset = {}

def load_filter(filepath):
    filter = []
    with open(filepath, "r") as f:
        for line in f:
            filter.append(line.replace("\n", ""))
    return filter

def load_avclass_labels():
    with open(PATH_NEW_AVCLASS_LABELS, "r") as f:
        for l in f:
            tmp = l.rsplit("\t")
            if tmp[1] != "[]\n":
                fam = tmp[1].rsplit("',")[0].replace("[('", "")
                hash = tmp[0]
                avclass[hash] = fam
                if fam in avclass_fams:
                    avclass_fams[fam] += 1
                else:
                    avclass_fams[fam] = 1

def remove_families_with_few_samples(families):
    rem = []
    for k, v in families.items():
        if v < MIN_NUM_OF_SAMPLES:
            rem.append(k)
    for fam in rem:
        del families[fam]
    remove = []
    for k, v in avclass.items():
        if v in rem:
            remove.append(k)
    for hash in remove:
        del avclass[hash]

def relabel():
    with open(PATH_DATASET, "r") as f:
        for s in f:
```

```
            s = s.rsplit(",")
            hash = s[HASH]
            if hash in avclass:
                dataset[hash] = [avclass[hash], s[TYPE], s[DATE], s[APK_SIZE],
                ↪  s[DETECT], s[DEX_SIZE], s[MARKET]]

def write_dataset():
    with open(PATH_NEW_DATASET, "w+") as f:
        for k, v in dataset.items():
            out = str(k)
            for i in v:
                out += "," + str(i)
            f.write(out)

def apply_filters():
    # Features could not be extracted from some of the samples
    # These are removed from the final dataset
    filter = load_filter(PATH_FILTER_MINSDK)
    filter += load_filter(PATH_FILTER_INVALID)
    filter += load_filter(PATH_FILTER_CANNOTRUN)

    print(len(filter))

    for apk in filter:
        hash = apk.replace(".apk", "")
        if hash in dataset:
            del dataset[hash]

def write_ranking():
    ranking = {}
    for k, v in dataset.items():
        fam = v[0]
        if fam in ranking:
            ranking[fam] += 1
        else:
            ranking[fam] = 1

    with open(PATH_RANKING, "w+") as f:
        for k, v in ranking.items():
            out = str(k) + "," + str(v) + "\n"
            f.write(out)

load_avclass_labels()
remove_families_with_few_samples(avclass_fams)
relabel()
apply_filters()
```

```
write_dataset()
write_ranking()
```

A.9 extract_hidden_code.py

```
#extract_hidden_code.py

"""
Requires dex2jar:
    -Download: https://github.com/pxb1988/dex2jar
    -Unzip
    -set environment path to folder
"""

import subprocess
import re
import shutil
from zipfile import ZipFile
import os
# pip install python-magic
import magic
import json
import time
import argparse
#pip install numpy
import numpy as np
from datetime import datetime
start = datetime.now()

PATH_CWD = os.getcwd() + "/"
PATH_SAMPLES = PATH_CWD + "download/done/"
PATH_HIDDEN_FILES = PATH_CWD + "hidden_dex/"
PATH_SMALI_DIR = PATH_CWD + "smali/"
PATH_LIB = PATH_CWD + "lib/"
PATH_TMP = PATH_CWD + "tmp_dex/"
PATH_JAR_TO_DEX = PATH_TMP + "jar.dex"

# loaded when looking for native code...
stat = {} # [num_dex, dex_ext, cannot disasemble, num_apk, ext_apk, num_jar,
 ↪  jar_ext, cannot disasemble]

def get_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument("-p", "--part", default=1337, type=int, help="Part of
     ↪  dataset to label 0,1,2")
```

```
39      parser.add_argument("-s", "--split", default=1337, type=int, help="spilt
        ↪  into n parts")
40      return parser.parse_args()
41
42  def write_stat():
43      with open(PATH_CWD + "hidden_stats.csv", "w+") as f:
44          for k, v in stat.items():
45              out = str(k)
46              for i in v:
47                  out += "," + str(i)
48              out += "\n"
49              f.write(out)
50
51  def remove_dir(dir):
52      shutil.rmtree(dir, ignore_errors=True)
53
54  def unzip(path):
55      with ZipFile(path, "r") as zf:
56          if "." in path:
57              dir = path.rsplit(".")[0]
58          else:
59              dir = path + "_noext"
60          zf.extractall(dir)
61          return dir + "/"
62
63  def find_hidden_dex_jar(dir, hash, features, depth=0):
64      dex = []
65      jar = []
66      ext = "noext"
67      if hash not in stat:
68          stat[hash] = [0,0,0,0,0,0,0,0]
69      for root, dirs, files in os.walk(dir):
70          for file in files:
71              file = os.path.join(root, file)
72
73              #find extension
74              if "." in file:
75                  ext = file.rsplit(".")[-1]
76
77              #find file type using magic header
78              try:
79                  filetype = magic.from_file(file)
80              except:
81                  filetype = "invalid"
82
83              #Look for files in zip archives or apk files
```

```
84              #APK and JAR files are of the same filetypes
85              #APK file will contain classes.dex file
86              if "JAR" in filetype:
87                  APK = False
88                  try:
89                      unziped = unzip(file)
90                      if os.path.exists(unziped + "classes.dex"):
91                          APK = True
92
93                      if APK:
94                          features["S_H_File_Apk_size"] += os.path.getsize(file)
95                          stat[hash][3] += 1
96                          if ext == "apk":
97                              stat[hash][4] += 1
98                          else:
99                              features["S_H_incorrectExtension"] += 1
100                             features["S_H_incorrectExtension_" + ext] = 1
101                         tmp_dex, tmp_jar, features =
                            ↪ find_hidden_dex_jar(unziped, hash, features, 1)
102                         dex += tmp_dex
103                         jar += tmp_jar
104
105                     else: #its most likley a .jar file
106                         features["S_H_File_Jar_size"] += os.path.getsize(file)
107                         stat[hash][5] += 1
108                         if ext == "jar":
109                             stat[hash][6] += 1
110                         else:
111                             features["S_H_incorrectExtension"] += 1
112                             features["S_H_incorrectExtension_" + ext] = 1
113                     jar.append(file)
114                 except:
115                     print("Not zip archive..?")
116
117
118             #Find dex files
119             if "Dalvik dex" in filetype:
120                 features["S_H_File_Dex_size"] += os.path.getsize(file)
121                 default_dex = file.replace(dir, "")
122                 if default_dex != "classes.dex" and default_dex !=
                    ↪ "classes2.dex":
123                     dex.append(file)
124                     stat[hash][0] += 1
125                     if ext == "dex":
126                         stat[hash][1] += 1
127                 elif depth > 0:
```

```
                    dex.append(file)
                    stat[hash][0] += 1
                    if ext == "dex":
                        stat[hash][1] += 1

    return dex, jar, features

def list_smali_files(dir):
    smali_files = []
    for root, dirs, files in os.walk(dir):
        for file in files:
            file = os.path.join(root, file)
            if file.endswith(".smali"):
                smali_files.append(file)
    return smali_files

def initiate_feature_dict():
    features = {}
    features["S_H_String_len100"] = 0
    features["S_H_String_len200"] = 0
    features["S_H_String_len300"] = 0
    features["S_H_String_len400"] = 0
    features["S_H_String_len500"] = 0
    features["S_H_String_len1000"] = 0
    features["S_H_String_len2500"] = 0
    features["S_H_String_len5000"] = 0
    features["S_H_String_len10000"] = 0
    features["S_H_String_len15000"] = 0
    features["S_H_String_len20000"] = 0
    features["S_H_Strings"] = 0
    features["S_H_NumAndroidApiCalls"] = 0
    features["S_H_NumOtherApiCalls"] = 0
    features["S_H_File_Dex_size"] = 0
    features["S_H_File_Jar_size"] = 0
    features["S_H_File_Apk_size"] = 0
    features["S_H_incorrectExtension"] = 0
    return features

def parse_smali_files(files, features, system_cmds, class_list, package_list,
↪   prefix):
    num_android_api_calls = 0
    num_other_api_calls = 0
    for file in files:
        with open(file, "r") as f:
            lines = f.readlines()
            for line in lines:
```

```
                line = line.strip()

                if "const-string" in line:
                    string = line.rsplit(", \"")[-1].replace("\"", "")
                    features["S_H_Strings"] += 1
                    if len(string) > 100:
                        features["S_H_String_len100"] += 1
                    if len(string) > 200:
                        features["S_H_String_len200"] += 1
                    if len(string) > 300:
                        features["S_H_String_len300"] += 1
                    if len(string) > 400:
                        features["S_H_String_len400"] += 1
                    if len(string) > 500:
                        features["S_H_String_len500"] += 1
                    if len(string) > 1000:
                        features["S_H_String_len1000"] += 1
                    if len(string) > 2500:
                        features["S_H_String_len2500"] += 1
                    if len(string) > 5000:
                        features["S_H_String_len5000"] += 1
                    if len(string) > 10000:
                        features["S_H_String_len10000"] += 1
                    if len(string) > 15000:
                        features["S_H_String_len15000"] += 1
                    if len(string) > 20000:
                       features["S_H_String_len20000"] += 1

                    #Look for cmds in strings
                    if string in system_cmds:
                        if "S_H_SystemCmd_" + string in features:
                            features["S_H_SystemCmd_" + string] += 1
                        else:
                            features["S_H_SystemCmd_" + string] = 1

                    if "invoke-" in string:
                        feature = "S_H_String_" + string
                        if feature in features:
                            features[feature] += 1
                        else:
                            features[feature] = 1

                elif "invoke-" in line:

                    line = line.rsplit("}, ")[1].rsplit(";->")
                    if len(line) == 1:
```

```
                continue

            method = line[1].rsplit("(")[0]
            if method.startswith("<init>"): #ignore init method
                continue

            package = line[0]
            if package.startswith("[L"):
                package = package[2:]
            elif package.startswith("L"):
                package = package[1:]
            else:
                continue

            package = package.replace("/", ".")
            api_call = package.rsplit(".")
            pkg = ""
            cls = ""
            android_package = False

            for api in api_call:
                if pkg + api in package_list:
                    pkg += api + "."
                    android_package = True
                else:
                    break

            if android_package:
                api_call = package.replace(pkg, "")
                api_call = api_call.rsplit(".")
                for api in api_call:

                    if cls + api in class_list:
                        cls += api + "."
                    else:
                        num_other_api_calls += 1
                        continue
                        # Not valid android api call

                api_call = pkg + cls + method

                num_android_api_calls += 1
                api = ""
                api_call = api_call.rsplit(".")
                for word in api_call:
                    api += word + "."
```

```
                                feature = prefix + api[:-1]
                                if feature in features:
                                    features[feature] += 1
                                else:
                                    features[feature] = 1
                        else:
                            num_other_api_calls += 1

    features["S_H_NumAndroidApiCalls"] += num_android_api_calls
    features["S_H_NumOtherApiCalls"] += num_other_api_calls

    return features

def disas_dex(dex_file):
    proc = subprocess.run(["d2j-dex2smali.sh", "--force", "--output",
    ↪  PATH_SMALI_DIR, dex_file], encoding='utf-8', stdout=subprocess.PIPE,
    ↪  stderr=subprocess.PIPE)
    smali_dir = proc.stderr.rsplit(" -> ")[-1].replace("\n", "").replace(" ",
    ↪  "")
    return smali_dir

def disas_jar(jar_file):
    proc = subprocess.run(["d2j-jar2dex.sh", "--force", "--output",
    ↪  PATH_JAR_TO_DEX, jar_file], encoding='utf-8', stdout=subprocess.PIPE,
    ↪  stderr=subprocess.PIPE)
    line = proc.stdout.rsplit("\n")
    dex_file = line[0].rsplit(" -> ")[-1].replace("\n", "").replace(" ", "")
    smali_dir = disas_dex(dex_file)
    return smali_dir

def extract_features(apk, system_cmds, class_index, package_index):
    features = initiate_feature_dict()
    hash = apk.replace(".apk", "")
    dir = unzip(PATH_SAMPLES + apk)
    dex, jar, features = find_hidden_dex_jar(dir, hash, features)

    for dex_file in dex:
        # try:
        prefix = "S_H_Dex_ApiCalls_"
        smali_dir = disas_dex(dex_file)
        smali_files = list_smali_files(smali_dir)
        features = parse_smali_files(smali_files, features, system_cmds,
        ↪  class_index, package_index, prefix)
        remove_dir(smali_dir)
        # except:
            # stat[hash][2] += 1
```

```
305             # print("Error while analysing dex file: ", dex_file)
306
307     for jar_file in jar:
308         # try:
309         prefix = "S_H_Jar_ApiCalls_"
310         smali_dir = disas_jar(jar_file)
311         smali_files = list_smali_files(smali_dir)
312         features = parse_smali_files(smali_files, features, system_cmds,
            ↪  class_index, package_index, prefix)
313         remove_dir(smali_dir)
314         # except:
315             # stat[hash][7] += 1
316             # print("Error while analysing jar file: ", jar_file)
317
318     remove_dir(dir)
319     return features
320
321 def load_info():
322     with open(PATH_LIB + "cmds.txt", "r") as f:
323         system_cmds = f.readlines()
324     system_cmds = [x.replace("\n", "") for x in system_cmds]
325     with open(PATH_LIB + "class_list.txt", "r") as f:
326         class_index = f.readlines()
327     class_index = [x.replace("\n", "") for x in class_index]
328     with open(PATH_LIB + "package_list.txt", "r") as f:
329         package_index = f.readlines()
330     package_index = [x.replace("\n", "") for x in package_index]
331     return system_cmds, class_index, package_index
332
333 def main():
334     global PATH_SMALI_DIR
335     #Load known feature names
336     system_cmds, class_index, package_index = load_info()
337
338     #parallelize
339     args = get_arguments()
340     part = args.part
341     split = args.split
342
343     apks = os.listdir(PATH_SAMPLES)
344
345     #Filter done
346     # done = os.listdir(PATH_HIDDEN_FILES)
347     # print("TODO: ", len(apks))
348     # for apk in done:
349         # apk = apk.replace("-hidden.json", ".apk")
```

```
        # if apk in apks:
            # apks.remove(apk)
    # print("filtered done - TODO: ", len(apks))

    #If split work
    if split != 1337:
        apks = np.array_split(apks, split)[part]
        PATH_SMALI_DIR = PATH_CWD + "smali" + str(part) + "/"
        print("after split - TODO: ", len(apks)/3)

    #Create dir for feature files
    if not os.path.exists(PATH_HIDDEN_FILES):
        os.mkdir(PATH_HIDDEN_FILES)
    if not os.path.exists(PATH_SMALI_DIR):
        os.mkdir(PATH_SMALI_DIR)
    if not os.path.exists(PATH_TMP):
        os.mkdir(PATH_TMP)


    #Extract features
    for i, apk in enumerate(apks):
        if ".apk" in apk:
            features = extract_features(apk, system_cmds, class_index,
             ↪  package_index)
            filename = apk.replace(".apk", "-hidden.json")
            with open(PATH_HIDDEN_FILES + filename, "w+") as f:
                f.write(json.dumps(features))
            if i % 100 == 0:
                print(i)
                print(datetime.now()-start)
                # break

    write_stat()
    print(datetime.now()-start)

main()
```

A.10 extract_native_code.py

```
#extract_native_code.py

import subprocess
import re
import shutil
from zipfile import ZipFile
import os
```

```
import magic
import json
import time
import argparse
import numpy as np
#pip3 install pyelftools
from elftools.elf.elffile import ELFFile
from elftools.elf.descriptions import describe_sh_flags
from datetime import datetime
start = datetime.now()

PATH_CWD = os.getcwd() + "/"
PATH_EXTENSIONS = PATH_CWD + "extensions.csv"
PATH_FILETYPES = PATH_CWD + "filetypes.csv"
PATH_SAMPLES = PATH_CWD + "/download/done/"
PATH_NC = PATH_CWD + "/nativecode/"

#sudo apt install gcc-arm-none-eabi
OBJDUMP_ARM = "arm-none-eabi-objdump"

#Valid chars for feature names
alp = "abcdefghijklmnopqrstuvwxyz"
num = "1234567890"
special = "._"
VALID_CHRS = num + alp + alp.upper() + special


# loaded when looking for native code...
ftypes = {}
extensions = {}
num_nc = {} # [nc, arm, arm fails, not arm same name arm]

def get_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument("-p", "--part", default=1337, type=int, help="Part of
    ↪  dataset to label 0,1,2")
    parser.add_argument("-s", "--split", default=1337, type=int, help="spilt
    ↪  into n parts")
    return parser.parse_args()

def is_printable(str):
    for ch in str:
        if ch not in VALID_CHRS:
            return False
    return True

```

```
52 def get_architecuture(filepath):
53     arch = magic.from_file(filepath)
54     tmp = arch.rsplit(",")
55     arch = tmp[1][1:].replace(" ", "")
56     return arch.lower()
57 
58 def write_dict(filepath, dict, list=False):
59     with open(filepath, "w+") as f:
60         for k, v in dict.items():
61             out = str(k)
62             if list:
63                 for i in v:
64                     out += "," + str(i)
65             else:
66                 out += "," + str(v)
67             out += "\n"
68             f.write(out)
69 
70 def remove_dir(dir):
71     shutil.rmtree(dir, ignore_errors=True)
72 
73 def unzip(path):
74     with ZipFile(path, "r") as zf:
75         if "." in path:
76             dir = path.rsplit(".")[0]
77         else:
78             dir = path + "_noext"
79         zf.extractall(dir)
80         return dir + "/"
81 
82 def find_native_code(dir):
83     nc = {}
84     for root, dirs, files in os.walk(dir):
85         for file in files:
86             file = os.path.join(root, file)
87             if "." in file:
88                 ext = file.rsplit(".")[-1]
89             else:
90                 ext = "noext"
91             #Statistics for report:
92             #Extensions in dataset
93             if ext in extensions:
94                 extensions[ext] += 1
95             else:
96                 extensions[ext] = 1
97             try:
```

```
                filetype = magic.from_file(file)
            except:
                filetype = "invalid"
            #File types in dataset
            if filetype in ftypes:
                ftypes[filetype] += 1
            else:
                ftypes[filetype] = 1

            if "archive" in filetype:
                try:
                    unziped = unzip(file)
                    nc.update(find_native_code(unziped))
                except:
                    print("Not zip archive..?")
            if "ELF" in filetype:
                nc[file] = ext
    return nc

def get_header_info(filepath, arch, features):
    """
    An APK might have multiple NC files, from multiple architectures,
    Features are counted per architecture.
    """
    #Number of NC files per architecture
    prefix = "S_NC_" + arch + "_"
    if prefix in features:
        features[prefix] += 1
    else:
        features[prefix] = 1

    with open(filepath, "rb") as f:
        elffile = ELFFile(f)
        elfheader = elffile.header

        #Number of program headers
        feature = prefix + "ProgramHeaders"
        if feature in features:
            features[feature] += elfheader["e_phnum"]
        else:
            features[feature] = elfheader["e_phnum"]

        #Program header size
        feature = prefix + "ProgramHeader_Size"
        if feature in features:
            features[feature] += elfheader["e_phentsize"]
```

```
144         else:
145             features[feature] = elfheader["e_phentsize"]
146 
147         #Number of sections headers
148         feature = prefix + "Sections"
149         if feature in features:
150             features[feature] += elffile.num_sections()
151         else:
152             features[feature] = elffile.num_sections()
153 
154         #Size of section headers
155         feature = prefix + "SectionHeader_Size"
156         if feature in features:
157             features[feature] += elfheader['e_shentsize']
158         else:
159             features[feature] = elfheader['e_shentsize']
160 
161         #Size of sections and flags
162         for section in elffile.iter_sections():
163             if is_printable(section.name):
164                 s_prefix = prefix + "Section_" + section.name
165 
166                 # Size
167                 feature = s_prefix + "Size"
168                 if feature in features:
169                     features[feature] += section["sh_size"]
170                 else:
171                     features[feature] = section["sh_size"]
172                 # Flags
173                 flags = describe_sh_flags(section["sh_flags"])
174                 for flag in flags:
175                     feature = s_prefix + "_Flag_" + flag
176                     if feature not in features:
177                         features[feature] = 1
178     return features
179 
180 def get_external_calls(filepath, arch, features):
181     if arch == "arm":
182         objdump = OBJDUMP_ARM
183     else:
184         print("Unknown architecture")
185         return features
186 
187     proc = subprocess.run([objdump, "-d", filepath], encoding='utf-8',
      ↪  stdout=subprocess.PIPE)
188     prefix = "S_NC_" + arch + "_pltCall_"
```

```
    for line in proc.stdout.rsplit("\n"):
        if "@plt" in line and line.endswith(">"):
            call = line.rsplit("<")[-1].rsplit("@")[0]
            feature = prefix + call
            if feature in features:
                features[feature] += 1
            else:
                features[feature] = 1
    return features

def extract_features(apk):
    features = {}
    dir = unzip(PATH_SAMPLES + apk)
    nc = find_native_code(dir)
    hash = apk.replace(".apk", "")
    num_nc[hash] = [0,0,0,0]
    num_nc[hash][0] = len(nc)
    names_arm = []
    names_not_arm = []
    for k, v in nc.items():
        # Inconsistant extension for .so file
        if v != "so" and v != "noext":
            # print("wrong ext: ", v)
            if "S_NC_IncorrectExtensions" in features:
                features["S_NC_IncorrectExtensions"] += 1
            else:
                features["S_NC_IncorrectExtensions"] = 1
            features["S_NC_IncorrectExtensions" + v] = 1

        arch = get_architecuture(k)
        name = k.rsplit("/")[-1]
        if arch == "arm":
            size = os.path.getsize(k)
            feature = "S_NC_ARM_Size"
            if feature in features:
                features[feature] += size
            else:
                features[feature] = size
            names_arm.append(name)
            num_nc[hash][1] += 1
            try:
                features = get_header_info(k, arch, features)
                features = get_external_calls(k, arch, features)
            except:
                print("failed to disasemble - get header info")
                num_nc[hash][2] += 1
```

```
                pass
        else:
            names_not_arm.append(name)
    remove_dir(dir)
    for n in names_not_arm:
        if n in names_arm:
            num_nc[hash][3] += 1
    return features

def main():
    #Create dir for NativeCode feature files
    if not os.path.exists(PATH_NC):
        os.mkdir(PATH_NC)


    #parallelize
    args = get_arguments()
    part = args.part
    split = args.split


    #Get list of apks
    apks = os.listdir(PATH_SAMPLES)


    #Filter out analysed
    # done = os.listdir(PATH_NC)
    # print("TODO: ", len(apks))
    # for apk in done:
        # apk = apk.replace("-nc.json", ".apk")
        # if apk in apks:
            # apks.remove(apk)
    # print("filtered done - TODO: ", len(apks))


    #If split work
    if split != 1337:
        apks = np.array_split(apks, split)[part]
        print("after split - TODO: ", len(apks)/3)


    for i, apk in enumerate(apks):  #split[p]
        if ".apk" in apk:
            features = extract_features(apk)
            filename = apk.replace(".apk", "-nc.json")
            with open(PATH_NC + filename, "w+") as f:
```

```
                f.write(json.dumps(features))
            if i % 100 == 0:
                print(i)
                print(datetime.now()-start)
                # break

    write_dict(PATH_CWD + "extensions", extensions)
    write_dict(PATH_CWD + "filetypes", ftypes)
    write_dict(PATH_CWD + "nc_stats.csv", num_nc, list=True)
    print(datetime.now()-start)

main()
```

A.11 apk_statistics.py

```
#apk_statistics.py
import operator
import os
import matplotlib.pyplot as plt
from datetime import datetime

PATH_CWD = os.getcwd() + "/"

#Input files
PATH_METADATA = PATH_CWD + "final_dataset.csv"
PATH_FILETYPES = PATH_CWD + "filetypes"
PATH_EXTENSIONS = PATH_CWD + "extensions"
PATH_NC_STAT = PATH_CWD + "runtime_data/nc_stats.csv"
PATH_HIDDEN_STAT = PATH_CWD + "hidden_stats.csv"

#Output dir
PATH_HISTOGRAMS = PATH_CWD + "histograms/"

#Output files
PATH_FILETYPE_ANALISIS = PATH_CWD + "filetypes_analysis.csv"
PATH_EXTENSIONS_ANALISIS = PATH_CWD + "extensions_analysis.csv"
PATH_NC_STATISTICS = PATH_CWD + "nc_statistics.csv"
PATH_HIDDEN_STATISTICS = PATH_CWD + "hidden_statistics.csv"


def read_metadata(dataset):
    metadata = {}
    errors = []
    with open(PATH_METADATA, "r") as f:
        for row in f:
            s = row.replace("\n", "").rsplit(",")
```

```
            hash = s[0]
            if hash in dataset:
                family = s[1]
                type = s[2]
                dexdate = datetime.strptime(s[3], "%Y-%m-%d %H:%M:%S")
                apksize = int(s[4])
                dexsize = int(s[6])

                metadata[hash] = [family, type, dexdate, apksize, dexsize]

    return metadata

def create_histogram(data, date_index, title, y_title):
    sorted_data = sorted(data, key=operator.itemgetter(1))
    start = min(sorted_data, key=operator.itemgetter(1))
    cur_year = start[1].year
    quarter = int((start[1].month-1)/3)
    index = []
    labels = []
    values = []
    i = 0
    counter = 0
    total = 0
    index.append(i)
    labels.append(" Q" + str(quarter + 1) + "-" + str(cur_year)[2:])

    for d in sorted_data:
        if cur_year == d[1].year and quarter == int((d[1].month-1)/3):
            total += 1
            if d[0] > 0:
                counter += 1
        else:
            cur_year = d[1].year
            quarter = int((d[1].month-1)/3)
            i += 1
            index.append(i)
            labels.append(" Q" + str(quarter + 1) + "-" + str(cur_year)[2:])
            values.append((counter*100)/total)
            counter = 0
            total = 0
    values.append((counter*100)/total)

    plt.close(1)
    font = {'size'   : 18}
    plt.rc('font', **font)
    plt.figure(figsize=(14,6))
```

```
78      plt.bar(index, values)
79      plt.xticks(index, labels)
80      plt.ylabel(y_title)
81      plt.title(title)
82      plt.savefig(PATH_HISTOGRAMS + title + ".png")
83
84  def add_to_dict(dict, feature, value):
85      if feature in dict:
86          dict[feature] += value
87      else:
88          dict[feature] = value
89      return dict
90
91  def analyze_filetypes(filepath):
92
93      with open(filepath, "r") as f:
94          lines = f.readlines()
95
96      ftypes = {}
97      for line in lines:
98          line = line.rsplit(",")
99          count = int(line.pop())
100         ft = line[0].rsplit(" ")[0]
101
102         ftypes = add_to_dict(ftypes, ft, count)
103
104     # Remove filetypes with less than 500 files
105     rem = []
106     for k, v in ftypes.items():
107         if v < 1500:
108             rem.append(k)
109     for k in rem:
110         del ftypes[k]
111
112     # Sort file types
113     sorted_ftypes = sorted(ftypes.items(), key=operator.itemgetter(1),
        ↪ reverse=True)
114
115     # get full file type of filetype starting with token
116     for i, d in enumerate(sorted_ftypes):
117         full_ft = ""
118         most = 0
119         for line in lines:
120             if line.startswith(d[0]):
121                 line = line.rsplit(",")
122                 count = int(line.pop())
```

```
                ft = line[0].rsplit(":")[0].rsplit("(")[0]
                if count > most:
                    most = count
                    full_ft = ft
        sorted_ftypes[i] += (full_ft, most)

    with open(PATH_FILETYPE_ANALISIS, "w+") as f:
        f.write("Filetype short,Count,Filetype,Count\n")
        for d in sorted_ftypes:
            out = ""
            for i in d:
                out += str(i) + ","
            out = out[:-1] + "\n"
            f.write(out)

def analyze_extensions(filepath):
    with open(filepath, "r", encoding="utf8") as f:
        lines = f.readlines()

    extensions = {}
    for line in lines:
        ext = line.rsplit(",")[0]
        count = line.rsplit(",")[1].replace("\n", "")
        count = int(count)
        if count > 1500:
            extensions[ext] = count

    with open(PATH_EXTENSIONS_ANALISIS, "w+") as f:
        f.write("Extension,Count\n")
        for k, v in extensions.items():
            out = str(k) + "," + str(v) + "\n"
            f.write(out)

def analyze_nc(filepath, metadata):
    #[nc, arm, arm fails, not arm same name arm]
    nc_stat = {}
    nc_date = []

    num_nc_files = 0
    num_arm_files = 0
    num_arm_error = 0
    num_not_arm_same_name = 0
    num_not_arm_not_same_name = 0
    has_nc = 0

    with open(filepath, "r") as f:
```

```
        for l in f:
            tmp = l.replace("\n", "").rsplit(",")
            hash = tmp[0]
            nc_stat[hash] = [int(tmp[1]), int(tmp[2]), int(tmp[3]), int(tmp[4])]

            #list for histogram

            if int(tmp[1]) > 0:
                nc = 1
            else:
                nc = 0
            try:
                nc_date.append([nc, metadata[hash][2]])
            except:
                print(hash)

    for k, v in nc_stat.items():
        if v[0] > 0:
            has_nc += 1
        num_nc_files += v[0]
        num_arm_files += v[1]
        num_arm_error += v[2]
        num_not_arm_same_name += v[3]
    num_not_arm_not_same_name = num_nc_files - num_arm_files -
    ↪ num_not_arm_same_name

    with open(PATH_NC_STATISTICS, "w+") as f:
        out = "Samples," + str(len(nc_stat)) + "\n"
        out += "Samples with NC," + str(has_nc) + "\n"
        out += "NC files," + str(num_nc_files) + "\n"
        out += "Architecture ARM," + str(num_arm_files) + "\n"
        out += "Failed to analyse ARM," + str(num_arm_error) + "\n"
        out += "Not Arm same name," + str(num_not_arm_same_name) + "\n"
        out += "Not Arm different name," + str(num_not_arm_not_same_name) + "\n"
        f.write(out)

    create_histogram(nc_date, 1, "Native code usage", "Percentage of samples
    ↪ with native code")

def analyze_hidden(filepath, metadata):
    hidden_date = []

    num_files = 0

    num_dex_files = 0
    num_dex_ext = 0
```

```
    num_dex_cannot_disas = 0

    num_apk_files = 0
    num_apk_ext = 0

    num_jar_files = 0
    num_jar_ext = 0
    num_jar_cannot_disas = 0

    has_hidden = 0

    with open(filepath, "r") as f:
        # Each line in file:
        # hash, num_dex, dex_ext, cannot disasemble, num_apk, ext_apk, num_jar,
        ↪  jar_ext, cannot disasemble
        for l in f:
            num_files += 1

            tmp = l.replace("\n", "").rsplit(",")
            hash = tmp[0]

            num_dex = int(tmp[1])
            num_apk = int(tmp[4])
            num_jar = int(tmp[6])

            num_dex_files += num_dex
            num_dex_ext += int(tmp[2])
            num_dex_cannot_disas += int(tmp[3])

            num_apk_files += num_apk
            num_apk_ext += int(tmp[5])

            num_jar_files += num_jar
            num_jar_ext += int(tmp[7])
            num_jar_cannot_disas += int(tmp[8])

            #Has jar, apk or dex files
            if num_dex > 0 or num_apk > 0 or num_jar > 0:

                has_hidden += 1
                hidden = 1
            else:
                hidden = 0
            try:
                hidden_date.append([hidden, metadata[hash][2]])
            except:
```

```
                print(hash)


    with open(PATH_HIDDEN_STATISTICS, "w+") as f:

        out = "Samples," + str(num_files) + "\n"
        out += "Samples with HC," + str(has_hidden) + "\n"

        out += "Dex files," + str(num_dex_files) + "\n"
        out += "Dex correct extensions," + str(num_dex_ext) + "\n"
        out += "Dex cannot disasemble," + str(num_dex_cannot_disas) + "\n"

        out += "APK files," + str(num_apk_files) + "\n"
        out += "APK correct extensions," + str(num_apk_ext) + "\n"

        out += "Jar files," + str(num_jar_files) + "\n"
        out += "Jar correct extensions," + str(num_jar_ext) + "\n"
        out += "Jar cannot disasemble," + str(num_jar_cannot_disas) + "\n"

        f.write(out)

    create_histogram(hidden_date, 1, "Hidden code usage", "Percentage of samples
    ↪  with hidden code")

#Load list of dataset hashes
dataset = os.listdir(PATH_CWD + "download/done/")
for i in range(len(dataset)):
    dataset[i] = dataset[i].replace(".apk", "")


metadata = read_metadata(dataset)
# analyze_filetypes(PATH_FILETYPES)
# analyze_extensions(PATH_EXTENSIONS)
# analyze_nc(PATH_NC_STAT, metadata)
analyze_hidden(PATH_HIDDEN_STAT, metadata)
```