DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Repackaged Malware Detection in Android

Jona Neumeier

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Repackaged Malware Detection in Android

Wiederverpackte Schadprogramm Erkennung in Android

Author:	Jona Neumeier
Supervisor:	Prof. Dr. Alexander Pretschner
Advisor:	Aleieldin Salem
Submission Date:	15.09.2017

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2017 Jona Neumeier

Acknowledgments

First, I want to thank my advisor Aleieldin Salem for the constant assistance during the making of this paper. Along with that I want to thank the Chair of Software Engineering at the Technical University Munich, which made it possible for me to write this thesis.

Furthermore, I want to give thanks to the team that developed GroddDroid, especially my primary contact person Mourad Leslous, who was always open for questions about their implementation and provided the latest version of GroddDroid.

Abstract

With the rise of mobile operating systems such as *Google's Android*, the number of malware instances for such platforms rose as well. *Repackaging* is a technique deployed by malware authors to evade detection. This raises the need for a better detection mechanism for exactly this type of malware instances. Currently, the approach to detect repackaged malware is to stimulate an application once, extracting features out of it and feeding them into some form of learning algorithm. With this approach there is no room for feedback, possibly gained by the learning part, to be channeled back and enhance the stimulation of the application. We argue that exactly this feedback, from the detection part back into the stimulation part, can enhance the results of the detection and therefore lead to a better malware detection. In this paper, we will be using the *GroddDroid*-tool to stimulate and manipulate applications. We create a feedback loop between *GroddDroid* as our stimulation engine and different machine learning algorithms, iterating through that loop until we have the best possible result to detect malware. In each round *GroddDroid* will force different sections by changing the source code of the application, hoping to enhance our results. The overall goal is to have a better detection of repackaged *Android* malware. Using samples from the *MalGenome* and *Piggybacked* dataset we evaluated our approach against conventional static- and dynamic-based detection techniques. The results show that we are able to enhance the results using the dynamic feature set with this continued forcing loop compared to the results achieved without this forcing loop.

Contents

Acknowledgments **iii**

Abstract **iv**

1. Introduction **1**
1.1. Motivation . . . 2
1.2. Proposed solution and objectives . . . 3
1.3. Literature review . . . 4
1.4. Organization . . . 5

2. Background **6**
2.1. Android malware . . . 6
2.1.1. Repackaged malware . . . 7
2.2. Behavior stimulation . . . 9
2.2.1. GroddDroid . . . 10
2.3. Machine learning . . . 14
2.3.1. Features . . . 15
2.3.2. Support Vector Machines . . . 16
2.3.3. Decision Trees and Random Forests . . . 16
2.3.4. *k*-Nearest Neighbors . . . 17
2.3.5. Performance measures . . . 17
2.3.6. Cross-Validation and *k*-Fold Cross-Validation . . . 18

3. Methodology **20**
3.1. State-of-the-art . . . 20
3.2. Methodology and architecture . . . 21
3.3. Stimulation . . . 22
3.4. Detection . . . 23
3.5. Feature selection . . . 23
3.6. Feedback to GroddDroid . . . 24

4. Implementation **25**
4.1. Development environment . . . 25

4.2. Used tools . . . 25
4.2.1. GroddDroid adaptations . . . 25
4.2.2. SciKit-Learn . . . 27
4.2.3. NumPy . . . 28
4.2.4. Genymotion and VirtualBox . . . 28
4.3. Phenax . . . 28
4.3.1. Inputs and outputs . . . 28
4.3.2. Feature extraction handler . . . 29
4.3.3. Config file handler . . . 29
4.3.4. Machine learning . . . 30
4.3.5. Main file . . . 30

5. Evaluation **32**
5.1. Experimentation Environment . . . 32
5.2. Experiments . . . 32
5.2.1. Data sources . . . 33
5.2.2. Variables . . . 34
5.2.3. Piggybacked dataset experiments . . . 35
5.2.4. MalGenome dataset experiments . . . 42
5.3. Discussion . . . 49

6. Conclusion **51**

7. Further Work **52**

List of Figures **53**

List of Tables **55**

Bibliography **57**

A. Appendix: List of all extracted features **61**

B. Appendix: Heuristics database **66**

C. Appendix: Main file source code **71**

1. Introduction

Google's mobile operating system **Android** is the dominant system for mobile phones and other mobile devices with around 90% market share [1]–[3]. Today, nearly every adult, teenager and even children are using smartphones for work, communication, entertainment and other fields on a daily basis. One reason for the popularity of this system might be the ability to easily acquire new applications (apps), not only through the official *Google Play Store*, but also through third party marketplaces. Well aware of this fact, malware authors try to exploit this by uploading malicious applications in these marketplaces with the goal of getting as many downloads as possible. Unfortunately, most users are not aware of this threat. New apps with malicious intentions are published and spread every day, even through official channels such as the *Google Play Store* [4], [5]. Because of this, most marketplaces try to counter this by deploying different defense mechanisms to detect possibly malicious applications within their marketplace. *Google*, for example, built a system called *Bouncer*, which automatically analyzes applications uploaded to the *Play Store* and locks them out, in case the system labels them malicious [1], [6]. Though, these systems are far from perfect. Partially this is due to the fact that authors of malicious applications use a range of different techniques. According to [4], [7], [8], **repackaging** or **piggypacking** is one such technique and one of the most common ones used. In essence, the authors describe this technique as follows: Attackers download applications from a marketplace, inject a malicious payload into the source code and make sure it is invoked. After that, they re-upload the altered APK file to the marketplace. The authors argue that with this, the altered applications are not easily detected, as they often only slightly differ from the official application. Another reason described is that the payload is often only invoked through specific events during the runtime of the applications and techniques, like code obfuscations, which make it hard to find such payloads through static analysis.

Today, most techniques to find malware in mobile operating systems use static and dynamic approaches to stimulate applications and extracting features or logs, which are then analyzed with the help of detection algorithms [4], [7]–[9].

In our opinion these approaches can be enhanced to be more flexible and therefore better for detecting malicious applications: The stimulation of each application is only done once, which means it cannot adapt to possible insights gained from the detection part.

For this reason, we want to propose a continuous feedback loop over the stimulation and detection part, seen in figure 1.1, to improve the results and provide a better detection of *Android* malware applications. The idea is to have a constant exchange of information between the stimulation and the detection part, in order to adapt one part to possible insights gained from the other part.

The next chapter will cover why we think this is a goal worth achieving.

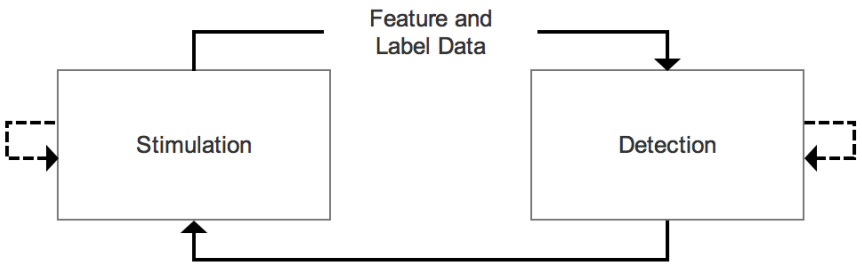

Figure 1.1.: Proposed feedback cycle: First we will stimulate an application to extract features. These will then be channeled into the detection part. Depending on the results, we may make another iteration of stimulating and detecting.

1.1. Motivation

The motivation behind our research is manifold, the most important points to us are the following.

Firstly, we are concentrating on the type of malware instances called repackaged. According to Yajin Zhou and Xuxian Jiang, 86% of the *MalGenome* dataset are of this type of malware. Projected on all applications published, this type of malware makes up the largest part of all malware types. We also acquired a relatively new dataset of malware instances based on the *AndroZoo* project, which primarily focuses on repackaged applications, called the *Piggybacked* dataset [8], [10]. The popularity of repackaging could be due to the fact that applications are not used standalone, but payloads are nested inside of an existing application and therefore hard to spot without a detailed investigation. Additionally, once the malicious payload has been written, it can be automatically injected into many different applications, which makes it easy for attackers to mass produce such malware instances [8]. This shows that these types of applications are very popular and form a large part of all malicious mobile applications, which makes them worth analyzing in more depth.

Secondly and more importantly, repackaged applications undermine the **trust of**

the user in mobile applications, mobile systems, their developers and the distribution channels. For normal and technical users, it is hard to identify applications as repackaged. One can never be sure if they are installing malware on their very own device or not. The paradox of self-installing harmful applications on one's own device is not a rarity anymore and has to be mitigated. Furthermore, the malicious parts of these applications can lie dormant for months or even years, before they get triggered by the attacker or through a specific event taking place. Then, if triggered and the applications behave unusual, the user is likely to write it off as a small error, not worth worrying about. This damages the relationship, and most important the trust, between the users and their applications, the developers of the application and the distribution channels of the apps. This may lead to the use of mobile systems and applications with higher caution, which counters the original intent of such platforms, to be easily accessible, trustworthy, easy to use and helpful.

1.2. Proposed solution and objectives

Our main objective is to establish a better framework for detecting *Android* malware instances with a focus on repackaged or piggybacked applications. As explained before, most currently used techniques for this task perform the stimulation and detection part only once, without sharing much information in between these two parts.

We argue that with the feedback structure seen in figure 1.1, we can build a more reliable and dynamic framework for detecting malicious applications. In this figure, we see two main components, namely the stimulation part and the detection part, similar to current approaches. Our goal is to let the information flow in between these two sides. For this, the stimulation part feeds the detection with features extracted from each application for analysis. The information we want to give back from the detection part to the stimulation part is to force specific, and possible malicious, parts of the applications. After each analysis, we compare the results with the previous run. If we get worse results, we stop the whole program and output our results. If we get better results, we perform another iteration over the stimulation and detection part.

For the stimulation part we specifically want to test out the tool *GroddDroid* and determine if it is applicable for the task at hand. On the detection side we work with Support Vector Machines, Decision Trees and k-Nearest Neighbors classifiers.

Repackaged malware instances are extended with malicious payloads, which are often shielded by specific conditions, examples for such conditions can be seen in figure 2.3. Continuously trying to force different parts of mobile applications will eventually lead to the execution of such shielded parts, resulting in a higher code coverage, concluding in better malware detection. The last thing which this paper tries

to achieve, is the comparison of the results of different malware and goodware datasets analyzed by our framework, as well as comparing the results by using different GUI stimulation methods.

1.3. Literature review

We did not find any works proposing a similar solution as we do in this paper. The only exception is the research of our advisor Aleieldin Salem, from whom we adopted the idea and specifics for this research. Our structure has two main parts, as seen in figure 1.1. The stimulation part and the detection part. Both fields have been heavily investigated within recent years. However, the specific combination, through a continuously running feedback loop that we are proposing, has not been done before, as of our best knowledge. We will now present some similar topics in these individual research fields.

Android malware analysis is a highly researched topic, mainly due to the increased mobile phone usage. We want to give an overview of a few research papers that cover this topic. Similar to one aspect of our main tool *GroddDroid*, Li Li et al. in [11] have developed a solution called *HookRanker*. It automatically finds and recommends possibly malicious packages or methods that have been inserted into the application using the repacking technique. Besides, the authors in [9] developed a tool called *HSOMiner*, to detect *Hidden sensitive operations* in *Android* applications. These operations are often added through repackaging. Lindorfer et al. in [1] built a tool called *Andrubis* which also does static and dynamic analysis on *Android* applications. The research in [12] focuses more on the GUI stimulation for detecting malware applications. There are numerous more examples of papers researching *Android* malware and their detection. Some of them focusing on dynamic, static or hybrid approaches, some focusing on other aspects like obtaining API traces, system or method call traces and other features from the applications. We listed the research we found closest to our used solution. Interested readers can find many other papers with other focuses in the same research field.

Stimulating and generating input for an application can be done in several different forms. Branch forcing, symbolic execution or the adapting of the environment are just some examples of possible ways to explore an application in its entirety. In [13] *FuzzDroid* is presented, which strives for building a perfect condition around an application, so that it reveals all malicious behavior. The research conducted in [14] explores different methods and techniques for generating input and stimulating an application and its GUI and compares the varied results afterwards. There is also the *Application Exerciser Monkey* which *Google* delivers with *ADB* (Android Debug Bridge)

[15]. Originally meant for testing purposes, it can also be used to generate input and stimulate the GUI for experiments on detecting malware.

For the detection part we will exclusively look at **machine learning algorithms**. There is a wide range of research done in the field of machine learning. Especially over the last years this topic gained a lot of popularity in the field of computer science and data science. Since we will be using classification, which is a common use-case for supervised learning and we do not have any specific or unusual requirements for our algorithms, we will not present any specific paper or book here.

1.4. Organization

The remaining part of this thesis document is organized as outlined in the following: The next chapter will cover all the background information needed to understand the individual components of our proposed solution. Therefore, we give a brief history about *Android* malware and will especially concentrate on repackaged applications. Following that, we will elaborate on the stimulation of applications, which is one of the two main parts of our implemented structure. The other main part is machine learning, which will be explained in the consecutive section. Chapter 3 will explain our methodology, architecture and high level design of the proposed framework, which we call **Phenax**. Chapter 4 will focus on the actual implementation of our solution. It covers all the software tools used and explains how we build the framework around them to integrate them to a whole. The next, and maybe most important chapter, will review the results drawn from our solution and insights gained from them. We conclude this work with a finishing statement as well as giving ideas on how this work could be continued in the future.

2. Background

The goal of this chapter is to give the reader a basic understanding of the different components with which this thesis is dealing. As seen in figure 1.1 the structure is built upon two main components, the stimulation and the detection part. Both will be explained in the following chapters. But first we want to give a brief explanation about *Android* apps with malicious intentions, their stimulation and machine learning in general.

2.1. Android malware

Android was introduced by *Google* in 2007. Unfortunately, the number of applications with malicious intentions, rose in nearly the same rate as new apps were published [4]. Attackers have many different reasons for making malicious applications, such as stealing private information, simple trolling, financial gain or building botnets. Yajin Zhou et al. further summarizes the intentions of attackers using mobile malware as: "privilege escalation, remote control, financial charges and personal information stealing" [4]. Therefore, various malicious apps can be found in nearly every marketplace.

According to Tam et al. in [5], developers in the 1980s became aware that the possibility of making money with malware was very real. The author continues that mobile malware was not popular before 2009, but exploded in popularity afterwards, because of the possibility of making money. Nowadays, *Android* malware contributes up to 79% of mobile malware and developers are possibly making several thousand dollars in profit a month with such techniques, which according to Tam et al. can be seen as a direct consequence of the above points.

In [4] Yajin Zhou and Xuxian Jiang show their observations on a dataset of 1260 malware instances. They conclude that most instances are only an adaptation of a malware family, therefore many of them have a lot of identical characteristics. In their case, they found around 49 families to make up the whole dataset. Another interesting finding of this paper is that the origin of such mobile malware can often be the official *Google Play Store*, next to other alternative marketplaces. Again, this shows the enormous threat that is imposed, as most users think that all applications from the official *Google* store are save. The authors further mapped most mobile malware

applications in the following categories, which describes their installation/distribution method:

- Repackaging
- Update Attack
- Drive-by Download
- Others

Since our research is mainly devoted to repacked applications, we will elaborate on this specific category in the following sub section. Interested readers can find the specifics for each of the above categories in [4].

2.1.1. Repackaged malware

As seen in the previous section malicious *Android* applications can be categorized in different categories. We now want to explain the category which we focus on the most in this writing: *Repackaged* or *piggybacked* applications. Some papers differentiate the two terms, in seeing piggybacking as a subcategory of repackaging [8]. However, in this paper we will use both terms as synonyms, meaning the injection of malicious payloads into an existing application by a third party.

One of the reasons this type of malware is so popular might be because the *Google Play store* allows developers to self-sign their applications, in order to get more developers to upload apps to this store [1]. Another reason could be that once the malicious payload is written, the injection process can be automated and a large number of applications can be altered to hold the payload [8]. *Bouncer*, a service deployed by *Google*, to detect and delete malicious applications from the *Play store*, is also not effective enough to mitigate this risk by a significant portion [6]. To examine more reasons for the choice to focus on this category, we will introduce the definition and common practices for repackaging.

We will take the definition for repackaging from [4]: "In essence, malware authors may locate and download popular apps, disassemble them, enclose malicious payloads, and then re-assemble and submit the new apps to official and/or alternative Android Markets." The analogy of a Trojan horse can be made here. Attackers want to trick users into downloading the repackaged application, because the repackaged app has a similar name or icon as the official one. Or even worse: the repackaged application replaces the official one in the marketplace. They want the users to infect themselves. Of course, the authors of these repackaged malware instances want to be undetected, which is why they use similar class names, obfuscation in the source code or shield the

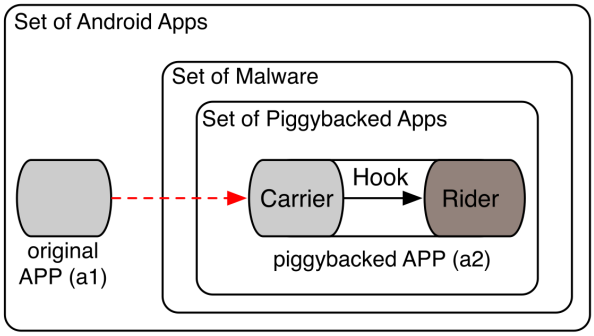

Figure 2.1.: Repackaging/Piggybacking structure [8]. The malicious rider code extends the original carrier application. The hook connects the rider to the carrier.

invocation of the malicious payload through specific conditions [4]. Examples for such trigger conditions can be seen in figure 2.3.

Figure 2.1 shows the basic, most often used structure of piggybacking. Explained in [8], [11], the literature refers to the original application as **carrier** and the malicious part that extends it as **rider**. The **hook**, which is also inserted by the attacker, is the connection between the carrier and the rider, as it is responsible to trigger the malicious payload. This triggering is often bound to specific conditions, such as a specific day or reaching a specific level in a game that needs to be true for the hook to fire.

Two types of hooks can be differentiated [8], [11]:

$Type_1$ hooks: These hooks explicitly insert a method call into to the rider source code and therefore connect the carrier to the rider through that method invoke. With this approach, the *Android* Manifest file does not have to be changed.

$Type_2$ hooks: In contrast to that, $type_2$ hooks need to be registered in the *Android* Manifest file, as they are realized as individual components and are not explicitly connected to any code within the carrier. Using the *Android* event system, these hooks listen for specific events such as the BOOT_COMPLETED broadcast from the *Android* system or a specific button click by the user until they trigger the malicious rider code.

As mentioned beforehand and by the authors of [4], [8], [11], hooks of $type_1$ and $type_2$ are often shielded by specific conditions, which have to be met for the hook to be executed. These conditions could be waiting for specific events from the user or from the system. As well as waiting for a specific state of the application to be reached. Examples are a click on a specific button, the arrival of a specific day, specific system broadcasts like BOOT_COMPLETED or the test if the application is running on an emulator or not.

2.2. Behavior stimulation

The main focus of stimulating an application is to get the app to reveal all its functionality, be it benign or malicious. Stimulation techniques are also used in a variety of different scenarios, apart from our use-case, like finding bugs during the software development. The stimulation can be done statically by looking at the source code of an application or it can be done dynamically by running the app on a device [6].

The stimulation can reach from rudimentary or random techniques, like the *Application Exerciser Monkey* delivered by *Google* which is shipped with the development tool ADB [15], to more complex techniques described in the following. The *Monkey* runner randomly creates input events, such as clicks and scrolling or changing the orientation for the stimulation. However, with this technique it is impossible to stimulate every aspect of an application. As described earlier, hooks that are shielded by specific conditions, like the ones shown in figure 2.3, are impossible to reach with this random approach. Because of this we need more sophisticated methods to stimulate an application:

Rasthofer et al. in [13] describes their approach to adapt the surroundings or the environment of the application so that it reveals possibly hidden features. They built a framework called *FuzzDroid*, which tries to fuzz the app towards reaching a targeted state by intercepting different API calls and sending back fuzzed returns. This is done until the targeted location is reached. The different fuzzed returns that were sent back from the framework to the app, then build the environment. Another approach known from other software analysis fields is *symbolic execution* [16]–[18]. In this technique, real inputs are replaced by symbolic ones, which leads to an output consisting of a function of the input. The path towards a desired location is then made up of boolean constraints placed upon the symbolic inputs. In *Android* and other software environments this can be used for testing during development and detecting malicious behaviors within an application.

The last here mentioned approach for stimulation is described in [6]. This will also be the technique we will be using in our implementation, which is why this will be explained in greater detail in the next section. The authors developed a framework to force different parts of an application by altering the source code with the goal to specifically run this part of an app.

At this point we want to refer interested readers to the research done in [14], which explores and compares different techniques for this task.

2.2.1. GroddDroid

We first want to give a general overview of the framework before going into more detail in the consecutive chapters. Abraham et al. in [6] developed **GroddDroid** in *python 3* and *Java* as a behavior stimulation tool that combines static and dynamic analysis to force suspicious code segments in *Android* applications. The core functionality of this tool is to analyze a given application instance on whether it is malicious or not. By using this tool we argue that we can cover most malicious payloads with a hook of $type_1$ in a repackaged application and even some of $type_2$ hooks. This is due to the fact that *GroddDroid* tries to force all conditional statements hiding possible malicious code segments. With that it can, for instance, also counter the ability of malware not to run if it is executed on a virtual machine, as this must be done through a conditional statement.

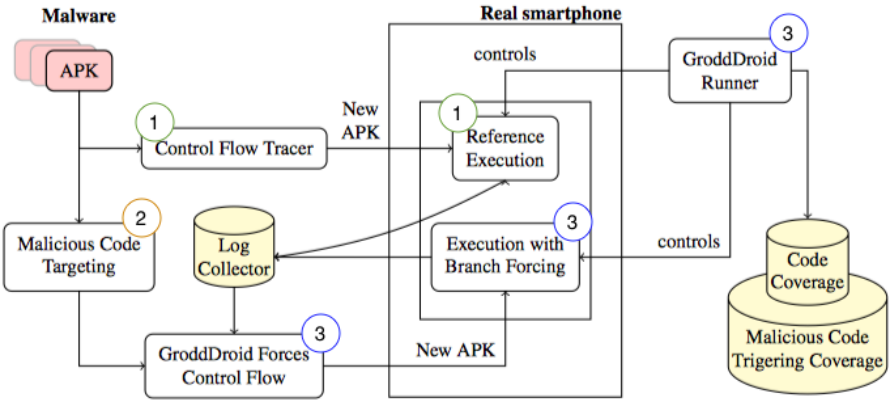

Figure 2.2.: GroddDroid framework overview [6]. Annotated with the information about what part is executed in which step of the GroddDroid execution.

As described in [6] and visualized in figure 2.2 the software runs through three steps during the execution. First, the given app is analyzed statically, which is referred to as the *reference execution*. In the step after that, the program tries to find every potentially malicious code segment within the source code. It is specifically looking for conditional statements, which are meant to hide the malicious intentions. This is also done through statical analysis. To identify which branches and methods are potentially malicious it compares the given code fragment against heuristics that are stored in a predefined database and then lists and ranks all suspicious segments of this application according to the risk scores, which are also stored in the database. In the last step, *GroddDroid* alters the source code of the application, so that the previously identified possibly malicious parts get triggered, resulting in changing the control flow of the application.

```
// first example
if(isEmulatorOn()){
    // execute not-malicious code
} else {
    // execute malicious code
}

//second example
targetYear = 2020;
if(currentYear < targetYear){
    // execute malicious code
} else {
    // execute not-malicious code
}
```

Figure 2.3.: Possible trigger points for GroddDroid [6]. The framework tries to evaluate the conditional statements, so that the malicious part will be executed.

Statements that *GroddDroid* tries to alter, typically have the form as seen in figure 2.3. The framework will try to evaluate the if-statements so that the malicious part will be executed [6]. To make changes to the source code, like altering the control flow and tagging each method and branch, *GroddDroid* is using the *Soot framework* [19]. This framework allows to make changes to the *Java* bytecode of the application using the *Jimple* representation of it. This altered application will be saved and then it will be run on a virtual device. The third step will be repeated several times, each time altering the control flow in a different way, to cover the next malicious segment in the list of possibly malicious code segments. Each run of an application gets monitored. The GUI stimulation for the dynamical analysis is either done by the *Android Monkey* Runner, *manual* or with the *GroddDroid* runner that gets shipped with the framework. The overall goal of this process is to get the highest code coverage rate possible.

Folder hierarchy

GroddDroid can be cloned or downloaded from `https://scm.gforge.inria.fr/anonscm/git/kharon/kharon.git`. It is made up of nine folders, holding the main functionality, which we will describe now [6].

- AcfgTools: In this folder all scripts and tools for building the *Application Control Flow Graph (ACFG)* are stored. First, all method control flow graphs are generated

and then merged into a complete control flow graph for the whole APK file. This folder also holds the functionality that analyzes the generated graphs and tries to find the nearest entry point for executing the desired target. All scripts in this folder are written in *python 3*.

- AndroidPlatforms: Here are all the different *Android* SDK versions, needed by the framework, are stored. They are downloaded during the installation.
- BranchExplorer: This is the main folder of the whole framework, where one can start the whole *GroddDroid* process from. It holds the different runners for the GUI stimulation, a general *config file*, handlers for the *Android* device, handlers for *apktool* (which is used to decompile the given APK file) [20], handlers for calling the *ForceCFI* program and other core functionalities. Again, every program in this folder is written in *python 3*.
- Demos: This directory simply contains two APK files that can be used to test *GroddDroid*.
- Documents: This folder holds all the documents for the different parts of the framework.
- ForceCFI: In this location the jar-file and the source code for the ForceCFI program are stored. This program is written in *Java* and utilizes *Soot* to force and tag each branch and method with a unique identifier. It can be given as input, whether to only tag the application or to also force specific paths. The result is a new, aligned, APK file.
- ManifestParser: The ManifestParser analyzes the *Android* Manifest file, which is present in every *Android* app. *GroddDroid* can retrieve various information from the Manifest file, such as the number of activities/services/broadcast receivers, the permissions the app is asking for and much more.
- MiniApp: A small custom application.
- SuspiciousHeuristics: This directory holds the suspicious.json file, which will be explained in the next section.

Heuristics

GroddDroid uses a database to decide whether a seen code segment could be malicious or not. The file *suspicious.json* holds this information and distinguishes between different categories. The authors of the framework included the following categories, each category combines different *Java* and *Android* classes that can be clustered together:

- *binary*: Execution of binaries
- *dynamic*: Dynamic code loading
- *crypto*: Cryptographic libraries
- *network*: Remote connections
- *telephony*: Telephony information
- *sms*: SMS

The framework also allows to execute a *grep*-command in the given classes. Every category has a *risk score* assigned to it. This score determines the harm-potential of the category. The authors of the framework used the research done in [21] to compute the risk scores. With this, *GroddDroid* assigns risk scores to every method that matches one of the categories in the suspicious.json file. After that it ranks all the possible malicious parts of the application according to their assigned risk score. The higher the risk score the higher the probability that the code fragment and with that the whole application is malicious. *GroddDroid* then works its way through the produced list, trying to force the part with the highest risk score first.

Inputs and outputs

GroddDroid has three inputs, two of them can be changed, as described in the following. Furthermore, for every run of every app *GroddDroid* has a number of outputs. We now want to take a closer look at those inputs and outputs, starting with the inputs:

One input is the application that we want to analyze. Other command line inputs are not covered here, because we will not be using them, but they can be seen on their website [22]. Besides the inputs on the command line, the user can also change settings in the following two files [6], [22]:

- config.ini: In this file we have to specify our directory for the *Android* SDK, the version of the *Android* tools we want to use with *GroddDroid*, the device and device code of the Android device on which we actually want to run the APK files. Further, *max_runs* determines how often *GroddDroid* should try to change and run the application. The runner type for the GUI stimulation can be set via *run_type* to either the *Android Monkey* runner, the *GroddDroid* runner or the *manual* runner. More inputs that can be adapted are the location of the *apktool* and other variables necessary for the framework to run.
- suspicious.json: As described before, this file holds the information on which code fragments should be investigated because of their potential risk.

Looking at the outputs, we have the following ones for each run:

- all_tags.log: A list of all methods and branches that *GroddDroid* has tagged.
- APK-directory: The new and altered APK file that got executed on this run is stored here.
- dot-directory: This directory holds the produced dot files, which represent the method control flow graphs.
- seen_tags.log: A list of all tags that *GroddDroid* has seen during this run.
- stats.json: This json-file holds general information about how much branches were targeted, how much of them were forced and the achieved code coverage.
- suspicious.log and targets.json: These two files basically hold the same information, viz. all the suspected methods and their risk score. The targets.json file is in the file format to best work with *GroddDroid*.
- to_force.log: In here all the branches that should be forced to execute are stored.
- xml.tmp: Temporary file, which does not have any relevance for our research.

For each application the framework also stores the output of the *apktool*, as well as other files, primarily used for analysis purposes.

2.3. Machine learning

Within the last years, machine learning became an often-used technique in the field of computer science and data science. This might also be the result of the mass amount of data everyone with a computer-like device produces every day, which is true for nearly everyone in the western world. For making inferences from a sample dataset to another dataset, the algorithms heavily rely on mathematical models [23]. In simple terms machine learning algorithms try to find common patterns among the data in the training set, so that they can make a prediction for a new, never before seen, dataset on the learnings of the training data set [23], [24]. Because of this ability, to learn and predict, it belongs to the field commonly known as artificial intelligence [23].

"Machine learning is programming computers to optimize a performance criterion using example data or past experience" [23]. Usually machine learning is divided up into two phases [23], [24]: The learning phase, which is referenced in the above quote with the past experience. In this phase, a lot of data is fed into the algorithm, to learn which features are common and which are the most characteristic. In the second

phase, it can make predictions, based on everything it has learned in the past, for a new instance that it has never before seen.

Machine learning can be divided into two main categories, namely: *Unsupervised learning and supervised learning* [23], [24]. **Supervised Learning** can be described as follows: Learn the mapping of an input X to an output Y from the training data set and apply it to the testing data set. Typically this means that one must manually or automatically label all instances within the training data set, so that the algorithm can learn the associations between different features and the corresponding label. Once learned, it can be applied to a never before seen instance, which the algorithms, then try to label into the fitting category. This is the category of machine learning that we will use in this research.

Unsupervised Learning. Other than supervised learning the data does not get labeled in unsupervised learning. The algorithm itself tries to find common patterns among the data instances. A typical application of unsupervised learning is *clustering*. Here the algorithm tries to find common patterns in a subset $\tilde{X} \subseteq X$ of the whole input data X. Once found, it clusters these data instances together, as they share a stronger connection in-between each other, than with the other data instances.

Next, we want to describe one typical use case for supervised learning, which we will also be using in our implementation:

Classification. We will give an example, fitting the topic of this paper: When looking at *Android* applications we can label them in two categories, *benign* and *malicious*. These labels represent two classes in which we can classify all other applications. In the case of classification, we now want to define a rule, or a set of rules, to state in which of the classes a newly seen application belongs to, depending on different attributes, configurations or features. The basic concept behind classification is *pattern recognition* [23], [24].

To perform the learning and analysis the algorithms need the data in a well-structured way. Usually, they take a feature vector X, which describes the characteristics of the instance to be analyzed. In supervised learning, the algorithms also take a vector y for the labels. We now want to proceed with the description of what features are and how we can retrieve them.

2.3.1. Features

Features are, as described in [24], a form of representing data. They transform a real-world instance i into a vector which is easily comprehensible for the computer and machine learning algorithms. The author continues that most algorithms take a complete vector of features as input for one data instance. In such a vector a feature typically is a numerical number describing one aspect or characteristic of that dataset.

This concludes that the representation of one data instance as features is only as good as the individual features. Therefore, the goal is to use features that describe the data as good as possible. This problem is known as *feature selection* [24], [25]. The relevance of one feature can be exactly defined and can be studied in [25]. A very simple example of features for a car, for instance, would be: the number of seats, maximum speed, acceleration time to 100 kilometres per hour, the day it was bought and so on.

We will explain the exact features we used for our experiments in a later section of this paper. Though, we now want to explain two main categories when it comes to features for a software application [26]. These are **dynamic** and **static** features. Static features are obtained by looking at the source code and extracting different characteristics, without running the application. Examples for this in the *Android* environment are the number of activities, the number of permissions, checking if certain permissions are required and much more. In contrast to static features, dynamic features are obtained during the actual runtime of a software application, for instance out of the method traces. Examples are the number of method calls, the number of API calls and others. Further, **hybrid** features combine the features from both categories described above.

2.3.2. Support Vector Machines

Support Vector Machines, or short SVMs, represent one type of machine learning algorithms and can be summarized in the category of kernel machines [23]. In [23], [24], [27] the functionality of Support Vector machines is defined as follows: These machines use the learning system of linear functions in a high dimensional feature space. The SVM paradigm is to search for large margin separators in the data samples. The form and curve of the hyperplane are defined by the type of kernel that is used. Typical kernels are the linear kernel, the polynomial kernel and the Gaussian kernel. We will be using the standard linear kernel.

2.3.3. Decision Trees and Random Forests

In accordance with [23], [24], a *Decision Tree* is hierarchical data structure, represented through a root node, inner nodes and leaf nodes, which makes them easy to understand for the human mind. In this thesis, we will be using trees for classification and therefore supervised learning. However, they can also be used for other prediction problems, such as regression. One tree is composed of several internal decision nodes, each one splitting the input space. Internal decision nodes implement a function $f(x)$ that determines in which branch the test data should proceed. This process is repeated recursively until the test data reaches a leaf node, which then constitutes the label for this test data.

Decision Trees often fall victim to *overfitting*, while *Random Forests* (TREES) can be used to mitigate this problem [24]. A Random Forest is made up of an ensemble of multiple decision trees. On each of these trees an algorithm A is applied onto the dataset and an additional random vector. The outcome is produced by a majority vote over the results of all individual trees.

2.3.4. k-Nearest Neighbors

The *k-Nearest Neighbor* classifier (KNN) is one of the most rudimentary machine learning algorithms existing [23], [24]. This algorithm assumes that the features that were given as input represent the corresponding labels with reasonable relevancy. It then labels a new data instance with the same label that the closest neighbor has, in terms of similarity in features. The distance between two data samples is calculated to find the next neighbor and therefore the most similar data instance already seen. k represents the number of neighbors that should be considered when searching for the nearest neighbor and their corresponding label. $k < N$, with N the number of data instances, has to apply.

2.3.5. Performance measures

In order to compare the results of the machine learning algorithms, we introduce five performance measures with values between 0 and 1, in accordance with [23], [28]. These measurements are all based on the following six primitives:

- P $\hat{=}$ positive $\hat{=}$ the number of malicious instances
- N $\hat{=}$ negative $\hat{=}$ the number of not-malicious instances
- TP $\hat{=}$ true positive $\hat{=}$ the number of malicious labeled instances that are malicious
- FN $\hat{=}$ false negative $\hat{=}$ the number of not-malicious labeled instances that are malicious
- TN $\hat{=}$ true negative $\hat{=}$ the number of not-malicious labeled instances that are not-malicious
- FP $\hat{=}$ false positive $\hat{=}$ the number of malicious labeled instances that are not-malicious

Accuracy: The *accuracy* score is calculated as follows:

$$\frac{TP + TN}{P + N}$$

This metric describes the percentage of instances that have been classified correctly, hence it is the percentage that a new instance gets classified by the correct label.

Recall: The calculation for the recall metric, or the *true positive* rate is:

$$\frac{TP}{P}$$

In other words, this is the percentage of a new malware getting classified as malware.

Specificity: This metric can be seen as the counterpart of the recall metric, as it is also known as the *true negative* rate and is calculated as follows:

$$\frac{TN}{N}$$

In other words, this is the percentage of a new goodware instance getting classified as goodware.

Precision: This metric is calculated as follows:

$$\frac{TP}{TP + FP}$$

It represents the percentage of true positives regarding all as positive classified instances. In other words, the probability that an instance classified as malware is really malware.

F1-Score: The F1-Score combines the previously explained metrics, *recall* and *precision*. It is computed as follows:

$$2 * \frac{precision * recall}{precision + recall}$$

It can be interpreted as a weighted average between recall and precision.

A single measurement from above on its own can be misleading, as each has their advantage and disadvantage. Whereas, in combination they can represent the results from the machine learning accurately.

2.3.6. Cross-Validation and *k*-Fold Cross-Validation

As described previously, machine learning inputs are divided up into a learning dataset and testing dataset. But if the dataset is too small the data cannot be divided without the loss of valuable information [23]. *Cross-Validation* can solve this problem. The author Ethem Alpaydin further describes the functionality of this method: Simply said, the given dataset gets divided into a number of smaller training and validation datasets. The goal is to have no overlappings between these datasets, while leaving the datasets as big as possible to keep the error rate as low as possible.

A special form of this method is *k-Fold Cross-Validation*, as explained in [23], [24]. In this case k is a variable that can be set to one's own preference. In this paper, we will

set $k = 10$. Here, the whole dataset is divided into k equal sized chunks. In each of these, one part will be chosen randomly to be the validation set, the other $k - 1$ parts will form the learning phase. This process gets repeated k times, each time leaving out another part of the set for the validation. After the k rounds each part was used as the validation set once. This method, is widely used, but of course also has its disadvantages, which can be read in [23].

3. Methodology

We present our proposed solution, the assumption on which it is based and why we believe it can successfully tackle the problem at hand.

3.1. State-of-the-art

In figure 3.1 we can see the structure that most current approaches are following [11]. Examples are papers like [1], [9], [29], [30] and others, which utilize the main idea of this approach. Most of the time it comprises of a stimulation part, where the application will be either statically or dynamically analyzed. From this, features will be extracted, which can then be used as input for the detection. The detection part then computes and outputs the results. In the literature review chapter we already listed research deploying this structure.

Figure 3.1.: Current approach for finding malware: First, an application will be stimulated once. Then, the extracted features are analyzed.

In our opinion this approach has a major flaw. Since there is no connection back from the detection part to the stimulation part, the stimulation cannot adapt and therefore cannot improve. However, this can be very relevant, for instance, when the detection part has concrete findings on how the stimulation could be improved. For this reason, we want to present our approach in the next section that aims to counter this.

3.2. Methodology and architecture

As seen in figure 1.1, in the introduction chapter, we extended the approach from figure 3.1 with a connection back from the machine learning to the stimulation part. By doing this, the stimulation can be responsive, depending on the outcomes of the machine learning. The next step is to define how this connection will look like and what data will be transferred back. Since we are using *GroddDroid*, we force different parts of the given application and with that try to enhance the corresponding machine learning output. The final structure for our solution is shown in figure 3.2.

input : Benign and malicious apk instances
output: Prediction and metrics of the machine learning algorithm

```
1  for apk ∈ not analyzed apks do
2  |  outputFiles = RunGroddDroid(apk);
3  |  features = ExtractFeatures(outputFiles);
4  end
5  prediction, metrics = AnalyzeFeaturesUsingMachineLearning(features);
6  while metrics[f1score] < metrics[previousf1score] do
7  |  for apk ∈ metrics[misclassified apks] do
8  |  |  outputFiles = RunGroddDroid(apk);
9  |  |  features = ExtractFeatures(outputFiles);
10 |  end
11 |  prediction, metrics = AnalyzeFeaturesUsingMachineLearning(features);
12 end
```

Algorithm 1: Overall structure of the proposed framework, outlining the main feedback loop.

The overall structure of the implementation of the framework is shown in algorithm 1 and will be explained in the following. The first thing we do, is to run every application once, without using the forcing part of *GroddDroid*. After that we analyze the results of the machine learning algorithms and then run the misclassified applications again with the forcing aspect of *GroddDroid* enabled. This second part lies within a for-loop, as it gets iterated over as long as we get better results from the machine learning. Once the results from the current run of the machine learning are worse than the ones from the previous run, we stop this loop and exit the program. In each iteration we force a different branch of the application. With this approach, the only thing we need are the APK files from the *Android* apps. *GroddDroid* uses *apktool* to decompile the files and receive the *Dalvik bytecode* in *Jimple* representation. As *GroddDroid* already does that for us, we can utilize these files as well to get access to the *Android* Manifest file for

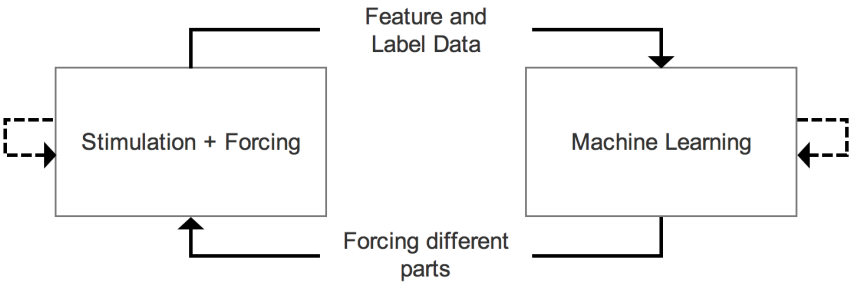

Figure 3.2.: Overall structure of our solution: We first stimulate an application, without forcing. After that, the extracted features are channeled into various machine learning algorithms. Depending on the F1-Score from the dynamic features we decide, whether to do another iteration or not. In each following iteration, we force a different part of the application.

instance.

Static features can be a good indicator for detecting malware. However, they cannot cover the whole application. Dynamic code loading, encryption or obfuscation are just some examples that static analysis cannot cover. Therefore, we need to be able to enhance the dynamic features, which can capture more precise characteristics of the application, in contrast to the static features. With our approach we argue that we can increase the branch coverage and therefore the code coverage of the application, which leads to better dynamic features, concluding in better results for detecting *Android* malware.

3.3. Stimulation

The stimulation part of our design is handled by the *GroddDroid* framework. It consists of two main components as outlined in the following:

Firstly, we use *GroddDroid* to analyze and force each application. We use the tool to force the different parts of each application. In each run of our feedback loop we use *GroddDroid* to force the next code segment according to the risk score list that the framework computed. This ranking of the code segments is highly dependent on the heuristics predefined in the database, therefore we decided to evaluate the impact of these with different experiments. Secondly, for the GUI stimulation, we can choose between three types of runners. These are the original *Google Monkey* runner and the *GroddDroid* runner itself, which came in two different versions, version one and two, and the *manual* input of clicks. The basic idea of the *GroddDroid* runner is to try every

possible combination of click-able elements on the APKs GUI [6]. Version two is a more enhanced variant of the first version. Unfortunately, we could not determine what exactly changed from version one to version two. Because of this and better results with the first version of the *GroddDroid* runner, we are only using this version. The *Google Monkey* runner produces pseudo random user inputs to stimulate the application [15]. In each execution of an application we perform 750 inputs with the Monkey runner.

3.4. Detection

In the detection part we are using different machine learning algorithms with k-Fold Cross-Validation with $k = 10$. The following twelve variations are used:

- k-Nearest Neighbor with $k = 10, 15, 20, 25, 30, 35$.
- Random Forests with estimators (number of trees to be built) $e = 10, 25, 50, 75, 100$.
- Support Vector Machine with a linear kernel.

Further, we build an ensemble of all the results from the above mentioned algorithms using a majority vote. With this we try to get the average result from all of the above algorithms. In each iteration we perform three calls to the machine learning algorithms. One time only with the static features, one time only with the dynamic and lastly with all features that we extract. Thus, we can compare our results between all the feature sets and evaluate our results. We are using the F1-Score metric from our ensemble, generated by the dynamic feature set to determine whether to do another iteration or not in our feedback loop.

3.5. Feature selection

This section describes the information that is communicated from the stimulation part to the detection part. Due to the fact that we are mainly extracting numerical features, all our results depend on the quality of them. In the following we want to give a compressed listing of the features we extract from each application. The exact features used are listed in the appendix.

- **Features from the traces file:** These features are obtained from the method traces file and are representing our dynamic features.
- **Features from the targets file:** The targets.json file from the *GroddDroid* outputs, is used to extract the total count of alerts from *GroddDroid*, along with the combined

risk score of these alerts. In the targets.json file the methods that are possibly malicious and therefore will be targeted in the next runs, are called alerts.

- **Features from the to_force file:** In this file *GroddDroid* stores the next segments to be forced. In our feature vector, we simply store the count of them.
- **Features from the stats file:** The stats file is produced after *GroddDroid* has finished the execution. We take different metrics from it, like the total methods seen in the different runs or the overall method and branch coverage.
- **Features from the Android Manifest file:** As commonly done, we use the *Android* Manifest file to extract static features from each application. The total number of activities, services, permissions and more specific queries for certain permissions are some examples of the features extracted here.

As seen above, we grouped the features into four categories, depending on which source they originated from, thereby mostly which *GroddDroid* output file. In total, we extract 77 features from each individual app.

3.6. Feedback to GroddDroid

This section describes the information that is communicated from the detection part to the stimulation part. In each iteration, if we decide to do another run, we instruct *GroddDroid* to force the code segment with the next highest score in its ranking of possibly malicious code segments. The goal of this is to achieve a higher code coverage and disclose the most malicious part of the application, so that classification from the machine learning can be done more accurately. The application should have revealed all its possibly malicious functionality once we forced every entry in the list.

4. Implementation

As part of this research we implemented a *python 3* framework, called **Phenax**. We got the inspiration for the name from the card game *Magic*. This open source framework is available at `https://github.com/JonaNeu/Phenax`. Essentially, it combines a set of tools in one interdependent software. As presented in section 1.2 this framework is mainly made up of two parts, which is also present in the implementation. We now want to give a more detailed look into, firstly, the individual tools themselves and, secondly, the Phenax framework which combines all these tools.

4.1. Development environment

Before we get into the specific implementation we briefly want to describe the environment on which this framework was developed and tested on. The implementation as well as the first smaller little tests have taken place on a laptop with the following specifications:

- OS: MacOS 10.12
- Processing Power: 4 Cores Intel I with 2,9 GHz (experiments were run on 2 cores)
- Memory: 8GB of RAM
- Storage: 500GB SSD

4.2. Used tools

In the next part we will briefly describe the tools on which our Phenax framework depends on. For each tool we will first give a general explanation, possible adaptations we have done and what role each one plays in our overall structure.

4.2.1. GroddDroid adaptations

As described before, *GroddDroid* makes up a huge part of our implementation. Unfortunately, we could not use the open source tool out of the box but had to make some

changes to the implementation. This took a large part of the overall implementation phase. We changed and added a number of features, which will be outlined in the following paragraphs.

The first thing we wanted to do was to get the method traces from the runtime of an app, which are used to extract all the dynamic features. We wanted to use the same approach *GroddDroid* already uses for other functionalities. We changed the *Soot* implementation of *GroddDroid* and added the code seen in figure 4.1, putting in a log message with a specific key as handle and with the invoke statement as content, before every invoke statement in the source code. Later, we searched for that specific handle in the *logcat* output, filtered and stored the values in a file called *traces.log*. *GroddDroid* already uses a similar method to assign every method and branch a unique handle (*BEGINXX, BRANCHXX*, where XX is an incrementally increased number).

Further, we added the ability to run an application once, without any forcing whatsoever. Called the *reference run*, *GroddDroid* before only did that statically, which means the app was not actually run on a device. We added the ability to specifically only running this reference run and also really executing the application on a virtual device.

Another thing we did to increase the efficiency of our application was to store the complete *python* object for the application control flow graph (ACFG) in a file, using the *python pickle* module. With this we could load that exact same object again to a later point in time. To use this, we added an optional input parameter that can be given to GroddDroid with a path to the stored file. We added that feature after the first experimental runs. These runs made clear that the merging of the method control flow graphs (CFGs) to the ACFG was the most time-consuming part of *GroddDroid*. With this feature we are able to only do this merging for every app once, instead of doing it over and over again, in every iteration of the feedback loop.

Additionally, we also extended the heuristics database 'suspicious.json' with the following categories: *Accounts*, *GPS*, *NFC*, *OS*, *IO*, *Recording*, as well as adding more classes to the existing categories with a risk score we found to be fitting. For a few of our experiments we also changed all the risk scores to 10. The whole heuristics file can be found in the appendix.

The last thing we added, was a restriction of the runtime of each app, to prevent a GUI rich app from running too long. For that, we added code by using the *python signal* module in the files for each specific runner under *BranchExplorer/branchexp/runners* in the *GroddDroid* files. For our experiments we set the runtime restriction to five minutes before ending the execution.

```
private void addSignatureTags(Unit unit, final Body body){
   unit.apply(new AbstractStmtSwitch() {
       public void caseInvokeStmt(InvokeStmt stmt) {
           InvokeExpr invokeExpr = stmt.getInvokeExpr();

           // Adding the log to the source code of the app
           SootMethod logMethod = Scene.v().getMethod(logSig);
           Value logTagConst = StringConstant.v(logTagSig);
           Value logMessageConst = StringConstant.v(invokeExpr.toString());

           StaticInvokeExpr logExpression = Jimple.v().newStaticInvokeExpr(
               logMethod.makeRef(), logTagConst, logMessageConst);
           Unit logStmt = Jimple.v().newInvokeStmt(logExpression);
           body.getUnits().insertBefore(logStmt, unit);

           System.out.println("Trace␣Log␣Message:␣" + logStmt);
       }
   });
   body.validate();
}
```

Figure 4.1.: Adaption to GroddDroids Soot implementation to mark all invoke statements in an application to later extract them for the method traces.

4.2.2. SciKit-Learn

According to Pedregosa et al. [31], SciKit-Learn is one of the most established frameworks, when it comes to machine learning in the programming language of *python*. This is mainly due to the framework's ease of use, while at the same time maintaining a high standard in code quality. Furthermore, in SciKit-Learn almost every modern machine learning algorithm is implemented.

In our implementation, we heavily depend on SciKit-Learn for all of the machine learning parts. Namely, we use SVMs, Decision Trees and k-Nearest Neighbor from this framework. We also used it to calculate all metrics, except *specificity*, as it is not supported by SciKit-Learn. We implemented that ourselves.

4.2.3. NumPy

One of the few dependencies of SciKit-Learn is NumPy [31]. It stands for Numerical Python and extends the *python* programming language with modules for scientific programming, as well as handling arrays and matrices in a very intuitive and easy way [32].

We are only scratching the possibilities of this powerful module, as we exclusively used NumPy to store the feature vectors, which serve as the inputs for the machine learning.

4.2.4. Genymotion and VirtualBox

Virtual machines were used to run all applications on the actual *Android* environment in our experiments. For that we used *Genymotion* and *VirtualBox*. VirtualBox is a commonly known virtualization software [33]. Genymotion builds upon VirtualBox. Developed by GenyMobile it especially concentrates on emulating Android devices for high speed and performance [34]. For all our experiments we used a *Google Nexus 5* virtual machine with *Android* version 4.4.4.

4.3. Phenax

Now that we have described all the tools, we want to show how the Phenax framework combines them all together.

The framework is based on the basic structure seen in algorithm 1.

4.3.1. Inputs and outputs

In this section we want to explain the inputs and outputs of our framework. Starting with the inputs we have the following list of mandatory and non-mandatory inputs:

- $-explorer\_dir$ (mandatory): The path to the subdirectory 'BranchExplorer' in the *GroddDroid* files.
- $-inputDir$ (mandatory): The path to the directory with the input APKs. It should contain two directories, 'malware' and 'goodware', with the individual APKs.
- $-outputDir$ (mandatory): The path to the directory for the output.
- $-VMname$ (non-mandatory): The name or ID of the VirtualBox virtual machine to use. A default is given.

- $-restore\_snapshot$ (not mandatory): The name or ID of the VirtualBox snapshot to restore the virtual machine. A default is given.
- $-kf$ (non-mandatory): Set to use k-Fold Cross-Validation or not. If yes, set the value of k. Default $k = 2$.
- $-selectBest$ (non-mandatory): Set to use feature selection or not. If yes, set the number of features that should remain (needs to be <= 30). Default 0 (no selection).

In the given output directory, the framework creates the following files and directories:

- the file 'results.txt', which stores all information about all results from our different machine learning algorithms for each round and for each feature type (hybrid, dynamic and static).
- the file 'ml_input.txt', which stores the input vectors for all APK files for each round.
- the files 'X_database.txt' and 'y_database.txt'. These files were created via the *python* module pickle to store the last feature vector X and the label vector y for the machine learning.
- a directory for every inspected APK file. In this directory two main folders are located: 'reference' and 'improvement'. The reference directory stores the *GroddDroid* output for the first run, in which nothing gets forced in the application. The improvement folder holds all other output files from *GroddDroid* which are produced in every iteration after the first run with forcing enabled.

4.3.2. Feature extraction handler

We created a class called *Extractor*, which handles all the feature extraction actions. In detail, it extracts all the features from the traces.log file, the target.json file, to_force.log file, the stats.json file and the Manifest file of the application. For the extraction of the features from the *Android* Manifest file we utilized the ManifestParser from the *GroddDroid* implementation.

4.3.3. Config file handler

The class *ConfigHandler* establishes an easy way to interact with the config.ini file from *GroddDroid*. It is used to set the *GroddDroid* output path, the device *GroddDroid* should use and most importantly to increase the number of runs for each app in every cycle of the feedback loop to force different parts of the application.

4.3.4. Machine learning

The machine learning aspect of our framework makes up an important part of the overall implementation. We used some implementations from our advisor. For this we created the file *Learn.py*, which holds the methods to actually make the call to the machine learning library SciKit-learn. Also present in this file, is a method to get the metrics from the machine learning results as well as a method to get the indices for misclassified apps. Currently implemented are methods for calls to machine learning algorithms using Support Vector Machines, Random Forests and *k*-Nearest Neighbors. Each of those methods comes in two variants: One only with the training phase and without a testing phase and the other one with a subdivision in training and testing. We only used the variant with the training phase without the testing phase. Furthermore, in each method call to the algorithms one can define if and with what number feature selection and *k*-Fold Cross-Validation should be used. Using the results of all twelve machine learning results, we built an ensemble of all of them using the majority vote by creating a new *predicted* array, which is then used to calculate all metrics again.

4.3.5. Main file

In this file we combine everything together. From here every, before explained, action is triggered. The complete source code for this file can be found in the appendix of this thesis. We now want to give more details about the methods and functionality of this by explaining the most important functions.

Main function. This function implements the actual feedback cycle. In here we retrieve the APK files from the input directory and create a python dictionary for them to hold the most important information. Namely, the path to the original APK file, whether the app is labeled malicious or not, the feature vector for this application and how many feedback loops it has already gone through. Then all the applications get executed once, using *GroddDroid* without forcing anything. The machine learning part is called three times: Once to analyze the hybrid features, once to analyze the static features and finally once to analyze the dynamic features. After that the results of the machine learning get analyzed and the actual feedback loop starts, to stimulate and analyze the misclassified apps again, this time with *GroddDroid* forcing enabled. This loop continues until the *F1-Score* of the ensemble machine learning algorithm produced by the dynamic features from the current run is smaller than the *F1-Score* of the previous one. After each iteration of our feedback loop we increase the *max_runs* variable of *GroddDroid*, so that next time the following most suspicious target gets forced.

Get paths to output files. With this function we retrieve the paths to the *GroddDroid*

output files, so that in the next steps we can extract the features from these files, using our *Extractor* instance.

Build the machine learning inputs. Here, we combine the individual feature vectors for each app into one large feature vector for the hybrid, dynamic and static features. We can then feed these into the machine learning algorithms. The label vector also gets build within this function, built up from the individual labels similar to the feature vector.

Doing the machine learning. In this function we call the machine learning functions located in the *Learn.py* file. It furthermore calls the method to retrieve the metrics and the indices of the misclassified applications. Currently, we are calling three different machine learning algorithms a total of twelve times with the specifications seen in the methodology chapter. From the results of all these algorithms we built the average metrics using the principle of the majority vote. These are also the metrics that get returned and which are used to decide whether to do another loop in our feedback cycle or not. The results are printed on the console as well as stored in the result.txt file.

5. Evaluation

The goal of this chapter is to describe the conducted experiments and the results drawn from them. The main question to answer was, how well the framework with the implemented feedback loop did perform and whether the continuous forcing of different parts of the misclassified applications through *GroddDroid* had a positive effect on our classification accuracies. But first, to give complete transparency and reproducibility we want to introduce the testing environment on which all tests were conducted.

5.1. Experimentation Environment

After the development and testing on a local machine, the actual evaluation with our program took place on a remote server with the following specifications:

- OS: Ubuntu 16.04 LTS (64-bit), with kernel version 4.4.0-79-generic
- Processing Power: 8 Intel Xeon(R) CPU E5-1630 v3 @ 3.70GHz (experiments were run on 4 cores)
- Memory: 62,8 GB of RAM and 63,9 GB Swap memory
- Storage: 1,9 TB HDD

5.2. Experiments

Once we finished the implementation of our solution, discussed in chapter four, we put the source code on a server, with the above seen specifications. This allowed us to run experiments over longer periods of time and with more efficiency. We ran the very first experiments a lower number of apps, to see how long the tests take and how stable our implementation was. Therefore, we firstly ran only a total number of 20 apps through our framework. Once that passed and finished within a reasonable amount of time, we started the experiments outlined in the next subsection using a total number of 100 applications.

In the following we want to show which tests we ran using our framework, the results of these tests and how we interpret and compare them.

For our analysis of each experiment we wanted to answer the following research questions (RQs):

1. How long did each experiment take?
2. How many applications did *GroddDroid* crash during execution?
3. On average, how many applications got misclassified by the machine learning algorithm per iteration?
4. How many iterations of the feedback loop did the experiment go through? What was the average for all experiments in order to achieve the maximum possible classification accuracies?
5. Did the type of features extracted affect the classification accuracy?
6. How did the different stimulation runners affect the outcome?
7. Did altering *GroddDroid* risk weights influence the classification accuracies?

For clarification, our definition for a crashed application was as follows: An app *crashed* when we were not able to retrieve any method traces from the runtime. Therefore, an application that got successfully installed on the device and ran, but got the typical *Android* error message 'Unfortunately, the application has stopped' did not count as a crash. The apps that did crash, in our definition, were not even installed on the device. This was mainly due to a bug in the *Soot* implementation of *GroddDroid*, where no aligned APK file was produced.

5.2.1. Data sources

The APK files to input into our application, were all received from our advisor. The benign instances were downloaded from the *Google Play Store* using an automated crawler script. One source for malicious applications was the *MalGenome* dataset [4]. It was published in 2012 and holds 1260 malware instances spanning all major *Android* malware families gathered from a variety of different *Android* markets. The second dataset we used, contains both the repackaged malicious applications as well as their original counterparts [8]. The authors of the *Piggybacked* dataset took the *AndroZoo* project [10] and different research datasets, such as the before mentioned *MalGenome* dataset as a basis. With this enormous input, they built a process to identify the

applications that got repackaged by another author than the original developer and doing a similarity analysis. With that they built pairs $< app_{original}, app_{repackaged} >$.

These datasets, cover almost every malware family existing in the *Android* space within the last years, back to 2010, as well as apps from several distribution channels, building a firm foundation.

For each experiments we used 50 goodware applications and 50 malware applications, leaving us with a total of 100 applications to be analyzed in each experiment. We chose the 100 applications randomly from each dataset. The same randomly-sampled applications were used for all experiments.

5.2.2. Variables

In order to evaluate our approach and answer the research questions, we vary a number of parameters/variables. We wanted to test out the *Monkey* runner in comparison to the *GroddDroid* runner as the stimulation engine, as well as the impact of the heuristics database from *GroddDroid* on the results and especially the number of iterations our framework went through. For the heuristics database, we used two variations. One leaving the risk scores as they were published by the *GroddDroid* authors. The categories we extended got a risk score assigned we found most fitting. In the other variation we set the risk score of every category within the database to 10 to be uniform. With this every possible malicious code snippet should be assigned the same risk score, so that we can test how heavily the risk scores influenced our results. Combining these options together we get a total of four experiments:

1. GroddDroid runner with original risk scores
2. GroddDroid runner with uniform risk scores
3. Monkey runner with original risk scores
4. Monkey runner with uniform risk scores

We decided to compare dynamic, static and hybrid features in each iteration and not to change these variables. However, the scores we consider in iterating/terminating the experiment are based on the score generated from using the dynamic features and the *F1-Score* of the ensemble machine learning algorithm.

We divided our experiments according to the datasets. In each dataset we conducted the aforementioned four experiments, leaving us with a total number of eight experiments for two datasets. Every time the machine learning was called we calculated the results for all twelve classification algorithms that we implemented. The decision, whether to make another loop or to stop the program, was made upon the *F1-Score* of

the dynamic features of the ensemble classification in each run. We chose this, because our aim with *GroddDroid* is to enhance these results from the dynamic features by targeting the suspicious code segments during the runtime. We will focus on the most important data points in the following sections.

5.2.3. Piggybacked dataset experiments

For the first four experiments we used 50 benign and 50 malicious applications from the *Piggybacked* dataset.

Experiment 1: GroddDroid runner with original risk scores

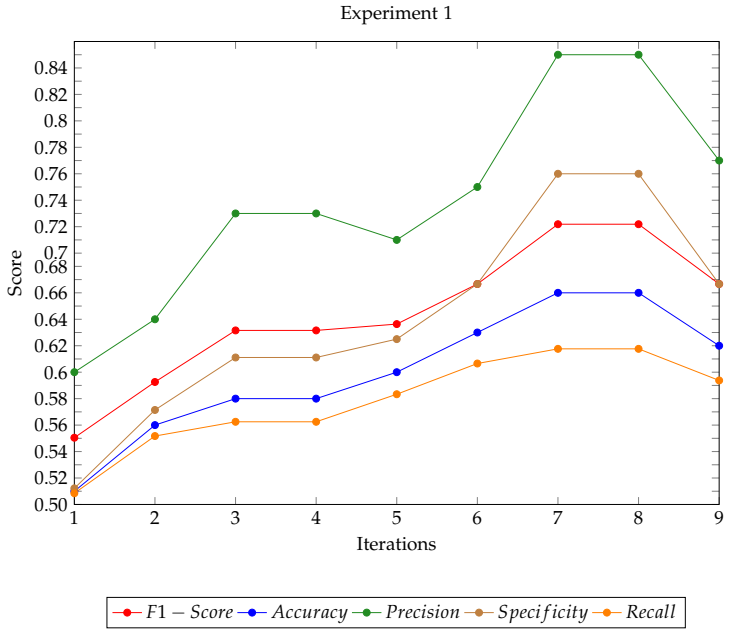

Figure 5.1.: Experiment 1 (GroddDroid runner with original risk scores): Evolution of the dynamic feature scores for each iteration from the ensemble classifier.

The first experiment took about 45 hours and gone through nine iterations in our proposed feedback loop. A total of 28 applications crashed during the runtime of our framework and an average of 44 applications were misclassified during each run. Looking at figure 5.1, we can see that all metrics for the dynamic feature set went up until the seventh iteration, the following iteration then had nearly the same results, before dropping to a lower level. This leaves the seventh and eighth iteration as our

Feature Set	Metrics	Best Iteration
Dynamic Features	F1-Score	0.71186440678
	Accuracy	0.66
	Precision	0.84
	Specificity	0.75
	Recall	0.617647058824
Static Features	F1-Score	0.767676767677
	Accuracy	0.77
	Precision	0.76
	Specificity	0.7647058823529411
	Recall	0.775510204082
Hybrid Features	F1-Score	0.673913043478
	Accuracy	0.7
	Precision	0.62
	Specificity	0.6724137931034483
	Recall	0.738095238095

Table 5.1.: Experiment 1 (GroddDroid runner with original risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features.

best results. It is also noteworthy that the *precision* score is significantly higher than all other metrics. Table 5.1 shows the precise results for the eighth iteration. The score of the first, last and best iteration for each individual algorithm can be seen in table 5.2. The results using the dynamic feature set were all higher in an advanced iteration compared to the first iteration without forcing. Except for the KNN20 and KNN25 classifier, the scores using the dynamic feature set are below the scores of the static feature set. Furthermore, we can observe that on the one hand the KNN classifiers had better results using the dynamic features than the hybrid features. On the other hand, the Random Forest classifiers and the SVM performed better on the hybrid feature set than on the dynamic one. Overall, the static features achieved the best results with only a few exceptions like the TREES75 classifier with the hybrid feature set most of the time.

Experiment 2: GroddDroid runner with uniform risk scores

The second experiment was the exact same as the first one with the only exception that all the scores in the heuristics database were set to 10. The experiment only went through five iterations in the feedback loop which results in the significantly lower runtime of about 37 hours. Nearly identical to the first experiment 44 applications got misclassified on average during each run and 26 application crashed completely. The scores in figure 5.2 show that the best iteration according to the *F1-Score* of the ensemble classifier using the dynamic feature set was the fourth. The *precision* metric especially sticks out in this chart, as it achieved a visibly better result before dropping to the lowest score compared to all metrics. Another point worth mentioning is that

Algorithm	Dynamic features	Static features	Hybrid features
KNN10	0.6387 | 0.6842 (8) | 0.7059	0.7033	0.3951 | 0.5714 (4) | 0.4884
KNN15	0.6400 | 0.7107 (9) | 0.7107	0.7174	0.4828 | 0.5625 (2) | 0.4889
KNN20	0.6240 | 0.7200 (9) | 0.7200	0.7174	0.4096 | 0.5112 (2) | 0.3846
KNN25	0.5763 | 0.7200 (7) | 0.7188	0.7158	0.4235 | 0.5000 (4) | 0.4445
KNN30	0.4948 | 0.6379 (4) | 0.6071	0.7312	0.4324 | 0.4675 (5) | 0.4675
KNN35	0.4884 | 0.5454 (8) | 0.4954	0.7312	0.4210 | 0.4675 (7) | 0.4675
TREES10	0.5270 | 0.7241 (8) | 0.5714	0.7755	0.7400 | 0.7446 (9) | 0.7446
TREES25	0.5545 | 0.6981 (4) | 0.6606	0.7879	0.6804 | 0.7676 (2) | 0.7327
TREES50	0.5556 | 0.7080 (7) | 0.5962	0.7921	0.7400 | 0.7800 (4) | 0.7755
TREES75	0.5263 | 0.7017 (7) | 0.6226	0.7879	0.6735 | 0.8080 (8) | 0.7647
TREES100	0.5400 | 0.7017 (8) | 0.6078	0.7879	0.7143 | 0.7755 (8) | 0.7423
SVM	0.4390 | 0.5474 (7) | 0.5169	0.8542	0.7238 | 0.7238 (1) | 0.6597
ALL	0.5505 | 0.7118 (7) | 0.6667	0.7677	0.6237 | 0.6875 (4) | 0.6374

Table 5.2.: First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 1 (GroddDroid runner with original risk scores): Best overall results of the F1-Score for each individual algorithm.

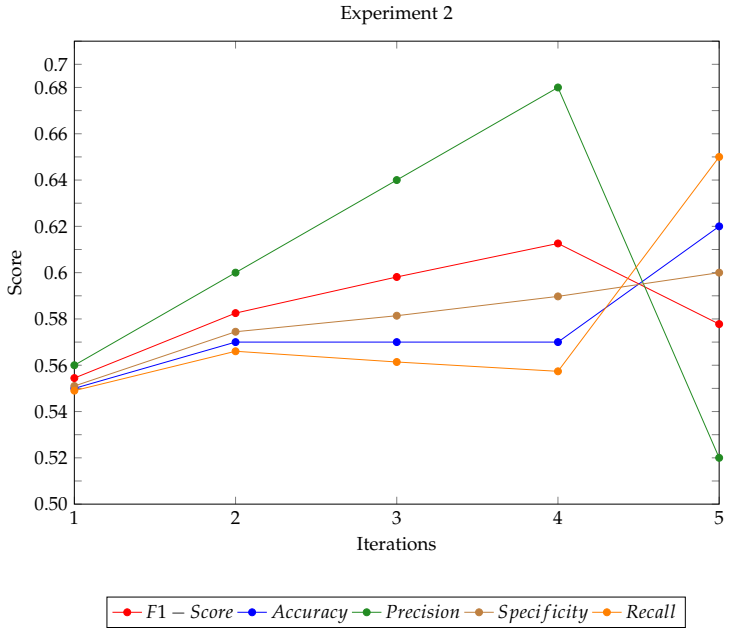

Figure 5.2.: Experiment 2 (GroddDroid runner with uniform risk scores): Evolution of the dynamic feature scores for each iteration from the ensemble classifier.

the *recall*, *accuracy* and *specificity* metric continued to grow in the last iteration. This might indicate a local maximum and that further iterations could have achieved better results. The detailed results in table 5.4 show that the scores using the static feature set

Feature Set	Metrics	Best Iteration
Dynamic Features	F1-Score	0.612612612613
	Accuracy	0.57
	Precision	0.68
	Specificity	0.5897435897435898
	Recall	0.55737704918
Static Features	F1-Score	0.767676767677
	Accuracy	0.77
	Precision	0.76
	Specificity	0.7647058823529411
	Recall	0.775510204082
Hybrid Features	F1-Score	0.666666666667
	Accuracy	0.67
	Precision	0.66
	Specificity	0.6666666666666666
	Recall	0.673469387755

Table 5.3.: Experiment 2 (GroddDroid runner with uniform risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features.

were better with each classifier compared to the results from the dynamic feature set. However, the scores from the dynamic feature set and the KNN classifiers performed better than the results from the hybrid feature set. With the remaining classifiers the score from the hybrid feature set were higher than the ones from the dynamic feature set but lower than the ones from the static feature set.

Experiment 3: Monkey runner with original risk scores

We again took the same one hundred applications, this time with the *Monkey* stimulation engine and the original *GroddDroid* risk scores. With this, the experiment went through even fewer iterations, viz. three, and took about 26 hours to complete. The number of misclassified applications also rose to 49 alongside the crashed applications with 29. As seen in figure 5.3, the *precision* metric and the *F1-Score* dropped after their high in the second iteration to their minimum in the third iteration. The scores of all other metrics, however grew in each iteration. Table 5.6 shows that all scores using the dynamic feature set got better using our forcing technique compared to the results obtained without forcing the application. Again, the scores from the hybrid feature set could not trump the scores from the static feature set, but using the KNN classifiers they could surpass the results from the hybrid feature set. These results, from the hybrid feature set, did achieve better results than the ones from dynamic feature set when looking at the Random Forrest and SVM classifier. They sometimes also got slightly better results than the static features.

Algorithm	Dynamic features	Static features	Hybrid features
KNN10	0.6379 | 0.6724 (4) | 0.6724	0.7033	0.3810 | 0.4719 (3) | 0.4524
KNN15	0.6167 | 0.6891 (4) | 0.6891	0.7174	0.4681 | 0.5393 (2) | 0.5169
KNN20	0.6555 | 0.7000 (4) | 0.6885	0.7174	0.4719 | 0.5227 (2) | 0.5169
KNN25	0.6018 | 0.6551 (5) | 0.6551	0.7158	0.4565 | 0.5106 (3) | 0.5054
KNN30	0.5471 | 0.5926 (3) | 0.5681	0.7312	0.4523 | 0.5161 (3) | 0.4889
KNN35	0.4545 | 0.5454 (4) | 0.5393	0.7312	0.4835 | 0.5161 (2) | 0.5161
TREES10	0.5346 | 0.5686 (3) | 0.5542	0.7679	0.6593 | 0.7416 (4) | 0.7045
TREES25	0.5053 | 0.6019 (3) | 0.5870	0.7723	0.6863 | 0.7917 (2) | 0.7500
TREES50	0.5455 | 0.6481 (3) | 0.5287	0.7800	0.7273 | 0.7327 (4) | 0.7216
TREES75	0.5192 | 0.6000 (4) | 0.5227	0.7879	0.7010 | 0.7677 (3) | 0.6947
TREES100	0.5686 | 0.6154 (3) | 0.5287	0.7879	0.7600 | 0.7789 (3) | 0.7368
SVM	0.4706 | 0.5316 (5) | 0.5316	0.8542	0.6400 | 0.7407 (3) | 0.7115
ALL	0.5545 | 0.6126 (4) | 0.5776	0.7677	0.6400 | 0.6869 (2) | 0.6596

Table 5.4.: First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 2 (GroddDroid runner with uniform risk scores): Best overall results of the F1-Score for each individual algorithm.

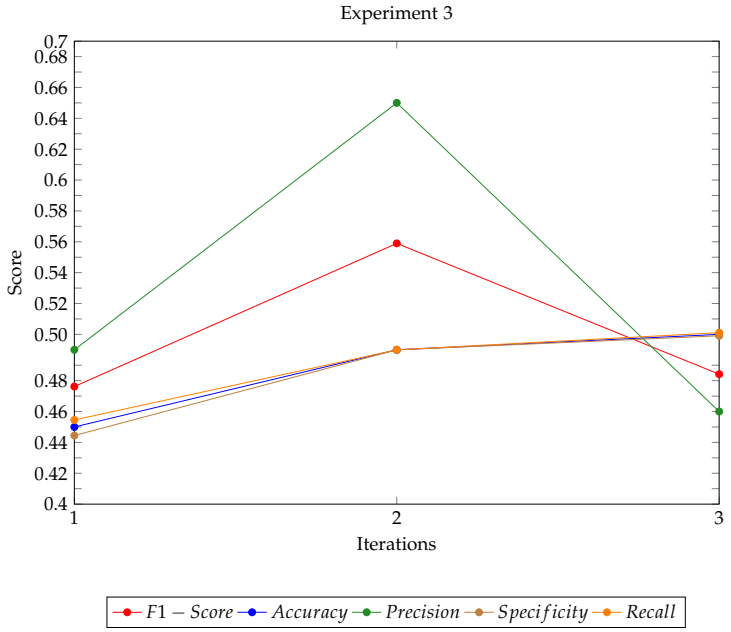

Figure 5.3.: Experiment 3 (Monkey runner with original risk scores): Evolution of the dynamic feature scores for each iteration from the ensemble classifier.

Experiment 4: Monkey runner with uniform risk scores

In the last experiment we conducted for this dataset, we only changed the risk scores to 10 compared to the previous experiment. The runtime was 55 hours with six iterations.

Feature Set	Metrics	Best Iteration
Dynamic Features	F1-Score	0.484210526316
	Accuracy	0.51
	Precision	0.46
	Specificity	0.509090909090909
	Recall	0.511111111111
Static Features	F1-Score	0.755102040816
	Accuracy	0.76
	Precision	0.74
	Specificity	0.75
	Recall	0.770833333333
Hybrid Features	F1-Score	0.680412371134
	Accuracy	0.69
	Precision	0.66
	Specificity	0.6792452830188679
	Recall	0.702127659574

Table 5.5.: Experiment 3 (Monkey runner with original risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features.

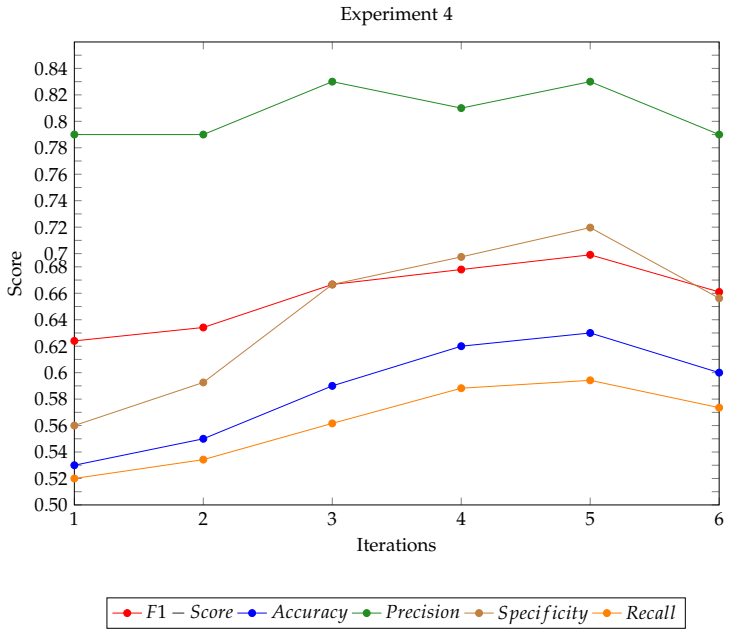

Figure 5.4.: Experiment 4 (Monkey runner with uniform risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier.

We had a total number of 27 crashed applications as well as 49 misclassified with this experiment setup. In figure 5.4 the *precision* metric stands out as it achieved

Algorithm	Dynamic features	Static features	Hybrid features
KNN10	0.5333 | 0.5913 (2) | 0.5738	0.7033	0.3678 | 0.4578 (2) | 0.3846
KNN15	0.5968 | 0.6562 (2) | 0.6562	0.7174	0.4894 | 0.5054 (2) | 0.4944
KNN20	0.5620 | 0.6190 (2) | 0.5714	0.7174	0.4318 | 0.4691 (3) | 0.4691
KNN25	0.5546 | 0.5593 (2) | 0.5098	0.7158	0.4731 | 0.4938 (3) | 0.4938
KNN30	0.4494 | 0.4516 (2) | 0.4156	0.7312	0.4524 | 0.4819 (2) | 0.4615
KNN35	0.4186 | 0.4494 (2) | 0.3733	0.7312	0.4828 | 0.5169 (2) | 0.4944
TREES10	0.4742 | 0.5740 (2) | 0.4782	0.7600	0.6237 | 0.6882 (3) | 0.6882
TREES25	0.4045 | 0.4902 (2) | 0.4632	0.7800	0.7059 | 0.7835 (3) | 0.7835
TREES50	0.4043 | 0.5000 (2) | 0.4516	0.7800	0.7142 | 0.7755 (3) | 0.7755
TREES75	0.4490 | 0.4694 (2) | 0.4536	0.7921	0.6465 | 0.7525 (2) | 0.7723
TREES100	0.4130 | 0.4952 (2) | 0.4946	0.7677	0.7010 | 0.7629 (2) | 0.7423
SVM	0.3750 | 0.4474 (3) | 0.4474	0.8542	0.7273 | 0.7184 (2) | 0.7184
ALL	0.5941 | 0.6804 (3) | 0.6804	0.7677	0.5941 | 0.6804 (3) | 0.6804

Table 5.6.: First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 3 (Monkey runner with original risk scores): Best overall results of the F1-Score for each individual algorithm.

Feature Set	Metrics	Best Iteration
Dynamic Features	F1-Score	0.689075630252
	Accuracy	0.63
	Precision	0.82
	Specificity	0.7096774193548387
	Recall	0.594202898551
Static Features	F1-Score	0.767676767677
	Accuracy	0.77
	Precision	0.76
	Specificity	0.7647058823529411
	Recall	0.775510204082
Hybrid Features	F1-Score	0.686274509804
	Accuracy	0.68
	Precision	0.7
	Specificity	0.6875
	Recall	0.673076923077

Table 5.7.: Experiment 4 (Monkey runner with uniform risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features.

higher scores than any of the other metrics in this experiment. Like in the previous experiments, the results for the *F1-Score* of the ensemble classifier using the dynamic features achieved the best results in an advanced iteration. Further and also similar to the previous experiments, the KNN classifier with the dynamic features achieved better results than the ones from the hybrid feature set while at the same time the results from the hybrid feature set were better using the remaining classifiers.

Algorithm	Dynamic features	Static features	Hybrid features
KNN10	0.6071 | 0.6721 (6) | 0.6721	0.7033	0.2891 | 0.4146 (4) | 0.3448
KNN15	0.6290 | 0.6720 (3) | 0.6612	0.7174	0.4717 | 0.5049 (6) | 0.5049
KNN20	0.5739 | 0.6316 (4) | 0.6018	0.7174	0.4694 | 0.5393 (4) | 0.5161
KNN25	0.6230 | 0.6333 (6) | 0.6333	0.7158	0.4952 | 0.5361 (6) | 0.5361
KNN30	0.6034 | 0.6324 (6) | 0.6324	0.7312	0.5102 | 0.5618 (5) | 0.5376
KNN35	0.5357 | 0.5811 (4) | 0.5614	0.7312	0.5253 | 0.5319 (6) | 0.5319
TREES10	0.5370 | 0.6491 (5) | 0.5789	0.7879	0.7000 | 0.7173 (2) | 0.7021
TREES25	0.5556 | 0.6667 (6) | 0.6667	0.7800	0.7312 | 0.7524 (5) | 0.6800
TREES50	0.6306 | 0.6724 (4) | 0.6167	0.7551	0.7475 | 0.7708 (5) | 0.7292
TREES75	0.6434 | 0.6667 (3) | 0.6557	0.7800	0.7600 | 0.7722 (2) | 0.7400
TREES100	0.5964 | 0.7107 (6) | 0.7107	0.7800	0.7579 | 0.7600 (2) | 0.7071
SVM	0.3673 | 0.6207 (6) | 0.6207	0.8542	0.6990 | 0.7115 (5) | 0.6990
ALL	0.6240 | 0.6890 (5) | 0.6610	0.7677	0.6408 | 0.7059 (4) | 0.6792

Table 5.8.: First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 4 (Monkey runner with uniform risk scores): Best overall results of the F1-Score for each individual algorithm.

5.2.4. MalGenome dataset experiments

In our next experiments we wanted to test our framework against the *MalGenome* database. Therefore, we took 50 applications from this dataset for our malware applications. The other 50 goodware applications were obtained randomly from the *Google Play Store*.

Experiment 5: GroddDroid runner with original risk scores

Feature Set	Metrics	Best Iteration
Dynamic Features	-Score	0.718446601942
	Accuracy	0.71
	Precision	0.74
	Specificity	0.723404255319149
	Recall	0.698113207547
Static Features	F1-Score	0.878504672897
	Accuracy	0.87
	Precision	0.94
	Specificity	0.9302325581395349
	Recall	0.824561403509
Hybrid Features	F1-Score	0.857142857143
	Accuracy	0.85
	Precision	0.9
	Specificity	0.8888888888888888
	Recall	0.818181818182

Table 5.9.: Experiment 5 (GroddDroid runner with original risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features.

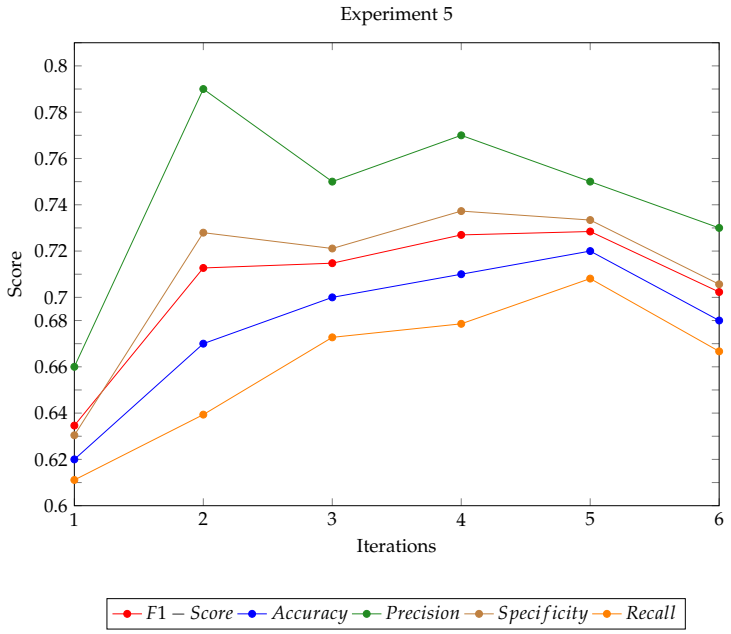

Figure 5.5.: Experiment 5 (GroddDroid runner with original risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier.

Similar to the aforementioned experiments, we first ran this experiment with the *GroddDroid* GUI stimulation engine and the original *GroddDroid* risk scores. This led to an experiment that lasted 64 hours, which is the highest we had in all our experiments. Even though, only six iterations were made. 28 applications crashed during the runtime and 32 got misclassified on average. As seen in figure 5.5, the *precision* metric peaked in the second iteration and fell after that, still dominating every other metric. Table 5.10 verifies that the results of the dynamic features got better with the forcing of an application in an advanced iteration. The only exception in this experiment was the KNN20 classifier. The score of this classifier could not be improved. Another point that this table shows, is that the results from the static and hybrid features trumped the results from the dynamic features. The best results within the dynamic feature set were achieved with the Random Forrest classifiers. These classifiers also led to better results using the hybrid feature set compared to the static feature set.

Algorithm	Dynamic features	Static features	Hybrid features
KNN10	0.5625 | 0.6232 (6) | 0.6232	0.8269	0.6481 | 0.7572 (3) | 0.6990
KNN15	0.6187 | 0.6382 (4) | 0.6345	0.8037	0.7059 | 0.7568 (3) | 0.7368
KNN20	0.6481 | 0.6481 (1) | 0.6095	0.7925	0.6897 | 0.7568 (5) | 0.7143
KNN25	0.6078 | 0.6486 (2) | 0.6200	0.7679	0.6780 | 0.7080 (2) | 0.6786
KNN30	0.5859 | 0.6200 (5) | 0.6061	0.7478	0.6724 | 0.6731 (5) | 0.6667
KNN35	0.5859 | 0.6200 (5) | 0.6061	0.7500	0.6549 | 0.6549 (1) | 0.6214
TREES10	0.6526 | 0.7660 (5) | 0.7253	0.9378	0.9032 | 0.9485 (4) | 0.8750
TREES25	0.6804 | 0.7143 (2) | 0.7097	0.9400	0.8842 | 0.9485 (4) | 0.9400
TREES50	0.6170 | 0.7473 (5) | 0.7447	0.9400	0.9600 | 0.9600 (1) | 0.9388
TREES75	0.6526 | 0.7447 (5) | 0.7292	0.9495	0.9167 | 0.9691 (2) | 0.9278
TREES100	0.6250 | 0.7708 (5) | 0.7447	0.9495	0.9388 | 0.9690 (3) | 0.9600
SVM	0.6889 | 0.6889 (1) | 0.5909	0.9072	0.8333 | 0.8571 (2) | 0.7959
ALL	0.6346 | 0.7184 (5) | 0.6923	0.8785	0.7586 | 0.8571 (5) | 0.8214

Table 5.10.: First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 5 (GroddDroid runner with original risk scores): Best overall results of the F1-Score for each individual algorithm.

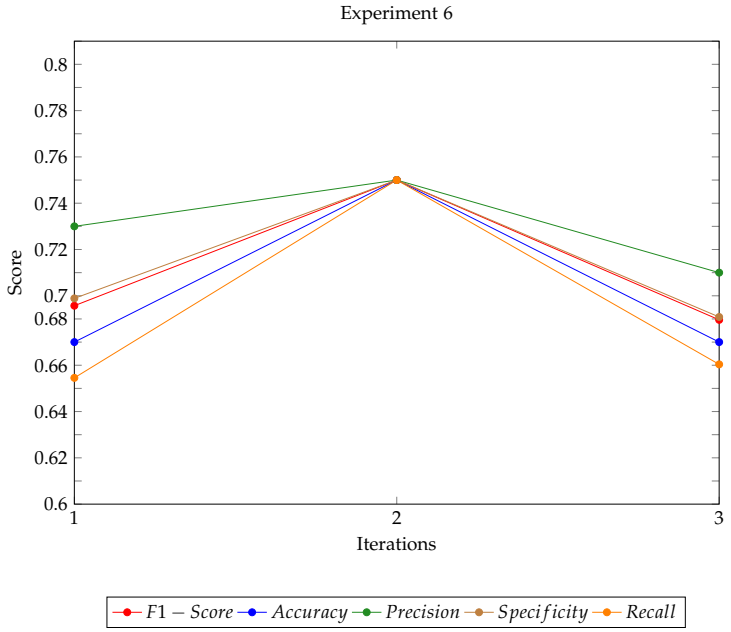

Figure 5.6.: Experiment 6 (GroddDroid runner with uniform risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier.

Experiment 6: GroddDroid runner with uniform risk scores

In this experiment we changed the risk scores to be uniform again. This led to a runtime of 40 hours with a total of three iterations. 30 applications got misclassified on average

Feature Set	Metrics	Best Iteration
Dynamic Features	F1-Score	0.74
	Accuracy	0.74
	Precision	0.74
	Specificity	0.74
	Recall	0.74
Static Features	F1-Score	0.878504672897
	Accuracy	0.87
	Precision	0.94
	Specificity	0.9302325581395349
	Recall	0.824561403509
Hybrid Features	F1-Score	0.793103448276
	Accuracy	0.76
	Precision	0.92
	Specificity	0.8823529411764706
	Recall	0.69696969697

Table 5.11.: Experiment 6 (GroddDroid runner with uniform risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features.

and 27 crashed. Interestingly enough in the second and best run of all, all metrics had a score of 0.74, which can be seen in figure 5.6 and table 5.11. Table 5.12 shows that we have increased the scores of each classifier with forcing again compared to without forcing, using the dynamic feature set. The table with the detailed results also presents that the Random Forrest classifiers achieved the best results within the dynamic feature sets. These classifiers also achieved better results with the hybrid features than with the static ones. Overall the results using the hybrid and static features trumped the results from the dynamic features.

Experiment 7: Monkey runner with original risk scores

After the first two experiments with this dataset, we changed the GUI stimulation to the *Monkey* runner for the last two experiments. This experiment took 38 hours to finish with a total of four iterations. 28 applications crashed and 40 got misclassified on average. Similar to the previous experiment all metrics reached their maximum at 0.74 in the third iteration, as shown in figure 5.7 and table 5.13. This could indicate that both experiments found the most malicious part of the applications in these iterations. Seen in table 5.14, we were able to better the results using the dynamic feature set for every classifier. Though, all results using the hybrid or static feature set, achieved better results than the ones from the dynamic feature set. By using the tree classifier the hybrid features achieved a better score than compared with the static features.

Algorithm	Dynamic features	Static features	Hybrid features
KNN10	0.5692 | 0.6046 (3) | 0.6046	0.8269	0.6139 | 0.6915 (2) | 0.6602
KNN15	0.5909 | 0.5968 (2) | 0.5839	0.8037	0.6783 | 0.7478 (2) | 0.7069
KNN20	0.6842 | 0.7193 (2) | 0.6909	0.7925	0.6842 | 0.7193 (2) | 0.6909
KNN25	0.6000 | 0.6667 (3) | 0.6667	0.7679	0.6897 | 0.6964 (3) | 0.6964
KNN30	0.6000 | 0.6060 (3) | 0.6060	0.7478	0.6726 | 0.6909 (3) | 0.6909
KNN35	0.6000 | 0.6060 (3) | 0.6060	0.7500	0.6726 | 0.6667 (2) | 0.6667
TREES10	0.6522 | 0.6966 (3) | 0.6966	0.9184	0.9184 | 0.9388 (3) | 0.9388
TREES25	0.7340 | 0.7391 (3) | 0.7391	0.9400	0.9109 | 0.9495 (3) | 0.9495
TREES50	0.6882 | 0.7660 (3) | 0.7660	0.9400	0.9600 | 0.9697 (3) | 0.9697
TREES75	0.7158 | 0.7292 (2) | 0.7083	0.9388	0.9053 | 0.9485 (2) | 0.9184
TREES100	0.6947 | 0.7609 (3) | 0.7609	0.9400	0.9278 | 0.9592 (2) | 0.9388
SVM	0.6667 | 0.6947 (2) | 0.6667	0.9072	0.7525 | 0.8687 (3) | 0.8687
ALL	0.6857 | 0.7400 (2) | 0.6796	0.8785	0.8070 | 0.7965 (3) | 0.7965

Table 5.12.: First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 6 (GroddDroid runner with uniform risk scores): Best overall results of the F1-Score for each individual algorithm.

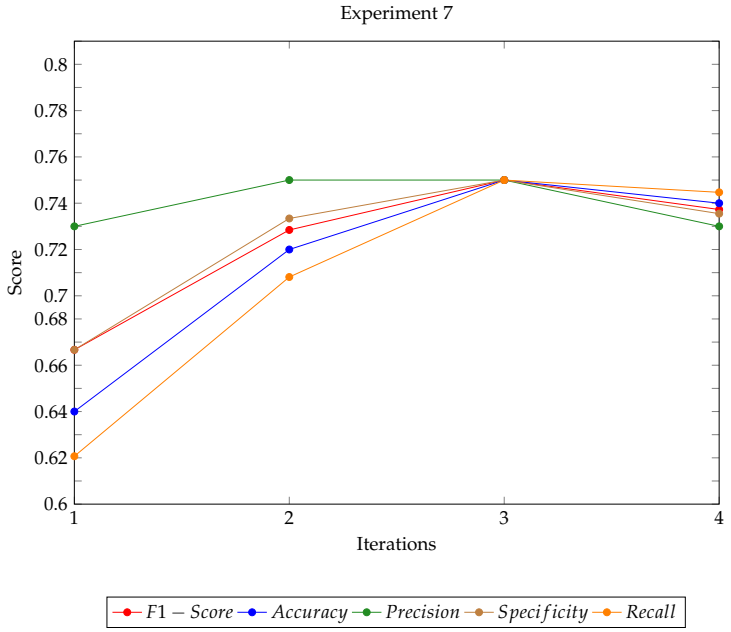

Figure 5.7.: Experiment 7 (Monkey runner with original risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier.

Experiment 8: Monkey runner with uniform risk scores

The last experiment we conducted took 32 hours and used the *Monkey* stimulation engine and uniform risk scores. In a total of three iterations, 35 applications got

Feature Set	Metrics	Best Iteration
Dynamic Features	F1-Score	0.74
	Accuracy	0.74
	Precision	0.74
	Specificity	0.74
	Recall	0.74
Static Features	F1-Score	0.878504672897
	Accuracy	0.87
	Precision	0.94
	Specificity	0.9302325581395349
	Recall	0.824561403509
Hybrid Features	F1-Score	0.87619047619
	Accuracy	0.87
	Precision	0.92
	Specificity	0.9111111111111111
	Recall	0.836363636364

Table 5.13.: Experiment 7 (Monkey runner with original risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features.

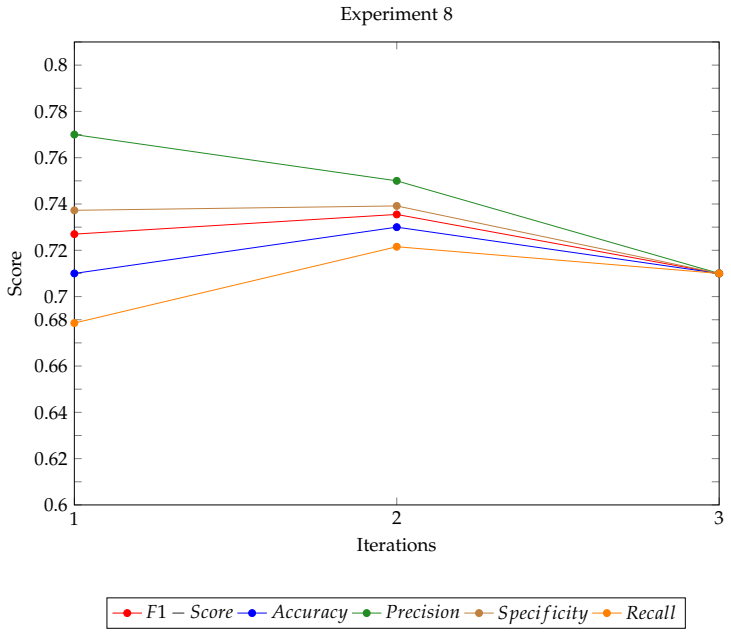

Figure 5.8.: Experiment 8 (Monkey runner with uniform risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier.

misclassified and 27 crashed. Figure 5.8 shows that the *precision* metric started with its highest score and fell during each iteration. The other metrics grew their score until

Algorithm	Dynamic features	Static features	Hybrid features
KNN10	0.5378 | 0.6821 (2) | 0.6718	0.8269	0.6038 | 0.7647 (2) | 0.7400
KNN15	0.6603 | 0.7115 (3) | 0.6560	0.8037	0.6837 | 0.7850 (3) | 0.7664
KNN20	0.6604 | 0.7071 (3) | 0.6939	0.7925	0.6552 | 0.7593 (3) | 0.7407
KNN25	0.6476 | 0.6730 (2) | 0.6250	0.7679	0.6890 | 0.7289 (3) | 0.7222
KNN30	0.6214 | 0.6327 (3) | 0.6263	0.7478	0.6552 | 0.6981 (2) | 0.6923
KNN35	0.6214 | 0.6327 (3) | 0.6263	0.7500	0.6552 | 0.6981 (2) | 0.6923
TREES10	0.7071 | 0.7234 (3) | 0.7174	0.9475	0.9400 | 0.9485 (4) | 0.9485
TREES25	0.7115 | 0.7619 (2) | 0.7475	0.9495	0.9400 | 0.9583 (4) | 0.9583
TREES50	0.7115 | 0.7600 (3) | 0.7010	0.9495	0.9292 | 0.9591 (3) | 0.9494
TREES75	0.7048 | 0.7358 (2) | 0.7083	0.9495	0.9307 | 0.9697 (3) | 0.9388
TREES100	0.6857 | 0.7347 (4) | 0.7347	0.9400	0.9495 | 0.9583 (2) | 0.9388
SVM	0.5747 | 0.5747 (1) | 0.5176	0.9072	0.7677 | 0.8541 (4) | 0.8541
ALL	0.6667 | 0.7400 (3) | 0.7273	0.8785	0.7833 | 0.8761 (3) | 0.8679

Table 5.14.: First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 7 (Monkey runner with original risk scores): Best overall results of the F1-Score for each individual algorithm.

Feature Set	Metrics	Best Iteration
Dynamic Features	F1-Score	0.725490196078
	Accuracy	0.72
	Precision	0.74
	Specificity	0.7291666666666666
	Recall	0.711538461538
Static Features	F1-Score	0.878504672897
	Accuracy	0.87
	Precision	0.94
	Specificity	0.9302325581395349
	Recall	0.824561403509
Hybrid Features	F1-Score	0.825688073394
	Accuracy	0.81
	Precision	0.9
	Specificity	0.8780487804878049
	Recall	0.762711864407

Table 5.15.: Experiment 8 (Monkey runner with uniform risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features.

the second iteration. In the third iteration every metric fell. In table 5.16 we can see that we were again able to better the results using the dynamic feature in an advanced iteration with the only exception of the KNN2020 classifier. Similar to the previous experiments the results from the hybrid and static features set did achieve better scores than the ones from the dynamic feature set. Using the Random Forest classifiers the hybrid features performed better than the static ones.

Algorithm	Dynamic features	Static features	Hybrid features
KNN10	0.5941 | 0.6667 (3) | 0.6667	0.8269	0.6600 | 0.7308 (2) | 0.7184
KNN15	0.6923 | 0.7143 (2) | 0.6538	0.8037	0.7018 | 0.7611 (3) | 0.7611
KNN20	0.6471 | 0.6392 (3) | 0.6392	0.7925	0.6667 | 0.7321 (2) | 0.7207
KNN25	0.6214 | 0.6535 (3) | 0.6535	0.7679	0.6949 | 0.7193 (2) | 0.7069
KNN30	0.6200 | 0.6465 (3) | 0.6465	0.7478	0.6903 | 0.6964 (2) | 0.6786
KNN35	0.6200 | 0.6465 (3) | 0.6465	0.7500	0.6786 | 0.6909 (2) | 0.6903
TREES10	0.7292 | 0.8085 (3) | 0.8085	0.9278	0.8824 | 0.9200 (3) | 0.9200
TREES25	0.7254 | 0.7475 (2) | 0.7273	0.9485	0.9400 | 0.9473 (2) | 0.9293
TREES50	0.7429 | 0.7572 (2) | 0.7292	0.9400	0.9278 | 0.9592 (3) | 0.9592
TREES75	0.7500 | 0.7629 (3) | 0.7629	0.9400	0.9505 | 0.9800 (3) | 0.9800
TREES100	0.7379 | 0.7647 (2) | 0.7551	0.9400	0.9293 | 0.9690 (2) | 0.9388
SVM	0.5909 | 0.6809 (2) | 0.5806	0.9072	0.7368 | 0.8421 (2) | 0.8333
ALL	0.7170 | 0.7255 (2) | 0.7000	0.8785	0.8108 | 0.8257 (2) | 0.8288

Table 5.16.: First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 8 (Monkey runner with uniform risk scores): Best overall results of the F1-Score for each individual algorithm.

5.3. Discussion

In the previous sections we have seen the individual results for each of our conducted experiments. We now want to outline the general insights we gained from our experiments. To do this, we first of all want to answer our main research questions.

First and foremost, we were able to achieve better results using our continued forcing loop compared to the results without forcing, with only very few exceptions using the KNN and SVM classifiers. The best results within the dynamic feature set were most of the time achieved by the KNN classifiers for the *Piggybacked* dataset and by the Random Forest classifiers for the *MalGenome* dataset. This confirms that with the continued forcing of different branches within an application in our proposed feedback loop does score better results compared to running an application only once and without forcing.

In the following we want to answer our research questions regarding the average of all eight experiments. The time needed to run is an important metric for us, as the experiments need to finish within a reasonable amount of time to be useful and efficient to detect malware instances. The average runtime for all eight conducted experiments is about 42 hours with a total of 5 iterations in our feedback loop. *GroddDroid* crashed 28 applications and misclassified 40 on average in each experiment. It has to be further investigated which part of our framework crashed the applications and why. It is not possible to eliminate all crashes, as sometimes the malware applications are very poorly implemented and are crashing because of this, but the number can be shrunk. In nearly every experiment the best results for all metrics were achieved in the second to last or last iteration.

Research question five can be answered with a simple yes. As we saw, most often the

results using the static features were better than the ones using the dynamic features, with only very few exceptions. However, in the *MalGenome* dataset the results from the hybrid feature set often trumped the results from the static feature set when using the Random Forest classifiers.

We could not draw a clear conclusion for the sixth research question from our results. The number of iteration as well as the overall scores of the metrics seem to be not influenced by the type of the GUI stimulation engine.

The last research question, if following the highest scoring branch leads to a faster classification of an application, is not easily answered with regard to our results. First of all, the original scores in the *GroddDroid* heuristics database were computed looking at how frequent these API calls were made in other malware applications. Computing the risk scores based on the significance of an API call with regard to malware applications, would have been another option to base the risk scores on. A specific API call, for example, can be used less frequent in malware applications, but if it is present it is a clear indicator of a malware application. It is left to be investigated in the future if this change would make a significant difference. There is no indicator in our results that changing the risk scores between the original *GroddDroid* and the uniform ones made a significant impact on the output. In the experiments using the *MalGenome* dataset, both experiments using the uniform risk scores only needed three iterations to finish, which is lower than the experiments using the original scores. However, in the *Piggybacked* dataset there is no clear sign of an impact from the risk scores.

For the *Piggybacked* dataset we achieved the best *F1-Score* of 0.7119 in the first experiment. Whereas the best *F1-Score* for the *MalGenome* dataset was achieved in the sixth and seventh experiment with a score of 0.7400. In total we achieved slightly better results for nearly every metric using the *MalGenome* dataset compared to the *Piggybacked* dataset.

We further want to point out that our implemented ensemble algorithm of all used machine learning algorithms gave us a good average over all twelve algorithms in each experiment.

This shows that our approach is not perfected yet, as often times the static features would classify most instances better than the dynamic feature set. However, we have shown that we can improve the scores of the dynamic feature set with forcing. Therefore, this method should be investigated and enhanced further until the results using the dynamic feature set are on the same level or better as the ones from the static features.

6. Conclusion

In this thesis we have presented our approach to build a better framework for detecting *Android* repackaged malware. We first introduced all necessary background information as well as our motivation for this research, followed by the architecture that we built for our goal. We used *GroddDroid* to stimulate and force branches within *Android* applications, with the hope to disclose all possibly malicious parts of an application. The features we extracted were then analyzed by machine learning algorithms and depending on that we decided whether to do another run in our feedback loop or not. If we did another run, we analyzed every application that got misclassified again with a different branch forced. We reviewed all steps we took to implement our framework and described the conducted experiments. After that we evaluated the results of our experiments with the following conclusion, based on our sampled data. We were able to achieve better results using our proposed feedback loop with the forcing aspect of *GroddDroid* compared to the results we achieved with running an application once and without forcing, with regard to the dynamic feature set. Though, the scores achieved by using the static features were most of the time better than the ones we achieved using the dynamic feature set. For us, it is a clear sign that our approach is working and heads in the right direction. However, there is still some research needed to elevate the results using the dynamic features to the same level as the results from the static feature set.

It is not foreseeable if the stream of new mobile phones will tear in the future, the trend, however, shows that there will be even more mobile phones on the market. This means, in the future even more people will trust and rely upon their mobile phone, at the same time attackers will have more valuable targets to attack and will therefore produce a static stream of new malware instances. For this reason, malware analysis, especially in the mobile field, will not lose any of its importance. On the contrary, it will most likely be even more important. Therefore, in the closing chapter of this thesis, we want to present some points building upon our research that might be worth to be investigated in the future.

7. Further Work

In this last chapter, we want to present some last points that we think would make sense to research in the future, building upon this thesis.

As described earlier, we are currently not detecting if an application showed the 'Unfortunately, the application has stopped', error message during the runtime. This would be worth achieving. One could then further investigate these instances and determine, if they crashed because of poor implementation or of an error within our framework.

In our research we conducted every experiment only once. For a higher statistical significance, each experiment should be run several times and compared to each other. Another thing that should be experimented with is to run more iterations and broaden the stopping condition we implemented in this research, with this one could test if the maxima in the *F1-Score* that we found are only local maxima or global ones.

Experiments with higher numbers of applications are also worthwhile, for instance 500 goodware and 500 malware instances. This would consolidate our results and might bring up new or more insights. Of course, such an experiment would be bound to a higher execution time, which is why we were not able to run it during our research.

We exclusively focused on *GroddDroid* as the tool for forcing different paths within the application, however there are several other tools doing a similar thing. A comparison between these tools would be another point worth investigating.

There are several other algorithms that could be used and compared in the space of machine learning. Support Vector Machines with a String Subsequence Kernel (SSK) to analyze the traces file or a neuronal network are just two examples of such algorithms. These could also be implemented in our given framework so that they can be compared to each other.

A last point that should be looked at in more detail, is the heuristics database of *GroddDroid*. In our evaluation we could not find a significant impact by using different risk scores. However, we think this should be investigated in greater detail with the goal to find a good fitting database for current and maybe future malware applications.

List of Figures

1.1. Proposed feedback cycle: First we will stimulate an application to extract features. These will then be channeled into the detection part. Depending on the results, we may make another iteration of stimulating and detecting. 2

2.1. Repackaging/Piggybacking structure [8]. The malicious rider code extends the original carrier application. The hook connects the rider to the carrier. 8
2.2. GroddDroid framework overview [6]. Annotated with the information about what part is executed in which step of the GroddDroid execution. 10
2.3. Possible trigger points for GroddDroid [6]. The framework tries to evaluate the conditional statements, so that the malicious part will be executed. 11

3.1. Current approach for finding malware: First, an application will be stimulated once. Then, the extracted features are analyzed. 20
3.2. Overall structure of our solution: We first stimulate an application, without forcing. After that, the extracted features are channeled into various machine learning algorithms. Depending on the F1-Score from the dynamic features we decide, whether to do another iteration or not. In each following iteration, we force a different part of the application. . 22

4.1. Adaption to GroddDroids Soot implementation to mark all invoke statements in an application to later extract them for the method traces. . . . 27

5.1. Experiment 1 (GroddDroid runner with original risk scores): Evolution of the dynamic feature scores for each iteration from the ensemble classifier. 35
5.2. Experiment 2 (GroddDroid runner with uniform risk scores): Evolution of the dynamic feature scores for each iteration from the ensemble classifier. 37
5.3. Experiment 3 (Monkey runner with original risk scores): Evolution of the dynamic feature scores for each iteration from the ensemble classifier. 39
5.4. Experiment 4 (Monkey runner with uniform risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier. . 40

5.5. Experiment 5 (GroddDroid runner with original risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier. 43
5.6. Experiment 6 (GroddDroid runner with uniform risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier. 44
5.7. Experiment 7 (Monkey runner with original risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier. . 46
5.8. Experiment 8 (Monkey runner with uniform risk scores) Evolution of the dynamic feature scores for each iteration from the ensemble classifier. . 47

List of Tables

5.1. Experiment 1 (GroddDroid runner with original risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features. 36

5.2. First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 1 (GroddDroid runner with original risk scores): Best overall results of the F1-Score for each individual algorithm. 37

5.3. Experiment 2 (GroddDroid runner with uniform risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features. 38

5.4. First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 2 (GroddDroid runner with uniform risk scores): Best overall results of the F1-Score for each individual algorithm. 39

5.5. Experiment 3 (Monkey runner with original risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features. 40

5.6. First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 3 (Monkey runner with original risk scores): Best overall results of the F1-Score for each individual algorithm. 41

5.7. Experiment 4 (Monkey runner with uniform risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features. 41

5.8. First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 4 (Monkey runner with uniform risk scores): Best overall results of the F1-Score for each individual algorithm. 42

5.9. Experiment 5 (GroddDroid runner with original risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features. 42

5.10. First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 5 (GroddDroid runner with original risk scores): Best overall results of the F1-Score for each individual algorithm. 44
5.11. Experiment 6 (GroddDroid runner with uniform risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features. 45
5.12. First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 6 (GroddDroid runner with uniform risk scores): Best overall results of the F1-Score for each individual algorithm. 46
5.13. Experiment 7 (Monkey runner with original risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features. 47
5.14. First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 7 (Monkey runner with original risk scores): Best overall results of the F1-Score for each individual algorithm. 48
5.15. Experiment 8 (Monkey runner with uniform risk scores): Results of the best iteration for each feature set from the ensemble classifier, according to the F1-Score of the dynamic features. 48
5.16. First iteration | Best iteration (iteration-number of the best result) | Last iteration
Experiment 8 (Monkey runner with uniform risk scores): Best overall results of the F1-Score for each individual algorithm. 49

Bibliography

[1] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[2] J. Vincent. (2017). 99.6 percent of new smartphones run Android or iOS - While BlackBerry's market share is a rounding error, [Online]. Available: `https://www.theverge.com/2017/2/16/14634656/android-ios-market-share-blackberry-2016` (visited on 06/15/2017).

[3] A. Kharpal. (2016). Google Android hits market share record with nearly 9 in every 10 smartphones using it, [Online]. Available: `http://www.cnbc.com/2016/11/03/google-android-hits-market-share-record-with-nearly-9-in-every-10-smartphones-using-it.html` (visited on 06/15/2017).

[4] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109, ISBN: 978-0-7695-4681-0. DOI: 10.1109/SP.2012.16.

[5] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Comput. Surv.*, 49, no., 76:1–76:41, Jan. 2017, ISSN: 0360-0300. DOI: 10.1145/3017427.

[6] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong, "Grodddroid: A gorilla for triggering malicious behaviors.," in *MALWARE*, IEEE, 2015, pp. 119–127, ISBN: 978-1-5090-0319-8.

[7] L. Li, D. Li, T. F. Bissyandé, J. Klein, H. Cai, D. Lo, and Y. L. Traon, "Automatically locating malicious packages in piggybacked android apps," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 170–174, ISBN: 978-1-5386-2669-6. DOI: 10.1109/MOBILESoft.2017.6.

[8] Li, D. Li, T. F. Bissyande, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *Trans. Info. For. Sec.*, 12, no., pp. 1269–1284, Jun. 2017, ISSN: 1556-6013. DOI: 10.1109/TIFS.2017.2656460.

[9] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps," no., 2017.

[10] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, Austin, Texas: ACM, 2016, pp. 468–471, ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2903508.

[11] L. Li, D. Li, T. F. Bissyandé, J. Klein, H. Cai, D. Lo, and Y. L. Traon, "Automatically locating malicious packages in piggybacked android apps," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 170–174, ISBN: 978-1-5386-2669-6. DOI: 10.1109/MOBILESoft.2017.6.

[12] Y. Cuixia, Z. Chaoshun, G. Shanqing, H. Chengyu, and C. Lizhen, "Ui ripping in android: Reverse engineering of graphical user interfaces and its application," 2016, pp. 160–167. DOI: 10.1109/CIC.2015.22.

[13] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 300–311, ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.35.

[14] S. R. Choudhary, A. Gorla, and A. Orso, "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)*, ser. ASE '15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 429–440, ISBN: 978-1-5090-0025-8. DOI: 10.1109/ASE.2015.89.

[15] (). Ui/application exerciser monkey, [Online]. Available: https://developer.android.com/studio/test/monkey.html (visited on 07/21/2017).

[16] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 00, no., pp. 461–471, 2015. DOI: doi.ieeecomputersociety.org/10.1109/ISSRE.2015.7381839.

[17] Y. Su, Y. Yu, Y. Qiu, and A. Fu, "Symfinder: Privacy leakage detection using symbolic execution on android devices," in *Proceedings of the 4th International Conference on Information and Network Security*, ser. ICINS '16, Kuala Lumpur, Malaysia: ACM, 2016, pp. 13–18, ISBN: 978-1-4503-4796-9. DOI: 10.1145/3026724.3026731.

[18] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *SIGSOFT Softw. Eng. Notes*, 37, no., pp. 1–5, Nov. 2012, ISSN: 0163-5948. DOI: 10.1145/2382756.2382798.

[19] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, ser. CASCON '10, Toronto, Ontario, Canada: IBM Corp., 2010, pp. 214–224. DOI: 10.1145/1925805.1925818.

[20] C. Tumbleson and R. Wiśniewski. (2017). Apktool, [Online]. Available: https://ibotpeaches.github.io/Apktool/ (visited on 06/23/2017).

[21] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Security and Privacy in Communication Networks: 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, Eds. Cham: Springer International Publishing, 2013, pp. 86–103, ISBN: 978-3-319-04283-1. DOI: 10.1007/978-3-319-04283-1_6.

[22] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong. (2015). Grodddroid, [Online]. Available: http://kharon.gforge.inria.fr/grodddroid.html (visited on 05/21/2017).

[23] E. Alpaydin, *Introduction to Machine Learning*, 3rd ed., ser. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press, 2014, ISBN: 978-0-262-02818-9.

[24] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014, ISBN: 978-1-107-05713-5.

[25] L. C. Molina, L. Belanche, and À. Nebot, "Feature selection algorithms: A survey and experimental evaluation," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, ser. ICDM '02, Washington, DC, USA: IEEE Computer Society, 2002, pp. 306–, ISBN: 0-7695-1754-4.

[26] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Review: Classification of malware based on integrated static and dynamic features," *J. Netw. Comput. Appl.*, 36, no., pp. 646–656, Mar. 2013, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2012.10.004.

[27] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000, ISBN: 978-0521780193.

[28] (2017). Scikit-learn api reference, [Online]. Available: `http://scikit-learn.org/stable/modules/classes.html` (visited on 07/12/2017).

[29] J. Sahs and L. Khan, "A machine learning approach to android malware detection.," in *EISIC*, IEEE Computer Society, 2012, pp. 141–147, ISBN: 978-1-4673-2358-1.

[30] L. Li, K. Allix, D. Li, A. Bartel, T. F. Bissyandé, and J. Klein, "Potential component leaks in android apps: An investigation into a new feature set for malware detection," in *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*, 2015, pp. 195–200. DOI: `10.1109/QRS.2015.36`.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, 12, no., pp. 2825–2830, 2011, cited By 2916.

[32] I. Idris, *NumPy 1.5 Beginner's Guide*. Packt Publishing, 2011, ISBN: 978-1-84951-530-6.

[33] (2017). Virtualbox, [Online]. Available: `https://www.virtualbox.org/` (visited on 06/30/2017).

[34] (2017). Genymotion, [Online]. Available: `https://genymotion.com/` (visited on 06/30/2017).

A. Appendix: List of all extracted features

Features from the traces file

1. Total number of invokes
2. Number of virtual invokes
3. Number of special invokes
4. Number of methods to send a sms message (Method: *sendTextMessage*())
5. Number of times an activity was started (Method: *startActivity*())
6. Number of times a service was started (Method: *startService*())
7. Number of times the method *onCreate*() was called
8. Number of times the method *onStart*() was called
9. Number of times the method *onResume*() was called
10. Number of times the method *onPause*() was called
11. Number of times the method *onStop*() was called
12. Number of times the method *onDestroy*() was called
13. Number of times the method *onReceive*() was called
14. Number of times a package for binary execution was called (We considered the following packages: *java.lang.Runtime*, *java.lang.Process* ,*java.lang.ProcessBuilder*, *java.lang.System*, *org.apache.commons.exec.∗*, *org.apache.commons.launcher.∗*, *android.os.Process*)
15. Number of times a package for dynamic code loading was called (We considered the following packages: *dalvik.system.BaseDexClassLoader*, *dalvik.system.PathClassLoader*, *dalvik.system.DexClassLoader*, *dalvik.system.DexFile*)

16. Number of times a package for cryptographic functions was called (We considered the following packages: *javax.crypto.∗*, *java.security.spec.∗*, *org.apache.commons.crypto.∗*)

17. Number of times a package for network functions was called (We considered the following packages: *java.net.Socket*, *java.net.ServerSocket*, *java.net.HttpURLConnection*, *java.net.JarURLConnection*, *java.net.URL*, *org.apache.commons.net.∗*, *org.apache.http.∗*, *org.apache.commons.mail.∗*, *android.webkit.∗*, *java.net.ssl.∗*, *java.net.HttpCookie*, *android.app.DownloadManager*, *android.net.Network*, *android.net.wifi.WifiManager*)

18. Number of times the TelephonyManager (*android.telephony.TelephonyManager*) was called

19. Number of tiems the AccountsManager was called (*android.accounts.AccountManager*)

20. Number of times the BluetoothManager was called (*android.bluetooth.BluetoothManager*)

21. Number of times a package related to GPS was called (We considered the following packages: *android.location.LocationManager*, *android.location.LocationProvider*)

22. Number of times the NfcManager was called (*android.nfc.NfcManager*)

23. Number of times something suspicious from the Operating System was called (we considered the following packages: *android.content.pm.PackageInstaller*, *android.os.PowerManager*)

24. Number of times a package related to IO was called (We considered the following packages: *java.io.DataInput*, *java.io.DataOutput*)

25. Number of times a package related to recording was called (We considered the following packages: *android.media.AudioRecord*, *android.media.MediaRecorder*)

26. Number of times the display manager was called (*android.hardware.display.DisplayManager*)

27. Number of times a content resolver was called (*android.content.ContentResolver*)

28. Number of times a context wrapper was called (*android.content.ContextWrapper*)

29. Number of times a package related to databases was called (*android.database.*∗)
30. Number of times an application accessed the camera (*android.hardware.Camera*)
31. Number of times an application issued a toast (*android.widget.Toast*)

Features from the targets file

1. The total count of different alerts
2. The total count of all alerts detected by GroddDroid
3. The total sum of all malicious scores from GroddDroid

Features from the to_force file

1. The total count of branches to force

Features from the stats file

1. Number of all methods
2. Number of all conditions
3. Number of all seen methods
4. Number of all seen branches
5. Number of executed branches
6. Number of targeted branches
7. Coverage of all targeted branches
8. Method coverage
9. Branch coverage

Features from the android manifest file

1. Minimum SDK
2. Target SDK
3. Number of activities
4. Number of all activity indents

5. Number of services
6. Number of content providers
7. Number of broadcast receivers
8. Number of all broadcast receivers indents

   Permissions:

9. SMS permissions (*android.permission.SEND_SMS*, *android.permission.RECEIVE_SMS*, *android.permission.READ_SMS*, *android.permission.WRITE_SMS*)
10. Internet permission (*android.permission.INTERNET*)
11. Contact permission (*android.permission.READ_CONTACTS*)
12. Package permissions (*android.permission.DELETE_PACKAGES*, *android.permission.INSTALL_PACKAGES*, *android.permission.REQUEST_INSTALL_PACKAGES*)
13. Setting permission (*android.permission.WRITE_SETTINGS*)
14. Location permissions (*android.permission.ACCESS_FINE_LOCATION*, *android.permission.ACCESS_coarse_LOCATION*)
15. Phone state permission (*android.permission.READ_PHONE_STATE*)
16. Call permissions (*android.permission.READ_CALL_LOGS*, *android.permission.PROCESS_OUTGOING_CALLS*, *android.permission.CALL_PHONE*)
17. MMS permissions (*android.permission.RECEIVE_MMS*)
18. APN permissions (*android.permission.WRITE_APN_SETTINGS*, *android.permission.READ_APN_SETTINGS*)
19. Wifi permissions (*android.permission.CHANGE_WIFI_STATE*, *android.permission.CHANGE_CONFIGURATION*, *android.permission.CHANGE_NETWORK_STATE*)
20. Recording permissions (*android.permission.RECORD_AUDIO*)
21. Clear app cache permission (*android.permission.CLEAR_APP_CACHE*)

22. Boot permissions (*android.permission.RECEIVE_BOOT_COMPLETED*)
23. Account permissions (*android.permission.GET_ACCOUNTS*)
24. Read phone numbers permission (*android.permission.READ_PHONE_NUMBERS*)
25. Run in background permission (*android.permission.RUN_IN_BACKGROUND*)

B. Appendix: Heuristics database

```
{
    "packages_to_ignore": [
        "android.support.v4",
        "android.support.v7",
        "android.support",
        "com.google.android",
        "org.apache.commons"
    ],
    "categories": [
        {
            "category": "binary",
            "description": "Execution of binaries",
            "classes": [
                "java.lang.Runtime",
                "java.lang.Process",
                "java.lang.ProcessBuilder",
                "java.lang.System",
                "org.apache.commons.exec.*",
                "org.apache.commons.launcher.*",
                "android.os.Process"
            ],
            "grep": [],
            "score": 6
        },

        {
            "category": "dynamic",
            "description": "Dynamic code loading",
            "classes": [
                "dalvik.system.BaseDexClassLoader",
                "dalvik.system.PathClassLoader",
```

```
            "dalvik.system.DexClassLoader",
            "dalvik.system.DexFile"
        ],
        "grep": [],
        "score": 8
    },

    {
        "category": "crypto",
        "description": "Cryptographic libraries",
        "classes": [
            "javax.crypto.*",
            "java.security.spec.*",
            "org.apache.commons.crypto.*"
        ],
        "grep": [
            "encrypt",
            "decrypt"
        ],
        "score": 3
    },

    {
        "category": "network",
        "description": "Remote connections",
        "classes": [
            "java.net.Socket",
            "java.net.ServerSocket",
            "java.net.HttpURLConnection",
            "java.net.JarURLConnection",
            "java.net.URL",
            "org.apache.commons.net.*",
            "org.apache.http.*",
            "org.apache.commons.mail.*",
            "android.webkit.*",
            "java.net.ssl.*",
            "java.net.HttpCookie",
            "android.app.DownloadManager",
            "android.net.Network",
```

```
            "android.net.wifi.WifiManager"
        ],
        "grep": [],
        "score": 3
    },

    {
        "category": "telephony",
        "description": "Telephony information (phone number, IMEI, ...)
            ",
        "classes": [
          "android.telephony.TelephonyManager"
        ],
        "grep": [],
        "score": 15
    },

    {
        "category": "sms",
        "description": "SMS",
        "classes": [
          "android.telephony.SmsManager"
        ],
        "grep": [],
        "score": 40
    },

    {
        "category": "accounts",
        "description": "Information about the accounts on the mobile",
        "classes": [
          "android.accounts.AccountManager"
        ],
        "grep": [],
        "score": 5
    },

    {
        "category": "bluetooth",
```

```
    "description": "Bluetooth",
    "classes": [
      "android.bluetooth.BluetoothManager"
    ],
    "grep": [],
    "score": 5
},

{
    "category": "GPS",
    "description": "gps",
    "classes": [
      "android.location.LocationManager",
      "android.location.LocationProvider"
    ],
    "grep": [],
    "score": 5
},

{
    "category": "NFC",
    "description": "nfc",
    "classes": [
      "android.nfc.NfcManager"
    ],
    "grep": [],
    "score": 8
},

{
    "category": "OS",
    "description": "os",
    "classes": [
      "android.content.pm.PackageInstaller",
      "android.os.PowerManager"
    ],
    "grep": [],
    "score": 20
},
```

```
        {
            "category": "IO",
            "description": "classes corresponding to input and output",
            "classes": [
              "java.io.DataInput",
              "java.io.DataOutput"
            ],
            "grep": [],
            "score": 2
        },

        {
            "category": "Recording",
            "description": "Recording",
            "classes": [
              "android.media.AudioRecord",
              "android.media.MediaRecorder"
            ],
            "grep": [],
            "score": 30
        },

        {
            "category": "Display",
            "description": "Display",
            "classes": [
              "android.hardware.display.DisplayManager"
            ],
            "grep": [],
            "score": 10
        }
    ]
}
```

C. Appendix: Main file source code

```
import os, subprocess, argparse, glob, time, shutil, pickle, sys
from Config_handler import Config_handler
from Utils import *
from MLA.extract_numerical_features import *
from MLA.Learn import predictKFoldSVM, predictAndTestKFoldSVM,
    predictKFoldSVMSSK, calculateMetrics,
    get_indices_of_misclassified_apps, predictKFoldTrees, predictKFoldKNN,
     predictKFoldSVM
from shutil import copyfile

def setup_Arguments():
    parser = argparse.ArgumentParser(prog="main.py")
    parser.add_argument("-explorer_dir", "--branchExplorer_dir", help="The
        ␣path␣to␣the␣subdirectory␣BranchExplorer␣in␣the␣GroddDroid␣files",
         required=True)
    parser.add_argument("-inputDir", "--inputDirectory", help="The␣path␣to
        ␣the␣directory␣with␣the␣input␣APKs.␣It␣should␣contain␣two␣
        directories␣malware␣and␣goodware", required=True)
    parser.add_argument("-outputDir", "--outputDirectory", help="The␣path␣
        to␣the␣directory␣for␣the␣output", required=True)
    parser.add_argument("-kf", "--kfold", help="Set␣the␣value␣of␣K␣for␣
        cross␣validation", default=2)
    parser.add_argument("-selectBest", "--selectBest", help="Set␣the␣
        number␣of␣the␣features␣that␣should␣be␣selected␣>=␣30", default=0)
    parser.add_argument("-VMname", "--vm_name", help="The␣name␣or␣ID␣of␣
        the␣Virtual␣Box␣virtual␣machine␣to␣use", default="7e44e530-9603-44
        af-9c03-72d5654130e7")
    parser.add_argument("-restore_snapshot", "--restore_snapshot", help="
        The␣name␣or␣ID␣of␣the␣Virtual␣Box␣snapshot␣to␣restore␣the␣virtual␣
        machine", default="117f78c8-97da-4acc-a6a8-d12e4d14d85e")
    return parser
```

```
genyProcess = None

def main():
    """Main method which handles the overall process"""

    argumentParser = setup_Arguments()
    arguments = argumentParser.parse_args()

    ######################################################################
    # Retrive the apk files form the input directory and prepare output
        directory
    ######################################################################
    if os.path.exists(arguments.outputDirectory + "/results.txt"):
        os.remove(arguments.outputDirectory + "/results.txt")
    if os.path.exists(arguments.outputDirectory + "/ml_input.txt"):
        os.remove(arguments.outputDirectory + "/ml_input.txt")
    if os.path.exists(arguments.outputDirectory + "/X_database.txt"):
        os.remove(arguments.outputDirectory + "/X_database.txt")
    if os.path.exists(arguments.outputDirectory + "/y_database.txt"):
        os.remove(arguments.outputDirectory + "/y_database.txt")
    for file in glob.glob(arguments.outputDirectory + "/*"):
        shutil.rmtree(file)

    if not os.path.exists(arguments.inputDirectory):
         prettyPrint("Unable␣to␣open␣the␣input␣directory", "error")
         return False

    if not os.path.exists(arguments.outputDirectory):
        os.makedirs(arguments.outputDirectory)

    f = open(arguments.outputDirectory + "/results.txt", "a")
    f.write("
        ##############################################################################\
        n")
    f.write("#␣Start␣time:␣%s␣#\n" % (getTimestamp()))
    f.write("#␣kFold:␣%s,␣Select␣K␣Best␣Features:␣%s␣#\n" % (str(arguments.
        kfold), str(arguments.selectBest)))
    f.write("
```

```
    ##############################################################################\
    n")
f.close()

f = open(arguments.outputDirectory + "/ml_input.txt", "a")
f.write("
    ##############################################################################\
    n")
f.write("#␣INSTRUCTIONS:␣Each␣iterations␣has␣3␣inputs.␣First␣all␣(
    hybrid)␣features.␣Second␣all␣static␣features␣and␣third␣all␣dynamic
    ␣features␣#")
f.write("
    ##############################################################################\
    n")
f.close()

# important variables
app_ids = {}
app_counter = 0
f1score = 0.0
previous_f1score = 0.0

#####################################################################
# Retrieve and run all the apks for the training phase
#####################################################################
prettyPrint("Running␣all␣the␣apps␣for␣the␣Learning␣phase", "info2")
input_dir = glob.glob("%s/*" % arguments.inputDirectory)

# loop trough the goodware and malware folder
for input_folders in input_dir:
    app_type = os.path.basename(input_folders)
    prettyPrint("Now␣processing␣all␣%s␣apps" % app_type, "info2")

    allAPKs = glob.glob("%s/%s/*.apk" % (arguments.inputDirectory,
        app_type))

    if len(allAPKs) < 1:
        prettyPrint("Could␣not␣find␣any␣APK's␣under␣\"%s/%s\".␣Exiting"
            % (arguments.inputDirectory, app_type), "error")
```

```
    return False

prettyPrint("Successfully␣retrieved␣%s␣APK's␣from␣\"%s/%s\"" % (
    len(allAPKs), arguments.inputDirectory, app_type), "info2")


# Loop through all the apk files, for the first time
for apk in allAPKs:
    app_ids[app_counter] = {"path": apk, "malicious": 0, "features"
        : [], "run_counter": 0}

    if app_type == "malware":
        app_ids[app_counter]['malicious'] = 1

    apk_name = os.path.basename(apk)
    prettyPrint("Now␣working␣with␣%s" % apk_name, "info2")

    ###############################################################
    # Create the output directories
    ###############################################################
    if not os.path.exists(arguments.outputDirectory + "/" +
        apk_name):
        os.makedirs(arguments.outputDirectory + "/" + apk_name)

    # create subdir for grodddroid outputs
    if not os.path.exists(arguments.outputDirectory + "/" +
        apk_name + "/reference/grodd_output"):
        os.makedirs(arguments.outputDirectory + "/" + apk_name + "/
            reference/grodd_output")

    ###############################################################
    # Restore the snapshot and start virtual machine
    ###############################################################
    prettyPrint("Restoring␣and␣starting␣virtual␣machine", "info2")
    avdIP = restore_and_start_vm(arguments.vm_name, arguments.
        restore_snapshot)

    ###############################################################
    # Run GroddDroid
```

```
############################################################
# change the outputdirectory for the grodddroid
config = Config_handler(arguments.branchExplorer_dir + "/
    branchexp/config.ini")
config.set_output_dir(arguments.outputDirectory + "/" +
    apk_name + "/reference/grodd_output/")
config.set_max_runs('0')
config.set_device_IP(avdIP)

# start gordddroid
main_cmd_call = ["python3", "-m", "branchexp.main", apk]
result = subprocess.Popen(main_cmd_call, cwd=arguments.
    branchExplorer_dir).communicate()[0]

############################################################
# Extract features
############################################################
prettyPrint("Extracting␣numerical␣features␣...", "info2")

# features for the individual apk
app_ids[app_counter]["features"] = extract_features_from_files
    (*get_grodd_output_file_paths(
    arguments.outputDirectory + "/" + apk_name + "/reference/
        grodd_output/"))

prettyPrint("Features␣extracted␣for␣this␣specific␣app:", "info2
    ")
print(app_ids[app_counter]["features"])
print(app_ids[app_counter]["malicious"])

############################################################
# Kill the virtual machine
############################################################
prettyPrint("Finished␣running␣this␣app,␣shutting␣down␣the␣
    virtual␣machine", "info2")
shutdown_vm(arguments.vm_name)

app_counter += 1
```

```
#####################################################################
# Analyze results using an svm machine learning algorithm
#####################################################################
print("\n\n\n")
prettyPrint("Analyzing␣the␣results", "info2")

# append the features of the individual apps to the overall X and y
X, y, X_static, X_dynamic = build_machine_learning_inputs(app_ids,
    arguments.outputDirectory, False)

# hybrid features - will only be logged in the results file
do_the_machine_learning(X, y, arguments.outputDirectory + "/results.
    txt", arguments.selectBest, arguments.kfold, 0, arguments.
    outputDirectory + "/ml_input.txt", 1)

# static features - will only be logged in the results file
do_the_machine_learning(X_static, y, arguments.outputDirectory + "/
    results.txt", arguments.selectBest, arguments.kfold, 0, arguments.
    outputDirectory + "/ml_input.txt", 2)

# dynamic features - iterate depending on the metrics of them
predicted, metrics, misclassified_apps = do_the_machine_learning(
    X_dynamic, y, arguments.outputDirectory + "/results.txt",
    arguments.selectBest, arguments.kfold, 0, arguments.
    outputDirectory + "/ml_input.txt", 3)

if metrics is not None:
    try:
        previous_f1score = f1score = metrics["f1score"]
    except Exception:
        prettyPrint("Error␣with␣the␣outputs␣of␣the␣machine␣learning", "
            info2")
else:
    prettyPrint("Machine␣Learning␣did␣not␣return␣any␣metrics....
        Finishing", "info2")
    return 1

#####################################################################
# FEEDBACK CYCLE
```

```
# Check which apps to run again, with different inputs
#####################################################################
print("\n\n\n")
prettyPrint("
    ------------------------------------------------------------",
    "info2")
prettyPrint("Running␣the␣misclassified␣apps␣again␣with␣groddroid␣
    forcing", "info2")

run = True

if misclassified_apps is not None:
    if f1score == 1.0 and len(misclassified_apps) == 0:
        ############################################################
        # Exit successfully
        ############################################################
        prettyPrint("Perfect␣f1score,␣finishing␣up", "info2")
        run = False
else:
    prettyPrint("Machine␣Learning␣did␣not␣return␣any␣misclassified␣
        apps....Finishing", "info2")

feedback_run_counter = 1
while(run):
    if misclassified_apps is None:
        prettyPrint("No␣misclassified␣apps", "info2")
        return 1

    prettyPrint("Overall␣run␣number␣%s␣for␣the␣missclassified␣apps" %
        feedback_run_counter, "info2")

    for apk_index in misclassified_apps:
        apk = app_ids[apk_index]["path"]
        apk_name = os.path.basename(apk)

        prettyPrint("Now␣working␣with␣%s" % apk_name, "info2")

        ############################################################
        # Restore the snapshot and start virtual machine
```

```
############################################################
prettyPrint("Restoring␣and␣starting␣virtual␣machine", "info2")
avdIP = restore_and_start_vm(arguments.vm_name, arguments.
    restore_snapshot)

################################################################
# Run GroddDroid
################################################################
# change the outputdirectory for the grodddroid
config = Config_handler(arguments.branchExplorer_dir + "/
    branchexp/config.ini")
config.set_output_dir(arguments.outputDirectory + "/" +
    apk_name + "/improvement/grodd_output/")
config.set_device_IP(avdIP)

# set the new max number of runs grodd should do
if app_ids[apk_index]["run_counter"] == 0:
    app_ids[apk_index]["run_counter"] += 2
else:
    app_ids[apk_index]["run_counter"] += 1

config.set_max_runs(str(app_ids[apk_index]["run_counter"]))

# get the location of the stored acfg
acfg_path = None
if os.path.exists(arguments.outputDirectory + "/" + apk_name +
    "/reference/grodd_output/acfg.txt"):
    acfg_path = arguments.outputDirectory + "/" + apk_name + "/
        reference/grodd_output/acfg.txt"

# start gordddroid if app did not crash before
if acfg_path is None:
    main_cmd_call = ["python3", "-m", "branchexp.main", apk]
    result = subprocess.Popen(main_cmd_call, cwd=arguments.
        branchExplorer_dir).communicate()[0]
else:
    main_cmd_call = ["python3", "-m", "branchexp.main", apk, "--
        acfg_path", acfg_path]
    result = subprocess.Popen(main_cmd_call, cwd=arguments.
```

```
            branchExplorer_dir).communicate()[0]


    # features for the individual apk
    app_ids[apk_index]["features"] = extract_features_from_files(
        *get_grodd_output_file_paths(arguments.outputDirectory + "/
            " + apk_name + "/improvement/grodd_output/"))

    prettyPrint("Features␣extracted␣for␣this␣specific␣app:", "info2
        ")
    print(app_ids[apk_index]["features"])
    print(app_ids[apk_index]["malicious"])
    ##############################################################
    # Kill the virtual machine
    ##############################################################
    prettyPrint("Finished␣running␣this␣app,␣shutting␣down␣the␣
        virtual␣machine", "info2")
    shutdown_vm(arguments.vm_name)

#####################################################################
# Analyze results using an svm machine learning algorithm
#####################################################################
prettyPrint("Analyzing␣the␣results", "info2")

# append the features of the individual apps to the overall X and
    y
X, y, X_static, X_dynamic = build_machine_learning_inputs(app_ids,
     arguments.outputDirectory, False)

# hybrid features - will only be logged in the results file
do_the_machine_learning(X, y, arguments.outputDirectory + "/
    results.txt", arguments.selectBest, arguments.kfold,
    feedback_run_counter, arguments.outputDirectory + "/ml_input.
    txt", 1)

# static features - will only be logged in the results file
do_the_machine_learning(X_static, y, arguments.outputDirectory + "
    /results.txt", arguments.selectBest, arguments.kfold,
    feedback_run_counter, arguments.outputDirectory + "/ml_input.
```

```
    txt", 2)

# dynamic features - iterate depending on the metrics of these
predicted, metrics, misclassified_apps = do_the_machine_learning(
    X_dynamic, y, arguments.outputDirectory + "/results.txt",
    arguments.selectBest, arguments.kfold, feedback_run_counter,
    arguments.outputDirectory + "/ml_input.txt", 3)

if metrics is not None:
    try:
        f1score = metrics["f1score"]
    except Exception:
        prettyPrint("", "info2")
        prettyPrint("Machine␣Learning␣did␣not␣return␣any␣metrics....
            Finishing", "info2")
else:
    prettyPrint("Machine␣Learning␣did␣not␣return␣any␣metrics....
        Finishing", "info2")
    return 1

# check the results and whether we should run the apps again
if misclassified_apps is not None:
    if len(misclassified_apps) > 0:
        if previous_f1score > f1score:
            run = False
            prettyPrint("The␣results␣of␣the␣previous␣(Run␣%s)␣run␣
                were␣better,␣finishing␣up.." %
                       str(feedback_run_counter - 1), "info2")
        else:
            previous_f1score = f1score
    else:
        run = False
        prettyPrint("TNo␣more␣misclassified␣apps,␣finishing␣up..",
            "info2")
else:
    prettyPrint("No␣more␣misclassified␣apps,␣finishing␣up..", "
        info2")

feedback_run_counter += 1
```

```
    ########################################################################
    # Exit successfully
    ########################################################################
    prettyPrint("Finished with everything", "info2")

    # simply save the X and Y learning feature vectors
    build_machine_learning_inputs(app_ids, arguments.outputDirectory, True
        )

    f = open(arguments.outputDirectory + "/results.txt", "a")
    f.write("
        ################################################################################\
        n")
    f.write("# End time: %s #\n" % (getTimestamp()))
    f.write("
        ################################################################################\
        n")
    f.close()

    return True

def build_machine_learning_inputs(dictionary, output_dir, extend=False):
    """Build the machine learning inputs form the features of each
        individual apk"""
    X, y = [], []
    for i in range(len(dictionary)):
        X.append(dictionary[i]["features"])
        y.append(dictionary[i]['malicious'])

    # build the subsets for the static and dynmaic features
    X_static = []
    for i in X:
        X_static.append(i[-33:])

    X_dynamic = []
    for i in X:
        X_dynamic.append(i[:31])
```

```
if extend == True:
    X2 = X
    y2 = y

    if os.path.exists("X_database.txt") and os.path.exists("y_database.
        txt"):
        # try to load the whole database of features
        try:
            with open("X_database.txt", "rb") as file:
                loaded_x = pickle.load(file)

            # print("LOADED FILE x")
            # print(loaded_x)

            X2 = loaded_x
            X2.extend(X)

            # print(X2)

            with open("y_database.txt", "rb") as file:
                loaded_y = pickle.load(file)

            # print("LOADED FILE y")
            # print(loaded_y)

            y2 = loaded_y
            y2.extend(y)

            # print(y2)
        except Exception:
            pass

        # append the new featuers to the database
        with open("X_database.txt", "wb") as file:
            pickle.dump(X2, file)

        with open("y_database.txt", "wb") as file:
            pickle.dump(y2, file)
```

```
            try:
                copyfile("X_database.txt", output_dir + "/X_database.txt")
                copyfile("y_database.txt", output_dir + "/y_database.txt")
            except Exception:
                pass

            return X2, y2
        else:
            prettyPrint("No␣X␣and␣y␣database␣found,␣only␣working␣with␣this␣
                runs␣features", "info2")
            return X, y

    return X, y, X_static, X_dynamic

def get_grodd_output_file_paths(grodd_output_path):
    """Find the paths to the gradddroid output files"""
    traces_file_path = ""
    targets_file_path = ""
    to_force_file_path = ""
    stats_file_path = ""
    manifest_file_path = ""

    run_directories = os.listdir(grodd_output_path)
    run_directory = max(run_directories,key=lambda run_directories:re.
        split(r"_|\.",run_directories))

    if not os.path.exists(grodd_output_path + "/" + run_directory):
        prettyPrint("Output-path␣does␣not␣exist")
        return traces_file_path, targets_file_path, to_force_file_path,
            stats_file_path, manifest_file_path

    for root, dirs, files in os.walk(grodd_output_path + "/" +
        run_directory):
        for name in files:
            if name == "traces.log":
                traces_file_path = os.path.join(root, name)
            elif name == "targets.json":
                targets_file_path = os.path.join(root, name)
            elif name == "to_force.log":
```

```
                to_force_file_path = os.path.join(root, name)
            elif name == "stats.json":
                stats_file_path = os.path.join(root, name)

    if os.path.exists(grodd_output_path + "apktool/AndroidManifest.xml"):
        manifest_file_path = grodd_output_path + "apktool/AndroidManifest.
            xml"

    return traces_file_path, targets_file_path, to_force_file_path,
        stats_file_path, manifest_file_path

def extract_features_from_files(traces_file_path, targets_file_path,
    to_force_file_path, stats_file_path, manifest_file_path):
    """Extract features form the manifest file, traces file and the
        grodddroid output files"""

    features = []

    extractor = Extractor()

    if traces_file_path:
        prettyPrint("Extracting␣features␣from␣the␣traces␣file␣...", "info2
            ")
        features.extend(extractor.extract_traces_features(traces_file_path
            ))
    else:
        # set all the traces features to 0, so that we do not have an
            error in the machine learning
        features.extend([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                        0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                            0.0, 0.0, 0.0, 0.0, 0.0])

    if targets_file_path:
        prettyPrint("Extracting␣features␣from␣the␣targets␣file␣...", "
            info2")
        features.extend(extractor.extract_targets_features(
            targets_file_path))
    else:
```

```
        # set all the targets features to 0, so that we do not have an
            error in the machine learning
        features.extend([0.0, 0.0, 0.0])

    if to_force_file_path:
        prettyPrint("Extracting␣features␣from␣the␣to_force␣file␣...", "
            info2")
        features.append(extractor.extract_to_force_features(
            to_force_file_path))
    else:
        # set all the force features to 0, so that we do not have an error
             in the machine learning
        features.extend([0.0])

    if stats_file_path:
        prettyPrint("Extracting␣features␣from␣the␣stats␣file␣...", "info2"
            )
        features.extend(extractor.extract_stats_features(stats_file_path))
    else:
        # set all the stats features to 0, so that we do not have an error
             in the machine learning
        features.extend([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

    if manifest_file_path:
        prettyPrint("Extracting␣features␣from␣the␣manifest␣file␣...", "
            info2")
        features.extend(extractor.extract_manifest_file_features(
            manifest_file_path))
    else:
        # set all the traces features to 0, so that we do not have an
            error in the machine learning
        features.extend([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                      0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                          0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

    return features

def do_the_machine_learning(X, y, output_dir, selectBest, kfold,
```

```
iteration, output2_dir=None, feature_type=1):
"""Do the actual machine learning and return the results"""

# Classifying using K-nearest neighbors
K = [10, 15, 20, 25, 30, 35]
metricsDict = {}
missclassifiedDict = {}

tmpPredicted = [0] * len(y)
for k in K:
    prettyPrint("Classifying␣using␣K-nearest␣neighbors␣with␣K=%s" % k)
    predicted = predictKFoldKNN(X, y, K=k, kfold=int(kfold),
        selectKBest=int(selectBest))

    for i in range(len(predicted)):
        tmpPredicted[i] += predicted[i]

    metrics = calculateMetrics(y, predicted)
    metricsDict["KNN%s" % k] = metrics

    tempMissclassified = get_indices_of_misclassified_apps(y,
        predicted)
    missclassifiedDict["KNN%s" % k] = {"missclassified":
        tempMissclassified, "#crashed_apps": len(tempMissclassified)}

# Classifying using Random Forests
E = [10, 25, 50, 75, 100]
for e in E:
    prettyPrint("Classifying␣using␣Random␣Forests␣with␣%s␣estimators"
        % e)
    predicted = predictKFoldTrees(X, y, kfold=int(kfold), selectKBest=
        int(selectBest), estimators=e)

    for i in range(len(predicted)):
        tmpPredicted[i] += predicted[i]

    metrics = calculateMetrics(y, predicted)
    metricsDict["Trees%s" % e] = metrics
```

```
        tempMissclassified = get_indices_of_misclassified_apps(y,
            predicted)
        missclassifiedDict["Trees%s" % e] = {"missclassified":
            tempMissclassified, "#crashed_apps" : len(tempMissclassified)}

    # Classifying using SVM
    prettyPrint("Classifying␣using␣Support␣vector␣machines")
    predicted = predictKFoldSVM(X, y, kfold=int(kfold), selectKBest=int(
        selectBest))

    for i in range(len(predicted)):
        tmpPredicted[i] += predicted[i]

    metrics = calculateMetrics(y, predicted)
    metricsDict["svm"] = metrics

    tempMissclassified = get_indices_of_misclassified_apps(y, predicted)
    missclassifiedDict["svm"] = {"missclassified": tempMissclassified, "#
        crashed_apps": len(tempMissclassified)}

    # Average the predictions in tempPredicted
    predicted = [-1] * len(y)
    for i in range(len(tmpPredicted)):
        predicted[i] = 1 if tmpPredicted[i] >= 12.0 / 2.0 else 0 # 12
            classifiers

    metricsDict["all"] = calculateMetrics(predicted, y)
    metrics = metricsDict["all"]

    missclassified_apps = get_indices_of_misclassified_apps(y, predicted)
    missclassifiedDict["all"] = {"missclassified": missclassified_apps, "#
        crashed_apps": len(tempMissclassified)}

    # Print and save the results:
    feature_type_string = "\n\n"
    if feature_type == 1:
        feature_type_string += "HYBRID␣FEATURES\n"
    elif feature_type == 2:
        feature_type_string += "STATIC␣FEATURES\n"
```

```
elif feature_type ==3:
    feature_type_string += "DYNAMIC␣FEATURES\n"

for m in metricsDict:
    # The average metrics for training dataset
    prettyPrint(feature_type_string, "info2")
    prettyPrint("Metrics␣using␣%s-fold␣cross␣validation␣and␣%s" % (
        kfold, m), "info2")
    prettyPrint("Accuracy:␣%s" % str(metricsDict[m]["accuracy"]), "
        info2")
    prettyPrint("Recall:␣%s" % str(metricsDict[m]["recall"]), "info2")
    prettyPrint("Specificity:␣%s" % str(metricsDict[m]["specificity"]),
         "info2")
    prettyPrint("Precision:␣%s" % str(metricsDict[m]["precision"]), "
        info2")
    prettyPrint("F1␣Score:␣%s" % str(metricsDict[m]["f1score"]), "
        info2")

    # Log results to the outfile
    f = open(output_dir, "a")
    f.write(feature_type_string)
    f.write("
        ##############################################################################\
        n")
    f.write("#␣Metrics:␣algorithm:␣%s,␣iteration␣%s,␣timestamp:␣%s␣#\n
        " % (m, iteration, getTimestamp()))
    f.write("
        ##############################################################################\
        n")
    f.write("Validation␣-␣accuracy:␣%s,␣recall:␣%s,␣specificity:␣%s,␣
        precision:␣%s,␣F1-score:␣%s,"
            "␣Misclassified␣apps:␣%s,␣Number␣of␣misclassified␣apps:␣%s\
                n\n" % (
    metricsDict[m]["accuracy"], metricsDict[m]["recall"], metricsDict[
        m]["specificity"],
    metricsDict[m]["precision"], metricsDict[m]["f1score"], str(
        missclassifiedDict[m]["missclassified"]),
    str(missclassifiedDict[m]["#crashed_apps"])))
    f.close()
```

```
    if output2_dir is not None:
        # Log the inputs of the machine learning into a file
        f = open(output2_dir, "a")
        f.write(feature_type_string)
        f.write("
            ##########################################################################\
            n")
        f.write("# Machine Learning input: iteration %s, timestamp: %s #\n
            " % (iteration, getTimestamp()))
        f.write("
            ##########################################################################\
            n")
        f.write("X input: %s \n\n y input: \n %s \n\n" % (str(X), str(y)))
        f.close()

    return predicted, metrics, missclassified_apps


def restore_and_start_vm(vm_id, snapshot_name):
    """Restore the snapshot of the virtual machine and launch it"""

    args_snap = ['vboxmanage', 'snapshot', vm_id, 'restore', snapshot_name
        ]
    prettyPrint("Restoring snapshot \"%s\"" % snapshot_name, "debug")
    result = subprocess.Popen(args_snap, stderr=subprocess.STDOUT, stdout=
        subprocess.PIPE).communicate()[0]
    attempts = 1

    result = str(result)
    while result.lower().find("error") != -1:
        print(result)
        # Retry restoring snapshot for 10 times and then exit
        if attempts == 25:
            prettyPrint("Failed to restore snapshot \"%s\" after 10
                attempts. Exiting" % arguments.vmsnapshot, "error")
            return False
        prettyPrint("Error encountered while restoring the snapshot \"%s\".
            Retrying ... %s" % (snapshot_name, attempts), "warning")
```

```
    # shut down the vm on virtual box
    args = ['vboxmanage', 'controlvm', vm_id, 'poweroff']
    subprocess.Popen(args, stderr=subprocess.STDOUT, stdout=subprocess.
        PIPE).communicate()[0]

    # Now attempt restoring the snapshot
    result = subprocess.Popen(args_snap, stderr=subprocess.STDOUT,
        stdout=subprocess.PIPE).communicate()[0]

    result = str(result)
    attempts += 1
    time.sleep(1)

# Start the Genymotion Android virtual device
prettyPrint("Starting␣the␣Genymotion␣machine␣\"%s\"" % vm_id, "debug")

args = ["/opt/genymobile/genymotion/player", "--vm-name", vm_id]
genyProcess = subprocess.Popen(args, stderr=subprocess.STDOUT, stdout=
    subprocess.PIPE)
prettyPrint("Waiting␣for␣machine␣to␣boot␣...", "debug")
time.sleep(23)

# Retrieve the IP address of the virtual device
getAVDIPCmd = ["VBoxManage", "guestproperty", "enumerate", vm_id]
result = subprocess.Popen(getAVDIPCmd, stderr=subprocess.STDOUT,
    stdout=subprocess.PIPE).communicate()[0]
result = str(result)
result = result.replace('␣', '')
if result.lower().find("error") != -1:
    prettyPrint("Unable␣to␣retrieve␣the␣IP␣address␣of␣the␣AVD", "error
        ")
    print (result)
index = result.find("androvm_ip_management,value:")+len("
    androvm_ip_management,value:")
avdIP = ""
while result[index] != ',':
    avdIP += result[index]
    index += 1
```

```
    adbID = "%s:5555" % avdIP

    print("\n\n")
    print(str(avdIP))
    print("\n\n")

    return str(avdIP)

def shutdown_vm(vm_id):
    """Shutdown the virtual machine"""

    args = ["/opt/genymobile/genymotion/player", "--poweroff", "--vm-name"
        , vm_id]
    subprocess.Popen(args, stderr=subprocess.STDOUT, stdout=subprocess.
        PIPE).communicate()[0]

    if genyProcess:
        genyProcess.kill()

    time.sleep(2)

    # shut down the vm on virtual box
    args = ['vboxmanage', 'controlvm', vm_id, 'poweroff']
    subprocess.Popen(args, stderr=subprocess.STDOUT, stdout=subprocess.
        PIPE).communicate()[0]

    time.sleep(2)

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