DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Comparing Application Stimulation Techniques for Repackaged Malware Detection in Android

Emir Demirdag

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Comparing Application Stimulation Techniques for Repackaged Malware Detection in Android

Vergleich von Anwendungssimulationstechniken für die Erkennung Wiederverpackter Android Schadprogramme

Author:	Emir Demirdag
Supervisor:	Prof. Dr. Alexander Pretschner
Advisor:	M.Sc. Aleieldin Salem
Submission Date:	15.05.2018

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

Garching, 15.05.2018 Emir Demirdag

Acknowledgments

I would like to thank my advisor Aleieldin Salem for his never-ending patience and constant assistance during the making of this thesis. I would also like to thank the Chair of Software Engineering at the Technical University Munich, which made it possible for me to write this thesis.

Abstract

As we migrate our daily tasks to our smartphones, attackers become more and more eager to write and distribute malware applications. Repackaging (Piggybacking) is one of the techniques adopted by authors of Android malware to avoid detection by decompiling an application, injecting malicious segments, and serving it back to the world. To detect this kind of malware, there needs to be mechanisms that stimulate the malicious parts within repackaged applications. Otherwise, the classified behavior of the application will not depict the true nature of the application itself. This will result in a false diagnosis, which is classifying a malicious application as benign, having only looked at the benign parts of the application. While there have been efforts for devising different stimulation techniques to analyze the true nature of repackaged malware, the lack of ground truth to detect hidden malicious segments throughout benign parts raises a concern. In this thesis, we will approach this lack of ground truth as a search problem, finding our way into the malicious segments of Android applications. To do this, we will compare two stimulation techniques, Random Stimulation Technique versus Forcing Execution Stimulation Technique; using a large dataset which has not been used to test these stimulation techniques before. We will be using Droidutan for Random Stimulation Technique, and GroddDroid for Forcing Executing technique. These two stimulation techniques will be used to generate runtime behaviors (i.e., in the form of API call traces) of Android benign and malicious applications, from which we will extract numerical features for classification using Droidmon, a tool that helps us monitor API calls. We will use machine learning classifiers, and will adopt an active learning setting to provide the stimulation methods with feedback about their performance. Lastly, we will evaluate these tools using the Piggybacked dataset and reach a conclusion.

Contents

1. Introduction

With 80% of the market share, Android mobile operating system attracts the most attention from malware authors. To cope with such attacks, researchers typically rely on various techniques (e.g. machine learning) to extract characteristic information about an application (hereafter app or apps) to detect malware. While these techniques continue to be addressed as significant by the research community, malware authors keep finding ways to escape detection of such tools developed using these techniques [13]. One of the ways that the malware authors have recently been adopting is referred as *piggybacking*, or *repackaging*. What makes this applicable for malware authors is that, in essence, Android apps are distributed as software packages (i.e. APK files) which includes developer bytecode, resource files, and a Manifest in XML format, presenting the essential information about an app: such as package name, permissions requested, list of components that the system makes use of before running the app [10]. In essence, this APK file is easily decomposable using various tools, for instance *apktool*, which makes the app's source code easily modifiable [25]. Malware authors leverage this ease of decompiling, build on top of the app, and redistribute the app with injected malicious segments. In the case that this app is a popular app, attackers ensure a wide diffusion of their malicious code within Android ecosystem [10]. Hereafter, we will address this effective and sophisticated technique as *piggybacking* and *repackaging* interchangebly.

Inside a repackaged app, a malware can have triggering protections. That is to say, it can wait for an event or expect a specific value for a variable before getting triggered [2]. The research community has tackled this detection problem by using static and dynamic approaches to **stimulate** applications and extracting features or logs, which are then analyzed with the help of detection algorithms [14, 2]. It was to our finding that different techniques of stimulation exist and they are proved to be useful in several Android malware analysis scenarios, such as [18, 26, 2], especially when malicious segments are protected by from getting triggered by malware analysis tools. Thus, in our opinion, stimulation techniques are valuable for further investigation.

Therefore, in this thesis, we will focus on comparing two stimulation techniques, Random Stimulation Technique versus Forcing Execution Technique. We propose the research efforts on these stimulation techniques as follows:

Firstly, we offer to test both stimulation techniques on a large dataset since they have only been tested using small, or outdated datasets [28]. Secondly, we acknowledge the problem of lack of ground truth in stimulation techniques, which is not knowing which

path, branch, or segment of the app was executed before the app itself was classified as benign or malicious. We propose that malicious apps can be classified as benign, if the classification was based on benign parts of the app. As a result, we offer to approach this situation as a search problem: we propose to execute almost all branches inside an app; determining a stopping point via machine learning classifiers to make the process faster. We then evaluate the results, comparing Random Stimulation Technique via Forcing Execution Technique. We will get the data for the Random Stimulation Technique from our advisor, while one can find the details how Forcing Execution evaluation is done in the following chapters.

1.1. Motivation

The main motivation behind our research is to being able to correctly classify repackaged malware as malicious or benign while pursuing the best method to do it. In our opinion, readers of this thesis may benefit keeping this main point in mind while reading the remaining important points to us, which can be found in following section.

1.1.1. Societal Impact

Societal impact of a repackaged app could result in disastrous consequences. That is to say, repackaging is one of the most common techniques adopted by malware authors to exploit users' trust and avoid detection. Since repackaged malware is, in essence, injecting malicious payloads into popular applications, and re-uploading back to the available markets; malware authors may locate and download popular apps, decompile them, enclose malicious payloads, and then serve it back to the world by submitting the new app to official Google Play Store and/or alternative Android Markets. Users, on the other hand, left with an illusion that they are downloading the famous Angry Birds, or the mobile banking app of a trusted bank. Needless to say, this vulnerability could become dangerous when users keep installing these infected apps [27]. The analogy of the Trojan Horse could be made here, as in [21, 14], since repackaged malware has a similar concept. Attackers want to invade users' phones by tricking the user to download the malicious code, in other words, "let the enemy inside the gates." Furthermore, repackaged malware authors can choose a variety of ways for leaking into user's mobile device; including but not limited to paid apps, popular game apps, powerful utility apps, and also security updates [28].

1.1.2. Technical Difficulty

One of the most important points that led us to this research is how a repackaged malware lives inside an app. To show this, consider such Control Flow Graph(hereafter

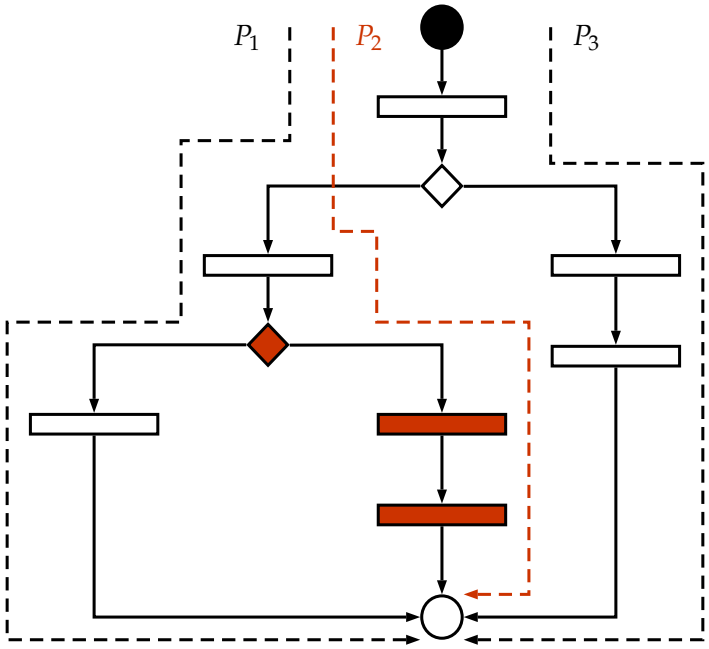

Figure 1.1.: Control Flow Graph of a repackaged app (a_i). The colored segment depicts the malicious payload injected into the app, whereas the dashed arrows represent different paths through the CFG [21].

CFG) in Figure 1.1. This CFG depicts the possible paths inside a piggybacked app (a_i) that has been injected with malicious segments (colored blocks).

Suppose we employ a stimulation technique to target and execute branches inside this app, a_i. To classify this app as malicious, one has to be able to devise test cases that execute P_2 -the malicious segment-; so that the malicious code will execute and the app would be classified as malicious. The feature vectors extracted from P_2 (e.g., API call trace) will depict the true nature of the app, leading us to the correct diagnonis and helping the trained classifier to classify this app as malicious. However, since the location of the injected malicious payload is unknown, and since the malicious payload is assumed to be smaller than the original benign code, the stimulation component is likely to target other branches/statements within the CFG, effectively executing other paths (i.e., P_1 or P_3). Feature vectors extracted from such paths are expected to represent benign behaviors, leading to the misclassification of the app. This misclassification will continue until a malicious path, such as P_2 is executed, and the stimulation component will continue to feed the classifier with benign path until such path is discovered.

The next section will cover how we will approach these problems, and how we tackle the problem of making the whole process more scalable.

1.2. Proposed Solutions and Objectives

As stated before, our main objective throughout this research is to correctly classify apps as benign or malicious. The biggest roadblock ahead of this goal is not knowing which path the malicious segment might belong. We found that, the most convenient way to circumvent this roadblock is to be able visit every path (e.g., paths in a CFG) inside an app.

Therefore, we argue that the problem of stimulating, analyzing, and detecting Android repackaged malware is a search problem. We have paths that include malicious behavior; to later extract features that will depict the app's true nature. Finding such paths, in other words malicious segments or branches, will allow us to train our classifier by having a more accurate view of the app's expected behavior. Furthermore, as a part of this search problem, again without exactly knowing the app's true nature, we need to execute multiple paths to increase the probability of finding a representation of the injected malicious segment. We also claim that, to approach this lack of ground truth as a search problem, we need active learning to tackle this problem for the following reasons. Needless to say, different repackaged malware could exhibit different behavior. That is; locations, behaviors and the nature of malicious segment can vary throughout different kinds of repackaged malware. Moreover, since we lack the ground truth about the malicious segments that an app might exhibit prior to executing different branches, we need to adopt a trial-and-error technique. In other words, we need be able to stop at a certain point before executing all the branches within an app, which would be significantly time and resource consuming.

That is to say, if we have n number of apps to execute and examine, and each app has m paths, that will leave us with m^n combinations of paths to execute.

$$p = m^n$$

If we decide to run our experiment several times, perhaps to represent more than one statistical combination -which we will do in our case-, then we need to explore p number of paths; every time we run the experiment. Needless to say, this is not ideal.

To escape this possibility, we aim to be able stop at a certain point. We assume that, in the training phase; if a feature vector (x_i) extracted from app (a_i) is misclassified, such feature vector represents the execution path that does not reflect the apps true nature (i.e., malicious or benign). Thus, the app (a_i) needs to be re-executed to explore another path that yields a feature vector (x_i) that depicts the app's true nature. This process can be iterated until the maximum training accuracy is achieved, which signals the best possible representation of benign and malicious behaviors embedded in the training apps.

We argue that, in a repackaged malware app, the number of re-stimulations to get the best training accuracy is equal to number of paths within which the app's true

nature will unveil. In other words, this is the number of executions within which the malicious behavior is likely to exhibit. After finding out such a number - X, and after extracting feature vectors from X number of apps, these feature vectors can be used to classify the app as malicious or benign using different methods, such as majority votes or one instance techniques, which we will explain in the later chapters.

This aforementioned trial-and-error technique, which is referred as active learning in Machine Learning context, will help us execute our plan of finding out this number and help us correctly classify the apps, and re-run the misclassified apps.

Therefore, in this thesis,

- We modify the novel architecture, Aion [19], to stimulate, analyze, and detect Android repackaged malware using active learning and compare the stimulation techniques using this tool.
- We evaluate and interpret the gathered data, and devise a comparison interpretation versus Forcing Executing Stimulation Technique and Random Stimulation Technique.
- We evaluate the stimulation techniques on the recently analyzed and released Piggybacking dataset [21].

Overall, our motivation behind this research could be gathered under the following points:

- We need to classify the app as malicious or benign.
- We assume that, an app (a_i) should be classified as malicious only if we have a representation of P_2.
- We need to know which path that our diagnosis was based on.
- We should be able to tell if our classifier is learning correctly.

1.3. Literature Review

In the context of repackaged malware, several efforts has been attempted by the research community, which also motivated us to pursue our research. We will now present the related efforts that has been done on repackaged malware, or similar topics.

Primarily, our work is largely based on Aion, a framework for analysis, stimulation, and detection of Android repackaged malware, which was written by our advisor Aleieldin Salem [21, 19]. We adopt the main idea from Aion for our methodology, which will be explained in the Methodology chapter.

In terms of the adoption of repackaged malware by malware authors; in [28], Zhou et al. studied 1260 Android malware instances, and concluded that more than 86% of them were repackaged. In 2015, TrendMicro reported that 77% of the top free 50 apps on the Google Play marketplace had fake versions, with a total of 890,482 fake apps being discovered 51% of which were found to exhibit unwanted and/or malicious behaviors. More recently, Li et al. managed to gather piggybacked versions of around 1,400 legitimate apps [21].

To our knowledge, there are multiple efforts on stimulation techniques, analyzing and detecting Android Repackaged Malware. That is, research community has been attempting to devise techniques that force malicious segments within Android apps to run, such as ([11, 16, 18]). However, either they have been evaluated using small datasets, outdated datasets (such as [28]), or not been evaluated on malware at all.

Related to our methodology, to the best of our knowledge, there have been also attempts that uses the method of extracting numerical features for training a machine learning classifier. In these cases, the trained classifier is used to classify test apps as malicious and benign. More detailed information on this background can be found in [21].

1.4. Organization

This thesis is structured as the following: Introduction, Background, Methodology, Implementation, Evaluation, Conclusion and Further Work. The current chapter, Introduction, intends to introduce the topic of repackaged malware and comparison of stimulation techniques, as well as our approach to problem the problem at hand and solution. The next chapter, Background, will cover all the background information necessary to get a better grasp of topics that will be mentioned in the rest of the document. Therefore, in the Background chapter, we will cover the roots of Android Repackaged Malware. After that, we will cover stimulation techniques that will be used during this thesis, and give brief information about different stimulation techniques and tools. We will especially concentrate on GroddDroid, which we will use as a tool that falls under the category of Forcing Execution, while briefly covering Random Stimulation Technique and Droidutan, which we will compare with GroddDroid. Following that, we will give background information about active learning, which, we believe, is essential to our research. The consecutive chapter will be covering our methodology, focusing on how we plan to use each tool combined with active learning, and how we apply our approach of classifiying apps. The next chapter, Implementation, will focus on the details of the implementation of the framework Aion, and the modifications we have done on Aion to make it work with GroddDroid. This chapter will cover all the tools used and cover will be briefly touch the parts of the code to give the readers of thesis a better understating also at the code level. The next chapter, Evaluation, will review the results drawn from our experiment, and will try to answer the research questions that

will be presented in the Evaluation chapter. We will conclude this thesis with a "Conclusion" section and future works that may be done to further extend our research, if need be.

2. Background

We now aim to give the readers of thesis the necessary background information to get a better understanding of our experiment. We start by giving a brief introduction of Android malware with the focus of repackaged malware, then on different stimulation techniques, which will go into more detail in GroddDroid, and then we will touch the surface about machine learning, with a focus of active learning, which is an essential part of our experiment.

2.1. Android Malware

Android was introduced by Google in 2007. Since then, it has become undoubtedly the most popular mobile operating system, with a market share of 80% [13]. Being the the market leader in mobile operating systems, it has been getting more and more attention from malware authors every year [14]. Its widespread distribution and wealth of application distribution channels besides the official Google Play Store also make it the undisputed market leader when it comes to mobile malware. Studies show that as many as 97% of mobile malware families target Android [13].

Needless to say, Google has been working on a counter-research to be able to defend its operating system against malware authors. The biggest reaction of Google was introducing Bouncer, a service that checks applications submitted to Google Play Store for malware. After the introduction of Bouncer, Google reported that this service led to a significant decrease of malware, nearly 40% [13].

However, attackers try to avoid detection by taking alternative routes. Firstly, one of the most common practices is serving from alternative markets; rather than official Google Play Store, where the automated checks may differ than Google's Bouncer. In fact, studies show that alternative markets host malicious applications up to 5-8% [13].

On the other hand, in [4], Yajin Zhou and Xuxian Jiang studied a dataset of 1260 malware instances and concluded that most instances are only an adaptation of a malware family, therefore many of them have a lot of identical characteristics. They categorize these malware families up to 49 different sets. Another interesting finding of this paper is that the official Google Play Store host the most number of malware, rather than alternative marketplaces.

Secondly, malware authors also adopt as a common practice when employing malware, *repackaging* or *piggybacking*, which we will go into more detail in the following subsection. Although there are many more types of attacks which attackers could

employ, which can be seen in detail in the paper [13], we will focus on repackaged malware since our research is directly related to repackaged malware. We explain, how we think our research could be adopted to different kinds of malware in the Further Work chapter.

2.1.1. Repackaged Malware

The main reason we are concentrating on this specific breed of malware is that the process of repackaging is fairly straightforward, and one might say "uncomplicated". As discussed in the Introduction section, Android apps are distributed as software packages (i.e., APK files, which are actually archives in the ZIP format) that include developer bytecode (i.e., DEX files), resource files and a manifest file in XML format, which, in essence, outlines the how the app behaves and what it needs to execute, as well as essential information like package name, permission requested and the list of components. These software packages that are downloaded from any marketplace can be decompiled easily (for instance using *apktool* [25]), to get this essential information. Malware authors then build on top of this package to spread their malware.

The other equally important reason is that repackaged malware authors can hide their malware really well. To avoid getting detected by several analysis and detection methods (i.e. Google's Bouncer), malware authors hide their malicious payloads by wrapping them with conditions that infrequently evaluate to true. Such conditions, also known as triggers, are manually designed by the malware author, and usually depend on system properties (e.g., date, time, GPS location, etc.), app/system intents and notifications (e.g., android.intent.action.BOOT COMPLETED), custom values forwarded to the app via its authors (e.g., via SMS messages), or a combination of those [10, 16]. In simplistic terms, a malware in a repackaged app can live inside the app, much like a Trojan horse, for days, before getting triggered.

Consider the example demonstration given by the authors of GroddDroid in [1], which we will get into more detail in the next sections. The malware encrypts the user files (ransomware) and is packaged with a morpion game. The malicious behavior is triggered when the user wins 10 levels of the game, and not before. This makes the executing, therefore detecting, the malware a difficult task.

Consequently, in order to be able to analyze and detect this breed of malware, there needs to be mechanisms that **stimulate** those dormant, malicious segments within repackaged apps. Otherwise, the resulting runtime behavior of the malware instances under test will not depict the true nature of the app. That is, automated detection mechanisms will not be able diagnose the repackaged malicious behavior as correctly.

2.2. Stimulation Techniques

The primary goal of stimulating a form of software is to get an application to reveal all its functionality, whether it is for detecting malicious behavior or for another reason like quality assurance testing - i.e. behavior testing, bug finding, stress testing and so on. Stimulating can be done in several ways: *staticly*, by analyzing the source code of an application; *dynamically*, by running the app on a real (or virtual) environment and interact with it, or in hybrid, which is a combination of both [2, 14].

In the context of *Android Malware Detection*, there has been several efforts of stimulation techniques, either by the research community or the industry. Google introduced *UI/Application Exerciser Monkey*, which is a program that runs on a virtual device or real device and generates "pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events" [9]. Needless to say, this technique, which is optimal for stress testing, is not ideal for stimulating every aspect of an application to detect malware. As in the cases of Figure 2.1 and Figure 2.2, where there is a conditional that drives the analysis tool away from executing the malicious code, *UI/Application Exerciser Monkey* will not be able to reach the malicious part of the code. However, there are some efforts of stimulation techniques by the research community where reaching the malicious code is possible than randomly interacting with the app. We now categorize stimulation techniques under three titles, which are essential background information and the foundation of the motivation of this thesis.

```
if (isEmulatorOn()) {
    // execute non-malicious code
} else {
    // execute malicious code
}
```

Figure 2.1.: Example conditional that drives away from malicious code [14, 2]

```
targetYear = 2020;
if (currentYear < targetYear) {
  // execute malicious code
} else {
  // execute non-malicious code
}
```

Figure 2.2.: Example conditional that drives away from malicious code [14, 2]

2.2.1. Random Stimulation Technique

First stimulation technique we will use to stimulate applications in thesis is, what we called, *Random Stimulation Technique.* Much like its name suggests, Random Stimulation Technique randomly interacts with the app, stimulating different branches randomly. Since the goal of a stimulation engine is to reach different paths even if it has made unreachable initially by the malware author; random stimulation based tools retrieve apps' UI Elements and interact with each element randomly, covering different branches. That being said, Random Stimulation Technique can be also used in different scenerios in addition testing the app's UI, such as testing applications. It is especially ideal for scenerios where the application owner/authors want to test if the application is compatible for different devices, testing the app "as is" on commodity devices. Futhermore, in malware analysis, random stimulation technique is useful where some malicious apps apply sandbox detection which might lead to different behaviors on instrumented testing devices and real devices [12].

Several example tools for this stimulation technique is **Droidutan**, and Droidbot; as well as Monkey Runner. We will explain *Droidutan* in detail in the next sections, and we also briefly explained Monkey in the previous section. As for Droidbot, we will use the official explanation by its authors in [12]: "DroidBot is a lightweight UI-guided test input generator, which is able to interact with an Android app on almost any device without instrumentation. The key technique behind DroidBot is that it can generate UI-guided test inputs based on a state transition model generated on-the-fly, and allow users to integrate their own strategies or algorithms. DroidBot does not require app instrumentation, thus doesn't cause inconsistency between the tested version and the original version."

2.2.2. Forcing Execution Technique

Second stimulation technique we will use to stimulate application in this thesis is the Forcing Execution Stimulation Technique. The goal of this technique is to stimulate different segments of the source code to force executing branches, the goal being the high coverage of branches. In malware analysis, this can be seen as executing suspicious part of the code, aiming to detect and analyze malicious parts. Primary obstacle this technique addresses are the code segments, supposedly malicious and injected by the malware author, that draw the stimulation engine away from execution (e.g. conditional statements). Thus, the stimulation tool that is using forcing execution technique aims to cancel such segments, overcoming the problem of malicious code not executing.

To our knowledge, the research community has taken several approaches to make use of this technique. In our thesis we will use **GroddDroid** [2], which we will go into more detail in the following sections. In a nutshell, GroddDroid took an approach of identifying the suspicious parts of the bytecode and compute a score (indicator of

risk) for each function of the malware, and interacting with the Graphical User Interface (hereafter GUI) with its own implemented stimulator. After that, it identifies the remaining parts of the malware that has not been executed and we force the required control flow statements to push the flow to the unexecuted parts previously scored [2].

On the other hand, one -supposed- disadvantage of this technique, in our opinion, is the modification on the code; such as canceling conditional statements to force execution on parts of the code. This differs from random stimulation technique, where no modification in the source code or byte code is made.

2.2.3. Adaptive Stimulation Technique

Third and final technique, which we will **not** experiment with in this thesis is Adaptive Stimulation Technique. This technique is based on adapting or altering the environment of the app to stimulate different parts of the code, thus detecting malware. The motivation behind this technique is that some malware exhibit their maliciousness only when being executed in a particular environment. For instance, findings of [18] say that, "some apps check whether they are running in an emulator or another analysis environment, and behave benignly in these cases. Other malware apps target specific countries and remain harmless unless the SIM card in the victims phone is registered in one of the target countries. Yet another kind of malware targets devices with a specific app installed, such as a vulnerable banking app." Therefore, the main idea is generating an Android execution environment where an app would express its maliciousness as clearly as possible.

We will not go into detail of this technique as we will not use this technique in our experiments. However, some research efforts have been done on this technique, such as tools like *FuzzDroid* [18], and *IntelliDroid* [26]. More detailed information on both tools can be found in the aforementioned references.

2.3. Tools Used for Stimulation

We will now explain the tools more in depth, tools that we will use in our experiments to compare the two stimulation techniques: *Forcing Execution Technique*, which we will test using **GroddDroid**; and *Random Stimulation Technique*, by **Droidutan**.

2.3.1. Droidutan

Droidutan (Android + Orangutan) is a simple Python API built to test Android apps, written at the Sofware Engineering Chair at Technical University of Munich, by our advisor, Aleieldin Salem. The source code of this API can be found on [20].

The API is built on top of AndroidViewClient [5]. Droidutan aims to emulate the interaction between a user and the app. It is non-invasive, meaning that a modification

in the source code of the app itself is not made. The tool starts the main activity of an app, retrieves its UI elements, chooses a random element out of the retrieved ones, and interactes with it. We categorize this tool under Random Stimulation Technique, however randomness on this tool differs from monkey-based tools. That is, Droidutan does not randomly load events to trigger to the app, but retrieves all the UI elements of the main activity (and following activies) and interacts with such elements one at a time. The API also broadcasts intents to the app and perform actions like swiping, touching random coordinates, and so on [20].

Droidutan also automatically restarts the main activity of an app in case of a crash. However, it is not ideal for bug-finding purposes, unlike the Monkey Runner [9]. While interaction is being made, it uses Droidmon [6], to monitor any API calls being made using a hook file.

While more detailed work on Droidutan can be found on [21], its workflow can be summarized in a brief manner like the following as described in [20]:

1. Starts the main activity of the app.
2. Retrieve app's UI elements
3. Choose a random element out of the retrieved ones
4. Start interaction with the retrieved element

That is to say, if the element retrieved is Radio Button, Droidutan will select the Radio Button; if it is a button, it will tap it; if it is a Text Field, random text will be typed into it.

2.3.2. GroddDroid

Abraham et al. in [2] developed GroddDroid in *python 3* and *Java*. Primarily motivated for malware analysis in mind, the tool automatically triggers and execute suspicious parts of the code. According to its authors, GroddDroid's goal is to take as input an application, run it on a real smartphone and modify as few as possible the control flow of the application in order to force the execution of the suspicious part of the code.

The starting point of GroddDroid was based on the assumption that although the static and dynamic approaches when detecting malware is promising, they stay limited to observe malicious behavior when used in turns, or one approach alone. As we covered in the previous chapters, malware authors have resources to drive the stimulation tool away to avoid getting detected.

GroddDroid authors claim that, GroddDroid does not care about covering the benign part of the code, but after identifying malicious parts of the code, they run a normal execution to trigger these parts of the code. If it has not been triggered by the normal execution, then GroddDroid forces the flow of execution in order to reach this

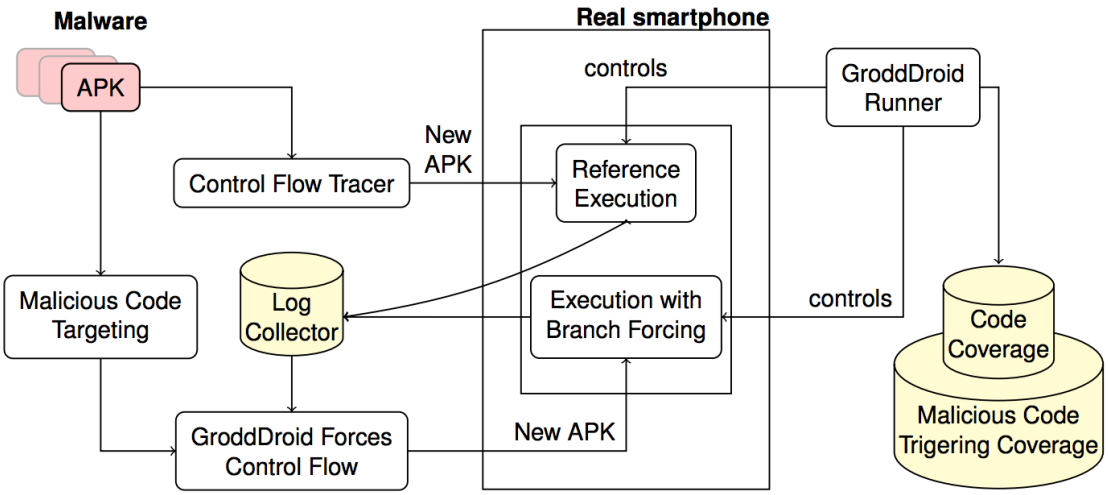

Figure 2.3.: GroddDroid framework overview [2].

part during subsequent executions. As a result, GroddDroid combines static analysis with dynamic analysis [2].

We now give an overview of how the framework works.

Workflow of GroddDroid

Overview of the GroddDroid framework is depicted by its authors in the Figure 2.3. In [2], they describe their approach in three steps as the following:

1. **Control Flow Tracer:** Firstly, GroddDroid examines the suspected application to observe its behavior. This is done via a reference execution. This examines enables the tool learn about which branches of the execution are taken during a run. After GroddDroid runs the application on a real devices, or on an emulator, it gets a reference execution. If reference execution suggests that there is nothing to suspect about this application, the tool stops after the reference execution.

2. **Malicious Code Targeting:** Secondly, GroddDroid identifies the possible malicious code inside the malware using a static analysis of its bytecode.

3. **Generating the New APK:** Thirdly, GroddDroid makes use of the execution log of the reference execution to determine which control flow has to be forced to reach the parts of the code identified as malicious. A new APK is produced where the control flow is modified accordingly. The GroddDroid runner executes the new APK on the device and new logs are generated. This step can be repeated for processing all the malicious parts of the identified code.

Inputs and Outputs

Inputs and outputs of GroddDroid are important for our work since we will use -and modify- GroddDroid in our experiments. Before we do that, we can summarize the work of GroddDroid in the following steps as in [1], to get a better picture of the use of inputs and outputs:

1. Load the APK file
2. Extract information from the AndroidManifest.xml file
3. Tag all the branches and the beginnings of methods
4. Set a score risk for methods using the suspicious2.json file
5. Identify target tags to force
6. Run the instrumented APK without forcing any branch
7. If the number of runs equal to 1, there will be no branch forcing. So, to force any branches in the APK, the number of runs must be greater than 1. Then, each run of the instrumented APK is done by forcing all necessary branches to execute one target method.

By default, GroddDroid takes the following inputs [1]:

- *apk_path:* path to the APK
- *–device DEVICE:* name of the device to use
- *–device-code DEVICE_CODE:* device code to use
- *–run-type RUN_TYPE:* type of automatic run to do
- *–max-runs MAX_RUNS:* maximum limit on number of runs
- *–output-dir OUTPUT_DIR:* output directory of run_# subdirs

GroddDroid will have the following output as a directory for each run [1]:

- *all_tags.log:* List of all tags
- *blare.log:* Blare log (if activated)
- *seen_tags.log:* Tags that are seen in this run
- *suspicious.log:* list of suspected methods and their risk scores
- *targets.json:* list of suspected statements and their risk scores

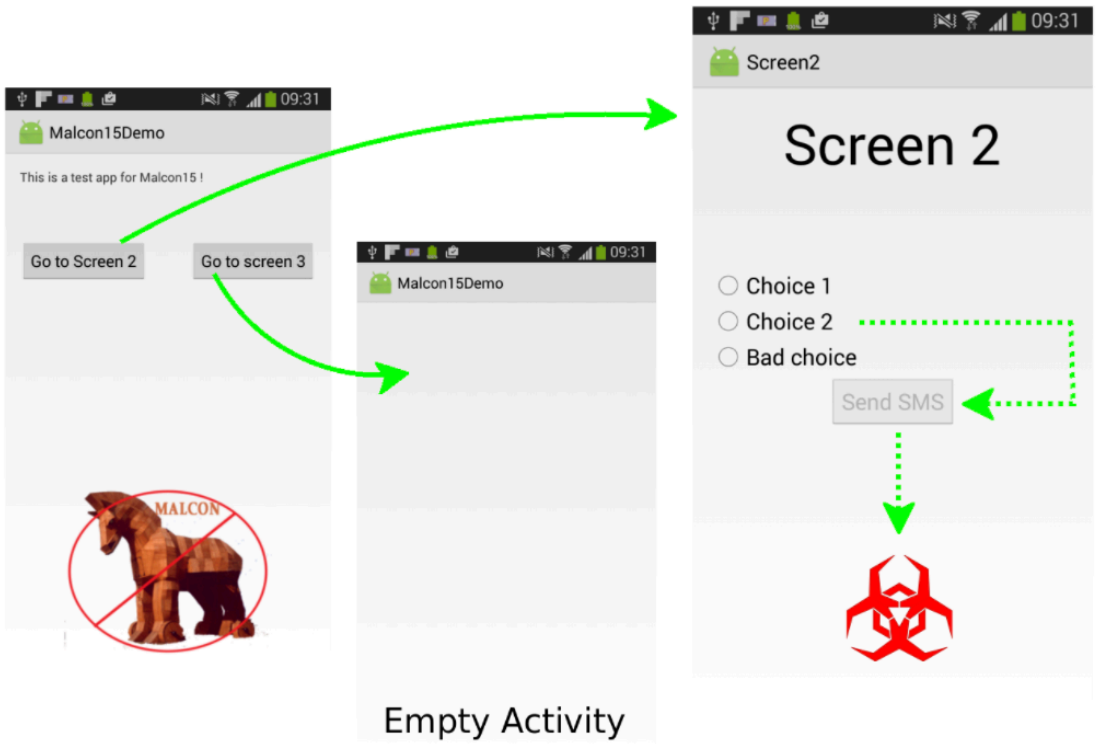

Figure 2.4.: Demo Application to demonstrate GroddDroid [1].

- *to_force.log:* list of branches to force
- The instrumented APK
- dot directory: contains methods CFGs

GroddDroid Runner

In our opinion, it is also important to understand how GroddDroid interacts with the GUI of an application to get a better picture of how it works. **The GroddDroid Runner** is based on uiautomator which is a python wrapper to the Google API for testing purpose [2]. After the aforementioned *New APK* is generated, GroddDroid opens the application on the devices. GroddDroid pushes the malware on the smartphone and launches it main activity. For each displayed activity, GroddDroid collects the graphical elements that may trigger additional code. If clicking on, say, a button, leads to a new activity, GroddDroid analyses it and repeat the same operation as before. Else, it triggers the next element of the activity or gets back to the previous activity [2]. Example Demo Application that was presented by the authors on MALCON 2015

conference can be seen on Figure 2.4. As seen in the figure, the GroddDroid runner works like the following as can be seen on the presentation of the authors [1]:

- Collects graphical elements
- Explores the app by clicking on the buttons
- Can go back
- Can launch the app again
- Detects loops
- Until all the different activities are explored

In this section we summarized GroddDroid that is related to our work in this thesis. However, more detailed information on GroddDroid can be found on [14, 2, 1]. We now would like the present other tool we will use to test Random Stimulation Technique, which is **Droidutan**.

2.4. Machine Learning

In [3], Alpaydin defines *machine learning* like the following: "Machine learning is programming computers to optimize a performance criterion using example data or past experience." If we apply this concept to our work on this thesis, we can integrate this definition like the following: In this thesis, we will use machine learning concepts to compare performances of stimulation techniques; while using this technique to analyze malware on an app that we have never seen before.

Machine learning can be categorized under 4 categories [3]:

- Association
- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

For the scope of thesis, we believe that it may be beneficial to inspected Supervised Learning and Unsupervised Learning in this section; for association and reinforcement learning, more information can be found under [3].

Supervised Learning

Supervised Learning is to learn a mapping from the input to an output whose correct values are provided by a supervisor [3]. In other words, *supervised learning* is to learn the mapping of an input to an output from the training dataset and apply the mapping to a test dataset. We can rephrase this as in one must manually or automatically label every element of the training dataset, so that the algorithm can learn the associations between different features and the corresponding label [14]. In the end, this learning can be generalized to multiple cases, where outputs are continuous.

Unsupervised Learning

Unsupervised learning is to learn what normally happens. This technique groups similar instances with no outputs. Unlike *supervised learning*, the data does not get labeled. In unsupervised learning, there is no such supervisor like in the *supervised learning* and we only have input data. The aim is to find the regularities in the input. There is a structure to the input space such that certain patterns occur more often than others, and we want to see what generally happens and what does not [3].

The majority of machine learning scenarios generally fall into one of two aforementioned learning tasks: supervised learning or unsupervised learning [24]. Related to our work, *Active Learning*, falls in between, and considered as semi-supervised. Before we go into Active Learning, we now want to give brief information about useful machine learning concepts that are related to our work.

Features A feature is 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. Therefore, the goal is to use features that describe the data as good as possible [14]. In [2], E. Alpaydin gives the following example task to understand the concept of *features*:

"Take a word, for example, machine. Write it ten times. Also ask a friend to write it ten times. Analyzing these twenty images, try to find ***features****, such as types of strokes, curvatures, loops, how you make the dots, and so on, that discriminate your handwriting from your friends."*

Classifiers The term "classifier" sometimes also refers to the mathematical function, implemented by a classification algorithm, that maps input data to a category [3]. Although there are many different classifiers, each equally important for different scenarios, in this thesis we will use **Ensemble** classifier. Although the reason for this will become more clear for the reader in the following chapters, for now, we can claim that we need to combine classifiers by "voting". This classifier is also known as "ensemble".

Performance Metrics

To make use of the aforementioned machine learning concept, we will make use of various performance measures, each between 0 and 1. We will calculate, in our experiments, the following performance measures, in correspondence to [14]:

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

Accuracy The *accuracy* score is calculated as follows, denoting the percentage of instances that have been classified correctly:

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

Recall The recall metric is calculated as follows, denoting the percentage of a new malware getting classified as malware.

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

Specificity This score is calculated as follows, keeping track on performances of the benign apps. In other words, how many of them did we correctly identify as benign apps?

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

Precision The precision score is calculated as follows, denoting the probability that an instance classified as malware is really malware.

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

F1-Score F1 score is calculated as follows, capturing how good our detection capability on the malware.

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

2.4.1. Active Learning

In [22], Active Learning is described as the following: "The key idea behind active learning is that a machine learning algorithm can achieve greater accuracy with fewer training labels if it is allowed to choose the data from which it learns. An active learner may pose queries, usually in the form of unlabeled data instances to be labeled by an oracle (e.g., a human annotator)."

In the context of Android malware detection techniques, to the best of our knowledge, passive learning is proved to be not sufficient, as stated in [21]. In a passive learning setting, it may be the case that a number of the test apps labels are incorrectly predicted, which negatively affects the classification accuracy. If the classification accuracy is inplausible, it results in the inability of the trained classifier to ask for more accurate representations of the misclassified apps in the form of different feature vectors [21].

Needless to say, this approach would prove inadequate in our case, because we do not have the ground truth about a malicious app's true nature and which path that our classification was based on.

In an active learning setting, however, if a feature vector of a test app is misclassified, the classifier is allowed to instruct the feature extraction mechanism to generate another feature vectors for the same app. This process can be repeated until either the misclassified app is correctly classified, or the overall classification accuracy of has converged to a maximum value [21].

3. Methodology

In this chapter we present the methodology of our proposed solution and objective to tackle the problem at hand and explain why we took such approach.

3.1. State-of-the-art

To the best of our knowledge, most of the current approaches in pursuit of detecting Android Repackaged Malware consist of three main parts: analyze, extract features, train machine learning classifier. Following the training, the trained *classifier* is used to classify the test apps [21].

As we have talked briefly in the Introduction chapter, particularly in Figure 1.1, a major flaw of this approach is the lack of ground truth when classifying apps: the uncertainty of which path or branch depict the grafted malicious behaviors. Lacking such ground truth would most likely lead to a misclassification, given that the probability of branch that unveils the true nature of the app may not be executed.

Also mentioned in the Introduction chapter, we will approach this "pursuit of ground truth" as a search problem. In the next section we will present the methodology and architecture behind our proposed solution.

3.2. Methodology and Architecture

Since the primary focus of this thesis is to compare two stimulation techniques, we had to run two experiments on two stimulation techniques: **GroddDroid**, for *Forcing Execution Stimulation Technique*; and **Droidutan**, for *Random Stimulation Technique*.

Figure 3.1 and Figure 3.2 depicts the overall methodology for two experiments; illustrating both experiments side by side. Although one may think that, in terms of comparison, keeping the architecture of both experiments identical is more reasonable; for varying purposes and/or requirements, we had to adapt the experiments for the *Droidutan* and *GroddDroid*. In the *Stimulation* subsection, we elaborate on these reasons and describe how both architectures work.

3.2.1. Droidutan

As we covered in the *Background* chapter, Droidutan is a random GUI stimulator that we use for the Random Stimulation. Since Droidutan experiments are conducted by

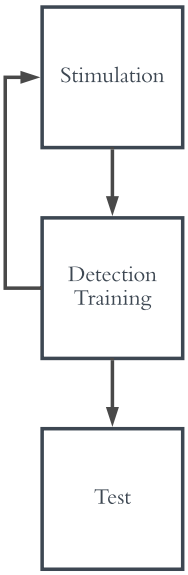

Figure 3.1.: Droidutan

Figure 3.2.: GroddDroid

our advisor, we will not get into detail about the experiments. Rather than that, we will use the results of our advisor to compare with GroddDroid. Experiments conducted via Droidutan can be found in [21].

However, we believe that elaborating on the **element of randomness** on both tools is rather significant for this thesis. Droidutan, as a random stimulation tool, acts completely random when stimulating different branches of the app. That is to say, one run of Droidutan will most likely will not generate the same results if it runs a second time. That is the main reason the architecture seen on the Figure 3.1, so as the following steps:

The experiment will start off with **stimulation**. That is to say, the apps inside the dataset will be seperated as test and training apps. Following this, Droidutan will be stimulated on each app in the training and test datasets, and features will be extracted from the dumped log on one run of Droidutan. This dumped log will be used to extract features using Droidmon, the api-call-monitor for each API call.

One important point to mention is that the analysis done by Droidutan is that every run will be limited to a time frame. Since Droidutan randomly interacts with the tool, limiting the execution into a time window helps us save an significant amount of time; since random stimulation tools do not have a roadmap when to stop executing.

After stimulation with Droidutan, **detection and training** will take place. Apps will be classified using only training apps with purpose of better classification accuracy. Following the classification, misclassified apps will be re-stimulated using Droidutan, going into a loop of stimulation and detection; back to the beginning. This means that Droidutan will keep stimulating apps more than once, generating a log for each

iteration.

To cover multiple statistical combinations, we will run the whole experiment multiple times, without changing any arguments or other data. The difference of the next runs following the first run of the experiment will be the splitting of the test and training apps.

3.2.2. GroddDroid

Unlike Droidutan, GroddDroid falls under Forcing Execution Stimulation technique, forcing the execution of different branches. Analysis done by GroddDroid differs in several aspects from Droidutan.

Firstly, **the element of randomness** is negligible in GroddDroid, to our best knowledge. In [2], authors of GroddDroid do not mention that seperate iterations on GroddDroid will not depict a randomness, unlike Monkey Runner. Specifically, we derived that multiple runs of GroddDroid on the same app will not yield significantly different representation of feature vectors. Therefore, for the sake of this experiment, we assume that running GroddDroid multiple times on the same app will depict the same behavior.

Secondly, since GroddDroid acts according to a roadmap (i.e. reference run, suspicious heuristics), in our opinion, limiting execution of GroddDroid to a time frame would yield unintended results. Therefore, unlike Droidutan, GroddDroid will keep executing as long as it finds something interesting to execute. However, to save time, adaptions had to be made on GroddDroid to some degree; which we get into more detail in the *Implementation* chapter.

For these reasons, our experiment on GroddDroid differs from Droidutan as the following:

Stimulation via GroddDroid will be done on all apps on our dataset before everything else. The number of *maximum runs* of GroddDroid will be in sync with the number of experiment iteration that will take place in the Detection part. That is to say, if maximum runs are specified as 10, GroddDroid will stimulate every app maximum 10 times, and generate 10 logs and 10 feature files, extracted via Droidmon. After this part is done, all GroddDroid execution will be completed and will not run again anywhere in the application.

Detection and Training that takes place after stimulation, similar to Droidutan, will classify apps using Ensemble classifier. However, one difference here is that the apps will be split into test and training apps in the detection phase. During the iteration of the detection phase, features belonging to the iteration number will be loaded and used for classification. That is to say, first iteration will load feature log files from GroddDroid's first run; second iteration will load the log number two, and so forth.

Special Case In our GroddDroid experiments, if the app is misclassified, the next feature from GroddDroid's next run will be loaded. However, there are special cases for apps which are misclassified although all their feature files are loaded. In this case, we will load a random feature file and continue the iteration with that feature.

We illustrate the workflow of our GroddDroid experiment in Figure 3.3, while Droidutan experiment workflow can be found in [21].

3.3. Test Phase

Test phase of our experiments is exactly the same for both our experiments, which is written by advisor and can be found in detail in [21].

In the test phase, we will retrieve the best classifier and use it to calculate the metrics. These metrics will be used to classify the app as malicious or benign, using **One Instance Technique** and **Majority Vote Technique.**

Majority Vote Technique Majority Vote technique looks for all best runs and takes the most present classification. For instance, in 5 iterations, if an app is classified as malicious 3 times, and benign 2 times; majority vote technique will classify the app as malicious.

One Instance Technique In contrast to the majority vote technique; one instance technique will look for only one malicious vote to classify the app as malicious. That is to say, in 5 iterations, if an app is classified as benign 4 times but as malicious 1 time; one instance technique will say that the app is considered as malicious.

For each of the aforementioned techniques, we will calculate metrics to calculate scores mentioned in the background chapter: true positive, true negative, false negative, false positive.

After the scores and labels from each techniques are calculated, our experiment will end, outputing each label as malicious or benign and from which technique.

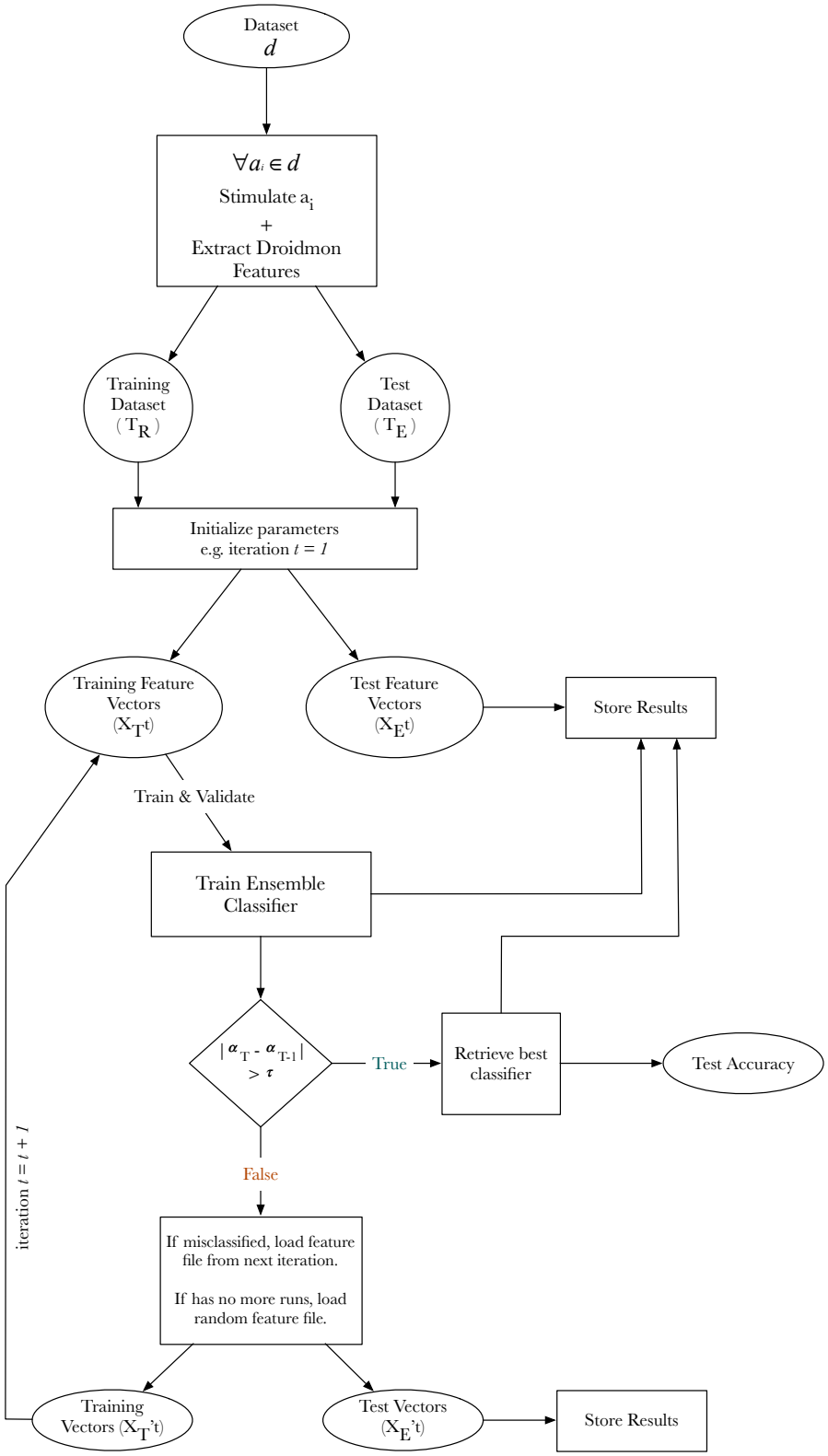

Figure 3.3.: Overview of GroddDroid Experiment

4. Implementation

In this thesis we implemented a framework to run experiments to compare stimulation techniques. While we get into significant details in the next sections, the source code can be downloaded from *http://github.com.demirdagemir/thesis* and can be found in the Appendix section. We did not name our framework since it is mostly based on *Aion* [19].

Our implementation consists of three main parts:

1. Adaptations to Aion [21]
2. Adaptations to GroddDroid [2]
3. Helper files to compute statistics and running the experiment

Before we get into the aforementioned main parts, we also would like to talk about dependencies in developement environment.

4.1. Development Environment

4.1.1. Hardware

Our source code is written and also tested with the following machine properties:

- Operating System: MacOS 10.13
- Processing Power: 4 Cores; 2,6 GHz
- Memory: 16GB of RAM
- Storage: 500GB SSD

For only running GroddDroid, an Ubuntu Linux machine provided by our chair was also used; which we will specify in the evaluation section.

4.1.2. Languages

While *Python* is the main language for our framework, there are couple of details that we believe are worth mentioning. Firstly, Aion [19] is written in Python 2.7, as well as Droidutan. Therefore, we also made our adaptations in Python 2.7; rather than

adapting the whole source code to a version above 3. However, since GroddDroid is written in Python 3, our adaptation on GroddDroid had to be written with Python 3. Although this does not make a significant difference in our code; we believe it is important to let the reader know about this version mixture. More detailed differences about version differences in Python can be found here [17]. Although GroddDroid is also written in Python, especially for the tasks to explore the apps and monitor runs; it also uses *Java* to transform the bytecode of an app. Most important frameworks that may prove significant to read about are the following:

- **Soot**: A Java optimization framework that provides intermediate representations for analyzing and transforming Java bytecode [23].
- **Soot-Infoflow**: Data Flow Tracker using Soot [7].
- **Soot-Infoflow-Android**: A framework statically computes data flows in Android apps and Java programs [4].

4.1.3. Tools

In the next part we will briefly describe the tools that our experiment runner was based on.

GroddDroid Adaptations

As described before, GroddDroid was huge part of our experiments to test the Forcing Execution Stimulation Technique. To serve our experiment purposes, we had to adapt GroddDroid because of the way that GroddDroid works.

Since our experiment is majorly based on *Aion* which is aimed to test Droidutan and Droidbot, GroddDroid adaptations was necessary for us in order to work better with Aion.

As drawn in the Methodology section, Aion is initially designed to run Droidutan each time as the iteration continues. We could not use this design out of the box for the following reason: It is not possible in GroddDroid, to continue with the third run when the previous iteration executed two runs of GroddDroid. As the iteration continues, we had to run GroddDroid from the beginning, executing 3 runs all over again.

Although this can be achieved by altering GroddDroid, after some efforts; we found that altering Aion according to GroddDroid would prove easier. Therefore, we made the decision to hand the control of stimulation completely to GroddDroid, while "modified" Aion picks up the rest of the experiment.

One modification on GroddDroid to support this decision is to move "extracting droidmon features" inside GroddDroid. Since GroddDroid will have complete control

on stimulating apps before the experiment starts, feature extraction had to be made after each run of GroddDroid.

After carefully examining the source code of GroddDroid, we found that the best place for extracting Droidmon features is inside the class *Explorer* which is, in its authors' words, "Class in charge of calling the tagger, the automatic tester and everyone at the right moment and with the right data." Inside Explorer, we had the chance of when to dump the log and when to extract features and which run is currently in process. Figure 4.1 depicts the aforementioned code part, deducted of the parts where it belongs to original GroddDroid. The whole part of the altered code as well as the referenced methods like *dump_logcat* and *filter_logcat_for_droidmon* can be found in Appendix A.

```
def run_apk(self):
  log.info("Running␣APK␣on␣device")
  # <Code regarding setting run_type and running the APK>
  logcatFileParam = self.output_dir + "/dumped_logcat_" +
       str(self.run_id) + ".log"
  self.dump_logcat(logcatFileParam)
  self.filter_logcat_for_droidmon(logcatFileParam)

  prettyPrint("----------␣Extracting␣Droidmon␣Features␣----------------",
       "warning")
  droidmonFeatures =
       extractDroidmonFeatures("%s/dumped_logcat_%s.log_filtered.log"
       \% (self.output_dir, str(self.run_id)))
  with open(self.output_dir + "/extractedDM_" + str(self.run_id)
       + "_traces.log", "w") as fp:
     fp.write('\n'.join('%s' % x for x in droidmonFeatures[0]))
  with open(self.output_dir + "/extractedDM_" + str(self.run_id)
       + "_features.log", "w") as fp:
     fp.write(str(droidmonFeatures[1]))
  prettyPrint("Extracted␣Droidmon␣Features␣:))", "success")
```

Figure 4.1.: Feature Extraction inside GroddDroid

After this code is ran, GroddDroid will generate the following output files if it finds something interesting to run, with <run_number> replacing with a number:

- original output of GroddDroid (e.g. apktool, run_0, run_1)
- **dumped_logcat_<*run_number*>.log:** All the log lines generated by a GroddDroid run, from AVD and taken from logcat.

- **dumped_logcat_filtered_<*run_number*>.log:** Filtered log lines, that includes the "droidmon" tag and the app's package name.
- **extractedDM_<*run_number*>_features.log:** Extracted features from filtered log file.
- **extractedDM_<*run_number*>_traces.log:** Extracted traces from filtered log file.

GenyMotion and VirtualBox

Android Virtual Devices are primarily used to test and run applications on emulators to make developers' life easier by further speeding up the development process. Since these emulators are almost the same as real devices, our decision was also run the experiments on emulators rather gathering couple real devices, which would really slow our experiment run time. For this purpose, we used Genymotion [8], which is built on top VirtualBox [15], the commonly known virtualization software. Genymotion was our first rather than the official Google Emulator because of its high performance, speed and integration with VirtualBox with allowed us to easily clone machines, take snapshots and restore snapshots to avoid high CPU usage after large number of apps are installed on the emulator. For our experiments, we used a machine named "Rikers", provided by our advisor; which runs Android version 4.1.1. For running several machines simultaneously we cloned this machine and used for parallel runs of GroddDroid.

Droidmon API Monitor

To extract features to train our classifier, we used Droidmon API monitor to monitor each API call made while the stimulation engine runs. Droidmon basically monitors applications inside a virtual machine and provides insight into an applications behavior by logging each specified API call with a tag [6]. The desired API calls to monitor can be speficied using a JSON file, called *hooks.json*. After this, each API call found in hooks.json, and triggered by the stimulation engine will be logged by *logcat* with the following format:

Droidmon-apimonitor-<*Package Name*>

Out of the log lines generated by the Android Virtual Device while executing, we will filter the logs looking for these specific tags, providing the app's package name; which can be found in the manifest file of the app after decompilation. After filtering, features will be extracted to for the test and training phases of our experiment.

4.2. Aion

As we have mentioned before, our implementation largely based on Aion, which is "a framework meant to apply the notion of active learning to the problem of stimulation, analysis, and detection of Android repackaged/piggybacked malware." [19]. *Aion* was designed to run the experiments on apps, stimulating via Droidutan or Droidbot. Since the way GroddDroid works differs in many ways than Droidutan; we had to make adaptations on Aion as well as GroddDroid.

4.2.1. Adaptations to Aion

Main modification of Aion from our side was to adapt to the way GroddDroid works. As mentioned in the Methodology chapter, since we moved the feature extraction feature inside GroddDroid; we also adapted Aion regarding this change.

Firstly, we moved the **"analysis of APKs"** part outside the main loop, where iteration continues as F1 score converges. Aion had already an argument where it allows to skip analysis, but we also added an additional argument to not get into the main loop and only run the analysis part.

Secondly, and following this, we also moved the **splitting dataset** to training and test apps right after the stimulation part; which will allow us run the program several times in order to represent several combinations.

Thirdly, we altered the main loop to change the way it uses the feature extraction. Since "original" Aion does extraction also in the main loop, we modified this to get only the features from the desired iteration. That is to say, first iteration will load the first feature vector, and second one will load the second feature vector. Here, we also implemented the special case mentioned in the Methodology chapter, where we get a random feature vector if no more runs from GroddDroid is available and the app is still misclassified.

Finally, we added a ***GroddDroidTest*** python class, in order to extend Aion to run GroddDroid as well as Droidutan. If, when running Aion, analysis engine is specified as GroddDroid; GroddDroidTest will run as a subprocess on specified Android Virtual Devices simultaneously. In case of a crash/GroddDroid Error/Timeout Error, GroddDroidTest will stops its process and Aion will run it again with the next APK specified.

Other parts of Aion are mostly kept the same, although we believe that several points are worth mentioning in this following section.

SQL-Lite Aion was implemented using SQL-Lite to store important data and activities as the experiment runs, in all phases. We decided to keep SQL-Lite exactly like Aion, since we could not find a good reason to switch another database language, or another database mechanism like NoSQL. SQL-Lite served our purposes enough to continue with the gathering experiment data and monitoring runs.

Test Phase Test phase is the part where the specified number of iterations are over, or the desired F1 score condition is reached and the final metrics and classification are expected to "diagnose" the app using One Instance Technique or Majority Vote Technique. Test phase consists of the following parts, numbered in the execution order.

1. Retrieve the best classifier and its iteration (X)
2. Classify feature vectors for each app in test apps.
   a) Retrieve all feature vectors up to X.
   b) Classify each feature vector using the loaded classifier.
   c) Decide upon the app's label according to majority vote technique versus one-instance.
   d) Declare the classification of the app in question
3. Calculate metrics
4. Store and print results

4.2.2. Main File

In the main file, which its source code and dependent source codes can be found in the appendix of this thesis, is the combination of all features we talked about in the previous sections and subsection and where all of them are triggered. It may be worthwhile to mentioned once again that, this file is an adaptation of *runExperimentII.py* in [19]. We now want to give an overview of most important aspects of this file.

Inputs and Outputs

Our main file takes the following inputs as run time arguments:

- *malwaredir*: The directory containing the malicious APK's to analyze and use as training/validation dataset.
- *goodwaredir*: The directory containing the benign APK's to analyze and use as training/validation dataset.
- *datasetname*: A unique name to give to the dataset used in the experiment (for DB storage purposes).
- *runnumber*: The number of the current run of the experiment (for DB storage purposes).

- *analyzeapks*: Whether to perform analysis on the retrieved APK's. item *onlyAnalyze*: If the experiment should stop after analysis phase.
- *analysistime*: How long to run monkeyrunner (in seconds).
- *analysisengine*: The stimulation/analysis engine to use
- *vmnames*: The name(s) of the Genymotion machine(s) to use for analysis (comma-separated).
- *algorithm*: The algorithm used to classify apps.
- *selectkbest*: Whether to select K best features from the ones extracted from the APK's.
- *featuretype*: The type of features to consider during training.
- *accuracymargin*: The margin (in percentage) within which the training accuracy is allowed to dip.
- *maxiterations*: The maximum number of iterations to allow.
- *branchExplorerDir*: Branch Explorer directory of GroddDroid, if the analysis engine is GroddDroid.

Out of these arguments; *analysistime* serves the purpose of setting a timeout to Droidutan, which we do not use in GroddDroid. For *algorithm*, and *selectkbest*; we consecutively use ensemble and 0 as default. For feature type; we will only look at dynamic features.

As outputs; each iteration of the experiment will create a classifier text file provided from the insert data from the database with the following format:

<algorithm>_<runnumber>_<iteration>_featuretype.txt

Also, data generated along the experiment will be inserted in the database with the schema in Figure 4.2.

Alternative GroddDroid Test and Statistical Calculations

Although, modified Aion works well with parallelized GroddDroid, we found that Python's handling of multithreading makes the process slower than usual. Therefore, in our repository, we also provide an alternative GroddDroid Test file, which runs only one instance of GroddDroid on all the apks in the specified directory and on specified Android Virtual Device. This file also checks the already analyzed APKs to continue where it has left of if the user wants to continue stimulating apps at a further point. Readers of this can run this file in parallel in seperated terminals/shells if need be.

```
CREATE TABLE learner(
  lrnID   TEXT PRIMARY KEY,
  lrnParams  TEXT
    );

CREATE TABLE run(
    runID  INTEGER,
    runDataset  TEXT,
    runStart   TEXT,
    runEnd   TEXT,
    runIterations INTEGER,
    PRIMARY KEY (runID, runDataset)
);

CREATE TABLE datapoint(
    dpID  INTEGER PRIMARY KEY AUTOINCREMENT,
    dpLearner  TEXT,
    dpIteration  INTEGER,
    dpRun   INTEGER,
    dpTimestamp  TEXT,
    dpFeature TEXT,
    dpType TEXT,
    dpAccuracy  REAL,
    dpRecall  REAL,
    dpSpecificity REAL,
    dpPrecision  REAL,
    dpFscore  REAL,
    FOREIGN KEY (dpLearner) REFERENCES parent(learnerID),
    FOREIGN KEY (dpRun) REFERENCES parent(runID)
);

CREATE TABLE testapp(
    taName  TEXT,
    taRun  INTEGER,
    taIteration  INTEGER,
    taType  TEXT,
    taClassified TEXT,
    taLog  TEXT,
    PRIMARY KEY (taName, taRun, taIteration),
    FOREIGN KEY (taRun) REFERENCES parent(runID)
);
```

Figure 4.2.: Database Schema to store experiment data

In addition to this we have written a Python script to remove all the reference APKs generated by GroddDroid save space on the development environment.

We also have written a script, answering statistical questions about the experiment. This script will answer questions, such as "how many apps did crash?", "what is the average on an app?", "how many apps did generate maximum runs?" and so on.

Source codes of both scripts can be found on the repository mentioned in the beginning of this chapter. In the next chapter, we will evaluate the data gathered by running our experiment and present it to the readers of this thesis.

5. Evaluation

In this chapter, we will present the experiments conducted to the readers of thesis, including how we conducted our experiments and results drawn from them. Because we used active learning while comparing two stimulation techniques for detecting Android repackaged malware, we will present both introduced stimulation techniques in the context of machine learning. Firstly, we would like to the specify the environment we ran our tests on.

5.1. Testing Environment

Our testing environment where we ran experiments was pretty similar to our development environment. In addition to the machine in development, we introduced one more machine supplied by our advisor to run our experiments faster to save time. This additional machine that we only used to run part of the dataset withheld the following specifications:

- Operating System: Ubuntu Linux 16.04
- Processing Power: 2 Cores; 2,7 GHz
- Memory: 16GB of RAM
- Storage: 235 SSD

5.2. Experiments

With the goal of having a better structure while describing our experiments, we organized this section like the following: We will first give the reader a brief introduction on the dataset we used to conduct our experiments. Secondly, we will present our experiments on Droidutan; therefore Random Stimulation Technique. Thirdly, we will present our experiment on GroddDroid; therefore Forcing Execution Stimulation Technique. Lastly, we will compare both technique having presented all the results from both techniques while discussing the results drawn from all the experiments.

For our goal of comparing Droidutan and GroddDroid; we believe that answering same questions for both stimulation engines would give better understanding of comparison between both techniques. Thus, while describing how did we manage to get

results, we will seek answers for the following questions for both techniques in their own subsections:

- On average, using different techniques, how many iterations does it take to reach the best classifier?
- How long does it take (in real time) to reach this number?
- Which technique better helps us; the majority vote technique, or the one instance technique?
- Does the aforementioned techniques help us correctly classify the test apps?

In addition to these we also present results for *number of survived apps* and *number of extracted features per app* for both tools, which, we believe, provide valuable insight for our research.

5.2.1. Dataset

In our experiments, we used a dataset that consisted malicious and benign Android apps, in the form of APK files. This dataset was called *Piggybacking* [10], and included about 1300 pairs of apps: one *apk* file as a benign, original app and one *apk* file as a malicious counterpart. According to [21], authored by our advisor, who also supplied us with this dataset; the process of gathering the apps to their repackaged counterparts started in 2014 and carried out until 2017. The dataset we used consisted of **1355 benign apps** and **1399 repackaged, malicious apps**, since some original apps had more than one repackaged versions. We kept all the app names hashed, in hexadecimal form.

5.2.2. Droidutan

Droidutan experiments was conducted by our advisor and more details can be found in [21]. However, we will discuss the result on this thesis, and also look for different results that will help us to compare Droidutan to GroddDroid.

Firstly, we ran Droidutan experiment **17 times**, meaning that the whole dataset was ran 17 times; with different combinations of testing and training apps split differently. We also included two more runs from recent experiments when presenting scores; because for duration and other statistics, the data was not readily presentable. However, we believe that this does not have an impact on our overall discussion. The reason for doing multiple runs is that, as also mentioned in the Methodology chapter, to be able cover different statistical combinations of training and test apps.

Table 5.1 provides how did each run take in hours, and how many iterations had taken place during each run. *Hours* in this table is rounded down to the closest integer value for a cleaner picture.

Run Number	Duration(*hours*)	Iterations
1	52	10
2	20	3
3	21	3
4	29	5
5	54	10
6	51	10
7	46	9
8	34	6
9	16	2
10	37	10
11	20	3
12	37	10
13	39	9
14	57	10
15	22	4
16	58	10
17	19	2

Table 5.1.: Droidutan: Experiment Runs

From this table, we derive that **average runtime for an iteration** of dataset via Droidutan is **4.9 hours.**

Secondly, we looked at how many apps survived when stimulating with Droidutan. This means that how many apps did generate minimum one log file filtered by Droidmon. We assumed that if a Droidmon log was not generated, this app either crashed or the stimulation engine did not find something interesting to execute.

Table 5.2 shows the amount of survived apps for each run of Droidutan from **a total of 2754** apps. We derive the average, which is 62% for each run. Which means that, **Droidutan generates a log file for an app 62% on average.**

Thirdly, we gathered data on how many feature vectors per survived app got generated. Table 5.3 shows the total number of extracted feature vectors, for each run; categorized by malware and goodware. By looking at these numbers; we derived the average number of extracted features per survived app as conveyed in the table in the last "Per app" column. Moreover, **the overall average is 2.19**, which means that stimulating with **Droidutan will result in 2 to 3 extracted feature vectors on average.**

As we also mentioned in the Introduction chapter, we are also in pursuit of the path that will unveil the app's true nature; meaning we are searching for number "X", a number that within which the malicious behavior should exhibit. Table 5.4 shows the number of iterations for each run of the experiments. From this, we derive that **on average 4.16 iterations** should get as to that number. Therefore; according to

Run Number	Malware	Goodware	Total	Survived(%)
1	1608	605	2213	80
2	812	685	1497	54
3	793	760	1553	56
4	896	792	1688	62
5	915	799	1714	62
6	872	744	1616	59
7	852	799	1651	60
8	873	773	1646	60
9	711	719	1430	52
10	800	755	1555	54
11	842	674	1516	55
12	775	730	1505	55
13	798	791	1589	58
14	864	866	1730	63
15	695	961	1656	60
16	1033	993	2026	74
17	1046	1011	2057	77

Table 5.2.: Droidutan: Number of Survived Apps (out of 2754)

Run Number	Malware	Goodware	Total	Per app
1	4872	3335	8207	3.71
2	1147	947	2094	1.40
3	1226	1129	2355	1.52
4	1681	1407	3088	1.83
5	3133	2691	5824	3.40
6	3022	2452	5474	3.39
7	2480	2145	4625	2.80
8	1754	1879	3633	2.21
9	828	913	1741	1.22
10	1520	1456	2976	1.91
11	1205	965	2170	1.43
12	1302	1458	2760	1.83
13	1865	1676	3541	2.23
14	3349	3290	6639	3.83
15	1061	1539	2600	1.57
16	1875	1703	3578	1.77
17	1202	1197	2399	1.17

Table 5.3.: Droidutan: Number of extracted feature vectors per survived app

Run Number	Iteration	F1 Score	Specificity Score
1	2	0.846511627907	0.967654986523
2	2	0.813559322034	0.955882352941
3	2	0.786301369863	0.949519230769
4	4	0.813333333333	0.970149253731
5	6	0.842572062084	0.971496437055
6	6	0.824539877301	0.946015424165
7	8	0.776439089692	0.986078886311
8	2	0.841142857143	0.970873786408
9	1	0.81875792142	0.958333333333
10	9	0.805774278215	0.955665024631
11	2	0.838709677419	0.992346938776
12	7	0.817089452603	0.965425531915
13	4	0.772861356932	0.988864142539
14	5	0.743262411347518	0.973509933775
15	3	0.663736263736264	1.0
16	1	0.811355311355311	0.952224052718
17	1	0.807102502017756	0.9234375
18	3	0.792279411764706	0.956973293769
19	8	0.825150732127476	0.979166666667

Table 5.4.: Droidutan: F1 Scores of Best Classifiers(Training)

our experiments, it is safe to claim that within **4.16 iterations**, Random Stimulation Technique should unveil the app's true nature.

Lastly, comparison of **majority vote technique** and **one-instance technique** is shown in Table 5.5. The data in this table is gathered via test apps after classification of malicious and benign apps. We derive the following result from this table: in 8 runs, both techniques have same F1 Score, while in remaining 11 runs; **one-instance technique has higher F1 Scores** in every run of the experiment.

5.2.3. GroddDroid

In this section, we will now present our findings by our experiments with GroddDroid. As described in the Methodology section, GroddDroid experiments differed from our Droidutan experiments in several different ways: GroddDroid is only ran on each app once; until the maximum runs specified to GroddDroid is reached; whereas Droidutan was ran by the main experiment and executed several times on an app until the experiment conditions are met. Second main difference is that, because of time reasons; we were not able to run all the apps in the dataset. We were able to ran **1214 apps** in total by GroddDroid; divided into 611 malicious and 613 benign apps. We experimented on GroddDroid answering the following questions.

Firstly, Table 5.6 shows the run times for GroddDroid. We believe that it is significant to elaborate on this table since it may not look consistent at first glance. The table shows the maximum and minimum taken for first run and the tenth run of as well as the average. As we mentioned on the Background chapter, GroddDroid subsection; first run of GroddDroid is most time consuming run since first run is the reference run and the control flow graph is generated during this run. Therefore after the first run; time spent on a single run significantly lowers. That is the main reason of why the tenth run average is approximantely ten times than the first run.

From this table, we derive that on average; one app of GroddDroid; with maximum run set to ten; approximately takes **44 minutes** in total. On average; **the first run takes 12 minutes**; whereas this value could be as low as 3 minutes and as high as 3.7 hours.

Secondly, we looked at **how many apps survived** stimulation with GroddDroid: That is to say, how many apps did manage to run at least one time. In addition to this; we looked at the **number of extracted feature vector per survived app;** as well as the number of apps that was able to run 10 times, in other words; **number of apps that were able to reach to the maximum runs.**

Table 5.7 illustrates the aforementioned statistics. From here, we derive that out of 1214 apps; **701 apps** were able to generate at least one log file; or been executed more than one time. Out of this 701 apps, 298 apps reached maximum number of runs; which were set to 10 by us when beginning our experiments. Again, these **701 apps generated in total of 4469 feature vectors;** extracted by filtered log file; as result of API monitoring by Droidmon. Average number of extracted feature vectors; however, is

Run Number	Iteration	F1 Score	Specificity	Classifier
1	5	0.7	0.656338028169	Majority Vote
1	5	0.713745271122	0.611267605634	One-Instance
2	2	0.714285714286	0.788617886179	Majority Vote
2	2	0.714285714286	0.788617886179	One-Instance
3	2	0.631404958678	0.76397515528	Majority Vote
3	2	0.631404958678	0.76397515528	One-Instance
4	4	0.706976744186	0.824840764331	Majority Vote
4	4	0.72131147541	0.786624203822	One-Instance
5	6	0.68044077135	0.745856353591	Majority Vote
5	6	0.700636942675	0.660220994475	One-Instance
6	6	0.695890410959	0.776470588235	Majority Vote
6	6	0.727037516171	0.729411764706	One-Instance
7	8	0.682080924855	0.840659340659	Majority Vote
7	8	0.717536813922	0.777472527473	One-Instance
8	2	0.664220183486	0.760383386581	Majority Vote
8	2	0.664220183486	0.760383386581	One-Instance
9	1	0.638403990025	0.776470588235	Majority Vote
9	1	0.638403990025	0.776470588235	One-Instance
10	9	0.665644171779	0.787356321839	Majority Vote
10	9	0.672619047619	0.755747126437	One-Instance
11	2	0.696078431373	0.716981132075	Majority Vote
11	2	0.696078431373	0.716981132075	One-Instance
12	7	0.652307692308	0.737288135593	Majority Vote
12	7	0.653846153846	0.689265536723	One-Instance
13	4	0.646962233169	0.836538461538	Majority Vote
13	4	0.661341853035	0.814102564103	One-Instance
14	5	0.666666666667	0.810473815461	Majority Vote
14	5	0.692506459948	0.770573566085	One-Instance
15	3	0.562277580071	0.942779291553	Majority Vote
15	3	0.570422535211	0.937329700272	One-Instance
16	1	0.678082191781	0.832835820896	Majority Vote
16	1	0.678082191781	0.832835820896	One-Instance
17	1	0.631578947368	0.749279538905	Majority Vote
17	1	0.631578947368	0.749279538905	One-Instance
18	3	0.669603524229	0.833333333333	Majority Vote
18	3	0.669527896996	0.805555555556	One-Instance
19	8	0.693042291951	0.764864864865	Majority Vote
19	8	0.716883116883	0.724324324324	One-Instance

Table 5.5.: Droidutan: One Instance Technique versus Majority Vote Technique

Total of Runs	**Maximum**	**Minimum**	**Average**
1	3.7	0.05	0.21
10	6.2	0.24	0.75

Table 5.6.: GroddDroid: Experiment Runs(in hours)

Number of	**Malware**	**Goodware**	**Average**
Apps Survived	350	351	%58
Apps to Reach Maximum Runs	179	119	%25
Extracted Feature Vectors	2387	2082	3.69

Table 5.7.: GroddDroid: App Statistics

lower than expected when looking at the survived malware and goodware apps. The reason for this is that 513 apps did not generate any log files; or did not make it past the first reference runs. The overall average is 3.69 for extracted feature vectors; which means that **stimulating with GroddDroid will generate 3.69 feature vectors per run** on average.

As we also mentioned in the Introduction chapter and Droidutan section, we are also in pursuit of the path that will unveil the app's true nature; meaning we are searching for number "X", a number that within which the malicious behavior should exhibit. Table 5.10 also shows the number of iterations for each run of the experiments. From this, we derive that **on average 1.88 iterations** should get as to that number. Therefore; according to our experiments, it is safe to claim that within **2 iterations**, Forcing Execution Technique should unveil the app's true nature.

Lastly, comparison of **majority vote technique** and **one-instance technique** is shown in Table 5.8 and Table 5.9 . The data in this table is gathered via test apps after classification of malicious and benign apps. We derive the following result from this table: in 17 runs, both techniques have same F1 Score, while in 7 runs **one-instance technique has higher F1 Scores**, and in 1 run **majority vote technique has higher F1 Scores**.

5.3. Discussion

In the previous sections we presented the results for both our experiments using Droidutan and GroddDroid objectively. We now want to elaborate on the results obtained on both experiments to answer our research questions. We also would like to point out the results which are unexpected from our point of view. We will present this section in three categories: **Usability**, **Reliability**, and **Effectiveness**.

Run Number	Iteration	F1 Score	Specificity	Classifier
1	3	0.637837837838	0.6	Majority Vote
1	3	0.629441624365	0.5	One-Instance
2	3	0.543352601156	0.644444444444	Majority Vote
2	3	0.598984771574	0.511111111111	One-Instance
3	1	0.577181208054	0.670731707317	Majority Vote
3	1	0.577181208054	0.670731707317	One-Instance
4	2	0.612244897959	0.703703703704	Majority Vote
4	2	0.612244897959	0.703703703704	One-Instance
5	1	0.413793103448	0.753246753247	Majority Vote
5	1	0.413793103448	0.753246753247	One-Instance
6	3	0.543046357616	0.681818181818	Majority Vote
6	3	0.583850931677	0.636363636364	One-Instance
7	1	0.526315789474	0.623529411765	Majority Vote
7	1	0.526315789474	0.623529411765	One-Instance
8	3	0.554838709677	0.706666666667	Majority Vote
8	3	0.59756097561	0.666666666667	One-Instance
9	3	0.613333333333	0.74025974026	Majority Vote
9	3	0.628205128205	0.701298701299	One-Instance
10	1	0.555555555556	0.771739130435	Majority Vote
10	1	0.555555555556	0.771739130435	One-Instance
11	1	0.514705882353	0.720588235294	Majority Vote
11	1	0.514705882353	0.720588235294	One-Instance
12	3	0.586826347305	0.75	Majority Vote
12	3	0.619565217391	0.652173913043	One-Instance
13	1	0.536912751678	0.679487179487	Majority Vote
13	1	0.536912751678	0.679487179487	One-Instance
14	3	0.604651162791	0.613636363636	Majority Vote
14	3	0.61797752809	0.579545454545	One-Instance
15	2	0.548780487805	0.610526315789	Majority Vote
15	2	0.548780487805	0.610526315789	One-Instance
16	2	0.574712643678	0.635294117647	Majority Vote
16	2	0.574712643678	0.635294117647	One-Instance
17	1	0.61935483871	0.717948717949	Majority Vote
17	1	0.61935483871	0.717948717949	One-Instance
18	1	0.535947712418	0.716049382716	Majority Vote
18	1	0.535947712418	0.716049382716	One-Instance
19	1	0.552631578947	0.670731707317	Majority Vote
19	1	0.552631578947	0.670731707317	One-Instance

Table 5.8.: GroddDroid: One Instance Technique versus Majority Vote Technique (1)

Run Number	Iteration	F1 Score	Specificity	Classifier
20	1	0.576470588235	0.547619047619	Majority Vote
20	1	0.576470588235	0.547619047619	One-Instance
21	2	0.601226993865	0.6	Majority Vote
21	2	0.601226993865	0.6	One-Instance
22	2	0.594594594595	0.717948717949	Majority Vote
22	2	0.594594594595	0.717948717949	One-Instance
23	1	0.539007092199	0.644736842105	Majority Vote
23	1	0.539007092199	0.644736842105	One-Instance
24	2	0.627906976744	0.597701149425	Majority Vote
24	2	0.627906976744	0.597701149425	One-Instance
25	3	0.583850931677	0.654761904762	Majority Vote
25	3	0.627906976744	0.607142857143	One-Instance

Table 5.9.: GroddDroid: One Instance Technique versus Majority Vote Technique (2)

Usability

To compare both stimulation techniques in terms of usability, we looked at how long does it take to stimulate *(n)* number of apps. Since we ran less apps in our GroddDroid experiments; we believe that looking at the averages is more reasonable for usability purposes. We state that, on average, GroddDroid takes 44 minutes to execute 10 runs per app with GroddDroid. For Droidutan, this number is 10 minutes. Therefore, Droidutan is faster than GroddDroid in this perspective. However; it is significant to mention that, in Droidutan, one can specify how long 1 run would take, by setting a parameter of "timeout", which we set to 60 seconds in our experiments. Since random interaction could take limitless amount of time; setting a timeout parameter would made us possible to conduct our experiments within a reasonable time frame.

Reliability

Next, we looked at the reliability of both stimulation techniques by an app having answered to questions: How many apps survived; and how many feature vectors/runs per app got generated. For Droidutan, we found out that on average, %62 of the apps survived; where as for GroddDroid %58 of the apps has survived. As we explained; survived means that at least one log file were generated where more than one run occurred on an app. In our opinion, it is logical that Droidutan is slightly higher in number of survived apps; because of GroddDroid's suspicious heuristics: GroddDroid will look for malicious parts of the code based on the suspicious heuristics: this may yield lower number of survived apps. It is important to mentioned that GroddDroid was ran on a smaller dataset; which may mean that this slight difference could get

Run Number	Iteration	F1 Score	Specificity Score
1	3	0.838926174497	0.881578947368
2	3	0.835087719298	0.910828025478
3	1	0.833876221498	0.886075949367
4	2	0.830065359477	0.894409937888
5	1	0.824675324675	0.914110429448
6	3	0.836879432624	0.941935483871
7	1	0.836879432624	0.974193548387
8	3	0.827067669173	0.939393939394
9	3	0.825622775801	0.927272727273
10	1	0.819923371648	0.932432432432
11	1	0.780487804878	0.901162790698
12	3	0.785992217899	0.943037974684
13	1	0.789473684211	0.969135802469
14	3	0.867549668874	0.903225806452
15	2	0.84	0.885906040268
16	2	0.80608365019	0.948717948718
17	1	0.817869415808	0.895061728395
18	1	0.834586466165	0.974842767296
19	1	0.829431438127	0.886075949367
20	1	0.783216783217	0.891025641026
21	2	0.842809364548	0.901234567901
22	2	0.792727272727	0.927272727273
23	1	0.776632302405	0.914634146341
24	2	0.842105263158	0.893081761006
25	3	0.837545126354	0.936708860759

Table 5.10.: GroddDroid: F1 Scores of Best Classifiers (Training)

higher on a larger dataset.

Moreover, Droidutan generated 2.19 extracted feature vectors per app whereas GroddDroid 3.69 feature vectors per app. Here, we can claim that GroddDroid will generate 1 to 2 more feature vectors per app than Droidutan, thus more reliable when extracting feature vectors are taken into account. Having better knowledge about the app; in our opinion, it is logical that GroddDroid were able extract more features than Droidutan.

Effectiveness

Firstly, we looked at **how many iterations does it take to reach the best classifier**. We found out that, on average; Droidutan took 4.16 iterations; and GroddDroid took 1.89 iterations to get to the best classifier. The reason for this, in our opinion, is that GroddDroid gathers much more information than Droidutan when executing paths which belongs to an app. Thus, since GroddDroid constructs its own Control Flow Graph and acts according to its reference run; it took almost 3 less iterations for GroddDroid to get to the best classifier. Since Droidutan acts randomly, the number of iterations to get to the best classifier varies throughout different runs. This can also been seen at looking at the maximum iteration a single run of an experiment went through: Droidutan was able to find the best classifier in the 9th iteration during our experiments; while GroddDroid did not find any best classifier past the third iteration. Therefore, we claim that, GroddDroid is more effective than Droidutan; to get the number *(X)*, within which the we find the path that will unveil the app's true nature.

Secondly, we asked the following question: **Which engine helps classify piggybacked apps better?** We found out that; Droidutan has average F1 Scores of 0.67, and specificity of 0.77 for test apps. One instance technique performed with 0.68 average F1 Score and 0.76 average specificity. Majority vote technique performed with 0.67 average F1 score and 0.78 average specificity. For GroddDroid on the other hand, we were able to get 0.57 average F1 Score and 0.66 average specificity. Majority vote technique had an average of 0.57 and 0.67 specificity, whereas 0.58 F1 Score and 0.65 specificity was obtained for one-instance technique.

We derive that one instance technique performed better in both stimulation engines, having a higher F1 Score for the most runs. We obtain that one vote of maliciousness performs better when classifying apps.

However, we found that Droidutan had higher F1 Scores; claiming that it performs better when classifiying apps. This came to our surprise; since we believed that a comprehensive tool like GroddDroid had to perform better than Droidutan, which acts randomly. We believe that GroddDroid did not perform well in terms of F1 Scores; because of its high effectiveness: Since GroddDroid's number *X* is significantly lower than Droidutan; it does not have too much sensitive information about the app. That is to say; while getting more sensitive information can yield better classifiers, iteration 1 does not give a lot of information when testing. Since GroddDroid's number X is 1.89; it has one or two feature vectors per test; which may result in low F1 Scores.

6. Conclusion

In this thesis, we have presented our approach to compare two stimulation techniques: Random Stimulation Technique versus Forcing Execution Technique. We started by introducing the problem at hand; stated our motivations and objectives and then presented all necessary background information to get a better understanding of our experiments. Methodology and architecture of our experiment followed this; and then we described how we implemented our architecture. Lastly, we demonstrated all the data from the experiments conducted and discussed how this data may help us determine compare both techniques. We used Droidutan and GroddDroid to compare Random Stimulation and Forcing Execution respectively. Our starting point was to determine what the problem is. Since repackaging was an emerging technique adopted by malware authors; we aimed to better classify repackaged application using stimulation techniques. We decided that, if an application is classified using benign paths; rather than injected malicious payloads or malicious paths; the classification of an app will most likely result in misclassification. We claimed that finding the path that was indicative of the app's true nature was a search problem. In pursuit of the malicious path; our motivation was to get a number (X) for both stimulation techniques; a number that within which an application's true nature will unveil. To achieve this, we design two sets of experiments using both techniques, and used active learning to find the number *(X)*. We found out that, different techniques performed well in different categories, and we discussed our findings under three parameters: usability, reliability, effectiveness. We found out that, although Droidutan performed slightly faster than GroddDroid and also resulted in more survived applications; GroddDroid performed better in extracting features and was highly effective on getting to the number *(X)* than Droidutan. Although our experiments suggested that Droidutan has higher scores in classifying apps; it was to our opinion that this was an effectiveness trade-off of GroddDroid and this may change in a different setting of an experiment. Although most part of the data showed that we were on the right track, there is still some research needed to pursue why GroddDroid's scores were lower than we expected.

Since Android Operating System's nature is attracted more and more malware authors, we believe that finding or improving effective techniques to prevent such authors from becoming widespread is crucial. In this thesis, we aimed to contribute to malware analysis; specifically repackaged malware detection while comparing stimulation techniques. In the closing chapter of this thesis, we would like to present further works that may prove significant to pursue in the future.

7. Further Work

In this chapter, we would like to emphasize the points we believe that may prove significant to pursue in the future.

As described earlier, we were unable to run all of the apps in the dataset via GroddDroid, while with Droidutan we were able to achieve this. We believe, to make an "exact" comparison of both these tools, there stimulation techniques, one could experiment with the same number of apps, supposedly with the whole piggybacked dataset.

Furthermore, one main difference in our experiment was that GroddDroid was run only one time, stimulating each app maximum 10 times. Although, we justified this by stating that running GroddDroid twice on a single app will not demonstrate major differences; we believe that it is significant emphasize the possibility that modifying GroddDroid to run more than one time may yield better results by eliminating minor randomness occurring in feature vectors.

Moreover, we stated that F1 Scores of GroddDroid was significantly lower than Droidutan. Although we stated that the reason for this maybe the high effectiveness of GroddDroid; it may be important pursue this problem in a further research by altering accuracy margin and forcing the experiment for more iterations when using GroddDroid. Although this may eliminate the effectiveness, it may result in better classification accuracy.

In addition to this, since this thesis was about comparing stimulation techniques for detecting repackaged malware, we only used repackaged malware instances and their benign counter parts to conduct our experiments. If one decides to extend our work to different kind of malware, this may bring new insights to how stimulation techniques would work with malware other than repackaged malware.

Lastly, we only compared Random Stimulation Technique and Forcing Execution Technique for timely reasons. In our opinion, extending our experiment to use another stimulation technique, such as Adaptive Stimulation Technique, may bring new possibilities in terms of malware detection.

List of Figures

List of Tables

Bibliography

[1] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong. *GroddDroid*. 2015. URL: http://kharon.gforge.inria.fr/grodddroid.html.

[2] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. V. T. Tong. "GroddDroid: a Gorilla for Triggering Malicious Behaviors." In: *In 10th International Conference on Malicious and Unwanted Software, Fajardo, Puerto Rico* (2015).

[3] E. Alpaydin. *Introduction to Machine Learning*. third. Cambridge, MA: MIT Press, 2014. ISBN: 978-0-262-02818-9.

[4] *Android-specific components of FlowDroid*. URL: https://github.com/secure-software-engineering/soot-infoflow-android.

[5] *AndroidViewClient*. URL: https://github.com/dtmilano/AndroidViewClient.

[6] *Droidmon - Dalvik Monitoring Framework for CuckooDroid*. URL: https://github.com/idanr1986/droidmon.

[7] *FlowDroid Static Data Flow Tracker*. URL: https://github.com/secure-software-engineering/FlowDroid/tree/master/soot-infoflow.

[8] *Genymotion Android Emulator*. URL: https://www.genymotion.com/.

[9] Google. *UI/Application Exerciser Monkey*. URL: https://developer.android.com/studio/test/monkey.html.

[10] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. L. Traon, D. Lo, and L. Cavallaro. "Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting." In: *IEEE Transactions on Information Forensics and Security* (2017).

[11] L. Li, D. Li, T. F. D. A. Bissyande, J. Klein, H. Cai, D. Lo, and Y. L. Traon. "Automatically locating malicious packages in piggybacked android apps." In: *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems* (2017).

[12] Y. Li, Z. Yang, Y. Guo, and X. Chen. "DroidBot: a lightweight UI-guided test input generator for Android." In: *ICSE-C '17 Proceedings of the 39th International Conference on Software Engineering Companion* (2017), pp. 23–26.

[13] 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: *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)* (2014).

[14] J. Neumeier. "Repackaged Malware Detection in Android." Bachelor's Thesis in Informatics. Technical University of Munich, 2017.

[15] *Oracle VM VirtualBox*. URL: https://www.virtualbox.org/.

[16] 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." In: (2017).

[17] *Python*. URL: https://www.python.org/doc/versions/.

[18] 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* (2017), pp. 300–311.

[19] A. Salem. *Aion*. 2017. URL: https://github.com/aleisalem/Aion.

[20] A. Salem. *Droidutan*. 2017. URL: https://github.com/aleisalem/Droidutan.

[21] A. Salem. "Stimulation and Detection of Android Repackaged Malware with Active Learning." In: *Technical University of Munich* (2018).

[22] B. Settles. "Active Learning Literature Survey." In: *Computer Sciences Technical Report 1648* (2010).

[23] *Soot - A Java optimization framework*. URL: https://github.com/Sable/soot.

[24] S. Tong. "Active Learning: Theory and Applications." PhD thesis. Stanford University, 2001.

[25] C. Tumbleson and R. Winiewski. *Apktool*. 2017. URL: https://ibotpeaches.github.io/Apktool/.

[26] M. Y. Wong and D. Lie. "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware." In: *NDSS Symposium* (2016).

[27] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. "Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces." In: *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY)* (2012).

[28] Y. Zhou and X. Jiang. "Dissecting android malware: Characterization and evolution." In: *In Security and Privacy (SP)* (2012), pp. 95–109.

Appendices

A. GroddDroid Adaptations Source Code

A.1. explorer.run_apk

```
def run_apk(self):
  log.info("Running␣APK␣on␣device")

  run_infos = self._get_run_infos()

  run_type = self.config["branchexp"]["run_type"]
  if run_type == "manual":
      self.runner = ManualRunner(run_infos)
  elif run_type == "monkey":
      self.runner = MonkeyUiExerciserRunner(run_infos)
  elif run_type == "grodd":
      self.runner = GroddRunner(run_infos)
  self.runner.run()
  if not isfile(self.seen_tags_file):
      quit( "No␣capture␣file␣found␣at␣" + self.seen_tags_file + ".␣"
            "You␣want␣to␣check␣if␣the␣logcatctl␣was␣correctly␣started." )
  logcatFileParam = self.output_dir + "/dumped_logcat_"
        + str(self.run_id) + ".log"
  self.dump_logcat(logcatFileParam)
  self.filter_logcat_for_droidmon(logcatFileParam)
  prettyPrint("----------␣Extracting␣Droidmon␣Features␣----------------",
        "warning")
  droidmonFeatures =
        extractDroidmonFeatures("%s/dumped_logcat_%s.log_filtered.log"
        \% (self.output_dir, str(self.run_id)))
  with open(self.output_dir + "/extractedDM_" + str(self.run_id)
        + "_traces.log", "w") as fp:
      fp.write('\n'.join('%s' % x for x in droidmonFeatures[0]))
  with open(self.output_dir + "/extractedDM_" + str(self.run_id)
        + "_features.log", "w") as fp:
      fp.write(str(droidmonFeatures[1]))
  prettyPrint("Extracted␣Droidmon␣Features␣:))", "success")
```

```
  init_blare(self.runner.device)
  self.runner = None
```

A.2. explorer.dump_logcat

```
def dump_logcat(self, logcat_file_param):
    adbID = self.config["branchexp"]["device"]
    # change path to Android SDK if necessary
    ANDROID_SDK = "/Users/emirdemirdag/android-sdks"
    ANDROID_ADB = ANDROID_SDK + "/platform-tools/adb"
    adbPath = ANDROID_ADB
    dumpLogcatCmd = [adbPath, "-s", adbID, "logcat", "-d"]

    # Dump the system log to file
    logcatFile = open(logcat_file_param, "w")
    log.info("Dumping␣logcat")
    subprocess.Popen(
        dumpLogcatCmd, stderr=subprocess.STDOUT,
                stdout=logcatFile).communicate()[0]
    logcatFile.close()
```

A.3. explorer.filter_logcat_for_droidmon

```
def filter_logcat_for_droidmon(self, logcat_file_param):
    # Filter droidmon entries related to the APK under test
    log.info("Retrieving␣\"Droidmon-apimonitor-%s\"␣tags␣from␣log"
        % self.manifest.package_name)
    catlog = subprocess.Popen(
        ("cat", logcat_file_param), stdout=subprocess.PIPE)
    try:
        output = subprocess.check_output(("grep", "-i",
                "droidmon-apimonitor-%s"
                % self.manifest.package_name), stdin=catlog.stdout)
    except subprocess.CalledProcessError as cpe:
        log.info("Could␣not␣find␣the␣tag␣\"droidmon-apimonitor-%s␣in␣the␣logs"
                % self.manifest.package_name,
                    "warning")
        return True
    logFile = open("%s_filtered.log" % logcat_file_param, "w")
    logFile.write(output.decode("utf-8"))
    logFile.close()
```

B. Run Experiment Source Code

B.1. main.py

```
# !/usr/bin/python2.7

from Aion.data_generation.reconstruction import *
from Aion.data_inference.learning import ScikitLearners
from Aion.utils.db import *
from src.GroddDroidTest import *

from sklearn.metrics import *
import hashlib, pickle

import os, sys, glob, shutil, argparse, subprocess, sqlite3, \
    time, threading, \
    pickledb, random


def defineArguments():
    parser = argparse.ArgumentParser(prog="main_new.py",
        description="Aion - GroddDroid Experiments")
    parser.add_argument("-x", "--malwaredir",
        help="Malicious APK directory",
        required=True)
    parser.add_argument("-g", "--goodwaredir",
        help="Benign APK directory",
        required=True)
    parser.add_argument("-d", "--datasetname",
        help="A unique name to give to the dataset",
        required=True)
    parser.add_argument("-r", "--runnumber",
        help="Number of the current run",
        required=True)
    parser.add_argument("-f", "--analyzeapks",
        help="Whether to perform analysis on the retrieved APK's",
```

```
    required=False, default="no",
    choices=["yes", "no"])
parser.add_argument("-t", "--analysistime",
    help="How␣long␣to␣run␣monkeyrunner␣(in␣seconds)",
    required=False,
    default=30)
parser.add_argument("-u", "--analysisengine",
    help="The␣stimulation/analysis␣engine␣to␣use",
    required=False,
    choices=["droidbot", "droidutan",
            "grodddroid"],
    default="droidutan")
parser.add_argument("-v", "--vmnames",
    help="The␣name(s)␣of␣the␣Genymotion␣machine(s)␣to␣use␣" \
         "for␣analysis␣(comma-separated)",
    required=False, default="")
parser.add_argument("-z", "--vmsnapshots",
    help="The␣name(s)␣of␣the␣snapshot(s)␣to␣restore␣" \
         "before␣analyzing␣an␣APK␣(comma-separated)",
    required=False, default="")
parser.add_argument("-a", "--algorithm",
    help="The␣algorithm␣used␣to␣classify␣apps",
    required=False,
    default="Ensemble",
    choices=["KNN10", "KNN25", "KNN50",
            "KNN100", "KNN250",
            "KNN500", "SVM", "Trees25",
            "Trees50",
            "Trees75", "Trees100",
            "Ensemble"])
parser.add_argument("-s", "--selectkbest",
    help="Whether␣to␣select␣K␣best␣features␣" \
         "from␣the␣ones␣extracted␣from␣the␣APK's",
    required=False,
    default=0)
parser.add_argument("-e", "--featuretype",
    help="The␣type␣of␣features␣to␣consider␣during␣training",
    required=False,
    default="hybrid",
    choices=["static", "dynamic",
            "hybrid"])
```

```
    parser.add_argument("-m", "--accuracymargin",
        help="The␣margin␣(in␣percentage)␣within␣which␣" \
             "the␣training␣accuracy␣is␣allowed␣to␣dip",
        required=False, default=1)
    parser.add_argument("-i", "--maxiterations",
        help="The␣maximum␣number␣of␣iterations␣to␣allow",
        required=False,
        default=25)
    parser.add_argument("-o", "--onlyAnalyze",
        help="If␣the␣experiment␣should␣stop␣after␣analysis␣phase",
        required=False,
        default="no", choices=["yes", "no"])
    return parser


def main():
    try:
        argumentParser = defineArguments()
        arguments = argumentParser.parse_args()
        runnumber = arguments.runnumber

        prettyPrint(
            "Welcome␣to␣the␣\"Aion\"'s␣dynamic␣GroddDroid␣Experiment␣II")

        if arguments.vmnames == "" and arguments.analyzeapks == "yes":
            prettyPrint(
                "No␣virtual␣machine␣names␣were␣supplied.␣Exiting",
                "warning")
            return False

        allVMs = arguments.vmnames.split(',')
        allSnapshots = arguments.vmsnapshots.split(',')
        availableVMs = [] + allVMs # Initially

        # Load GroddDroid outputs if analysis phase will be skipped.
        if arguments.analyzeapks == "no":
            goodAPKs, malAPKs = [], []
            prettyPrint(
                "Loading␣GroddDroid␣outputs␣from␣\"%s\"␣and␣\"%s\"" % (
                    arguments.malwaredir,
                    arguments.goodwaredir))
```

```
    # Retrieve malware GD
    for fileName in glob.glob(
            "%s/*_grodddroid" % arguments.malwaredir):
        apk = fileName.replace("_grodddroid",
                               ".apk")
        malAPKs.append(apk)
    if len(malAPKs) < 1:
        prettyPrint(
            "Could␣not␣find␣any␣malicious␣APK's",
            "warning")
    else:
        prettyPrint(
            "Successfully␣retrieved␣%s␣malicious␣instances" % len(
                malAPKs))
    # Retrieve goodware GD
    for fileName in glob.glob(
            "%s/*_grodddroid" % arguments.goodwaredir):
        apk = fileName.replace("_grodddroid",
                               ".apk")
        goodAPKs.append(apk)

    if len(goodAPKs) < 1:
        prettyPrint(
            "Could␣not␣find␣any␣benign␣APK's",
            "warning")
    else:
        prettyPrint(
            "Successfully␣retrieved␣%s␣benign␣instances" % len(
                goodAPKs))

# Initialize and populate database
hashesDB = pickledb.load(getHashesDBPath(), True)
aionDB = AionDB(int(runnumber),
                arguments.datasetname)

################################################
## Run GroddDroid on all supplied APKs ##
################################################

if arguments.analyzeapks == "yes" and arguments.analysisengine == "grodddroid":
    malAPKs = glob.glob(
```

```
        "%s/*.apk" % arguments.malwaredir)
    if len(malAPKs) < 1:
        prettyPrint(
            "Could␣not␣find␣any␣malicious␣APK's",
            "warning")
    else:
        prettyPrint(
            "Successfully␣retrieved␣%s␣malicious␣instances" % len(
                malAPKs))
    # Retrieve goodware APK's
    goodAPKs = glob.glob(
        "%s/*.apk" % arguments.goodwaredir)
    if len(goodAPKs) < 1:
        prettyPrint(
            "Could␣not␣find␣any␣benign␣APK's",
            "warning")
    else:
        prettyPrint(
            "Successfully␣retrieved␣%s␣benign␣instances" % len(
                goodAPKs))

    allAPKs = goodAPKs + malAPKs
    prettyPrint(
        "Starting␣GroddDroid␣Analysis.␣%s␣apps␣to␣go" % len(
            allAPKs),
        "info")
    currentProcesses = []
    while len(allAPKs) > 0:
        prettyPrint("Starting␣analysis␣phase")
        # Step 1. Pop an APK from "allAPKs" (Defaut: last element)
        currentAPK = allAPKs.pop()
        # Step 2. Check availability of VMs for test
        while len(availableVMs) < 1:
            prettyPrint(
                "No␣AVD's␣available␣for␣analysis.␣Sleeping␣for␣10␣seconds")
            print[p.name
            for p in currentProcesses]
            print[p.is_alive()
            for p in currentProcesses]
            # 2.a. Sleep for during analysis
            time.sleep(60)
```

```
# 2.b. Check for available machines
for p in currentProcesses:
    if not p.is_alive():
        if verboseON():
            prettyPrint(
                "Process␣\"%s\"␣is␣"
                "dead.␣" \
                "A␣new␣AVD␣is␣"
                "available␣" \
                "for␣analysis" %
                p.name,
                "debug")
        availableVMs.append(p.name)
        currentProcesses.remove(p)
        # Also restore clean state of machine
        if len(allAPKs) % 25 == 0:
            vm = p.name
            snapshot = allSnapshots[
                allVMs.index(vm)]
            prettyPrint(
                "Restoring␣snapshot␣"
                "\"%s\"␣" \
                "for␣AVD␣\"%s\"" % (
                    snapshot, vm))
            restoreVirtualBoxSnapshot(
                vm, snapshot)

    elif \
    checkAVDState(p.name, "stopping")[
        0] or \
            checkAVDState(p.name,
                          "powered␣off")[
                0]:
        prettyPrint(
            "AVD␣\"%s\"␣is␣stuck.␣"
            "Forcing␣a␣restoration"
            % p.name,
            "warning")
        vm = p.name
        snapshot = allSnapshots[
            allVMs.index(vm)]
```

```
                restoreVirtualBoxSnapshot(vm,
                                          snapshot)

            print[p.name
            for p in currentProcesses]
            print[p.is_alive()
            for p in currentProcesses]

        # Step 3. Pop one VM from "availableVMs"
        currentVM = availableVMs.pop()

        if verboseON():
            prettyPrint(
                "Running␣\"%s\"␣on␣AVD␣\"%s\"" % (
                    currentAPK, currentVM))

        # Step 4. Start the analysis thread
        if len(currentProcesses) < 2:
            pID = int(time.time())
            p = GroddDroidAnalysis(pID, currentVM,
                                   currentVM,
                                   currentAPK, "")
            p.start()
            currentProcesses.append(p)

        prettyPrint("%s␣APKs␣left␣to␣analyze" % len(
            allAPKs), "output")

    # Just make sure all VMs are done
    while len(availableVMs) < len(allVMs):
        prettyPrint(
            "Waiting␣for␣AVD's␣to␣complete␣analysis")
        # 2.a. Sleep for during analysis
        time.sleep(60)
        # 2.b. Check for available machines
        for p in currentProcesses:
            if not p.is_alive():
                availableVMs.append(p.name)
                currentProcesses.remove(p)

    prettyPrint("GroddDroid␣is␣done!", "success")
```

```
if arguments.onlyAnalyze == "yes":
    prettyPrint(
        "onlyAnalyze is set to yes. Stopping after GroddDroid.",
        "success")
    return False

# Set initial metrics and parameters
iteration = 1 # Initial values
reanalysis = False
currentMetrics = {"accuracy": 0.0, "recall": 0.0,
                  "specificity": 0.0,
                  "precision": 0.0, "f1score": 0.0}
previousMetrics = {"accuracy": -1.0, "recall": -1.0,
                   "specificity": -1.0,
                   "precision": -1.0,
                   "f1score": -1.0}
# Use this as a cache until conversion
reanalyzeMalware, reanalyzeGoodware = [], []


# Split the data into training and test datasets
malTraining, malTest, allFeatureFiles = [], [], []
goodTraining, goodTest = [], []
malTestSize, goodTestSize = len(malAPKs) / 3, len(
    goodAPKs) / 3
# Start with the malicious APKs
while len(malTest) < malTestSize:
    malTest.append(malAPKs.pop(
        random.randint(0, len(malAPKs) - 1)))
malTraining += malAPKs
prettyPrint(
    "[MALWARE] Training dataset size is %s, "
    "test dataset size is %s" % (
        len(malTraining), len(malTest)))
# Same with benign APKs
while len(goodTest) < goodTestSize:
    goodTest.append(goodAPKs.pop(
        random.randint(0, len(goodAPKs) - 1)))
goodTraining += goodAPKs
prettyPrint(
```

```
    "[GOODWARE] Training dataset size is %s, "
    "test dataset size is %s" % (
        len(goodTraining), len(goodTest)))

####################################
#### Experiment Phase ####
####################################

### Start iterating the experiment ###
while (round(
        currentMetrics["f1score"] - previousMetrics[
            "f1score"],
        2) >= -(
        float(
            arguments.accuracymargin) / 100.0)) and (
        iteration <= int(arguments.maxiterations)):

    # Retrieve only training features to train the classifiers
    allApps = malTraining + goodTraining if not \
        reanalysis else reanalyzeMalware + reanalyzeGoodware

    # Set/update the reanalysis flag
    reanalysis = True if iteration > 1 else False
    prettyPrint(
        "GroddDroid Experiment: iteration #%s" % iteration,
        "info2")
    # Update the iteration number
    aionDB.update("run", [
        ("runIterations", str(iteration))],
                 [("runID", runnumber), (
                     "runDataset",
                     arguments.datasetname)])

    ######################################################################
    # Load the JSON and feature files as traces before classification #
    ######################################################################

    # Retrieve all feature files from the current iteration
    if arguments.analysisengine == "grodddroid":
        if not reanalysis:
            for app in allApps:
```

```
                appFeatureFile = \
                    app.replace\
                        (".apk",
                            "_grodddroid/extractedDM_%s_features.log"
                            % str(
                            iteration - 1))
                if os.path.exists(appFeatureFile):
                    allFeatureFiles.append(
                        appFeatureFile)
            else:
                for app in reanalyzeGoodware + reanalyzeMalware:
                    currFeatureFile = app.replace(
                        ".apk",
                        "_grodddroid/extractedDM_%s_features.log" % str(
                            iteration - 2))
                    nextFeatureFile = app.replace(
                        ".apk",
                        "_grodddroid/extractedDM_%s_features.log" % str(
                            iteration - 1))
                    if currFeatureFile in allFeatureFiles:
                        for (ix, itemx) in enumerate(
                                allFeatureFiles):
                            if itemx == currFeatureFile:
                                if os.path.exists(
                                        nextFeatureFile):
                                    allFeatureFiles[
                                        ix] = nextFeatureFile
                                    prettyPrint(
                                        "Misclassified␣%s␣is␣" \
                                        "swapped␣with␣%s␣in␣iteration␣%s" % (
                                            currFeatureFile,
                                            nextFeatureFile,
                                            iteration))
                                elif not os.path.exists(
                                        nextFeatureFile):
                                    randFeatureFile = fileName.replace(
                                        ".apk",
                                        "_grodddroid/extractedDM_%s_features.log"
                                        % str(
                                            random.randint(1,
                                                    iteration - 2)))
```

```
                            allFeatureFiles[
                                ix] = randFeatureFile
                            prettyPrint(
                                "Still misclassified"
                                " %s is swapped "
                                "with random %s" % (
                                    currFeatureFile,
                                    randFeatureFile))
    prettyPrint(
        "Retrieved %s feature files" % len(
            allFeatureFiles))

if len(allFeatureFiles) < 1:
    prettyPrint(
        "Could not retrieve any feature files. Exiting",
        "error")
    return False

prettyPrint("Retrieved %s feature files" % len(
    allFeatureFiles))
# Split the loaded feature files as training and test
Xtr, ytr = [], []
numberEmptyFeatures = 0
for ff in allFeatureFiles:
    fileName = "%s.apk" % ff[:ff.find(
        "_grodddroid")]
    x = Numerical.loadNumericalFeatures(ff)
    if len(x) < 1:
        numberEmptyFeatures += 1
        continue
    if fileName in malTraining:
        Xtr.append(x)
        ytr.append(1)
    elif fileName in goodTraining:
        Xtr.append(x)
        ytr.append(0)
prettyPrint(
    "%s were empty feature vectors" % numberEmptyFeatures,
    "warning")

metricsDict = {}
```

```
############
# Training #
############

# Classifying using [algorithm]
prettyPrint(
    "Classifying␣using␣%s" % arguments.algorithm)
clfFile = "%s/db/%s_run%s_itn%s_%s.txt" % (
    getProjectDir(), arguments.algorithm,
    runnumber, iteration,
    arguments.featuretype)
# Train and predict
if arguments.algorithm.lower().find(
        "trees") != -1:
    e = int(arguments.algorithm.replace("Trees",
                                        ""))
    clf, predicted, predicted_test = \
    ScikitLearners.predictAndTestRandomForest(
        Xtr, ytr, estimators=e,
        selectKBest=int(
            arguments.selectkbest))
elif arguments.algorithm.lower().find(
        "knn") != -1:
    k = int(
        arguments.algorithm.replace("KNN", ""))
    clf, predicted, predicted_test = ScikitLearners.predictAndTestKNN(
        Xtr, ytr, K=k, selectKBest=int(
            arguments.selectkbest))
elif arguments.algorithm.lower().find(
        "svm") != -1:
    clf, predicted, predicted_test = ScikitLearners.predictAndTestSVM(
        Xtr, ytr, selectKBest=int(
            arguments.selectkbest))
else:
    K = [10, 20, 50, 100, 150, 200]
    E = [10, 25, 50, 75, 100]
    allCs = ["KNN-%s" % k for k in K] + [
        "FOREST-%s" % e for e in
        E] + ["SVM"]
    MLResults = ScikitLearners.predictAndTestEnsemble(
```

```
            Xtr, ytr,
            classifiers=allCs,
            selectKBest=int(
                arguments.selectkbest))
        clf = MLResults[0]
        predicted = MLResults[1]
        if len(MLResults) > 2:
            predicted_test = MLResults[2]

    # Write to file
    open(clfFile, "w").write(pickle.dumps(clf))
    metrics = ScikitLearners.calculateMetrics(ytr,
                                              predicted)
    metricsDict = metrics

    # Print and save results
    prettyPrint(
        "Metrics␣using␣%s␣at␣iteration␣%s" % (
            arguments.algorithm, iteration),
        "output")
    prettyPrint("Accuracy:␣%s" % str(
        metricsDict["accuracy"]), "output")
    prettyPrint(
        "Recall:␣%s" % str(metricsDict["recall"]),
        "output")
    prettyPrint("Specificity:␣%s" % str(
        metricsDict["specificity"]),
                "output")
    prettyPrint("Precision:␣%s" % str(
        metricsDict["precision"]),
                "output")
    prettyPrint("F1␣Score:␣%s" % str(
        metricsDict["f1score"]), "output")
    # Insert datapoint into the database
    tstamp = getTimestamp(includeDate=True)
    learnerID = "%s_run%s_itn%s" % (
        arguments.algorithm, runnumber, iteration)
    aionDB.insert(table="learner",
                  columns=["lrnID", "lrnParams"],
                  values=[learnerID, clfFile])
    aionDB.insert(table="datapoint",
```

```
                columns=["dpLearner",
                         "dpIteration", "dpRun",
                         "dpTimestamp",
                         "dpFeature", "dpType",
                         "dpAccuracy", "dpRecall",
                         "dpSpecificity",
                         "dpPrecision",
                         "dpFscore"],
                values=[learnerID, str(iteration),
                        runnumber, tstamp,
                        arguments.featuretype,
                        "TRAIN", str(
                        metricsDict["accuracy"]),
                        str(metricsDict[
                                "recall"]),
                        str(metricsDict[
                                "specificity"]),
                        str(metricsDict[
                                "precision"]),
                        str(metricsDict[
                                "f1score"])])

    # Save incorrectly-classified training instances for re-analysis
    # Reset the lists to store new misclassified instances
    reanalyzeMalware, reanalyzeGoodware = [], []
    for index in range(len(ytr)):
        if predicted[index] != ytr[index]:
            cfeat = allFeatureFiles[index]
            capk = "%s.apk" % cfeat[:cfeat.find(
                "_grodddroid")]
            if capk in malTest + goodTest:
                prettyPrint(
                    "Skipping␣adding␣test␣file␣"
                    "\"%s\"␣to␣the␣reanalysis␣lists"
                    % cfeat)
            else:
                # Add to reanalysis lists
                if cfeat.find("malware") != -1:
                    reanalyzeMalware.append(capk)
                else:
                    reanalyzeGoodware.append(capk)
```

```
        prettyPrint(
            "Reanalyzing␣%s␣benign␣apps␣and␣%s␣malicious␣apps" % (
                len(reanalyzeGoodware),
                len(reanalyzeMalware)),
            "debug")

        # Swapping metrics
        previousMetrics = currentMetrics
        currentMetrics = metricsDict

        # Commit results to the database
        aionDB.save()

        # Update the iteration number
        iteration += 1

    # Final Results
    prettyPrint(
        "Training␣results␣after␣%s␣iterations" % str(
            iteration - 1),
        "output")
    prettyPrint(
        "Accuracy:␣%s" % currentMetrics["accuracy"],
        "output")
    prettyPrint("Recall:␣%s" % currentMetrics["recall"],
                "output")
    prettyPrint("Specificity:␣%s" % currentMetrics[
        "specificity"], "output")
    prettyPrint(
        "Precision:␣%s" % currentMetrics["precision"],
        "output")
    prettyPrint(
        "F1␣Score:␣%s" % currentMetrics["f1score"],
        "output")

    # Update the current run's end time
    aionDB.update("run", [
        ("runEnd", getTimestamp(includeDate=True))],
                  [("runID", runnumber)])
```

```
#########################################################
# Commence the test phase using the "best classifier" #
#########################################################
# 1. Retrieve the best classifier and its iteration (X)
results = aionDB.execute(
    "SELECT␣*␣FROM␣datapoint␣WHERE␣dpRun='%s'␣"
    "AND␣dpFeature='%s'␣ORDER␣BY␣dpFScore␣DESC" % (
        runnumber, arguments.featuretype))
if not results:
    prettyPrint(
        "Could␣not␣retrieve␣data␣about␣the␣training␣phase.␣Exiting",
        "error")
    aionDB.close()
    return False

data = results.fetchall()
if len(data) < 1:
    prettyPrint(
        "Could␣not␣retrieve␣data␣about␣the␣training␣phase.␣Exiting",
        "error")
    aionDB.close()
    return False

# 1.a. Best classifier should be the first entry
bestClassifier, bestItn, bestF1score, bestSp = \
data[0][1], data[0][2], \
data[0][11], data[0][9]
if verboseON():
    prettyPrint(
        "The␣best␣classifier␣is␣%s␣at␣iteration␣%s␣"
        "with␣F1score␣of␣%s␣and␣Specificity␣score␣of␣%s" % (
            bestClassifier, bestItn, bestF1score,
            bestSp), "debug")
# 1.b. Load classifier from hyper parameters file
results = aionDB.execute(
    "SELECT␣*␣FROM␣learner␣WHERE␣lrnID='%s'" % bestClassifier)
if not results:
    prettyPrint(
        "Could␣not␣find␣the␣hyperparameters␣file␣"
        "for␣\"%s\".␣Exiting" % bestClassifier,
        "error")
```

```
    aionDB.close()
    return False

data = results.fetchall()
if len(data) < 1:
    prettyPrint(
        "Could␣not␣find␣the␣hyperparameters␣file␣"
        "for␣\"%s\".␣Exiting" % bestClassifier,
        "error")
    aionDB.close()
    return False

clfFile = data[0][
            3] or ''
prettyPrint(clfFile, "warning")
if not os.path.exists(clfFile):
    prettyPrint(
        "The␣file␣\"%s\"␣does␣not␣exist.␣Exiting" % clfFile,
        "error")
    aionDB.close()
    return False

prettyPrint(
    "Loading␣classifier␣\"%s\"␣from␣\"%s\"" % (
    bestClassifier, clfFile))
clf = pickle.loads(open(clfFile).read())

# 2. Classify feature vectors
P, N = 0.0, 0.0
# To keep track of majority vote classification
TP_maj, TN_maj, FP_maj, FN_maj = 0.0, 0.0, 0.0, 0.0
# To keep track of one-instance classification
TP_one, TN_one, FP_one, FN_one = 0.0, 0.0, 0.0, 0.0
for app in malTest + goodTest:
    prettyPrint("Processing␣test␣app␣\"%s\"" % app)
    # 2.a. Retrieve all feature vectors up to [iteration]
    appVectors = {}
    for i in range(1, bestItn + 1):
        appFeatureFile = \
            app.replace(".apk",
                "_grodddroid/extractedDM_%s_features.log" % i)
```

```
        if os.path.exists(appFeatureFile):
            v = Numerical.loadNumericalFeatures(
                appFeatureFile)
            if len(v) > 1:
                appVectors["itn%s" % i] = v

    if len(appVectors) < 1:
        prettyPrint(
            "Could␣not␣retrieve␣any␣feature␣vectors.␣Skipping",
            "warning")
        continue

    prettyPrint(
        "Successfully␣"
        "retrieved␣%s␣feature␣vectors␣of␣type␣\"%s\"" % (
            len(appVectors), arguments.featuretype))
    # 2.b. Classify each feature vector using the loaded classifier
    appLabel = 1 if app in malTest else 0
    if appLabel == 1:
        P += 1.0
    else:
        N += 1.0
    labels = ["Benign", "Malicious"]
    appMalicious, appBenign = 0.0, 0.0
    for v in appVectors:
        predictedLabel = \
        clf.predict([appVectors[v]]).tolist()[0]
        prettyPrint(
            "\"%s\"␣app␣"
            "was␣classified␣as␣\"%s\"␣"
            "according␣to␣iteration␣%s" % (
                labels[appLabel],
                labels[predictedLabel],
                v.replace("itn", "")), "output")
        classifiedCorrectly = "YES" if labels[
                                        appLabel] == \
                                    labels[
                                        predictedLabel] else "NO"
        itnmx = v.replace("itn", "")
        aionDB.insert("testapp",
                      ["taName", "taRun",
```

```
                    "taIteration", "taType",
                    "taClassified", "taLog"],
                   [app, runnumber, itnmx,
                    labels[appLabel],
                    classifiedCorrectly,
                    app.replace(".apk",
                                "_grodddroid/"
                                "extractedDM_%s_features.log"
                                % itnmx)])
    if predictedLabel == 1:
        appMalicious += 1.0
    else:
        appBenign += 1.0

# 2.c. Decide upon the app's label
# according to majority vote vs. one-instance
majorityLabel = 1 if (appMalicious / float(
    len(appVectors))) >= 0.5 else 0
oneLabel = 1 if appMalicious >= 1.0 else 0
if appLabel == 1:
    # Malicious app
    if majorityLabel == 1:
        TP_maj += 1.0
    else:
        FN_maj += 1.0
    if oneLabel == 1:
        TP_one += 1.0
    else:
        FN_one += 1.0
else:
    # Benign app
    if majorityLabel == 1:
        FP_maj += 1.0
    else:
        TN_maj += 1.0
    if oneLabel == 1:
        FP_one += 1.0
    else:
        TN_one += 1.0
# 2.d. Declare the classification of the app in question
prettyPrint(
```

```
        "\"%s\" app has been declared as \"%s\" "
        "by majority vote and as \"%s\" "
        "by one-instance votes" % (
            labels[appLabel], labels[majorityLabel],
            labels[oneLabel]),
        "output")

# 3. Calculate metrics
accuracy_maj, accuracy_one = (TP_maj + TN_maj) / (
            P + N), (
                                    TP_one + TN_one) / (
                                        P + N)
recall_maj, recall_one = TP_maj / P, TP_one / P
specificity_maj, specificity_one = TN_maj / N, TN_one / N
precision_maj, precision_one = TP_maj / (
            TP_maj + FP_maj), TP_one / (
                                       TP_one + FP_one)
f1score_maj, f1score_one = 2 * (
            precision_maj * recall_maj) / (
                                precision_maj + recall_maj), 2 * (
                                precision_one * recall_one) / (
                                precision_one + recall_one)

# 4. Display and store metrics
prettyPrint("Test metrics using %s at run %s" % (
    arguments.algorithm, runnumber), "output")
prettyPrint(
    "Accuracy (majority): %s versus accuracy (one-instance): %s" % (
        str(accuracy_maj), str(accuracy_one)),
    "output")
prettyPrint(
    "Recall (majority): %s versus recall (one-instance): %s" % (
        str(recall_maj), str(recall_one)),
    "output")
prettyPrint(
    "Specificity (majority): %s versus specificity (one-instance): %s" % (
        str(specificity_maj), str(specificity_one)),
    "output")
prettyPrint(
    "Precision (majority): %s versus precision (one-instance): %s" % (
        str(precision_maj), str(precision_one)),
```

```
        "output")
    prettyPrint(
        "F1 Score (majority): %s versus F1 score (one-instance): %s" % (
            str(f1score_maj), str(f1score_one)),
        "output")

    # 4.b. Store in the database
    aionDB.insert(table="datapoint",
                  columns=["dpLearner", "dpIteration",
                           "dpRun",
                           "dpTimestamp", "dpFeature",
                           "dpType",
                           "dpAccuracy",
                           "dpRecall", "dpSpecificity",
                           "dpPrecision",
                           "dpFscore"],
                  values=[bestClassifier, bestItn,
                          runnumber, tstamp,
                          arguments.featuretype,
                          "TEST:Maj",
                          accuracy_maj, recall_maj,
                          specificity_maj,
                          precision_maj, f1score_maj])
    # Same for one-instance classification scheme
    aionDB.insert(table="datapoint",
                  columns=["dpLearner", "dpIteration",
                           "dpRun",
                           "dpTimestamp", "dpFeature",
                           "dpType",
                           "dpAccuracy",
                           "dpRecall", "dpSpecificity",
                           "dpPrecision",
                           "dpFscore"],
                  values=[bestClassifier, bestItn,
                          runnumber, tstamp,
                          arguments.featuretype,
                          "TEST:One",
                          accuracy_one, recall_one,
                          specificity_one,
                          precision_one, f1score_one])
```

```
        # Don't forget to save and close the Aion database
        aionDB.close()

    except Exception as e:
        prettyPrintError(e)
        return False

    prettyPrint("Good␣day␣to␣you␣^_^")
    return True


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

B.2. GroddDroidTest.py

```
#!/usr/bin/python

# Aion imports
from Aion.utils.graphics import *
from Aion.utils.misc import *
# Python imports
import os, subprocess, signal
from multiprocessing import Process


class GroddDroidAnalysis(Process):
    """
    Represents a GroddDroid-driven test of an APK
    """

    def __init__(self, pID, pName, pVM, pTarget, pSt=""):
        """
        Initialize the test
        :param pID: Used to identify the process
        :type pID: int
        :param pName: A unique name given to a proces
        :type pName: str
        :param pVM: The Genymotion AVD name to run the test on
        :type pVM: str
        :param pTarget: The path to the APK under test
```

```
        :type pTarget: str
        :param pSt: The snapshot of the AVD in case restoring is needed
        :type pSt: str
        :param pDuration: The duration of the Droidutan test in seconds (default: 60s)
        :type pDuration: int
        """
        Process.__init__(self, name=pName)
        self.processID = pID
        self.processName = pName
        self.processVM = pVM
        self.processTarget = pTarget
        self.processSnapshot = pSt

    def run(self):
        """
        Runs the GroidDroid
        """
        prettyPrint("%s␣is␣running␣on␣%s" % (
        self.processTarget, self.processVM), "success")
        try:
            # Step 1. Notify beginning
            if verboseON():
                prettyPrint(
                    "Analyzing␣APK:␣\"%s\"" % self.processTarget,
                    "debug")
            if self.processTarget.find(".apk") == -1:
                prettyPrint(
                    "Could␣not␣retrieve␣an␣APK␣to␣analyze.␣Skipping",
                    "warning")
                return False
            # Step 2. Get the Ip address assigned to the AVD
            getAVDIPCmd = ["VBoxManage", "guestproperty",
                           "enumerate", self.processVM]
            avdIP = ""
            result = subprocess.Popen(getAVDIPCmd,
                                      stderr=subprocess.STDOUT,
                                      stdout=subprocess.PIPE).communicate()[
                0].replace('␣', '')
            if result.lower().find("error") != -1:
                prettyPrint(
                    "Unable␣to␣retrieve␣the␣IP␣address␣of␣the␣AVD",
```

```
                "error")
            print
            result
            return False
        index = result.find(
            "androvm_ip_management,value:") + len(
            "androvm_ip_management,value:")
        while result[index] != ',':
            avdIP += result[index]
            index += 1
        adbID = "%s:5555" % avdIP

        ##########################################
        # Step 3 Unleash GroddDroid
        ##########################################

        # 3.1 Change the Output Directory For the GroddDroid
        branchExplorerDir = "" # insert branch exp here
        head, tail = os.path.split(self.processTarget)
        parsedAPKName = "%s/%s" % (head, tail)
        groddDroidOut = parsedAPKName.replace(".apk",
                                              "_grodddroid")
        runType = "grodd"
        maxRuns = 10
        # start gordddroid
        main_cmd_call = ["python3", "-m",
                         "branchexp.main",
                         self.processTarget, "--device",
                         adbID, "--output-dir",
                         groddDroidOut, "--max-runs",
                         "%s" % maxRuns, "--run-type",
                         runType]
        prettyPrint("Command is running %s on %s" % (
        main_cmd_call, self.processVM), "info")
        success = subprocess.call(main_cmd_call,
                                  cwd=branchExplorerDir)
        if success == 0: prettyPrint(
            "Run on %s terminated!!!!!!" % tail)
    except Exception as e:
        prettyPrintError(e)
```

```
        return True

    def stop(self):
        """
        Stops this analysis process
        """
        try:
            prettyPrint(
                "Stopping␣the␣analysis␣process␣\"%s\"␣on␣\"%s\"" % (
                self.processName, self.processVM),
                "warning")
            os.kill(os.getpid(), signal.SIGTERM)

        except Exception as e:
            prettyPrintError(e)

        return True
```