IMPROVING MOBILE MALWARE DETECTORS USING CO-EVOLUTION TO CREATE AN ARTIFICIAL ARMS RACES

by

Raphael Bronfman-Nadas

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2018

© Copyright by Raphael Bronfman-Nadas, 2018

Table of Contents

List of Tables . . . . **v**

List of Figures . . . . **vii**

Abstract . . . . **viii**

Acknowledgements . . . . **ix**

Chapter 1 Introduction **1**

Chapter 2 Literature Review **3**

2.1 Evolutionary Malware Detection 3
2.1.1 Feature Based Detectors 3
2.1.2 Malware Detectors 4

2.2 Evolutionary Malware Generation 4

2.3 Co-evolution and Artificial Arms Races 5

2.4 Android and the Mobile Malware Landscape 7

2.5 Summary 8

Chapter 3 Methodology **9**

3.1 C5.0 Decision Tree 9

3.2 Overview of the Proposed System 11
3.2.1 Malware Generation Module 11
3.2.2 Malware Detection Module 13
3.2.3 Co-evolution 13

3.3 Genetic Algorithm Implementations 15
3.3.1 Solution Representation 16
3.3.2 Solution Modification 17

3.4 Implementation of Genetic Programming 20
3.4.1 Preparation 20
3.4.2 Execution 21
3.4.3 Evaluation 24
3.4.4 A GP Example 24

3.5 Malware-Detection 25
3.5.1 Fitness Score 25
3.5.2 Variation Method 25
3.5.3 Detector Inputs 26

3.6 Malware-Generation 28
3.6.1 Malware evolution 28
3.6.2 Malware Generation 30

3.7 Co-evolution Setup 32
3.7.1 Malware Generator Co-evolution Changes 33
3.7.2 Malware Detector Co-evolution Changes 33

3.8 Evaluated Algorithms 36

3.9 Datasets Used in This Thesis 37
3.9.1 Dataset Sources 37
3.9.2 Dataset Setup 39

3.10 Summary 39

Chapter 4 Evaluations 41

4.1 Preliminary Experiments 41
4.1.1 Permission Feature Selection 42
4.1.2 Code Feature Selection 42
4.1.3 Dataset Size 45
4.1.4 Co-evolved Solutions 47

4.2 Experiments Performed 50
4.2.1 Portability of Solutions 52
4.2.2 Using F-Droid and Google Play 54
4.2.3 Complexity 55
4.2.4 Variation and Exploratory Results 56
4.2.5 Real World Results 56

4.3 Summary 57

Chapter 5 Conclusion 63

5.1 Interesting Features 63

5.2 Limitations of the Malware Generator 64

5.3 Future Works 64

Bibliography 66

Appendix A Sample Results **69**

Appendix B Expanded Table of Results **71**

Appendix C Malware Feature List **78**

C.1 Sink 78

C.2 Source 78

C.3 Trigger 78

C.4 Evasion 80

List of Tables

3.1 Selected 15 Permission features 27

3.2 Android Code Features 28

4.1 Listing of trained detectors 42

4.2 Precision and Recall of different data sizes 46

4.3 Results of training C5.0 trees 50

4.4 Average GP Results at 30 generations without co-evolution . . 52

4.5 Average GP Results at 30 generations with co-evolution 53

4.6 Average GP Results at 100 generations without co-evolution . . 53

4.7 Average GP Results at 100 generations with co-evolution 53

4.8 Selection of apps from GetJar with detection rates 57

A.1 Results of GP with 149 permissions 69

A.2 Results of C5.0 with 149 permissions 69

A.3 Results of GP with 15 permissions only 69

A.4 Results of C5.0 with 15 permissions only 69

A.5 Results of GP with 15 permissions and 8 code features 70

A.6 Results of C5.0 with 15 permissions and 8 code features 70

A.7 A Sample Result of GP with 15 permissions and 8 code features with co-evolution 70

B.1 Table Legend 71

B.2 Results on 30 generations of evolution tested on F-Droid 72

B.3 Results on 30 generations of evolution tested on Google Play apps 73

B.4 Results on 30 generations of evolution tested on Generated Malware 74

B.5 Results on 100 generations of evolution tested on F-Droid . . . 75

B.6 Results on 100 generations of evolution tested on Google Play apps . . . 76

B.7 Results on 100 generations of evolution tested on generated malware . . . 77

List of Figures

3.1 Simple example of a decision tree 10

3.2 The proposed artificial arms race 14

3.3 Co-evolution diagram 32

4.1 Permission features used by C5.0 when trained with 149 Android permissions on a 700 F-Droid dataset 43

4.2 C5.0 tree using 15 permissions and 8 code features, trained on F-Droid 44

4.3 Accuracy vs. Complexity 45

4.4 Accuracy vs. Number of Features 46

4.5 Accuracy vs. Populations accumulated knowledge 49

4.6 Co-Evolved Tug of war: How detectors get better over time . 49

4.7 C5.0 tree using 15 permissions and 8 code features, trained on Google Play 51

4.8 C5.0 tree using 15 permissions and 8 code features, trained on both F-Droid and Google Play 59

4.9 Training over time for a co-evolved detector trained using F-Droid and Google play apps with Drebin malware for 100 generations. (left) The fitness of the best individual on the static dataset. (middle) The fitness of the best individual on the test dataset. (right) The fitness of the population on generated malware. 60

4.10 Boxplot of the number of features used in program solution for F-Droid and Google play trained detectors at 100 generations 60

4.11 Boxplot of program complexity for F-Droid and Google play trained detectors at 100 generations 61

4.12 Non Co-Evolved score of best performer on test set 61

4.13 Co-Evolved the score of best performer on test set 62

Abstract

On the Internet today, mobile malware is one of the most common attack methods. These attacks are usually established via malicious mobile apps. One technique used to combat this threat is the deployment of mobile malware detectors. In this thesis, I aim to explore the similarity between artificial evolution and the cycle of developmental adaptation between malware and cyber security developers. Mobile malware is often a derivative of past results, only modified slightly to avoid detection. In turn, this requires the security, malware detectors, to react and improve. The result is a cycle of modifications of malware and improvements of security. Using this cycle, I shape an artificial evolutionary arms race between mobile malware and malware detectors to consider how this structure will allow for the adaptation of detectors to evolving threats. To model this interaction, I present a co-evolution of two genetic algorithms in the roles of malware and malware detector. The experimental evaluations on publicly available malicious / non-malicious mobile apps and their variants generated by the artificial arms race show that this approach improves the detector's understanding of the problem. During the experiments, the detectors generated were simpler then when not using an artificial arms race, and required less data from each malware sample to detect the malicious behaviours. Given the variety of apps available, I also considered how this approach performs when trained with different sources of non-malicious apps. I considered apps from: F-Droid, an open source app repository for Android; and Google Play, the default installed app store on Android devices. Each source was used to train detectors with one set as a baseline and then testing performance with the other set. I found that the F-Droid trained detectors performed better than the Google Play trained detectors at differentiation between malware and non-malicious apps outside of the source they were trained on. In conclusion, although my evaluations were performed using Android malware, this approach is sufficiently generic that it could be extended to other forms of malware on other platforms.

Acknowledgements

Thank you to everyone that kept me sane as I was writing this: Brandon, Travis, Ross, Irena, Dudley, Florian, and Gabriela. My family of course was also very important to me over this time: my parents, grandmother, and sister. Especially thank you to my Dad who read too many drafts of this thesis. Thank you to the NIMS lab. Thank you to Lucca for getting me involved with the NIMS lab and security. Thank you to my readers: Malcolm and Nauzer. Thank you to my thesis chair Raghav. Thank you to my supervisor Nur. Thank you Raytheon for supporting my research.

Chapter 1

Introduction

Malware detection remains a challenging task as malware is created specifically to avoid discovery by malware detectors. In turn, malware detectors are designed to overcome this elusive opponent by constantly improving their detection methods and techniques. From this vantage point, there exists a type of arms race between those attempting to manufacture malware and those attempting to detect it. As one's performance improves, the other's is reduced. Malware strength is partly based on how undetectable it is, while the performance of detectors is entirely based on how well it correctly classifies malware from benign files.

Evolutionary arms races are a natural phenomenon and can be compared to co-evolution. With research into the evolutionary creation of malware, such as Mystique[17], the adaptation for use with an evolutionary malware detector becomes a natural path to explore[28].

As such, my new contribution to the field of mobile malware security presented in this thesis is the introduction of a methodology to create a simulated competition between mobile malware and malware detectors that improves the ability of the detector to adapt to the evolving threats / malware. I aim to create the artificial arms race framework by using a bio-inspired malware generator and a bio-inspired malware detector under a co-evolutionary paradigm. In this framework, the malware detection module could function in conjunction with the malware generation module, or as an independent module. To better simulate this direct adversarial intent of malware, this thesis presents a method of training a malware detector using a malware generator as a co-evolved genetic algorithm.

The methodology used in this work defines two populations with competitive goals, each individually striving towards its own goal and being rewarded based on how well it performs. Given that the parts of the populations that survive are those that outperformed, or were closest to outperforming, members of the other

population, this grows the knowledge of the population beyond the static sets used in other algorithms. The first population consists of a genetic program deployed to detect malware using feature-based analysis. This serves as the detector which can be evolved to classify as either malware or benign. The second population consists of malware which attempts to become as strong a sample as possible, given that it must remain simultaneously undetectable. This is the population representing the changing malware in the arms race. The malware begins simple and is rewarded for both achieving greater levels of control of the target system, and by being misclassified by the detector population. The premise is based on the logic that the generated malware is used to show what the detectors are lacking and inform them of what future changes are required. Simultaneously, the malware will learn from the results of the detectors as to the kind of malware that is classified incorrectly, providing a dataset for new malware that is more difficult to classify.

The malware generator and malware detector are both independently explored in the literature, however, the novel process of combining these parts within a co-evolution framework is the main contribution introduced here. The aim of this approach is to improve the detector's understanding of the problem, as well as to provide meaningful datasets to represent real life situations for further analysis and measurements. My results show that the signatures / solutions obtained for malware detectors are simpler then not using an artificial arms race, and require less data from each malware sample to be detected / classified accurately. I have also generated results for the adaptability of the solutions to other datasets using different sources of benign software. The comparison of this new method is made to both a traditional genetic program, and a deterministic algorithm, namely the C5.0 Decision Tree.

The rest of the thesis is organized as follows: Literature review and background works are summarized in Chapter 2. The methodology used to generate malware and the detector side of the artificial arms race, along with an explanation of their co-evolution and the experimental setup, is introduced in Chapter 3. The results are presented next in Chapter 4, and finally, conclusions are drawn and the future works are discussed in Chapter 5.

Chapter 2

Literature Review

This chapter will discuss the background behind the proposed methodology. There are several factors involved to create the chosen methodology. First, in section 2.1, I will discuss the methods of detecting malware using genetic algorithms. Next, in section 2.2, I will continue with the discussion of genetic algorithms, this time for the creation of malware. Following the discussion of both sides of this evolution, I move on to the artificial arms race in section 2.3. Finally in section 2.4, I will discuss the benefits and prior research using the Android environment.

2.1 Evolutionary Malware Detection

As with other malware detector, the mobile malware detector must be able to classify malware and benign software in some meaningful way. The state of mobile malware, such as malware on the Android platform, is changing as new malware is developed, creating a moving target for malware detectors. Although the methods become more complex, the basic ideas behind the detection methods remain the same.

2.1.1 Feature Based Detectors

Feature based detection requires using a preprocessor to prepare the specimen into a sequence of identifiable values. The performance of feature based detectors is better when the feature set is tailored to the task [13]. As apps have layers of encryption and structures where byte patterns have different meanings depending on context, human thought needs to go into what features may be relevant to the detection process.

Permission based detection with Android malware has proven to be very successful [26]. A large amount of malware can be classified using only permissions.

This method works because the permissions decide what actions the app is able to perform, with some shown to be more dangerous then others. Permission evaluation also becomes an issue as apps may request more permissions then they actually need [4]. Even if an app is not malicious, requesting additional permissions than required is a security risk.

2.1.2 Malware Detectors

Previous research indicates that by building a more scalable deeper understanding of the malware's method, the application itself is able to better predict future trends [18]. Furthermore, for the task at hand, when feature based analysis was used, previous research using machine learning has produced results showing that indeed malware may be categorized through machine learning approaches. For a Bayesian classifier, it was found that 15 to 20 features were optimal for the detection of android malware [27]. The mobile malware detection space has expanded research on using external app resources to help detect malware. Some detectors have been built to use Battery Life and App Permission for heuristically predicting what may be malware. Skovoroda and Gamayunov discussed that external values such as these could be combined with both static and dynamic analysis, and machine learning to create more robust detectors[24].

2.2 Evolutionary Malware Generation

Malware, in the mobile space, is focused on apps. These apps are often from different malware families, each with their own goals and malicious intents. Due to the structure of Android systems, most malware are modified versions of existing apps that contain added hidden malicious functions. Many apps of the same family have similar feature combinations, required for the exploits or actions they intend to use[2]. Earlier research in this field include vulnerability analysis tools which aim to evolve new variants of known malicious behaviour such as buffer overflow attacks[19, 17] , where Kayacik et. al. [15, 14] evolved using genetic programming against open source anomaly detectors (stide, ph) [9] and intrusion detection systems (snort)[21]. Furthermore, Fraser et al. employed a similar approach

to demonstrate a similar concept in "Return-oriented programme evolution with ROPER: a proof of concept" for evolving a ROP-chain to be used as a payload by jumping between different chunks of code within a static binary[10]. Additionally, Noreen et al. and Meng et al. researched similar approaches for evolving mobile malware in [19] and [17], respectively. These methods depend on external malware detectors that do not change over the course of the evolutionary system. The detectors can take many forms, such as some pre-trained machine learning methods used in Mystique[17]. In summary, the above previous works demonstrate that by using existing malware, a form of abstract representations, and an evolutionary algorithm, new malware can be created to evolve beyond the capacity of the static detectors. Evolved malware is given the opportunity to find the flaws in the detectors. In doing so, the researchers aim to improve the detectors before such malware is introduced into any Android marketplace.

2.3 Co-evolution and Artificial Arms Races

Arms race is used here to represent the struggle between opposing goals, creating tension which leads to the necessity for adaptation. A good representation found in nature is that of an evolutionary arms race. A predator may have its survival linked to its goal of catching prey. Prey on the other hand would have its survival linked to the goal of avoiding the predator. Both sides' survival depends on the others ability to not perform its goal. The individuals that are able to complete their goal are more likely to create offspring with individual traits that made it able to complete said goal. The predators over time build up there ability to hunt as the prey improve there ability to evade. The predator and prey relationship expressed here is an asymmetric evolutionary arms race[6]. The model is similar to the engineered artificial evolutionary arms race between malware developers and cyber security developers. The continued development of new malware techniques is compounded with the number of old attacks that still function[16]. Reacting and improving security becomes a full time endeavour to counteract the adapting and improving malware.

With the advent of the use of AI to aid in the creation of malware, some protections must be taken on the defensive side. The purely reactionary model for new

technology is less effective when the system performing the attacks is adapting as its attacks are caught.

Evolutionary algorithms are naturally exploitative, in a similar way that malware developers poke around existing systems. This makes it a good option for using this technique as a method of developing malware, as was done with Mystique[17]. The framework of co-evolution allows for the building of a detector that simulates attacks against itself, thereby finding ways to defend against attacks tailored to the detector.

Concurrent research involving a similar methodology was performed by Sen et. al. [23]. The method used a co-evolved system to allow for the same reasoning of discovery. The method involves many of the same aspects, such as a population of malware and a population of malware detectors, referred to as anti-malware software.

The detector in [23] used a different malware generator extended from previous work by the same authors[3]. The abilities of that generator greatly outperform the generator created for this thesis, which allows for a more versatile malware generation process, beginning with an existing piece of malware and mutating it. In contrast, the malware generation process presented in this thesis is based on Mystique[17] and uses a framework app combined with custom pieces of malicious functionality to build a new piece of malware without the need of an initial sample.

In [3], the performance of the malware sample was entirely based on how well it was able to avoid detection, and not about the actual ability of the malware. Here, the payload was fixed from an existing malware family, only the obfuscation of the malware was evolved to avoid detection. Due to the structure of the proposed system, some malware would not fit the structure required to evolve.

In [23], the actual detection used feature based analysis, however, the features chosen were quite different than the ones used in this thesis. In total, their detector used up to 146 features, many of which are similar to those use in my detector. Forty features were a selection of permissions, with another hundred, the presence or absence of specific API calls. This means that there was a total of 140 analysis points of chosen binary features. In addition, six numerical values were chosen:

half were related to counting permissions and API calls, with the remainder counting the number of classes, number of methods, and number of goto statements[1].

The detector used was a genetic program tree[2]. This tree constructs a single comparison. This method allows a tree to be trained to distinguish between two classes. Tree GP is only able to distinguished between two classes.

2.4 Android and the Mobile Malware Landscape

The Android operating system has a key place in the mobile market, making it ideal for the experiments done for this thesis. Android has been the most widely used mobile operating system since 2011 and, as with previous popular leaders in the mobile space, it has been the target of choice for most of today's malware.

The operating system, developed by Google, was built on top of Linux and attempts to reduce the attack surface accessible to user installed software. These apps have a very defined structure and are sand-boxed to limit their interaction with the rest of the system [7]. This has not stopped Android malware, but has shaped it, distinctly from that of traditional operating systems, but similarly to other mobile operating systems, such as iOS which also locks apps with restrictive sand boxing [12].

The structure of an Android app is quick and easy to read with a mostly consistent structure. All apps require external actions to be noted statically in a manifest file, categorized into small groups of actions called permissions. These permissions alone are an effective way to determine what exploits and potential harm an app could accomplish [22]. Also, as previously described, there has already been some direct research in the automatic evolution of Android malware with the intent of avoiding this kind of detector[17].

The above reasons indicate that Android is a good starting point on which to perform exploratory research on mobile devices. Thus, in my research, I make use of the Android platform and available Android apps to be able to evaluate my proposed system and present it in conjunction with the previous work in this area.

[1] two code features are the same as my chosen code features: Number of Classes, and Number of Methods

[2] The trees function similarly to the linear GP that was used by me during this thesis, although they produce a different form of solution, and may favour different solutions.

2.5 Summary

From the review of the literature, it would seem that the next logical step in co-evolution is to combine a malware generator and malware detector. Concurrent research proving that this method works well indicates this.

The variable methods of creating detectors shows that many different paths are viable. The minimal inputs needed to provide a performant detector is one such variable that differs between methods. Machine learning, the literature indicates, require few input features to reach an acceptable performance.

Android, being the largest market share of mobile devices makes it the key platform on which to focus. With research into creating and detecting malware on Android being the most advanced of the mobile platforms, it is clear that the next step of combining these methods into one would likely yield interesting results.

Chapter 3

Methodology

This chapter will detail the exact methods that have been designed and developed to implement the artificial arms race proposed in this thesis. Firstly, in section 3.1, I will go over the C5.0 algorithm that will be used as a control and standalone baseline in this research. Secondly, section 3.2 will be an overview of: the larger structure; how the genetic algorithms work; and the various component purposes, inputs, and outputs will be described. Thirdly, section 3.3 will go over in more detail the genetic algorithm structure separately from the modules implementations. Next, section 3.4 will discuss the details of the created implementation along with an example of how a program within the implementation is run. The following two sections, 3.5 and 3.6, detail how each module, Malware detector and Malware generator respectively, is built into a genetic algorithm. Section 3.7 will cover the details of bringing the modules disused into the co-evolution framework.

The remainder of the chapter details how I prepared experiments for this method. Section 3.8 details the frameworks that are used for my experiments. Next, section 3.9 will discuss the datasets used.

Finally, to conclude this chapter, a short summery can be found in section 3.10

3.1 C5.0 Decision Tree

The C5.0 algorithm is based on the C4.5 algorithm, a decision tree and rule building algorithm. I have used the decision tree part of the algorithm as a control and baseline method, using a proprietary release of the software.

C4.5 uses data entropy to decide what feature holds the most information gain for predicting the target feature, in this case, malware or not malware. The dataset is split into subsets based on the decided feature, and then this same tree process is repeated until the data has been split into homogeneous sets of only malware samples or non-malware samples. The tree is formed from the decisions of what

```
SEND_SMS = t: malware
SEND_SMS = f:
:..READ_SMS = t: malware
   READ_SMS = f: clean
```

Figure 3.1: Simple example of a decision tree

features were used to split the dataset at each step. For example, the tree found in figure 3.1 was split using the attribute of `SEND_SMS` first, the 't' half must have been homogeneous with respect to the target value, where as the 'f' half must have been split again. This half would now have the most data entropy on `READ_SMS`, causing a split again to produce two leaf nodes.

When a created dataset is homogeneous on the target attribute, it becomes a leaf node. The leaf node is marked with the value of the target attribute. In the case of one of these leaf datasets not being homogeneous, then the most common feature is chosen to represent the set. This can happen because all other features have been used, leaving no other option but to create a leaf node, or because a decision was made to not over fit the data[1]. In the example tree in figure 3.1, the dataset where `SEND_SMS = t` must have a target attribute label homogeneously (or majority) with the label of malware. The same would have been the case for the sets `SEND_SMS = f and READ_SMS = t` and `SEND_SMS = f and READ_SMS = f`, which were chosen to have the labels malware and clean respectively.

To decide the class of a new sample, the features of the sample are compared along the tree until a result is reached. As an example, see figure 3.1. This tree would take in a new sample, and check the value of `SEND_SMS`. If the value is 't', here meaning true or present, then the sample would be marked as malware. If the value of `SEND_SMS` is 'f', meaning false or not present, then we must make a new decision. So, by checking the result of the `READ_SMS`, we get to either a malware classification or a clean classification. Either way, our tree will finish the classification with a result for the sample.

C5.0 is an improvement that addresses the limitations of C4.5 [11]. This new

[1] One of the features added in C5.0

implementation features elements such as the ability to process other forms of data, such as numerical inputs and boosting which improves the resulting tree. The results of C5.0 are often smaller trees featuring fewer splits and nodes which improves the computational resources required [11].

3.2 Overview of the Proposed System

The method proposed is a co-evolved system involving two genetic algorithms, each with their own goal and requirements. The two share incremental results over the generations of each algorithm and progress towards their individual goals.

As discussed earlier, the primary goal of this research is to explore whether creating this artificial arms race / simulated competition between mobile malware and detectors could improve the ability of the detector to adapt to the evolving threats / malware. This will be the output of the malware detector module.

As a secondary goal, this methodology creates new versions of potential malware as a dataset to analyze and to produce signatures / detectors. This byproduct of the methodology naturally provides us with a dataset that can be shared and used among other researches working in the field of mobile malware detection. This will be the output of the malware generator module.

To augment a genetic malware detector, I aim to create the artificial arms race framework by using a bio-inspired malware generator and a bio-inspired malware detector under a co-evolutionary paradigm. To this end, I will use genetic algorithms and genetic programming. This will be how the two independent modules / components interact.

3.2.1 Malware Generation Module

Independently, the malware generation module has the goal of generating a population of effective malware. The process of generating malware is simple enough with the complexity coming from the fact that results created must be effective. The exact nature of what is considered to be effective differs greatly based on what kind of malware is being considered.

I mean to separate this reliance on the specific goals of malware and focus

purely on a more generic approach that could be applied to any form of malware. To accomplish this goal, the method of evaluation for the malware generator is defined by three objectives:

- Aggressiveness: A catch all term for any aggressive action that could be taken by the malware.

- Detectability: This is a measure of how well the malware is able to go undetected. Malware, by its very nature, is malicious and therefore unwanted. The less detectable the malware is, the more effective it is at reducing the number of unwanted detections and subsequent removals. In practice, this means blending in with benign software to such a degree that there is no meaningful way of distinguishing the malware from other apps.

- Evasion: Evasive actions are non-vital and purely exist to to avoid detection. It helps malware increase its effectiveness by ensuring that less aggressive action will be taken towards it. However, using a minimum amount of evasive tactics is preferable for malware, as it adds to the complexity of construction.

Aggressiveness and Evasion are internal factors that depend solely on the app itself, where Detection is an external factor depending not just on the malware itself, but the abilities of an external tool.

The malware generation process must begin with a malware template that can be tweaked and altered to find the efficient solution. The generation process for an individual sample of malware is the application of these alterations. The alteration process performs a search of the parameter space to find an effective solution with the goal of optimizing for high aggressiveness, low detection, and low evasion.

The malware generation module used in this thesis is based on the Mystique method introduced by Meng et. all [17]. While sufficiently generic, this method's definition of effectiveness and search procedure was created with the intent of finding the effectiveness of privacy leaking malware. As this form of malware was also the target malware for this thesis, this model is a good fit for my methodology.

3.2.2 Malware Detection Module

Independently, the malware detection module has the goal of generating a singular solution with the ability to classify various malware and benign samples.

The detection process can be considered as a classification task for the purposes of applying machine learning with the two classes of malware and benign.

In this case, the method of evaluation for the malware detector is defined by a single goal:

- Performance: The degree to which the detector is able to properly classify.

Due to the risks of misclassification, adding unbalanced penalties to performance is an option. For example, it may be less damaging to mark benign software as malware than it would be to classify malware as benign. In this case, unbalanced penalties may be an option, providing less of a penalty for one type of misclassification than another. For the series of experiments in this thesis, this was not performed, as penalties to performance were treated the same for any kind of misclassification.

3.2.3 Co-evolution

The important factors to understand why an arms race is a logical next step in this research is how the metrics of ’Detection’ for the malware generator and of ’Performance’ for the detector are the same concept viewed from two different perspectives. These two are based on external factors, whereas ’Aggressiveness’ and ’Evasion’ are based on internal factors.

There are performance metrics that depend on outside factors and are therefore influenced by the external information. Moreover, the two external factors are related to the same goal of detection. These are the active and passive forms of detection[2].

To illustrate, the measure of ’Detection’ is based on external detectors, as the act is performed on the malware. The detector is asked if a sample presented is malware. Therefore the amount of detectability is not only determined by the sample,

[2]Active being the detector attempting to classify, and passive being the action of being classified by the detector.

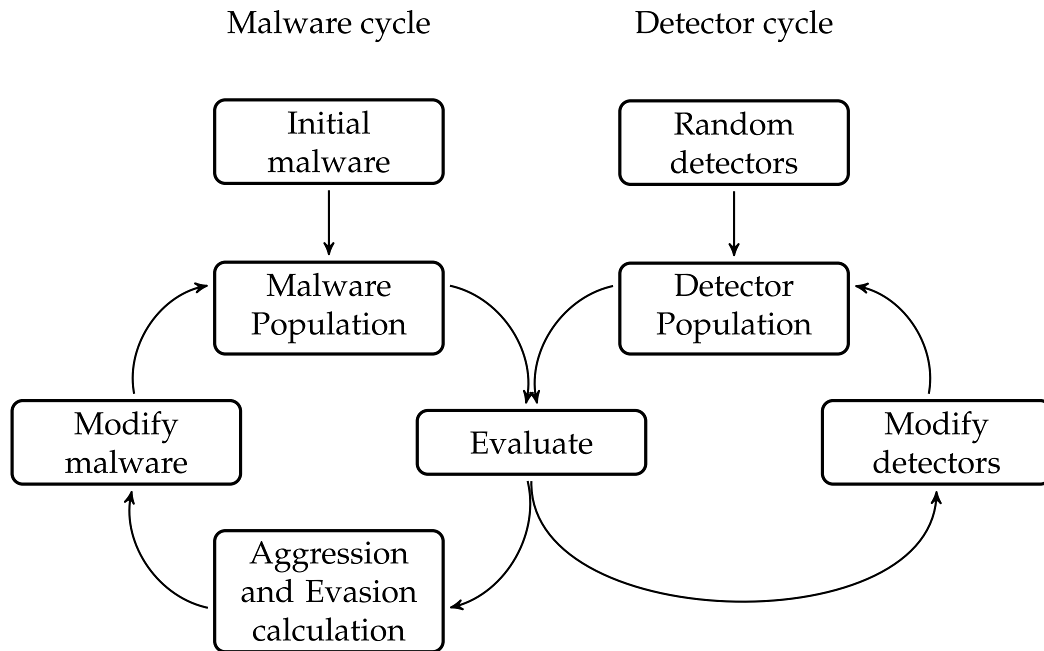

Figure 3.2: The proposed artificial arms race

but also the detector. Having a consistent detector, as when evaluating during co-evolution between modifications of the population, removes the factor of being tested and leaves the only changing part to be the malware being tested.

Similarly, the same logic can be applied to malware detection 'Performance'. If the malware remains static, the opposite case, then only the active role of detecting is modified. The detector can have its performance compared against other detectors as a separate measure that is unrelated to the 'Detection' measurement.

Both methods are simultaneously measured when testing every combination, ranking malware samples against detectors, and detectors in terms of malware samples. This dual use is what leads to the core idea of co-evolution as the core idea behind the artificial arms race.

The arms race is a summary of the intended behaviour of this relationship. A diagram of the process can be seen in Figure 3.2. The process starts with initializing some form of malware, and some form of detectors. These detectors can be random, however applying rules to create useful programs in the initial population may be used. The evaluation process discussed above measures the performance for the detector and detection for the malware. Each sits in a population

that is initially random. The evaluation process compares both the malware and the detectors. This information is then used to modify both the malware and the detectors, with the included calculated values of Aggression and Evasion to create the next population. This cycle may be repeated any number of times in order to produce a population of detectors and a population of malware that approach the solutions of their respective tasks. The advantage of the shared evaluation allows for incremental improvements for the next generation to be based off of the failings from the current.

3.3 Genetic Algorithm Implementations

Genetic algorithms are based on biological evolution. In the biological analogy:

- There exists a population.
- The population changes over time.
- New elements in the population are created from the existing ones, either one or more.
- When an element creates offspring, that offspring is similar to each of its parents.
- Any element has the potential to create new offspring, but those that are more fit will be more likely to create more offspring.

Applying these basic points to the a computational problem creates a genetic algorithm. The population is used to create a solution by delving into a search space and use the principals of evolution to arrive at a solution.

For this to function, here is what is needed:

- A representation of the solution in a modifiable state, called a genotypic state.
- Each solution can be modified to create new solutions, either by taking a random step, called mutation, or by combining multiple solutions, called crossover.

- Given that offspring are similar to their parents, the application of the solution should be as well.
- The value of fitness is now based on how well the given task was performed.

By making this comparison, many of the ways that biological evolution succeeds can be replicated to create a genetic algorithm. The main focus depends on a few key factors: the representation of a solution, the methods of modification, and the method of evaluating the individual solutions.

3.3.1 Solution Representation

The representation of a solution has a great impact on the way the algorithm is implemented, however, the method of representation may have a lot of variety. Most problems will require solutions in a form that would be hard to modify. Therefore, most problems have two representations, genotypic and phenotypic, each form having a specific use. The genotypic form has two requirements: to be modifiable and to be convertible into a usable form; and the phenotypic form has just one requirement: to be an active solution for the problem.

Phenotypic solutions are the end goal, whereas genotypic ones are the forms built for alteration through the genetic algorithm. The analogy with biology would be DNA for the genotypic form, and the active protein as the phenotypic form.

An advantage of the genotypic representation is a mapping, such that every genotypic representation should be directly mappable to a phenotypic solution. These should be viewed as one and the same solution, and the fitness expressed in the phenotypic state should apply to the genotypic representation. Secondly, most, if not all, genotypic solutions should be valid. The more errors found, the less quickly the algorithm will be able to find valid solutions. Finally, solutions' representation should be transferable, that is, a part of a solution in the genotypic form should have the same or similar effect in other solutions.

In this thesis, two methods of genotypic representation are used. The first representation that is used is feature based, which will be discussed in section 3.3.2. The other representation is genetic programming, which will be discussed in more detail in section 3.3.2.

3.3.2 Solution Modification

Solution modification has a few requirements. Firstly, the domain can be a large search space, and finding a good solution may be difficult without guidance. Secondly, by modifying old solutions, we start from an evaluated point within the search space before taking a step within the search space.

- **Mutation**, using a single parent
- **Crossover**, using two or more parents

The solution will be modified versions of the parents. Mutation will take the single value and modify it randomly. Crossover, in this implementation, will take two solutions, combines them, to create a single solution that has aspects of both[3]. During crossover, some mutation of the resulting solution may also be performed. It is also possible that crossover can create as many children as parents.

Feature Based Representation

One way of breaking down a solution is into independently acting features. Each feature is unique and can only be modified with valid options for that feature. It becomes easy to see that such features can be representative of a large or small concept and the search space will have one dimension per feature. For example, evolving a byte string can be implemented where each feature is a binary value. The values for each feature are 0 and 1, and the number of features is the number of bits in the string.

When more complex features are considered, there must be an implemented method of transferring between the features and the phenotypic representation. Although, the reverse is not necessary, it can be useful to have a one to one mapping. This feature based system has the advantage where complex ideas, that can be in multiple states, can be made as important as any small change. At the genetic level, they behave the same.

The breakdown of a solution into features requires a large amount of thought into how the representation should work. However, the method is adaptive to

[3]To determine how the solutions are combined, randomness can be introduced.

many different kinds of values. The methodology of modifying a feature depends on the kind of feature it is.

Of the notable options, the most important for this work is that of the binary feature. A binary feature can either be on or off, as with the byte example from earlier. These features are easy to work with, but have no middle ground or alternative forms. Many features may be required before a large enough search space is created. However, they do not directly represent bytes, which aids in the expressiveness of the features.

Feature based representations have a simple mechanism for mutation. Each feature is considered independently and is mutated randomly. This can either take the current state into account, which means that the values are similar to the original, or be completely random. The option must be decided based on each particular feature. In either case, the mutation should be with respect to the way the feature is specified.

Crossover for feature based representations can be achieved in a few different ways. Firstly, if there is no given order to the features, they can be randomly assembled between the two chosen parents chosen. This creates two new feature sets built from the previous. If there is order, a less chaotic, often better, solution is to order the features and then perform one or two point crossover. To perform crossover, randomly select the intended number of points between features (e.g. for two point crossover, two points would be selected, perhaps after the third feature and another before the last). Then, select one parent, and begin copying features in order to the child. At the selected point, switch to the other parent as the source of values, repeating this process for each point. Additional methods of crossover may be performed. For example, if it is possible for the parents' values to be averaged, that may be a valid alternative.

Genetic Programming

Genetic Programming is the representation of a solution in a programming language. As such, this method has rules and must function according to some simulation of programming.

This method relies on the fact that, like assembly languages, simple commands

can build up to create more complex strategies. Here, a program will be the intended solution. The representation can be stored as a sequence of instructions. Each instruction is executed independently and affects the state of the machine. After all instructions are executed, the result comes from the final state.

As programs can get highly unpredictable, as many programs as possible should be valid. Validity also includes mention of the halting problem, as there is no assurance that any program will ever complete. Considering that these programs may be initially random, the problem is amplified. In summary, the programs should be valid in as many circumstances as possible.

One solution to the halting problem comes with the definition of the instruction set to use. If there are no instructions that could cause an infinite loop, then the problem is solved. Although reducing the expression of the programs, it is a possible solution to the problem.

For dealing with the complexity of the instruction set, edge cases that are usually considered to be programmer error must be considered valid. Operations should be consistent when performing the unexpected behaviours. For example, when dividing by zero, the program should not crash and instead recover; The actual effect is not important, as long as some consistent result takes place and the remainder of the program is run.

To create a state that is understandable and limited enough to function with short programs, many, including this one, use registers to act as a fixed number of variables. The state of the machine when read will compare the registers to decide the result. To have a singular output result, a single value from the first register could be used. If more are needed, more registers can be considered for the results.

Instructions themselves do not have an intended order and can be moved around and still maintain their context. However, the order of the instructions in small chunks remains important, maintaining the logic learned into the programs.

Mutation can be performed in a few ways. Firstly, individual instructions can be modified. Instruction modification can be the replacement of a full instruction or just a part of an instruction. Secondly, new instructions can be added randomly to programs. Finally, instructions can be removed. All these mutations can be done

on a single program, as long as the order of remaining instructions is preserved.

Crossover is performed similarly to that of feature based representation. Using crossover points, the first half of one program is mixed with the ending of another. The points chosen to break apart the programs do not need to be the same. Using one or two point crossover, the endings, or a middle section can be replaced within the programs. This is also useful in preserving the order and placement of instructions. However, not requiring a consistent size nor exact placement leads to more variation within the limitations of crossover.

3.4 Implementation of Genetic Programming

A genetic program, or GP, will be used as a classifier, to decide whether a given app is malware or not. The steps involved to achieve this are Preparation(3.4.1), Execution(3.4.2), and Evaluation(3.4.3).

3.4.1 Preparation

To prepare for evaluation, a app must first be broken down into input that can be accepted. Then, this acceptable data can be modified to allow for better distribution of the input.

This thesis' implementation of GP uses all floating point values. To prepare the app, it is broken down into input features from a few different sets of chosen inputs.

Feature set A, the app permissions, has the representation of either true or false values. In this case, the numbers 1 and 0 are used respectively. Feature set B, the code features, has integer values of different code features seen within the program. These integers are converted to floating point, otherwise these values are ready to be used.

For the preprocessing to be more effective, knowing ranges of values as well as averages aid in mean and variance standardization. For most experiments, the part of the dataset that will be seen by the detectors will be used to collect this information. Mean and variance standardization has the advantage of creating a diverse range of values caused by the distribution of probabilities of each feature

in the training set, including those that are binary options.

After features are converted to numerical form, mean and variance standardization of the constants is applied to the input values. Once this is performed, the input values are now ready to be used as inputs for the detector. Where possible, the results are saved in order to avoid this process again as this input will be the same for every detector in every generation.

3.4.2 Execution

This thesis' implementation uses a fixed number of registers. Each register is given a number to uniquely identify it. Before the evolutionary algorithm is started, the number of of registers must be decided. The more registers, the larger the search space will be. While this may slow down the process of discovering good solutions, the number of registers matters for both how to get a result and intermediary values.

All registers start off the program at 0.

Some instructions are taken from the input stream, and as these values are not in registers and cannot be altered, they are only used as inputs.

Constants, as will be described later, are integers from 0 to 255 as decided by the format.

The chosen implementation uses the following options for instructions:

- Arithmetic operations
- If statement
- Function call

These basic operations are mostly independent from each other and can be built up together into a single program. Afterwards, the instructions can be reviewed and reduce to those those actually executed based on the inputs and outputs of each.

To visually represent which values are being used, the following syntax is be applied:

- Registers are indicated by `r[0]`, where the number `0` is replaced by the number of the register.
- Inputs are named, and are written as `in[inputName]`, where `inputName` is replaced by the name of the input.

Register Operations

Arithmetic operations are the most common type of instruction and involves an operation between a register and either another register, an input value, or a constant. The operation is executed using these two values, and the result is stored in a register.

The four options chosen that the register operations can be used with are:

- Addition
- Subtraction
- Multiplication
- Division

Examples of instructions printed in a programming language fashion are as follows:

- `r[1] = r[5] / r[2]`
- `r[4] = r[1] * in[value 5]`
- `r[6] = r[6] + 9`

If Statements

If statements control access to the next instruction based on a comparison. If the comparison succeeds, then the next instruction is executed, otherwise, the instruction is skipped. Sequential if statements get compounded with 'and' logic. The first non if statement instruction will be conditionaly executed only if all if statements before it are evaluated as true.

The actual comparisons that can be performed are register to register, register to input, and register to constant, using one of the following:

- Less than or equal to
- Greater than

Equal to is not often used, so it is added onto *less than* arbitrarily.

Some examples of instructions printed in a programming language fashion are as follows:

- `if( r[6] <= r[5] )`
- `if( r[0] <= 162 )`
- `if( r[5] > in[value 3] )`

Function Calls

The final type of instruction is a single parameter modifier, or function call. These are usually mathematical functions which can be useful in building up more complex equations in the process of solving the problem.

The chosen functions are:

- sin
- cos
- square root
- exponent
- log

These instructions can be performed with either registers or inputs. Some examples of instructions printed in a programming language fashion are as follows:

- `r[6] = sin(r[5])`
- `r[5] = sqrt(r[7])`
- `r[5] = exp(in[value 1])`

3.4.3 Evaluation

To evaluate a program, the program will be decoded into a list of the above mentioned instructions. Any optimization will be performed to remove any instructions that will have no result on the output. For each input given to be evaluated, the program will begin with zeros in each register and run linearly through the program.

To get a result from a program, the chosen method depends on the number of registers. To decide between n options, you can use n registers to bid for each option. Each register holds a floating point value, and the final returned result is the option associated with one of the first n registers. This requires that the number of registers must be greater than n from which to select the options.

3.4.4 A GP Example

The process described uses simple instructions to build up complex behaviours. Running through a simple example, here is a short genetic program:

```
r[0] = r[1] + in[value 5]
r[1] = r[0] * in[value 2]
if( r[0] > 2 )
r[0] = r[0] * r[1]
r[2] = cos(r[4])
```

This five line program is simple enough to compute by hand.

First, I begin by initializing all values related to this program. All registers start off as zero.

The first line sets register 0 to the 5th value of the input, since the value of register 1 is still the default value of 0. Next, the value of register 1 is then set to the value of the 5th input multiplied by the 2nd input. Then, if the 5th input is greater then 2, the value of register 0 will be set to itself multiplied by register 1. This would set register 0 to the value of the 5th input squared times the value of input 2. Finally the value of register 2 is set to the value of 1 because register 4 is still its initial value of 0. The program is now over and the registers are compared to each other.

Depending on how many output values are expected, calculating a result may be different. Assuming three output values, the first three registers will be compared. Registers 0, 1 and 2 will be compared to see which contains the higher value. This output hides much of the complexity of the program, and yet results in a clear general use method for constructing and using what can amount to complex mathematical comparisons which are built, expressed and executed using very few computational resources.

3.5 Malware-Detection

The detector begins with the implementation of a linear GP as described in Section 3.4, along with a framework for evolving and evaluating the programs involved.

3.5.1 Fitness Score

The program is tested on each item in the dataset. The fitness value is based on how many elements it correctly classifies. If the inputs from a malware sample gets classified as malware, or if a benign input is marked as benign, then a fitness point is awarded to that sample. Alternative approaches which increased the complexity of this fitness process were not effective at improving the score. Therefore, this counting approach was chosen.

3.5.2 Variation Method

The approach taken to evolve the generation is proportional selection. This depends on the relative scores between different fitness values. Before that, the top performers are preserved for the next generation. This is to ensure that there is not a regression in the best performers for the training set.

To perform proportional selection, I select an individual from the population with a probability related to its fitness value. This probability is $\frac{\text{fitness score}}{\text{total fitness of all}}$. The values used here ensure that similar fitness values are treated equally, with greater distinctions leading to larger changes within the population. When selecting multiple individuals, such as for crossover, the same individual cannot be chosen more

than once.

With the inclusion of the best previous individuals, new populations will be comprised of better performers, modified in an attempt to improve the solutions with each generation. New material is introduced, as any instruction, when being moved to a new individual, has a small chance of being changed into a new instruction. These new instructions are mutations applied to the child program and ensure that the entire population does not converge too quickly on a single solution. This new material also reduces the likelihood of an initial population that may be missing key instructions being unable to recover an unlucky start.

3.5.3 Detector Inputs

The inputs selected are from two sources. Previous research indicated that detectors of this kind used a variety of sources, leaving the number of features around 20, so this was also my target from the outset. Two sources were selected: 15 Android permissions, and 8 code features.

Android permissions that an app uses are declared in a metadata file known as the Android manifest. Permissions can enable hardware access, operating system features, or access to other apps. There are over 100 officially supported permissions, however, an app can create a new permission for it to accept. For this reason, any number of permissions may be found, in addition to the official Android permissions.

In the first phase of this research, I employed the full list of official permissions from 600 of the apps I planned as the training set, 300 malware and 300 benign, and evaluated two classifiers on these permissions to test which classifier would identify a malicious apps more accurately. To this end, I used the C5.0 decision tree classifier as a representative of the state of the art classifiers, and the linear GP classifier, which I intended to use in my artificial arms race framework. The results of these tests are displayed in Appendix A, Tables A.1 and A.2. The results show that the C5.0 classifier reaches up to 91% using all permissions, whereas the linear GP classifier reaches in average 76% accuracy using all the permissions. To improve this, I focused on a smaller set of permission list features based on previous research [27, 26] and further empirical evaluations. At the end of these

Table 3.1: Selected 15 Permission features

INTERNET
READ_SMS
SEND_SMS
READ_CONTACTS
READ_EXTERNAL_STORAGE
WRITE_EXTERNAL_STORAGE
INSTALL_PACKAGES
BIND_DEVICE_ADMIN
BIND_ACCESSIBILITY_SERVICE
RECEIVE_BOOT_COMPLETED
READ_PHONE_STATE
CAMERA
RECORD_AUDIO
READ_CALENDAR
ACCESS_FINE_LOCATION

evaluations, I choose the most relevant 15 permissions that are likely to be used for malware detection. These 15 permissions are listed in Table 3.1, and results of the evaluations are shown Appendix A Tables A.3 and A.4. Given that both classifiers are very comparable in their accuracy, I concluded that these 15 permissions are more consistent at representing the normal and benign behaviours of the apps.

On the other hand, Android code features are associated with the creation of the code of an app. Every app will include these features to a lesser or greater degree as they are the number of occurances of common features within the code. They include, but are not limited to: the number of classes, the number of interfaces, and the number of instanced variables. In the literature, these features are considered to be representative of the structure and the use of the code [25]. The chosen Android code features are listed in Table 3.2.

Using the aforementioned Android permission and code based features, the linear GP based detector is trained on a subset of Android app samples to set up the artificial arms race framework. In this case, the method of evaluation for this malware detector is defined by a single goal: performance – the number of correctly classified malware samples. It should be noted here that the performance value is measured evenly, so that the imbalances in the dataset will not affect the overall accuracy.

Table 3.2: Android Code Features

Number of Classes
Number of Classes using interfaces
Number of Classes containing annotations
Number of Direct methods
Number of Virtual methods
Number of Abstract methods
Number of Static Member variables
Number of Instanced Member variables

3.6 Malware-Generation

The malware generation breaks down into two sub-modules, the evolutionary method and the malware generation process.

3.6.1 Malware evolution

The core task of this module is to produce better malware. As I am evolving apps, I would like to produce a malware sample that is both effective as a piece of malware and performs well in my test of undetectability. This leads to a multi-objective optimization for the proposed co-evolutionary process. Also, note that this property of the proposed framework makes it different from [23].

A population of malware will be created. Each sample is given random features turned on and off from a list of features, fully listed in Appendix C. The measurement of aggressiveness, evasion, and detectability is performed for each member of the population.

These measures of aggressiveness, evasion, and detectability form the three dimentions used in the multi-objective optimization. To match these measures to the objectives, the objectives are matched with the goals presented in Section 3.2.1. Aggressiveness is optimized for the maximum value. Evasion is optimized for the minimum value. Detectability, while minimized for, instead targets a value of 80% maximum possible score to prevent disconnect during co-evolution. The purpuse of this multiobjective structure is because the surviving elements of each generation are the non-dominated elements within this three dimensional space.

A sample is said to be dominated by another sample when, for all dimensions

it is scored on, no values are greater than in the other sample. For instance, a point at (3, 4) is dominated by the points (4, 4), because of the eclipsed first value with identical other dimensions, and (4, 5), because both values are eclipsed. If one value is lower, as in (2, 5), then the point is not dominated even if other dimensions are higher.

The non-dominated samples are preserved, and the rest are forgotten with each generation, making this similar to dominance rank sorting and only keeping the members of the generation that are zero. In accordance with the feature based representation discussed earlier, the input for any given app should be a number of features to either include or not in the sample. The features are marked based on the functionality that the feature will have on the solution. Features are either aggressive(meaning the feature adds some malicious action to the payload or method for the payload to activate) or defensive(the feature provides some evasive action), and from this I get the Aggressiveness and Evasion of an individual. Aggressiveness is the count of the aggressive features, whereas Evasion is the number of defensive features used to attempt avoidance of detectors.

To get the detectability of the app, it needs to be constructed and let the detectors examine it. These detectors can be external, as within the previous research of Mystique, or, as in the case of this research, the detectors are created by an evolutionary process. In either case, the result for detectability is the number of detectors that are alerted.

The detectability should be minimized. However, I would like to discourage the malware becoming a runaway success, leaving the detector with too difficult of a problem to solve. This is a problem known as disengagement and results in the detectors being unable to differentiate bad solutions from good solutions as all solutions fail to classify most or all malware. Therefore, a target value for the malware to hit was created. The arbitrary value of 80% was chosen as the target number of detectors to deceive. Obtaining 100% undetected would therefore be 20% off of the goal, and a 0% undetected rate would, likewise, be 80% off of the goal.

After the values are computed for the entire population, the algorithm continues with the selection process[4]. A selection of the best samples are stored from this generation. These chosen samples are those that were non-dominated by other elements in the generation. The next generations are built on this stored repository, through mutation or crossover with samples from this storage, which gets larger with each generation.

This process can be ended at any point. In previous work, the ending was reached when no new malware was added to the library due to duplication. However, for this thesis, the generations are forcibly continued for co-evolution. The ending result is not an individual, but a library, pruned to only include the non-dominated.

3.6.2 Malware Generation

Malware generation is done through an independent process, separate from the evolutionary process improving the selection of features. The input is a list of features to include. The output is a signed APK file, an app package file that can be installed directly on the target version of Android.

The initial list of features is a list of desired end results. To actually assemble the app, the full list of requirements is extracted from a dependency table. These dependencies are mostly Android permissions, and may be used in multiple features. Therefore, the process removes duplicate code, giving the app more authenticity, as code duplication looks more formulaic and auto generated.

The feature list, now complete, can be applied to the Android app template. The template shares the structure of the source code directory of the app. The template files create a normal app, with some added tagged locations waiting for the template to replace them. These tags are replaced with the code specified by each feature.

Not all features introduce code. Many of the evasive features require post processing of the app to apply scripts to the byte code after it has been generated. In addition to tags, features can specify some special values which alter the build

[4]This method is similar to IBEA((Indicator-Based Evolutionary Algorithm) search as used in Mystique[17]

process, such as including optional files of the template, or specify build process functions. In this latter case, the build process functions that are changed are settings for the compiler or the use of features within Droidchameleon[20], a tool for adding malware obfuscation to an app.

After tags are inserted, all the files are moved into an Android project and the compiling process is performed. The tool, Droidchameleon [20] is used after compiling the app. The selected features add parameters to the Droidchameleon command and turn on functionality in the obfuscation software. The instantiation process can be done independent of the genetic algorithm, making it possible to create custom malware of this sort very easily for a user.

However, the proposed malware evolution methodology may raise a large problem that cannot be ignored, which is the automated process of turning out a new variant of a malware family very quickly.

Targeted Malware

The malware generated from this method is targeting a simple-to-make and customized malware, Privacy Leakers. Privacy Leakers malware does not rely on exploits or other sensitive material. This class of malware takes information it has access to and slips it off of the device to be collected somewhere else. In this case, Android has a wealth of information that apps can request, so all this malware needs to do is to send it out of the phone to be read by a waiting server.

The type of software determines the types of features. Therefore, this malware has four kinds of features, of which at least one source, one sink, and one trigger are required. There are no such restrictions imposed on the forth type of feature, evasion strategies.

A source is the source of data from which the app will be drawing. Most of these are part of the Android API to retrieve information such as phone number, model, or more personal information such as the history of received SMS messages.

A sink is the method of moving the information retrieved out of the device. Usually this is through HTTP requests or SMS traffic, something that would go unnoticed, and is best not to be a completely new protocol.

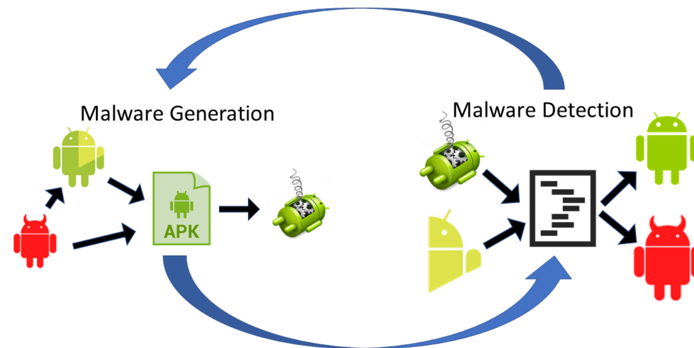

Figure 3.3: Co-evolution diagram

A trigger is the method that starts the source and sink processes. Starting on boot is usually easier to detect, however, if some odd check is done and the correct event triggers, then the malware will run. It can also be helpful to trigger in multiple ways along multiple malware paths.

Evasion strategies are the features that determine which obfuscation techniques to use to hide the true purpose of the app. Most evasion tasks are handled though a tool known as Droidchameleon[20] which was used to perform the automated obfuscation of these apps.

The features considered for the aggressiveness fitness are: the sources, sinks, and triggers. Evasion strategies add to our evasiveness score.

3.7 Co-evolution Setup

Competitive co-evolution is the technique I employ to implement the artificial arms race between the malware detector and the malware generator. With the details of both methods described independently, I consider how they work together within the co-evolution framework.

The structure of the technique is to start up both processes with a message passing system to share data between each other. While both processes loop independently, the internal structure is more of a loop as seen in figure 3.3.

Within this diagram, I illustrate the intended behaviour of the processes. The Devil Android mascot represent the malicious intent. Starting with the malware

features, the malware generator will generate all the android permissions, followed by forming an APK file. This file is then examined and the results are sent to the malware detector. The detector uses as input a selected 15 permissions, represented as the gold section of the android mascot, along with the code features of the malware, and determines it to be malware or not. The co-evolved section of code relies on small changes to both modules and a message passing system that keeps the process ordered.

3.7.1 Malware Generator Co-evolution Changes

For the generator, the change is simple. The app is preprocessed the same way as other apps, revealing the features that will be input for the detector. A summary is created for each sample in the population, and then the entire sample set is passed to the evaluator process.

The response is a number of labels and scores representing how many detectors were able to detect that particular sample. The labels are matched and the evaluation completed. The malware then continues with the evolution process.

This method is similar to the non-co-evolved version of this method, since an external detector used for testing already required this summary and response process. The biggest difference, largely done out of convenience, is to create a summary that is able to be passed to the detector in the form required, rather than passing a full APK file and having to extract the small amount of data required.

3.7.2 Malware Detector Co-evolution Changes

Upon review of the goals of each module (malware generator vs malware detector), it becomes clear that each task is directly at odds with the other. Thus, the natural tension between the goals of the malware generator module and the malware detector module are used to increase the performance of each module.

- The malware detector attempts to classify malware, while avoiding to classify safe software as malware. Naturally, the classifier is only considered to be successful, if it can detect the malware.

- The malware generator creates malware. Malware is only considered to be successful, if it can evade detection of the malware detector.

Due to the malware specific goal of avoiding classification, the classifier is a direct adversary. Likewise, malware attempting to be unclassifiable becomes an adversary of any detector. Detectability can swing between the malware and the detectors, but both cannot be optimal at once.

Algorithm 1 Co-evolution main loop

Input: input_dataset, generation_count, archive_max
Output: malware_detector_population

Initialisation :
1: pop_d ← random_population_GP()
2: pop_m ← random_population_Malware()
3: archive ← []
LOOP for each generation
4: **for** $i = 0$ to *generation_count* **do**
5: malware_dataset = input_dataset ∪ pop_m ∪ archive
6: eval ← Evaluate(pop_d, malware_dataset)
7: Sort pop_m by eval
8: Append first *archive_amount* of pop_m to archive
9: Append last *archive_amount* of pop_m to archive
10: Reduce archive to *archive_max*
11: pop_d ← GP_Evolve(pop_d, eval)
12: pop_m ← IBEA_Evolve(pop_m, eval ∩ pop_m)
13: **end for**
14: **return** pop_d

To operate under the co-evolution, the process summarized in Algorithm 1 operates the cycle of evolution and evaluation. The inputs are a static dataset to be used during evaluation, the number of generations to be performed and the maximum size of the archive. Initialization is performed on lines 1-2. An empty archive is initialized on line 3. Each generation goes through the lines 5-12. The malware used to evaluate the current generation of malware detectors is compiled on line

5 from the static input dataset, the malware in the current malware population along with malware in the archive. Line 6 performs the evaluation testing each malware on each detector, along with calculating any additional required evaluations such as the calculation of Aggressiveness and Evasiveness. *eval* represents the results calculated, with enough detail that a count of how many detectors correctly identified a malware sample as malware could be derived. Lines 7-10 are considered with the archive. Line 7 performs a sort of the malware population by the dominance rank[5]. Line 8 and 9 add the best performers and the worst performers to the archive. The intention here is to add diversity to the content in the archive by purposefully picking what should be easy to identify malware, given that having the worst dominance rank means that sample was the most detected. When appending malware to the archive, duplicates should be removed. Line 10 performs maintenance on the archive, ensuring that the maximum is retained by removing random elements from the archive until a predetermined maximum size is reached[6]. Line 11 and 12 performs the evolution steps of selection and modification of both the malware detector population and the malware population respectively. Finally, the detector population is returned at the end of the process on line 14.

Starting with random initialized populations, to perform a generation, the malware detector population is evaluated using both a static dataset and the malware population. Given that the malware detectors were evaluated using the population of malware, the intersection of that evaluation and the malware population is the detection rate of the malware. The aggressiveness and evasion values are computed using only the malware population, as those measures only use the sample's genotypic form. With the population evaluations done, both are evolved using their respective evolutionary procedures.

To better adapt to the changes in the apps / malware, and to ensure that the final result is robust, as the populations change, an archive is used to select features of the malware population to preserve and to use for detection (evaluations) over

[5]Dominance rank is sorting by the number of other members in the population that dominate a given point. The samples that perform better are closer to zero. [8]

[6]During all experiments discussed within this thesis, the size of the archive was not reached, as such this step was not performed.

generations.

3.8 Evaluated Algorithms

As discussed in previous chapters, I have employed two machine learning algorithms, namely C5.0 and GP, to generate a malware detector. Thus, I performed the following sets of evaluations in order to explore how far I could push the proposed framework:

1. The C5.0 based detector uses both a malware dataset and a benign app set. This creates a decision tree to classify the apps into malware and benign. The complexity of the generated decision tree is the number of nodes present in the tree. Additionally, because C5.0 algorithm uses information gain, the decision tree is capable of choosing the most important features among all the features given. This enabled me to choose the most relevant 15 permissions from the set of all available permissions (149) that represents an app.

2. The GP based detector also uses both a malware dataset and a benign app set without the addition of generated malware. In this case, I can also measure the complexity of the classifier using the number of instructions in the chosen solution's program. Again, GP classifier is able to identify the most important features from the set of all features given. This enabled me to choose the most relevant 15 features agreed on by all used classifiers.

3. Finally, the proposed co-evolved method can be used in one of two ways. Either a benign dataset alone can be used or a benign dataset along with a malware dataset. As malware is created through the process, it is not required to have static dataset. The GP based detector, chosen because of its co-evolution capability, is used in the artificial arms race framework. The same measurements of complexity can be directly compared to other GP evolved detectors, giving a good comparative benchmark between the static and the co-evolved classifiers.

3.9 Datasets Used in This Thesis

The malware provided was a large dataset, and as such, subsampling was employed. The C5.0 classifier was used to verify that the randomly selected apps were comparable to the full version of the dataset. I achieved this by splitting the dataset into the intended size of a training set using random selection of app samples from the full dataset. Building a tree using the subsection compared to the full dataset, I measured the accuracy and calculated the number of features most used in the tree. Randomly selecting the subset for the training set, and verifying the similarity of a well tested algorithm's ability to train, enables the verification of the selected subset. It is important to choose the training set to be as small as possible to decrease the training time. However, at the same time, it is important not to lose accuracy for detection. In the end, the selected training dataset has 0.4% difference in accuracy and includes the top 10 features which were also present in the tree generated by the full dataset.

3.9.1 Dataset Sources

There were two kinds of datasets used during this thesis: Malware and Benign datasets. As discussed in section 3.8, C5.0 and GP without co-evolution require one of each, where the co-evolved algorithm requires only a benign source with an optional malware source.

Malware Sources

Several sources of malware software were used during this research. They are as follows:

1. The Drebin project, which provides a malware dataset specifically for research use [2]. This dataset contains thousands of instances of malware taken as they were submitted to the Google Play Store. In this thesis, I took a subset of one thousand apps. Importantly, 300 apps were chosen to be the test set used in both the F-Droid and Google Play test sets to be discussed later.

2. The other source of malware used during training, as discussed in the methodology, is the malware generator used during co-evolution. A collection of ten thousand apps was collected for testing, however, when performing training, the dataset was generated during the training for co-evolved malware detectors.

3. For a final verification, a small selection of malware was found outside of the dataset provided. This was done to confirm the effectiveness of the detectors on real world malware. The website GetJar, a public Android app repository, was found to possess a large amount of malware. The majority of the apps found on this website are identified as malware according to VirusTotal, an on-line collection of actively used, publicly available, malware detectors. Some apps were investigated as well, to see if improvements could be made over other automated methods.

Benign Software Sources

Benign apps are notoriously difficult to download en masse. Many of these app sources are online app repositories or stores. A large portion of the apps available are free, which allows for a very diverse array of benign samples. A random subset of 1000 apps was selected to make the data easier to manage.

There were three benign app sources used in this thesis, compared against the three aforementioned malware app sources. The benign app sources are:

1. F-droid open source Android repository, abbreviated in the results as Fdroid

2. Google play app store, abbreviated in the results as Gplay

3. Data created from the co-evolution of previous runs, abbreviated as *live*, as the data is created from a living dataset.

These sources are all sampled to have 1000 apps at least. The live dataset was sourced for 10 000 apps, because the source was simple enough to collect from.

3.9.2 Dataset Setup

The dataset used in this work for the input malware samples is a random subset, 1000 malware apps, of the Drebin dataset [2]. Along with this, I also use 1000 benign apps to train a malware detector. Selecting these apps was performed randomly. In total, I have 2000 apps where half of them are malware and half of them are benign. I then use 70% of this dataset for training and 30% for testing. Therefore, there are 1400 samples in the training datasets and 600 samples in the testing datasets.

3.10 Summary

The method described here is best summarized in Figure 3.2. The co-evolution happens in two loops, one for malware generation, and one for malware detection. There is a population for both, and they are evaluated using each other. Each population is then independently modified based on the performance from their previous evaluation.

Malware generation is based on creating malware that is aggressive and undetectable, while discouraged from being overly evasive. A single sample knows how aggressive and evasive it is, but can only know how undetectable it is when tested with a malware detector, or in this case, many malware detectors. The malware generated used a template of a privacy leaking malware, as well as a variety of components implementing the features of such malware, to construct a single malware sample.

Malware detection requires many samples to learn the difference between what is, and what is not, malware. The malware detector writes many short programs and uses Genetic Programming to combine and modify them to construct a superior program that is capable of classifying inputs properly. The malware generated is combined with a static dataset of malware samples to help guide the evolution.

To judge the detectors, I took benign malware samples from several sources. F-Droid was my first source, as it is an open source app repository. My other source was Google Play, the largest store for apps on Android. For malware, I used Drebin, a dataset provided by Google for research. I also collected many of

the samples generated when performing the co-evolution process for testing. In addition, some malware was collected from a website called GetJar to represent malware found currently for android.

There are two controls used during this set of experiments: a decision tree algorithm called C5.0, and a standalone linear GP, not co-evolved, that is trained using only static datasets.

Chapter 4

Evaluations

The results of evaluating the proposed artificial arms race are presented in this chapter.

In section 4.1, I discuss the results of the early co-evolution tests that were performed. I start by evaluating the proposed system against only the F-Droid dataset, then look into how the inputs were reduced to become the 23 that were used.

In section 4.2, all subsequent tests are discussed. The detectors trained were the 6 control and 6 experimental configurations seen in Table 4.1. This adds five additional experimental configurations to the one present in the preliminary research, along with four other control configurations. I examine various interesting features of individual and average results generated, to consider why the results presented themselves the way they did. Finally, I look at some values generated from a recent real world test that shows the potential capability and benefits of an artificially co-evolved and improved mobile malware detector.

A full listing of the summary results, along with best samples, are available in Appendix B.

4.1 Preliminary Experiments

In the preliminary test phase of this thesis, ten runs were performed for each technique to get an idea of the performance differences. The measures used for these evaluations were: (i) the number of apps/samples correctly classified, (ii) the minimization of the number of features used, and (iii) the simplicity of the solutions generated.

The goal of detecting mobile malware accurately is the only guiding principal in the proposed artificial arms race framework. Features used and the complexity of the solution were selected during the training phase of the classifiers. While there is no bias for simple solutions or minimal features, creating simpler solutions

Table 4.1: Listing of trained detectors

Method	Training Benign	Training Malware
C5.0	F-Droid	Drebin
C5.0	Google Play	Drebin
C5.0	F-Droid & Google Play	Drebin
GP	F-Droid	Drebin
GP	Google Play	Drebin
GP	F-Droid & Google Play	Drebin
GP	F-Droid	Drebin & Co-evolved malware
GP	Google Play	Drebin & Co-evolved malware
GP	F-Droid & Google Play	Drebin & Co-evolved malware
GP	F-Droid	Co-evolved malware
GP	Google Play	Co-evolved malware
GP	F-Droid & Google Play	Co-evolved malware

seems to be a bi product of this method.

Appendix A shows some results used to define the methodology. These incremental results show various inputs on the methods involved. Using 149 permissions with both GP and C5.0 show the difficulty of GP to handle large amounts of input on a short training period of 30 generations.

4.1.1 Permission Feature Selection

As discussed in section 3.5.3, a decision was made to reduce the number of features used as input to the malware detector. The method of determining what features were used began with tests using a set of 149 permissions. Figure 4.1 shows which features were used and their level of importance in the tree generated by C5.0 trained on this set.

4.1.2 Code Feature Selection

As the next step, I employed the aforementioned eight code features listed in Figure 3.2 together with the 15 permission features listed in Figure 3.1 to represent an app using 23 features to the detector. Table A.5 shows the performance of the GP based standalone malware detector under this representation. Table A.6 and Figure 4.2 show the performance of the C5.0 based detector and the resulting tree rules using 23 features on the same data respectively. By adding the code features

```
100%  android.permission.SEND_SMS
 68%  com.android.launcher.permission.UNINSTALL_SHORTCUT
 63%  android.permission.RECEIVE_BOOT_COMPLETED
 62%  android.permission.CHANGE_WIFI_STATE
 56%  android.permission.READ_SMS
 54%  android.permission.DELETE_PACKAGES
 50%  android.permission.ACCESS_WIFI_STATE
 42%  android.permission.READ_LOGS
 41%  android.permission.SYSTEM_ALERT_WINDOW
 40%  android.permission.ACCESS_FINE_LOCATION
 32%  android.permission.CHANGE_COMPONENT_ENABLED_STATE
 11%  android.permission.READ_PHONE_STATE
 10%  android.permission.CAMERA
  8%  android.permission.WRITE_EXTERNAL_STORAGE
  4%  android.permission.SET_WALLPAPER
  3%  android.permission.ACCESS_NETWORK_STATE
  3%  android.permission.CALL_PHONE
  3%  android.permission.ACCESS_COARSE_LOCATION
  2%  android.permission.RESTART_PACKAGES
  2%  android.permission.INTERNET
  1%  android.permission.READ_EXTERNAL_STORAGE
```

Figure 4.1: Permission features used by C5.0 when trained with 149 Android permissions on a 700 F-Droid dataset

```
READ_PHONE_STATE = f:
:..SEND_SMS = t:
:  :..classes_with_annotation <= 37: malware
:  :   classes_with_annotation > 37: clean
:  SEND_SMS = f:
:  :..abstract_count > 562: clean
:     abstract_count <= 562:
:     :..INTERNET = f: clean
:        INTERNET = t:
:        :..static_count <= 70: malware
:           static_count > 70:
:           :..classes_with_annotation <= 2: clean
:              classes_with_annotation > 2:
:              :..classes_with_annotation <= 23: malware
:                 classes_with_annotation > 23: clean
READ_PHONE_STATE = t:
:..classes_with_interface > 558:
   :..ACCESS_FINE_LOCATION = t: malware
   :   ACCESS_FINE_LOCATION = f: clean
   classes_with_interface <= 558:
   :..BIND_ACCESSIBILITY_SERVICE = t: clean
      BIND_ACCESSIBILITY_SERVICE = f:
      :..INTERNET = f:
         :..SEND_SMS = t: malware
         :   SEND_SMS = f: clean
         INTERNET = t:
         :..READ_CALENDAR = t:
            :..WRITE_EXTERNAL_STORAGE = t: malware
            :   WRITE_EXTERNAL_STORAGE = f: clean
            READ_CALENDAR = f:
            :..abstract_count <= 51: malware
               abstract_count > 51:
               :..direct_count > 418: malware
                  direct_count <= 418:
                  :..classes_with_interface <= 37: malware
                     classes_with_interface > 37: clean
```

Figure 4.2: C5.0 tree using 15 permissions and 8 code features, trained on F-Droid

to the representation of an app, I obtained a 5% increase in the performance of GP and 8% increase in the performance of the C5.0. When I analyzed what the detectors learned in these experiments, I observed that GP used, on average, 12 of the 23 features while C5.0 used 11.

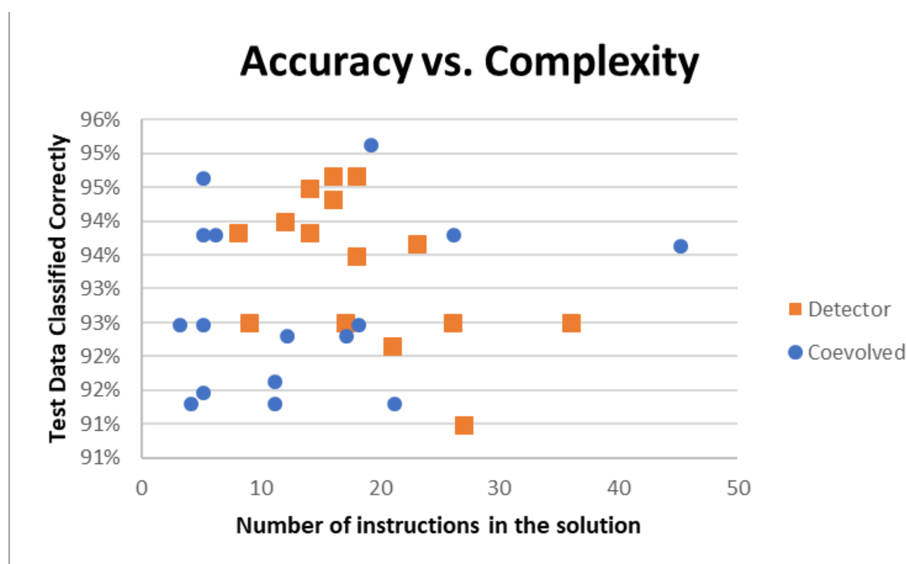

Figure 4.3: Accuracy vs. Complexity

Then, using the same training and test data, I implemented the artificial arms race framework and added in the co-evolution system. Table A.7 shows the performance of one of the GP based malware detectors evolved via this arms race. The results of the evolved detectors are comparable with the results of the C5.0 and GP based detectors that were trained standalone, i.e. outside of the artificial arms race framework. Moreover, the co-evolved malware detector, on average, uses fewer features compared to the standalone ones. The degree to which this average changes is not significant and may be circumstantial, including how the same test performed on 100 generations increases in complexity rather then reducing it.

4.1.3 Dataset Size

Table 4.2 shows the results of a set of experiments I conducted to observe the sensitivity of the training data size on the performance of standalone detectors. In these experiments, I varied the training set size for each category of apps, from 300 to 1000, and observed the precision and recall rates to determine the most suitable training dataset size. Based on these results, I chose the training dataset size to be 700 benign and 700 malware apps.

While there is no significant difference in accuracy with this kind of approach, other aspects begin to show differences, although not significant enough in it's

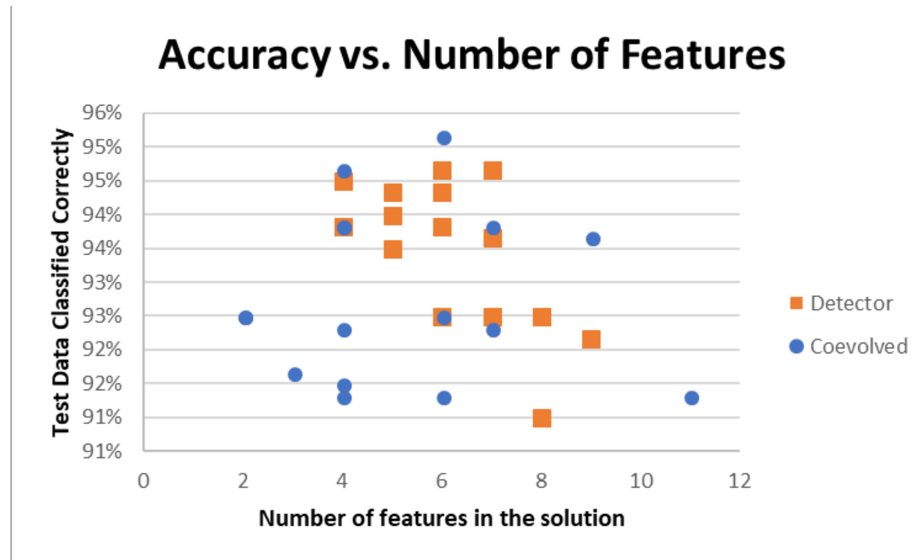

Figure 4.4: Accuracy vs. Number of Features

Table 4.2: Precision and Recall of different data sizes

Test	Precision	Recall
GP 300	97%	88%
GP 500	96%	89%
GP 700	97%	89%
GP 1000	96%	89%
C5.0 300	99%	96%
C5.0 500	86%	96%
C5.0 700	96%	97%
C5.0 1000	96%	97%

present state. Figure 4.3 shows that the average number of instructions, which is used to measure complexity, is reduced without significantly alternating the accuracy of the process. The same holds true regarding the average number of features selected by the learning algorithm from the given set of features, as is seen in Figure 4.4.

4.1.4 Co-evolved Solutions

Below is one of the simpler linear GP programs, automatically generated by the co-evolved solution to detect a malware using the proposed artificial arms race framework. As discussed earlier, in this framework, the detector is co-evolved against the malware generator to detect the generated malware. The resulting detector solution is in the form of a program:

```
r[6] = exp(in[READ_PHONE_STATE])
r[1] = r[6] - 200
r[0] = exp(in[direct method count])
```

where the final value of `r[1]` is the bid for malware and `r[0]` is the bid for benign, the higher one becoming the program's resulting choice.

This particular solution focuses on two values:

1. READ_PHONE_STATE
2. Number of Direct methods

The resulting rules followed by this program amount to: If app has READ_PHONE_STATE and a low value for Direct methods, then it is malware. The performance of this solution program's rules is presented in Table A.7.

On the other hand, if we analyze a simple solution of a GP based detector that was trained standalone, outside of the artificial arms race framework, the solution program appears as the following:

```
if( r[5] <= 62 )
if( r[0] > r[1] )
r[4] = r[1] / in[INSTALL_PACKAGES]
if( r[0] > in[SEND_SMS] )
```

```
r[0] = r[4] - 31
r[4] = log(r[0])
if( r[4] > in[READ_PHONE_STATE] )
r[1] = sin(in[RECEIVE_BOOT_COMPLETED])
```

where, once again, `r[1]` is the bid for malware and `r[0]` is the bid for benign.

The first part can be manually removed as the statement on the second line is always false. This particular solution focuses on the three values:

1. SEND_SMS

2. READ_PHONE_STATE

3. READ_BOOT_COMPLETED

Although this is also a simple program, on average, non-co-evolved detectors have 1 more feature, and 5 more instructions. The performance of the above program is given in Table A.5. The resulting rules followed by this program amount to: If the program has READ_PHONE_STATE and SEND_SMS, then it is malware. READ_BOOT_COMPLETED in this situation has no effect on the final result.

To better understand how the evolved solutions - the population - under the artificial arms race framework compares with the single best solution, I analyzed the diversity of the solutions in Figure 4.5. The population's combined coverage of knowledge reaches a very high accuracy (100%) given the collaboration of the top 20 solutions (programs) out of the 100 solutions evolved. This indicates that the solutions evolved have enough diversity to be able to recognize different malicious behaviours in the apps. From a co-evolutionary perspective, this ensures the survival of useful programs, even if they are not the single best solution. In other words, this diversity enables us to generate (evolve) different rules that have the potential of detecting different variants of malware.

Over the course of the evolution, it can be seen that there is indeed an arms race happening. In figure 4.6, the lines representing score move from being in favour of the malware to that of the detector. The detectors over generations are learning and improving overall.

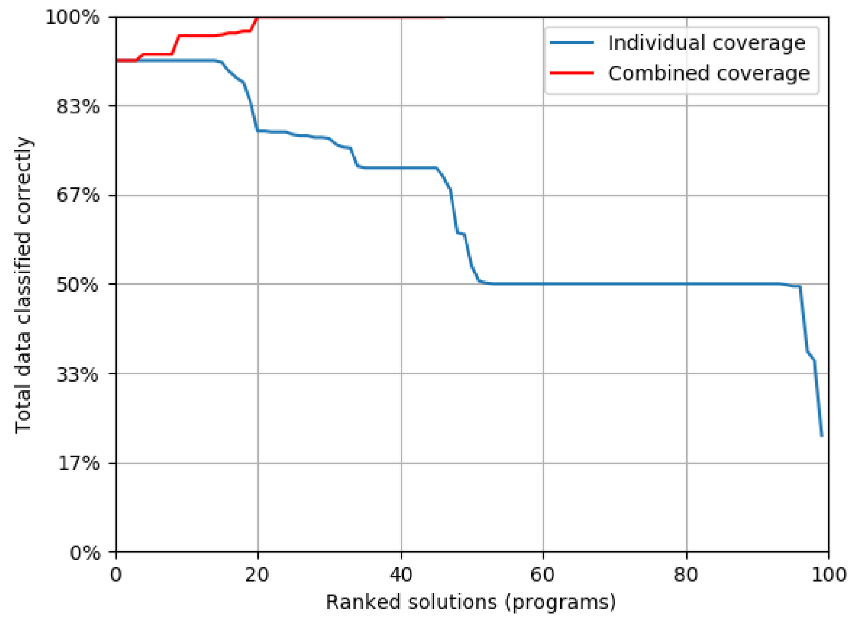

Figure 4.5: Accuracy vs. Populations accumulated knowledge

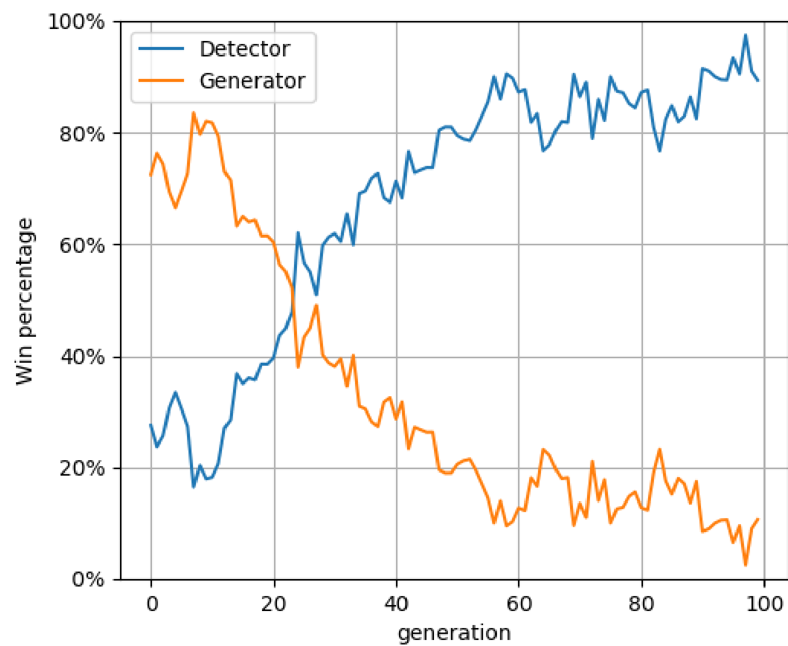

Figure 4.6: Co-Evolved Tug of war: How detectors get better over time

Table 4.3: Results of training C5.0 trees

Trained with	F-Droid		Google Play		F-Droid & Google Play	
Complexity	17		11		23	
Feature Count	11		8		13	
Tested on	fdroid	gplay	fdroid	gplay	fdroid	gplay
Accuracy	96%	88%	84%	98%	96.3%	97.5%

4.2 Experiments Performed

Given the collected datasets, experiments were performed for each of the three methods discussed in 3.8. A total of twelve configurations, listed in table 4.1, were chosen given the two benign training app sources and the single external training Malware source. The three benign sources trained on were F-Droid, Google Play, and a combination of the two.

For C5.0, three trees were built given the three benign app sources and the required malware source:

- C5.0 tree trained using F-Droid apps, and Drebin malware, shown in figure 4.2
- C5.0 tree trained using Google Play apps, and Drebin malware, shown in figure 4.7
- C5.0 tree trained using F-Droid and Google Play apps, and Drebin malware, shown in figure 4.8

A table of C5.0 results can be seen in Table 4.3.

Next, the same training sets were used as input for a GP without co-evolution. Unlike C5.0, this method includes randomness. In addition to the need for multiple solutions, individual solutions are often unclear at first glance, as explained in section 4.1.4. Therefore, where possible, I focus on the solution statistics by averaging multiple tests.

All GP results were tested at two configurations. One allowed for 30 generations of evolution, and the other 100.

The different GP without co-evolution experiments were:

```
class_count_with_interface <= 458:
:.READ_EXTERNAL_STORAGE = t:
:  :.READ_SMS = t: malware
:  :  READ_SMS = f:
:  :  :.ACCESS_FINE_LOCATION = f: clean
:  :     ACCESS_FINE_LOCATION = t:
:  :     :.class_count_with_interface <= 103: clean
:  :        class_count_with_interface > 103: malware
: READ_EXTERNAL_STORAGE = f:
:  :.READ_PHONE_STATE = t: malware
:     READ_PHONE_STATE = f:
:     :.class_count_with_interface <= 173: malware
:        class_count_with_interface > 173:
:        :.direct_count <= 3756: clean
:           direct_count > 3756: malware
class_count_with_interface > 458:
:.virtual_count > 11710: clean
   virtual_count <= 11710:
   :.READ_PHONE_STATE = f: clean
      READ_PHONE_STATE = t:
      :.RECORD_AUDIO = t: clean
         RECORD_AUDIO = f:
         :.class_count_with_interface <= 876: malware
            class_count_with_interface > 876: clean
```

Figure 4.7: C5.0 tree using 15 permissions and 8 code features, trained on Google Play

- GP trained using F-Droid apps, and Drebin malware
- GP trained using Google Play apps, and Drebin malware
- GP trained using F-Droid and Google Play apps, and Drebin malware

A table of GP results for 30 generations can be seen in Table 4.4, and for the corresponding 100 generations, in Table 4.6.

The final method, with the artificial arms race, used six different configurations of additional training sets. External malware is not strictly required for the co-evolution process, as malware is created through the process itself. This makes the six configurations the same three used for both C5.0 and GP without co-evolution plus the same three stripped of the Drebin malware.

Table 4.4: Average GP Results at 30 generations without co-evolution

Trained with	F-Droid			Google Play			F-Droid & Google Play		
Complexity	17.7			18.3			16.2		
Feature Count	5.7			5.3			5.4		
Scored on	fdroid	gplay	live	fdroid	gplay	live	fdroid	gplay	live
Precision	96.5%	91.7%	37.9%	68%	92.5%	38.3%	96.6%	96.6%	49.8%
Recall	88.7%	91.4%	37.9%	99.1%	98.5%	38.3%	84.8%	88%	49.8%

Again, all GP results were tested at both 30 generations and 100 generations of evolution, which applies to the co-evolved solutions as well.

The different co-evolved GP experiments were therefore as follows:

- Co-evolved GP trained using F-Droid, and Drebin malware
- Co-evolved GP trained using Google Play apps, and Drebin malware
- Co-evolved GP trained using F-Droid and Google Play apps, and Drebin malware
- Co-evolved GP trained using F-Droid
- Co-evolved GP trained using Google Play apps
- Co-evolved GP trained using F-Droid and Google Play apps

A table of co-evolved GP results for 30 generations can be seen in Table 4.5, and for the corresponding 100 generations, in Table 4.7.

4.2.1 Portability of Solutions

The results of comparing datasets show the differences inherent between the kinds of benign apps being used for training. Naturally, when asked to classify the test set comprised of benign software from the same grouping, the detectors are at their best. When the same detector is asked to classify the test set containing benign software from a different source, the performance drops. The portability of a detector is dependent on how well its performance is maintained given such a difference in source. A lower drop in performance means an increases in its portability, which is a useful feature of the detector overall.

Table 4.5: Average GP Results at 30 generations with co-evolution

With Drebin malware									
Trained with	F-Droid			Google Play			F-Droid & Google Play		
Complexity	13.3			19.4			16.6		
Feature Count	5.1			6.25			5.4		
Scored on	fdroid	gplay	live	fdroid	gplay	live	fdroid	gplay	live
Precision	96.3%	82.8%	100%	65.1%	91.4%	100%	97.6%	95.5%	100%
Recall	89.0%	92.5%	100%	99.5%	99.5%	100%	87.2%	91.5%	100%
Without external malware									
Trained with	F-Droid			Google Play			F-Droid & Google Play		
Complexity	8.7			14.3			8.7		
Feature Count	3.6			4.5			3.9		
Scored on	fdroid	gplay	live	fdroid	gplay	live	fdroid	gplay	live
Precision	92.9%	67.7%	100%	78.28%	90.5%	100%	92.0%	87.1%	100%
Recall	18.8%	20.29%	87.6%	40.1%	43.2%	78.7%	13.4%	15.5%	49.7%

Table 4.6: Average GP Results at 100 generations without co-evolution

Trained with	F-Droid			Google Play			F-Droid & Google Play		
Complexity	38.7			30.0			41.2		
Feature Count	8.3			6.6			7.9		
Scored on	fdroid	gplay	live	fdroid	gplay	live	fdroid	gplay	live
Precision	97.2%	93.0%	16.8%	70.6%	93.6%	59.2%	97.0%	97.3%	54.5%
Recall	90.8%	92.7%	16.8%	99.3%	98.8%	59.2%	89.8%	92.2%	54.5%

Table 4.7: Average GP Results at 100 generations with co-evolution

With Drebin malware									
Trained with	F-Droid			Google Play			F-Droid & Google Play		
Complexity	53.0			34.9			29.7		
Feature Count	11			7.8			6.2		
Scored on	fdroid	gplay	live	fdroid	gplay	live	fdroid	gplay	live
Precision	96.8%	88.7%	99.97%	67.7%	93.0%	99.98%	97.5%	97.8%	100%
Recall	90.5%	93.4%	99.97%	99.5%	99.1%	99.98%	88.6%	92.1%	99.97%
Without external malware									
Trained with	F-Droid			Google Play			F-Droid & Google Play		
Complexity	22.5			23.6			22.1		
Feature Count	5.8			7			6.2		
Scored on	fdroid	gplay	live	fdroid	gplay	live	fdroid	gplay	live
Precision	81.2%	50.0%	100%	50.7%	72.6%	100%	89.0%	92.9%	100%
Recall	4.5%	5.1%	98.6%	25.0%	24.0%	92.3%	25.2%	26.3%	90.5%

From early tests, it was seen that portability was not symmetric between the main datasets. F-Droid trained detectors have a much higher portability than those trained on Google Play apps. There is a surprisingly large gap between the performance of F-Droid trained detectors' ability to classify Google Play apps as benign, and the ability of Google Play trained detectors' ability to classify F-Droid apps as benign. This is more apparent in the GP results, with and without co-evolution.

4.2.2 Using F-Droid and Google Play

Perhaps one of the best overall detectors would be one of the combined dataset detectors, trained using F-Droid and Google Play along with the co-evolved malware. The solution is 11 lines long, and uses 5 inputs:

1. READ_CALENDAR
2. Direct function count
3. Virtual function count
4. READ_PHONE_STATE
5. Class count

Obtaining a precision of 97% and recall of 93% for F-Droid's test set, and 98% precision and 95% recall for Google Play's test set, this may be one of the best performers, even though it was only trained for 30 generations.

Training using both F-Droid and Google play appears to have a more compounding impact on the results. The performance retained within the same amount of complexity and limited features results in an overall improved detector. What is not clear is how the co-evolution of malware affected this dataset.

At 30 generations, the statistics for the co-evolved detectors are very similar to the control GP not using co-evolution. Higher recall overall, along with higher precision for F-Droid, is all that can be seen. There is loss of complexity or feature reduction. However, the 100 generation tables (4.6 and 4.7) show the trend over time to become more complex without the influx of new malware samples.

Considering a detector that was trained for 100 generations, may give insight into how co-evolution effects the training process. In the Figure 4.9, a breakthrough

is discovered, that would have gone unnoticed without co-evolution. The amount of fitness given to the detectors that use this new breakthrough quickly spread through the population. This can be seen by the spike in detector performance early in the 40th generation. This hidden feature is likely what dramatically improves test scores a few generations later. This shift is all but invisible to the static set, as around the 40 generation mark, there is hardly any change in performance, as can seen in the left graph.

As seen in figure 4.10, the number of features used is not significantly different from non co-evolved solutions. There is however a larger drop in the average number of features used.

4.2.3 Complexity

Consider the results of C5.0 in table 4.3. The behaviour is as expected with a large number of features in use. Although the Google Play dataset performed the best, creating a small tree, it suffered from the portability issue previously described, making it likely that it was a property of the dataset itself.

Looking at the complexity range in Figure 4.11, we cannot say that there is a significant change in complexity. However, there seems to be more consistency with results obtained, leading to an overall lower average, when co-evolution is employed.

When comparing the equivalent experiments at 30 generations and at 100 generations, the complexity climbs. This is most likely a failing of the method used as larger programs act as protection against changes and therefore produce offspring making smaller changes though crossover then small programs. This was unintentional as there was no incentive to have a short program. In addition a longer program gives more chances that there is useful active features present within the program. While there was no intentional bias in the changing of a programs length, it appears as if this problem caused growth simply over generations.

The average results of training without co-evolution are shown in table 4.4, and the equivalent with co-evolution is seen in table 4.5. The most interesting aspect to note is that the results of reduced complexity and feature count are unique to the F-Droid trained data. There is, however, an across the board improvement for

recall, with no significant loss in performance of precision.

The largest improvement is the incorporation of the live data. The live instances are taken from multiple runs and are a representative sample of the malware being generated for co-evolution. Most detectors had little trouble in identifying the malware generated, however, without co-evolution, only about half of the detectors were able to detect the same malware. The averages here hide the fact that most detectors were achieving either 100% or 0% with very few detectors in the middle ground.

Our malware analysis shows that the generated malware was not as diverse and varied as other malware families, and future improvements to this process could include the generation of other malware families.

The combination of training with a source of benign samples from both F-Droid and Google Play create the best all-around results. Unlike the C5.0 results, the GP results do not use more features, and remain at the same performance level as when trained using only F-Droid samples.

4.2.4 Variation and Exploratory Results

Results of the co-evolved system show a higher level of quality of exploration caused by the introduction of the co-evolution. The real impact of the search can be seen by analyzing the performance of the test set as in figure 4.9. Furthermore, figure 4.12 and 4.13 show the results of the best performing non co-evolved and co-evolved GP on the test set, respectively.

4.2.5 Real World Results

To confirm the effectiveness of the detectors on real world malware, apps from the GetJar[1] website were analyzed with the co-evolved mobile malware detectors and the standalone malware detectors. It should be noted here that the majority of the apps found on this website are malware, either viruses or adware.

By tossing random apps found on this site into VirusTotal, which is a collection of malware detection and classification tools, most were marked as malware. VirusTotal offers a breakdown of which detectors detected positive (malware), and which detected negative (benign). Moreover, for the purpose of this research, any

Table 4.8: Selection of apps from GetJar with detection rates

	Virus Total	C5.0	GP	Arms race GP
MicrowaveRecipies	31%	100%	100%	100%
God of war Wall paper	36%	100%	100%	100%
Facebook Password Hacker	22%	100%	100%	100%
Footcare salon	0%	0%	0%	100%
Application	46%	100%	100%	100%
Saavn_getjar	5%	100%	100%	100%
PS4 emulator	11%	100%	0%	100%
Subway Servers Hack and cheat	9%	100%	100%	100%
Miss You - Whatsapp	24%	100%	100%	100%
Cam Scanner License	27%	100%	100%	100%

GetJar app that was detected as malware was considered to be malware after my manual inspection also confirmed that it was malware. In these tests, the game 'Footcare Salon' was manually inspected. Opening up the app using APK Tool, it was determined to be adware using a unique package that was not detected by VirusTotal.

A selection of apps presented in table 4.8 shows that the standalone trained models of C5.0 and GP, as well as co-evolved GP, were able to detect apps at varying degrees of effectiveness. The proposed co-evolved detector was able to detect more malicious malware in this situation, and that may be the case in others as well. More importantly, limitations that are present in the standalone approach are avoided by using the proposed artificial arms race, i.e. co-evolution, approach.

Furthermore, the resulting detectors of this thesis identified the app version of the GetJar site itself to be structured like malware, as C5.0, GP, and our co-evolved GP, all were convinced of the malicious nature of the app store itself, while VirusTotal did not see it as malicious.

4.3 Summary

The results of my evaluations show that the co-evolution method allows for a more exploratory search. The resulting solutions, while not necessarily performing significantly better in terms of accuracy, are learning the problem faster and more diversely, resulting in solutions that are not significantly different from previous

attempts at solving this classification problem.

The results show that detectors trained on F-Droid were more portable solutions, while those that were trained on Google Play performed worse on malware that was from F-Droid. The portability also extends to the generated malware family, as those that were not trained using co-evolution may not have detected these malware as being malicious.

```
abstract_count > 619:
:..static_count > 3519: clean
:  static_count <= 3519:
:  :..READ_PHONE_STATE = f: clean
:     READ_PHONE_STATE = t:
:     :..RECORD_AUDIO = t: clean
:        RECORD_AUDIO = f: class_count <= 2322: malware otherwise clean
abstract_count <= 619:
:..READ_PHONE_STATE = t:
   :..INTERNET = f:
   :  :..SEND_SMS = t: malware otherwise clean
   :  INTERNET = t:
   :  :..READ_CALENDAR = t:
   :     :..WRITE_EXTERNAL_STORAGE = t: malware
   :     :  WRITE_EXTERNAL_STORAGE = f: clean
   :        READ_CALENDAR = f:
   :        :..READ_EXTERNAL_STORAGE = t:
   :           :..READ_SMS = t: malware
   :           :  READ_SMS = f:
   :           :  :..ACCESS_FINE_LOCATION = f: clean
   :           :     ACCESS_FINE_LOCATION = t:
   :           :     :..class_count <= 262: clean otherwise malware
   :           READ_EXTERNAL_STORAGE = f:
   :           :..abstract_count <= 51: malware
   :              abstract_count > 51:
   :              :..direct_count > 418: malware
   :                 direct_count <= 418:
   :                 :..class_count_with_interface <= 37: malware otherwise clean
   READ_PHONE_STATE = f:
   :..SEND_SMS = t: malware
      SEND_SMS = f:
      :..INTERNET = f: clean
         INTERNET = t:
         :..static_count <= 70:
            :..READ_EXTERNAL_STORAGE = t: clean otherwise malware
            static_count > 70:
            :..class_count_with_annotation <= 2: clean
               class_count_with_annotation > 2:
               :..class_count_with_interface <= 132: malware
                  class_count_with_interface > 132:
                  :..virtual_count <= 6548: clean otherwise malware
```

Figure 4.8: C5.0 tree using 15 permissions and 8 code features, trained on both F-Droid and Google Play

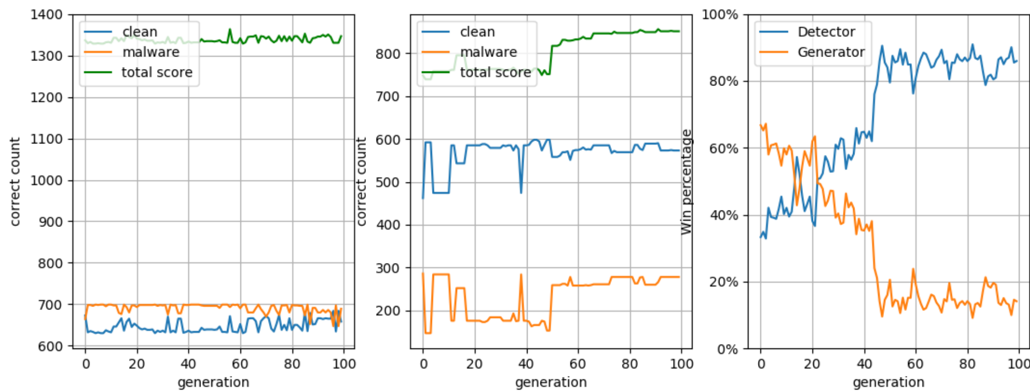

Figure 4.9: Training over time for a co-evolved detector trained using F-Droid and Google play apps with Drebin malware for 100 generations. (left) The fitness of the best individual on the static dataset. (middle) The fitness of the best individual on the test dataset. (right) The fitness of the population on generated malware.

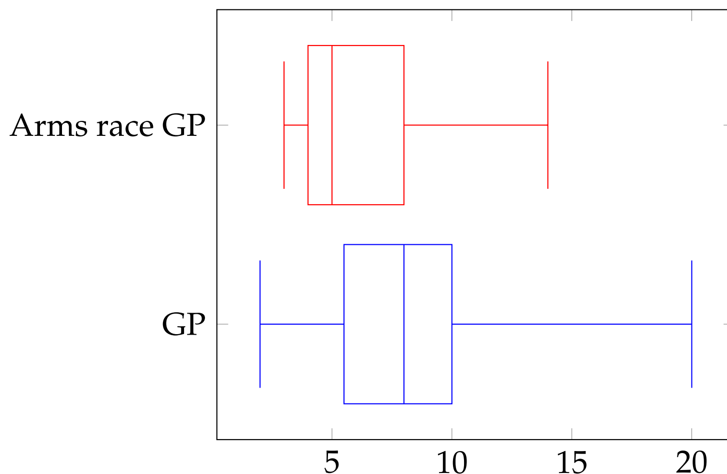

Figure 4.10: Boxplot of the number of features used in program solution for F-Droid and Google play trained detectors at 100 generations

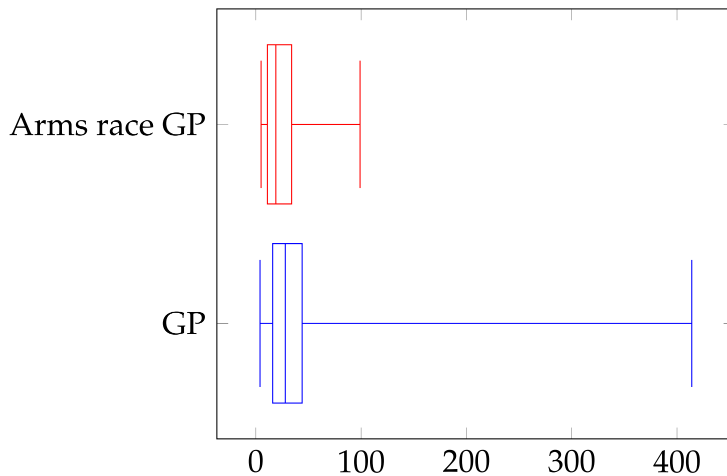

Figure 4.11: Boxplot of program complexity for F-Droid and Google play trained detectors at 100 generations

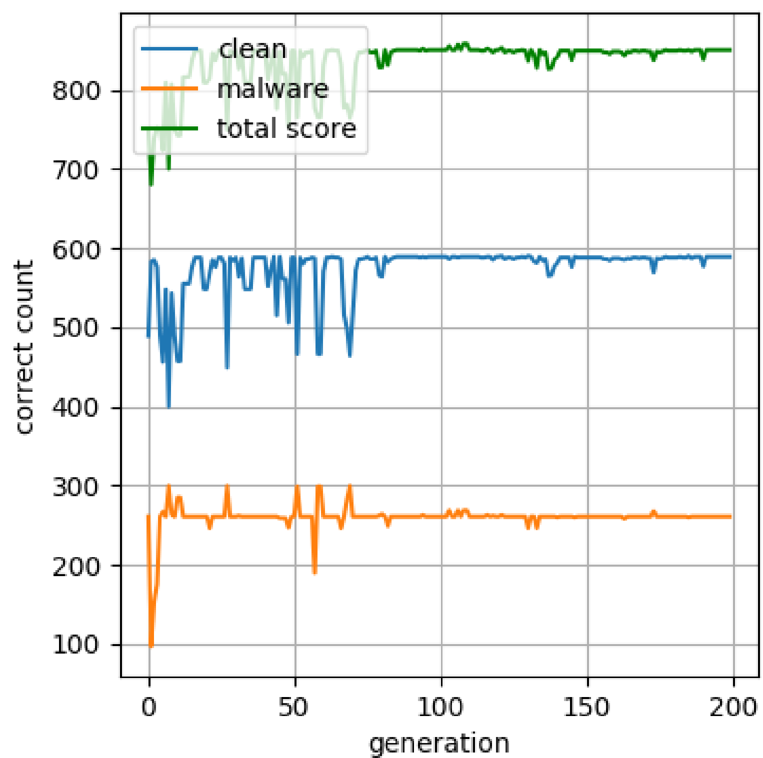

Figure 4.12: Non Co-Evolved score of best performer on test set

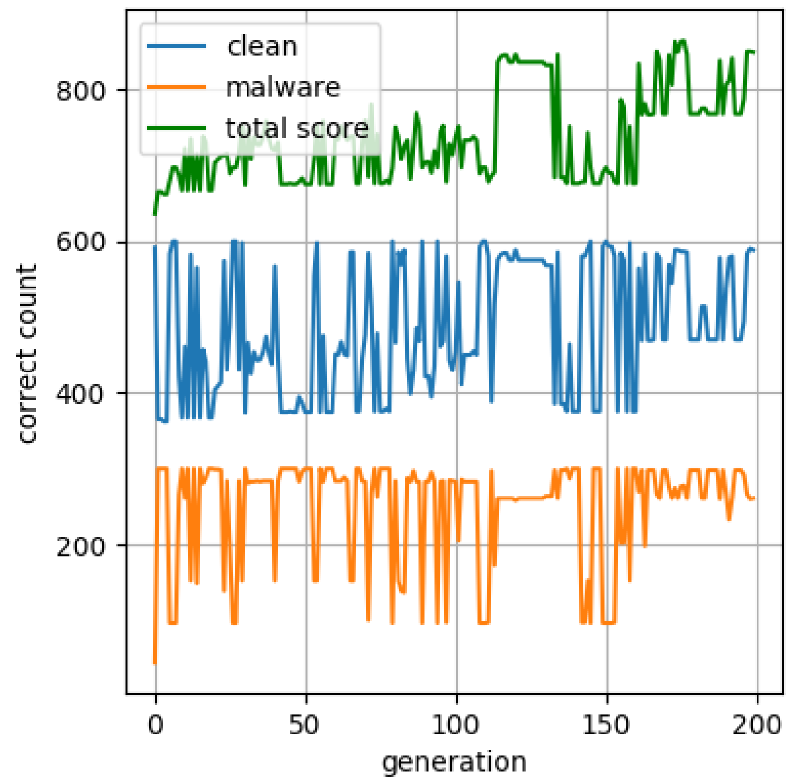

Figure 4.13: Co-Evolved the score of best performer on test set

Chapter 5

Conclusion

By looking at how mobile malware changes over time, I was able to identify, and provide a solution to an existing problem of the evasive nature of malware families. The solution was to internalize the problem and create a method that simulates the change in malware to more completely understand it. I think that simulating this change could be an effective way of countering malware changes in the wild. While presenting this idea, I also present some interesting results of this proposed system.

Beginning with an idea of how I could automate a battle going on lead to the idea of using a form of co-evolution. This developed into the methodology proposed and created. Using previous research into the subject, I combined a malware generator and a malware detector into a single solution that developed a detector that was better able to understand the problem it was set to solve.

5.1 Interesting Features

While there was a core area of research I wanted to focus on, there were many interesting side notes that were not directly related to the core idea of simply creating the artificial arms race. This includes the interesting distinction found between Google Play apps and F-Droid apps, as well as how the population changes throughout the process.

Initial findings indicate that the building of teams from the co-evolved detectors may lead to a more robust solution compared to building teams from stand alone detectors. This is thought to be due to the diversity within an individual population. As discussed in the previous chapter, a team of solutions outperform an individual solution. Using teams of programs seems to result in a more robust performance, while producing / discovering small efficient solutions with low computation.

In short, the performance of the framework demonstrates that it is possible to generate new rules with high accuracy against the new variants of malware using a co-evolution process to automatically emulate an artificial arms race between malware and detectors. The programs, if processed carefully, may be converted to a list of rules, given the simple nature of the instructions built. The internal workings of the programs give hints towards performing rule mining, and a clear representation of what the inputs do.

Finally, previous work in [18] indicates that the co-evolved training within an artificial arms race framework shows an increase in knowledge about the task. This observation seems to be the case in my research as well. Moreover, it appears that the more knowledge the detector has, the better it can adapt to future malware behaviours.

5.2 Limitations of the Malware Generator

While this method seems like a solid solution, its main issue appears to be the malware generator. This component functions well enough for a proof-of-concept research, as within this thesis. However, the malware generated by this component lacks variety, since the generated malicious behaviour / malware all belong to the same family, namely privacy leaking. The malware does not rename files or packages, and the template is the same for each. It is a limitation that produces detectors only for this type of malicious behaviour / malware. In other words, the malware generator can be improved to include other malware families as well.

5.3 Future Works

The advantage of the proposed artificial arms race is the modularity of the different elements. The framework is not strictly held to the requirements of a particular malware generator or detector. As long as the algorithms used are evolutionary computation techniques, the malware generator can be replaced with another. From the perspective of the proposed framework, a malware generator needs to produce a population of malware, output these for evaluation, and then accept

feedback for its evolution. Any program that can replicate those inputs and outputs may fill the role of malware generator.

The same abstraction as above may be made with the malware detector. The malware detector must accept additions to the training set, and provide granular results using the additions it is provided. By this definition, other machine learning approaches can be used in place of linear genetic programming, while still incorporating the core methodology.

In this thesis, the proposed framework was evaluated using Android apps. By making both detector and generator modular, other platforms may also be considered. Given that the approach is not using features dependent on the platform and / or architecture, it could be applied to other mobile platforms, apps, and malware. While sufficiently different to not be considered yet, IOS feature based detection is indeed a viable solution [5]. The features of the detector and generator were only a limited selection of possible Android features. Other features, Android or otherwise, could be added and extracted from the malware of whichever platform it was designed for.

As discussed above, the proposed framework opens up several potential avenues for future research. The possibilities that arise from introducing different malware generators are enormous. There is even the potential to consider multiple malware generators, focusing on different families, methods, or even platforms. All together, they may aid in the performance of a detector. The method employed by the detector as well may be improved to evaluate how other learning algorithms compare with this research.

Bibliography

[1] Getjar. `https://www.getjar.com`. Accessed: 2018.

[2] Daniel Arp, Michael Spreitzenbarth, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. 2014.

[3] Emre Aydogan and Sevil Sen. Automatic generation of mobile malwares using genetic programming. In *European conference on the applications of evolutionary computation*, pages 745–756. Springer, 2015.

[4] P. Chester, C. Jones, M. Wiem Mkaouer, and D. E. Krutz. M-perm: A lightweight detector for android permission gaps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 217–218, May 2017.

[5] Aniello Cimitile, Fabio Martinelli, and Francesco Mercaldo. Machine learning meets ios malware: Identifying malicious applications on apple environment. In *ICISSP*, pages 487–492, 2017.

[6] João Filipe da Cruz, Helena Gaspar, and Gonçalo Calado. Turning the game around: toxicity in a nudibranch-sponge predator–prey association. *Chemoecology*, 22(1):47–53, Mar 2012.

[7] Google Developers. Application fundamentals, April 2018.

[8] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[9] Prahlad Fogla and Wenke Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 59–68. ACM, 2006.

[10] Olivia Lucca Fraser, Nur Zincir-Heywood, Malcolm Heywood, and John T. Jacobs. Return-oriented programme evolution with roper: A proof of concept. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1447–1454, New York, NY, USA, 2017. ACM.

[11] Badr Hssina, Abdelkarim Merbouha, Hanane Ezzikouri, and Mohammed Erritali. A comparative study of decision tree id3 and c4. 5. *International Journal of Advanced Computer Science and Applications*, 4(2), 2014.

[12] Apple Inc. Security, June 2018.

[13] Rafiqul Islam, Ronghua Tian, Lynn M Batten, and Steve Versteeg. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, 36(2):646–656, 2013.

[14] Hilmi Güneş Kayacık, A. Nur Zincir-Heywood, and Malcolm I. Heywood. Evolutionary computation as an artificial attacker: generating evasion attacks for detector vulnerability testing. *Evolutionary Intelligence*, 4(4):243–266, Dec 2011.

[15] Gunes Kayacık, A Zincir-Heywood, and Malcolm I. Heywood. Can a good offense be a good defense? vulnerability testing of anomaly detectors through an artificial arms race. 11:4366–4383, 10 2011.

[16] Steve Mansfield-Devine. The malware arms race. *Computer Fraud & Security*, 2018(2):15 – 20, 2018.

[17] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 365–376. ACM, 2016.

[18] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu. Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):157–175, June 2017.

[19] Sadia Noreen, Shafaq Murtaza, M Zubair Shafiq, and Muddassar Farooq. Evolvable malware. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1569–1576. ACM, 2009.

[20] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA, 2013. ACM.

[21] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[22] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *Intelligence and security informatics conference (eisic), 2012 european*, pages 141–147. IEEE, 2012.

[23] S. Sen, E. Aydogan, and A. I. Aysan. Coevolution of mobile malware and anti-malware. *IEEE Transactions on Information Forensics and Security*, 13(10):2563–2574, Oct 2018.

[24] A. Skovoroda and D. Gamayunov. Review of the mobile malware detection approaches. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 600–603, March 2015.

[25] Lichao Sun, Xiaokai Wei, Jiawei Zhang, Lifang He, Philip S Yu, and Witawas Srisa-an. Contaminant removal for android malware detection systems. *arXiv preprint arXiv:1711.02715*, 2017.

[26] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang. Exploring permission-induced risk in android applications for malicious application detection. *IEEE Transactions on Information Forensics and Security*, 9(11):1869–1882, Nov 2014.

[27] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new android malware detection approach using bayesian classification. In *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pages 121–128, March 2013.

[28] R. Bronfman-Nadas N. Zincir-Heywood and J. Jacobs. An artificial arms race: Could it improve mobile malware detectors? In *Network Traffic Measurement and Analysis Conference (TMA)*, Jun 2018.

Appendix A

Sample Results

Results displayed here were quick results performed to help define the methodology. The dataset used was 700 malware samples from the Drebin dataset [2], along with 700 non malware samples from F-Droid.

Table A.1: Results of GP with 149 permissions

		True class		
		Malware	Benign	Total
Predicted class	Malware	167	12	93%
	Benign	133	288	68%
	Total	56%	96%	76%

Table A.2: Results of C5.0 with 149 permissions

		True class		
		Malware	Benign	Total
Predicted class	Malware	272	28	91%
	Benign	27	273	91%
	Total	91%	91%	91%

Table A.3: Results of GP with 15 permissions only

		True class		
		Malware	Benign	Total
Predicted class	Malware	272	28	90%
	Benign	56	244	81%
	Total	83%	90%	86%

Table A.4: Results of C5.0 with 15 permissions only

		True class		
		Malware	Benign	Total
Predicted class	Malware	260	29	90%
	Benign	40	271	87%
	Total	87%	90%	88%

Table A.5: Results of GP with 15 permissions and 8 code features

		True class		
		Malware	Benign	Total
Predicted class	Malware	278	22	93%
	Benign	15	285	95%
	Total	95%	93%	94%

Table A.6: Results of C5.0 with 15 permissions and 8 code features

		True class		
		Malware	Benign	Total
Predicted class	Malware	289	12	96%
	Benign	11	288	96%
	Total	96%	96%	96%

Table A.7: A Sample Result of GP with 15 permissions and 8 code features with co-evolution

		True class		
		Malware	Benign	Total
Predicted class	Malware	294	6	98%
	Benign	39	261	87%
	Total	88%	98%	92%

Appendix B

Expanded Table of Results

Table B.1: Table Legend

Column	Description	Desired Value
Complexity	The number of Instructions used in the chosen solution.	Low values are desirable
Clean raw	The number of benign software samples correctly classified	The value of the Benign count for the chosen test set
Malware raw	The number of malware samples correctly classified	The value of the Malware count for the chosen test set
Score	The sum of the Clean raw and Malware raw. This is the raw numerical version of the CR	The value of the Total sample count for the chosen test set
Total	The total sample count for the chosen dataset, this is Benign count + Malware count	This value is a constant for each dataset.
CR	Classification rate. The ratio of correctly classified instances and the total number of instances: $(TP+TN)/(TP+FP+FN+TN)$ or Score/Total	A value of 100% is desirable
Input Count	The number of inputs provided that were consulted in a chosen solution	Low values are desirable
check_time	The number of solutions required to have full correct coverage of the entire dataset	Low values indicate a more diverse population and therefore are more desirable
TP	Percentage of the dataset that is malware and was correctly classified as malware	High values are desirable. The maximum possible value is 50% for fdroid and gplay and 100% for live data
FP	Percentage of the dataset that is Benign software and was incorrectly classified as malware	Low values are desirable The maximum possible value is 50% for fdroid and gplay and live data will always have a value of 0%
FN	Percentage of the dataset that is malware and was incorectly classified as Benign software	Low values are desirable. The maximum possible value is 50% for fdroid and gplay and 100% for live data
TN	Percentage of the dataset that is Benign software and was correctly classified as Benign software	High values are desirable. The maximum possible value is 50% for fdroid and gplay will live data will always have a value of 0%
Precision	The fraction of data instances predicted as positive that are actually positive: TP/(TP+FP)	High values are desirable
Recall	The fraction of Malware that is correctly classified. Also known as Detection Rate: TP/(TP+FN)	High values are desirable

Table B.2: Results on 30 generations of evolution tested on F-Droid

Average	Complexity	Clean raw	Malware raw	Score	Total	CR	Input Count	check_time	TP	FP	FN	TN	Precision	Recall
Drebin + fdroid	17.77	290.40	266.25	556.65	600	92.78%	5.70	64.92	44.38%	1.60%	5.62%	48.40%	96.52%	88.75%
Drebin + gplay	18.31	161.68	297.56	459.25	600	76.54%	5.34	50.52	49.59%	23.05%	0.41%	26.95%	68.27%	99.19%
Drebin + fdroid + gplay	16.21	291.15	254.49	545.64	600	90.94%	5.42	45.36	42.42%	1.47%	7.58%	48.53%	96.64%	84.83%
Drebin + fdroid + Mystique co-evolution	13.27	289.83	267	556.83	600	92.81%	5.11	63.38	44.50%	1.69%	5.50%	48.31%	96.33%	89.00%
Drebin + gplay + Mystique co-evolution	19.43	140	298.56	438.56	600	73.09%	6.25	64.62	49.76%	26.67%	0.24%	23.33%	65.11%	99.52%
Drebin + fdroid + gplay + Mystique co-evolution	16.68	293.62	261.81	555.43	600	92.57%	5.43	56.56	43.64%	1.06%	6.36%	48.94%	97.62%	87.27%
Mystique co-evolved + fdroid	8.66	295.73	56.4	352.13	600	58.69%	3.6	47.4	9.40%	0.71%	40.60%	49.29%	92.97%	18.80%
Mystique co-evolved + gplay	14.30	266.61	120.30	386.92	600	64.49%	4.53	35.07	20.05%	5.56%	29.95%	44.44%	78.28%	40.10%
Mystique co-evolved + gplay + fdroid	8.73	296.53	40.26	336.8	600	56.13%	3.92	35.26	6.71%	0.58%	43.29%	49.42%	92.07%	13.42%
Best Precision														
Drebin + fdroid	19	297	262	559	600	93.17%	8	81	43.67%	0.50%	6.33%	49.50%	98.87%	87.33%
Drebin + gplay	14	202	298	500	600	83.33%	6	25	49.67%	16.33%	0.33%	33.67%	75.25%	99.33%
Drebin + fdroid + gplay	20	300	256	556	600	92.67%	8	39	42.67%	0.00%	7.33%	50.00%	100.00%	85.33%
Drebin + fdroid + Mystique co-evolution	19	295	276	571	600	95.17%	6	20	46.00%	0.83%	4.00%	49.17%	98.22%	92.00%
Drebin + gplay + Mystique co-evolution	9	195	296	491	600	81.83%	4	14	49.33%	17.50%	0.67%	32.50%	73.82%	98.67%
Drebin + fdroid + gplay + Mystique co-evolution	33	297	261	558	600	93.00%	10	59	43.50%	0.50%	6.50%	49.50%	98.86%	87.00%
Mystique co-evolved + fdroid	3	300	1	301	600	50.17%	2	92	0.17%	0.00%	49.83%	50.00%	100%	0%
Mystique co-evolved + gplay	27	300	79	379	600	63.17%	7	29	13.17%	0.00%	36.83%	50.00%	100%	26%
Mystique co-evolved + gplay + fdroid	9	300	23	323	600	53.83%	5	25	3.83%	0.00%	46.17%	50.00%	100%	8%
Best Recall/Detection Rate														
Drebin + fdroid	17	268	288	556	600	92.67%	4	79	48.00%	5.33%	2.00%	44.67%	90.00%	96.00%
Drebin + gplay	3	104	300	404	600	67.33%	3	17	50.00%	32.67%	0.00%	17.33%	60.48%	100.00%
Drebin + fdroid + gplay	27	242	296	538	600	89.67%	7	76	49.33%	9.67%	0.67%	40.33%	83.62%	98.67%
Drebin + fdroid + Mystique co-evolution	26	285	278	563	600	93.83%	7	93	46.33%	2.50%	3.67%	47.50%	94.88%	92.67%
Drebin + gplay + Mystique co-evolution	28	99	300	399	600	66.50%	7	16	50.00%	33.50%	0.00%	16.50%	59.88%	100.00%
Drebin + fdroid + gplay + Mystique co-evolution	24	292	278	570	600	95.00%	9	78	46.33%	1.33%	3.67%	48.67%	97.20%	92.67%
Mystique co-evolved + fdroid	11	287	261	548	600	91.33%	3	9	43.50%	2.17%	6.50%	47.83%	95%	87%
Mystique co-evolved + gplay	7	25	300	325	600	54.17%	2	4	50.00%	45.83%	0.00%	4.17%	52%	100%
Mystique co-evolved + gplay + fdroid	10	300	97	397	600	66.17%	4	68	16.17%	0.00%	33.83%	50.00%	100%	32%

Table B.3: Results on 30 generations of evolution tested on Google Play apps

Average	Complexity	Clean raw	Malware raw	Score	Total	CR	Input Count	check_time	TP	FP	FN	TN	Precision	Recall
Drebin + fdroid	17.77	275.38	274.18	549.56	600	91.59%	5.70	62.56	45.70%	4.10%	4.30%	45.90%	91.76%	91.39%
Drebin + gplay	18.31	276.24	295.74	571.98	600	95.33%	5.34	75.26	49.29%	3.96%	0.71%	46.04%	92.56%	98.58%
Drebin + fdroid + gplay	16.21	290.80	264.09	554.89	600	92.48%	5.42	42.25	44.02%	1.53%	5.98%	48.47%	96.64%	88.03%
Drebin + fdroid + Mystique co-evolution	12.31	242.31	277.57	519.89	600	86.65%	5.05	63.63	46.26%	9.61%	3.74%	40.39%	82.79%	92.53%
Drebin + gplay + Mystique co-evolution	19.43	272.06	298.5	570.56	600	95.09%	6.25	81.06	49.75%	4.66%	0.25%	45.34%	91.44%	99.50%
Drebin + fdroid + gplay + Mystique co-evolution	16.68	286.56	274.5	561.06	600	93.51%	5.43	53	45.75%	2.24%	4.25%	47.76%	95.33%	91.50%
Mystique co-evolved + fdroid	8.66	270.93	60.86	331.8	600	55.30%	3.6	47.8	10.14%	4.84%	39.86%	45.16%	67.68%	20.29%
Mystique co-evolved + gplay	14.30	286.38	129.69	416.07	600	69.35%	4.53	25.07	21.62%	2.27%	28.38%	47.73%	90.50%	43.23%
Mystique co-evolved + gplay + fdroid	8.73	293.13	46.4	339.53	600	56.59%	3.92	44	7.73%	1.14%	42.27%	48.86%	87.11%	15.47%
Best Precision														
Drebin + fdroid	14	296	281	577	600	96.17%	7	53	46.83%	0.67%	3.17%	49.33%	98.60%	93.67%
Drebin + gplay	14	285	298	583	600	97.17%	6	76	49.67%	2.50%	0.33%	47.50%	95.21%	99.33%
Drebin + fdroid + gplay	5	300	161	461	600	76.83%	3	7	26.83%	0.00%	23.17%	50.00%	100.00%	53.67%
Drebin + fdroid + Mystique co-evolution	17	296	273	569	600	94.83%	7	90	45.50%	0.67%	4.50%	49.33%	98.56%	91.00%
Drebin + gplay + Mystique co-evolution	23	282	299	581	600	96.83%	5	87	49.83%	3.00%	0.17%	47.00%	94.32%	99.67%
Drebin + fdroid + gplay + Mystique co-evolution	11	295	274	569	600	94.83%	5	70	45.67%	0.83%	4.33%	49.17%	98.21%	91.33%
Mystique co-evolved + fdroid	3	300	5	305	600	50.83%	2	67	0.83%	0.00%	49.17%	50.00%	100%	2%
Mystique co-evolved + gplay	27	300	92	392	600	65.33%	7	20	15.33%	0.00%	34.67%	50.00%	100%	31%
Mystique co-evolved + gplay + fdroid	9	300	27	327	600	54.50%	5	37	4.50%	0.00%	45.50%	50.00%	100%	9%
Best Recall/Detection Rate														
Drebin + fdroid	7	276	288	564	600	94.00%	4	80	48.00%	4.00%	2.00%	46.00%	92.31%	96.00%
Drebin + gplay	3	261	300	561	600	93.50%	3	86	50.00%	6.50%	0.00%	43.50%	88.50%	100.00%
Drebin + fdroid + gplay	27	262	295	557	600	92.83%	7	9	49.17%	6.33%	0.83%	43.67%	88.59%	98.33%
Drebin + fdroid + Mystique co-evolution	9	201	284	485	600	80.83%	5	41	47.33%	16.50%	2.67%	33.50%	74.15%	94.67%
Drebin + gplay + Mystique co-evolution	7	271	300	571	600	95.17%	5	87	50.00%	4.83%	0.00%	45.17%	91.19%	100.00%
Drebin + fdroid + gplay + Mystique co-evolution	24	294	284	578	600	96.33%	9	81	47.33%	1.00%	2.67%	49.00%	97.93%	94.67%
Mystique co-evolved + fdroid	11	202	275	477	600	79.50%	3	9	45.83%	16.33%	4.17%	33.67%	74%	92%
Mystique co-evolved + gplay	7	243	300	543	600	90.50%	2	17	50.00%	0.00%	0.00%	40.50%	100%	100%
Mystique co-evolved + gplay + fdroid	10	300	110	410	600	68.33%	4	68	18.33%	0.00%	31.67%	50.00%	100%	37%

Table B.4: Results on 30 generations of evolution tested on Generated Malware

Average	Complexity	Score	Total	CR	Input Count	check_time	TP	FP	FN	TN	Precision	Recall
Drebin + fdroid	18.24	3794.22	10000	37.94%	6.02	100	37.94%	0.00%	62.06%	0.00%	37.94%	37.94%
Drebin + gplay	18.31	3831.05	10000	38.31%	5.34	100	38.31%	0.00%	61.69%	0.00%	38.31%	38.31%
Drebin + fdroid + gplay	16.21	4981.82	10000	49.82%	5.42	100	49.82%	0.00%	50.18%	0.00%	49.82%	49.82%
Drebin + fdroid + Mystique co-evolution	12.31	9997	10000	99.97%	5.05	100	99.97%	0.00%	0.03%	0.00%	99.97%	99.97%
Drebin + gplay + Mystique co-evolution	19.43	9998.5	10000	99.99%	6.25	100	99.99%	0.00%	0.02%	0.00%	99.99%	99.99%
Drebin + fdroid + gplay + Mystique co-evolution	16.68	9997	10000	99.97%	5.43	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Mystique co-evolved + fdroid	8.66	8768.2	10000	87.68%	3.6	100	87.68%	0.00%	12.32%	0.00%	100.00%	87.68%
Mystique co-evolved + gplay	14.30	7868.38	10000	78.68%	4.53	100	78.68%	0.00%	21.32%	0.00%	100.00%	78.68%
Mystique co-evolved + gplay + fdroid	8.73	4976.66	10000	49.77%	3.92	100	49.77%	0.00%	50.23%	0.00%	100.00%	49.77%
Best Precision												
Drebin + fdroid	8	9997	10000	99.97%	4	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + gplay	3	10000	10000	100.00%	3	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Drebin + fdroid + gplay	12	9997	10000	99.97%	4	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + fdroid + Mystique co-evolution	17	9997	10000	99.97%	7	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + gplay + Mystique co-evolution	10	10000	10000	100.00%	5	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Drebin + fdroid + gplay + Mystique co-evolution	11	9997	10000	99.97%	5	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Mystique co-evolved + fdroid	9	9997	10000	99.97%	5	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Mystique co-evolved + gplay	7	10000	10000	100.00%	2	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Mystique co-evolved + gplay + fdroid	4	6945	10000	69.45%	1	100	69.45%	0.00%	30.55%	0.00%	100.00%	69.45%
Best Recall/Detection Rate												
Drebin + fdroid	8	9997	10000	99.97%	4	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + gplay	3	10000	10000	100.00%	3	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Drebin + fdroid + gplay	12	9997	10000	99.97%	4	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + fdroid + Mystique co-evolution	17	9997	10000	99.97%	7	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + gplay + Mystique co-evolution	10	10000	10000	100.00%	5	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Drebin + fdroid + gplay + Mystique co-evolution	11	9997	10000	99.97%	5	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Mystique co-evolved + fdroid	9	9997	10000	99.97%	5	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Mystique co-evolved + gplay	7	10000	10000	100.00%	2	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Mystique co-evolved + gplay + fdroid	4	6945	10000	69.45%	1	100	69.45%	0.00%	30.55%	0.00%	100.00%	69.45%

Table B.5: Results on 100 generations of evolution tested on F-Droid

Average	Complexity	Clean raw	Malware raw	Score	Total	CR	Input Count	check_time	TP	FP	FN	TN	Precision	Recall
Drebin + fdroid	38.69	292.09	272.50	564.59	600	94.10%	8.31	83.13	45.42%	1.32%	4.58%	48.68%	97.18%	90.84%
Drebin + gplay	30.03	175.70	297.83	473.54	600	78.92%	6.61	73.04	49.64%	20.72%	0.36%	29.28%	70.56%	99.28%
Drebin + fdroid + gplay	41.24	291.73	269.50	561.24	600	93.54%	7.93	67.98	44.92%	1.38%	5.08%	48.62%	97.03%	89.84%
Drebin + fdroid + Mystique co-evolution	53	291.25	271.5	562.75	600	93.79%	11	91.75	45.25%	1.46%	4.75%	48.54%	96.88%	90.50%
Drebin + gplay + Mystique co-evolution	34.87	157.62	298.62	456.25	600	76.04%	7.75	83	49.77%	23.73%	0.23%	26.27%	67.72%	99.54%
Drebin + fdroid + gplay + Mystique co-evolution	29.64	293.23	265.76	559	600	93.17%	6.17	75.17	44.29%	1.13%	5.71%	48.87%	97.52%	88.59%
Mystique co-evolved + fdroid	22.5	296.87	13.5	310.37	600	51.73%	5.87	42.25	2.25%	0.52%	47.75%	49.48%	81.20%	4.50%
Mystique co-evolved + gplay	23.6	227	75.2	302.2	600	50.37%	7	44.4	12.53%	12.17%	37.47%	37.83%	50.74%	25.07%
Mystique co-evolved + gplay + fdroid	22.07	290.71	75.5	366.21	600	61.04%	6.28	49.07	12.58%	1.55%	37.42%	48.45%	89.05%	25.17%
Best Precision														
Drebin + fdroid	22	297	268	565	600	94.17%	7	88	44.67%	0.50%	5.33%	49.50%	98.89%	89.33%
Drebin + gplay	37	216	291	507	600	84.50%	11	95	48.50%	14.00%	1.50%	36.00%	77.60%	97.00%
Drebin + fdroid + gplay	13	299	246	545	600	90.83%	5	95	41.00%	0.17%	9.00%	49.83%	99.60%	82.00%
Drebin + fdroid + Mystique co-evolution	13	294	261	555	600	92.50%	4	96	43.50%	1.00%	6.50%	49.00%	97.75%	87.00%
Drebin + gplay + Mystique co-evolution	13	191	298	489	600	81.50%	4	96	49.67%	18.17%	0.33%	31.83%	73.22%	99.33%
Drebin + fdroid + gplay + Mystique co-evolution	99	297	262	559	600	93.17%	11	76	43.67%	0.50%	6.33%	49.50%	98.87%	87.33%
Mystique co-evolved + fdroid	20	300	10	310	600	51.67%	5	82	1.67%	0.00%	48.33%	50.00%	100%	3%
Mystique co-evolved + gplay	15	300	23	323	600	53.83%	7	100	3.83%	0.00%	46.17%	50.00%	100%	8%
Mystique co-evolved + gplay + fdroid	70	300	62	362	600	60.33%	10	33	10.33%	0.00%	39.67%	50.00%	100%	21%
Best Recall/Detection Rate														
Drebin + fdroid	73	291	282	573	600	95.50%	14	62	47.00%	1.50%	3.00%	48.50%	96.91%	94.00%
Drebin + gplay	55	178	300	478	600	79.67%	11	89	50.00%	20.33%	0.00%	29.67%	71.09%	100.00%
Drebin + fdroid + gplay	34	244	292	536	600	89.33%	7	98	48.67%	9.33%	1.33%	40.67%	83.91%	97.33%
Drebin + fdroid + Mystique co-evolution	57	289	278	567	600	94.50%	10	84	46.33%	1.83%	3.67%	48.17%	96.19%	92.67%
Drebin + gplay + Mystique co-evolution	41	167	300	467	600	77.83%	9	93	50.00%	22.17%	0.00%	27.83%	69.28%	100.00%
Drebin + fdroid + gplay + Mystique co-evolution	14	280	278	558	600	93.00%	4	32	46.33%	3.33%	3.67%	46.67%	93.29%	92.67%
Mystique co-evolved + fdroid	33	297	97	394	600	65.67%	6	2	16.17%	0.50%	33.83%	49.50%	97%	32%
Mystique co-evolved + gplay	5	122	101	223	600	37.17%	2	1	16.83%	29.67%	33.17%	20.33%	36%	34%
Mystique co-evolved + gplay + fdroid	14	291	261	552	600	92.00%	5	36	43.50%	0.00%	6.50%	48.50%	100%	87%

Table B.6: Results on 100 generations of evolution tested on Google Play apps

Average	Complexity	Clean raw	Malware raw	Score	Total	CR	Input Count	check_time	TP	FP	FN	TN	Precision	Recall
Drebin + fdroid	38.69	278.95	278.08	557.04	600	92.84%	8.31	81.05	46.35%	3.51%	3.65%	46.49%	92.97%	92.69%
Drebin + gplay	30.03	279.70	296.52	576.23	600	96.04%	6.61	89	49.42%	3.38%	0.58%	46.62%	93.59%	98.84%
Drebin + fdroid + gplay	41.24	292.44	276.66	569.11	600	94.85%	7.93	59.79	46.11%	1.26%	3.89%	48.74%	97.34%	92.22%
Drebin + fdroid + Mystique co-evolution	53	264.25	280.25	544.5	600	90.75%	11	88.25	46.71%	5.96%	3.29%	44.04%	88.69%	93.42%
Drebin + gplay + Mystique co-evolution	34.87	277.62	297.37	575	600	95.83%	7.75	90.75	49.56%	3.73%	0.44%	46.27%	93.00%	99.13%
Drebin + fdroid + gplay + Mystique co-evolution	29.64	293.70	276.23	569.94	600	94.99%	6.17	83.76	46.04%	1.05%	3.96%	48.95%	97.77%	92.08%
Mystique co-evolved + fdroid	22.5	284.5	15.5	300	600	50.00%	5.87	56.25	2.58%	2.58%	47.42%	47.42%	50.00%	5.17%
Mystique co-evolved + gplay	23.6	272.8	72	344.8	600	57.47%	7	62.6	12.00%	4.53%	38.00%	45.47%	72.58%	24.00%
Mystique co-evolved + gplay + fdroid	22.07	294	78.78	372.78	600	62.13%	6.28	54.07	13.13%	1.00%	36.87%	49.00%	92.92%	26.26%
Best Precision														
Drebin + fdroid	21	296	282	578	600	96.33%	5	26	47.00%	0.67%	3.00%	49.33%	98.60%	94.00%
Drebin + gplay	11	289	293	582	600	97.00%	5	94	48.83%	1.83%	1.17%	48.17%	96.38%	97.67%
Drebin + fdroid + gplay	65	300	184	484	600	80.67%	10	61	30.67%	0.00%	19.33%	50.00%	100.00%	61.33%
Drebin + fdroid + Mystique co-evolution	82	295	284	579	600	96.50%	15	81	47.33%	0.83%	2.67%	49.17%	98.27%	94.67%
Drebin + gplay + Mystique co-evolution	13	285	299	584	600	97.33%	4	96	49.83%	2.50%	0.17%	47.50%	95.22%	99.67%
Drebin + fdroid + gplay + Mystique co-evolution	21	296	272	568	600	94.67%	6	85	45.33%	0.67%	4.67%	49.33%	98.55%	90.67%
Mystique co-evolved + fdroid	20	300	7	307	600	51.17%	5	82	1.17%	0.00%	48.83%	50.00%	100%	2%
Mystique co-evolved + gplay	22	298	85	383	600	63.83%	10	100	14.17%	0.33%	35.83%	49.67%	98%	28%
Mystique co-evolved + gplay + fdroid	35	300	20	320	600	53.33%	7	1	3.33%	0.00%	46.67%	50.00%	100%	7%
Best Recall/Detection Rate														
Drebin + fdroid	32	230	287	517	600	86.17%	9	90	47.83%	11.67%	2.17%	38.33%	80.39%	95.67%
Drebin + gplay	23	277	300	577	600	96.17%	11	93	50.00%	3.83%	0.00%	46.17%	92.88%	100.00
Drebin + fdroid + gplay	34	275	292	567	600	94.50%	7	99	48.67%	4.17%	1.33%	45.83%	92.11%	97.33%
Drebin + fdroid + Mystique co-evolution	82	295	284	579	600	96.50%	15	81	47.33%	0.83%	2.67%	49.17%	98.27%	94.67%
Drebin + gplay + Mystique co-evolution	41	279	299	578	600	96.33%	9	93	49.83%	3.50%	0.17%	46.50%	93.44%	99.67%
Drebin + fdroid + gplay + Mystique co-evolution	14	293	284	577	600	96.17%	4	81	47.33%	1.17%	2.67%	48.83%	97.59%	94.67%
Mystique co-evolved + fdroid	33	277	110	387	600	64.50%	6	89	18.33%	3.83%	31.67%	46.17%	83%	37%
Mystique co-evolved + gplay	59	262	90	352	600	58.67%	11	50	15.00%	6.33%	35.00%	43.67%	70%	30%
Mystique co-evolved + gplay + fdroid	14	289	274	563	600	93.83%	5	27	45.67%	0.00%	4.33%	48.17%	100%	91%

Table B.7: Results on 100 generations of evolution tested on generated malware

Average	Complexity	Score	Total	CR	Input Count	check_time	TP	FP	FN	TN	Precision	Recall
Drebin + fdroid	38.69	1684.09	10000	16.84%	8.31	100	16.84%	0.00%	83.16%	0.00%	16.84%	16.84%
Drebin + gplay	30.03	5924.36	10000	59.24%	6.61	100	59.24%	0.00%	40.76%	0.00%	59.24%	59.24%
Drebin + fdroid + gplay	41.24	5444.98	10000	54.45%	7.93	100	54.45%	0.00%	45.55%	0.00%	54.45%	54.45%
Drebin + fdroid + Mystique co-evolution	53	9997	10000	99.97%	11	100	99.97%	0.00%	0.03%	0.00%	99.97%	99.97%
Drebin + gplay + Mystique co-evolution	34.87	9998.12	10000	99.98%	7.75	100	99.98%	0.00%	0.02%	0.00%	99.98%	99.98%
Drebin + fdroid + gplay + Mystique co-evolution	29.64	9997	10000	99.97%	6.17	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Mystique co-evolved + fdroid	22.5	9863.62	0	98.64%	5.87	100	98.64%	0.00%	1.36%	0.00%	100.00%	98.64%
Mystique co-evolved + gplay	23.6	9229.2	10000	92.29%	7	100	92.29%	0.00%	7.71%	0.00%	100.00%	92.29%
Mystique co-evolved + gplay + fdroid	22.07	9050.85	10000	90.51%	6.28	100	90.51%	0.00%	9.49%	0.00%	100.00%	90.51%
Best Precision												
Drebin + fdroid	42	9997	10000	99.97%	8	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + gplay	113	10000	10000	100.00%	12	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Drebin + fdroid + gplay	15	9997	10000	99.97%	5	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + fdroid + Mystique co-evolution	60	9997	10000	99.97%	15	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + gplay + Mystique co-evolution	74	10000	10000	100.00%	13	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Drebin + fdroid + gplay + Mystique co-evolution	19	9997	10000	99.97%	4	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Mystique co-evolved + fdroid	40	10000	10000	100.00%	9	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Mystique co-evolved + gplay	59	10000	10000	100.00%	11	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Mystique co-evolved + gplay + fdroid	24	10000	10000	100.00%	4	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Best Recall/Detection Rate												
Drebin + fdroid	42	9997	10000	99.97%	8	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + gplay	113	10000	10000	100.00%	12	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Drebin + fdroid + gplay	15	9997	10000	99.97%	5	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + fdroid + Mystique co-evolution	60	9997	10000	99.97%	15	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Drebin + gplay + Mystique co-evolution	74	10000	10000	100.00%	13	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Drebin + fdroid + gplay + Mystique co-evolution	19	9997	10000	99.97%	4	100	99.97%	0.00%	0.03%	0.00%	100.00%	99.97%
Mystique co-evolved + fdroid	40	10000	10000	100.00%	9	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Mystique co-evolved + gplay	59	10000	10000	100.00%	11	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%
Mystique co-evolved + gplay + fdroid	24	10000	10000	100.00%	4	100	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%

Appendix C

Malware Feature List

C.1 Sink

HTTP::URL_CONNECTION_GET, HTTP::URL_CONNECTION_POST,
HTTP::SOCKET_GET, HTTP::SOCKET_POST

C.2 Source

CONTACT::CONTACT, TELEPHONY::IMEI, TELEPHONY::IMSI,
TELEPHONY::PHONE_NUMBER, TELEPHONY::SIM_COUNTRY,
TELEPHONY::SIM_SERIAL, TELEPHONY::SIM_OPERATOR,
TELEPHONY::SIM_OPERATOR_NAME, TELEPHONY::NETWORK_COUNTRY,
TELEPHONY::NETWORK_OPERATOR, TELEPHONY::NETWORK_OPERATOR_NAME,
SMS::INBOX

C.3 Trigger

BROADCAST::Android.intent.action.BATTERY_CHANGED, MAIN::STARTUP,
BROADCAST::Android.provider.Telephony.SMS_RECEIVED,
BROADCAST::Android.intent.action.WALLPAPER_CHANGED,
BROADCAST::Android.intent.action.PACKAGE_ADDED,
BROADCAST::Android.intent.action.PACKAGE_REMOVED,
BROADCAST::Android.intent.action.PACKAGE_CHANGED,
BROADCAST::Android.intent.action.TIME_TICK,
BROADCAST::Android.intent.action.TIME_SET,
BROADCAST::Android.intent.action.TIMEZONE_CHANGED,
BROADCAST::Android.intent.action.BOOT_COMPLETED,

```
BROADCAST::Android.intent.action.PHONE_STATE,
BROADCAST::Android.intent.action.SCREEN_ON,

BROADCAST::Android.intent.action.SCREEN_OFF,
BROADCAST::Android.intent.action.NEW_OUTGOING_CALL,
BROADCAST::Android.intent.action.MEDIA_UNMOUNTED,
BROADCAST::Android.intent.action.MEDIA_MOUNTED,
BROADCAST::Android.intent.action.MEDIA_REMOVED,
BROADCAST::Android.net.wifi.WIFI_STATE_CHANGED,
BROADCAST::Android.intent.action.AIRPLANE_MODE,
BROADCAST::Android.bluetooth.adapter.action.STATE_CHANGED,
BROADCAST::Android.intent.action.ACTION_POWER_CONNECTED,
BROADCAST::Android.intent.action.INPUT_METHOD_CHANGED,
BROADCAST::Android.intent.action.USER_PRESENT,
BROADCAST::Android.net.wifi.STATE_CHANGE,
BROADCAST::Android.intent.action.PACKAGE_INSTALL,
BROADCAST::Android.intent.action.PACKAGE_REPLACED,
BROADCAST::Android.media.RINGER_MODE_CHANGED,
BROADCAST::Android.intent.action.BATTERY_LOW,
BROADCAST::Android.intent.action.BATTERY_OKAY,
BROADCAST::Android.intent.action.GTALK_CONNECTED,
BROADCAST::Android.intent.action.PACKAGE_RESTARTED,
BROADCAST::Android.intent.action.MEDIA_SCANNER_SCAN_FILE,
BROADCAST::Android.intent.action.MEDIA_SCANNER_STARTED,
BROADCAST::Android.intent.action.REBOOT,
BROADCAST::Android.intent.action.MEDIA_EJECT,
BROADCAST::Android.intent.action.MEDIA_SHARED,
BROADCAST::Android.intent.action.CAMERA_BUTTON,
BROADCAST::Android.intent.action.ACTION_SHUTDOWN,
BROADCAST::Android.intent.action.DEVICE_STORAGE_LOW,
```

```
BROADCAST::Android.intent.action.DEVICE_STORAGE_OK,
BROADCAST::Android.intent.action.MEDIA_BAD_REMOVAL,
BROADCAST::Android.intent.action.MEDIA_SCANNER_FINISHED,
BROADCAST::Android.intent.action.MEDIA_CHECKING,
BROADCAST::Android.bluetooth.adapter.action.DISCOVERY_FINISHED,
BROADCAST::Android.bluetooth.adapter.action.DISCOVERY_STARTED,
BROADCAST::Android.bluetooth.adapter.action.LOCAL_NAME_CHANGED,
BROADCAST::Android.bluetooth.adapter.action.SCAN_MODE_CHANGED,
BROADCAST::Android.bluetooth.device.action.ACL_DISCONNECTED,
BROADCAST::Android.bluetooth.device.action.NAME_CHANGED,
BROADCAST::Android.bluetooth.device.action.FOUND,
BROADCAST::Android.bluetooth.device.action.BOND_STATE_CHANGED
```

C.4 Evasion

```
renameclasses,
reverseorder,
encString,
encArrays,
remDebugInfo,
reorder,
nontrivialjunk,
insertnops,
insertFunctionIndirection,
doci
```