Application of Evolutionary Machine Learning in Metamorphic Malware Analysis and Detection

Kehinde Oluwatoyin Babaagba

School of Computing
Edinburgh Napier University

A thesis submitted in partial fulfilment of the requirements of Edinburgh Napier University, for the award of
Doctor of Philosophy.

April 2021

This work is dedicated to my beloved husband, daughter and son, Rotimi, Araoluwa and Oluwagbemiga Oloye, for their endless love, encouragement and support. To my dear parents, Richard and Sonia Babaagba and my sisters, Anne, Taiwo and Sharon, thanks for your inexhaustible supply of belief in me. To Almighty God, for being my rock and shield, I also dedicate this work to you.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Kehinde Oluwatoyin Babaagba
April 2021

Acknowledgements

The journey to the completion of my PhD studies has been filled with highs and lows, moments of joy and tears. In all, I am indeed grateful and indebted to a number of people who have inspired me to be the best version of myself and motivated me to pursue and complete my doctoral degree. I am forever grateful to them for their relentless support and presence.

I say a big thank you to my supervisors and academic advisory team who have directed me throughout this process and have been committed to ensuring that I achieve all goals and objectives required. Thank you, Dr. Zhiyuan (Thomas) Tan, my director of studies for pushing me to be my best, listening to my contributions while providing me in-depth direction and insights to the achievement of my goals. Discussions with you led to the discovery of key insights and several new research directions.

I am very grateful to my second supervisor, Professor Emma Hart who has been a constant source of support and encouragement to me in this journey. Thanks for teaching me work life balance. Not to mention all the invaluable research and academic insights you have provided to me. Your technical prowess and your ability to dissect and provide solutions to complex scientific problems is beyond admirable.

I acknowledge my panel chair, Dr. Neil Urquart whose valuable direction has ensured I stayed on track in the PhD studies. Thanks for also providing guidance and advice on milestones to take and improved methods of tackling problems.

Thanks to the School of Computing, Edinburgh Napier University, who funded my PhD research and provided numerous resources for my sustained development. I also want to appreciate all the wonderful people who have made my PhD successful, all members of NIIS research group, my colleagues turned friends, staff members like Dr. Kevin Sims, Dr. Laura Muir, Kathy Thierens, Cheryl Thompson that have supported me in one way or the other.

I recognise my parents (Mr and Mrs Babaagba) who have invested heavily in the pursuance of all my dreams and never gave up on me. Thanks to my siblings (Anne, Taiwo and Sharon) and in-laws for being sources of joy and support to me. Thanks to Sharon for proof reading my thesis, I am indeed grateful. My gratitude specially goes to my husband (Rotimi Oloye), daughter (Araoluwa Oloye) and son (Oluwagbemiga Oloye) who put up

with all my imperfections during this journey. They ensured I stayed on track and was constantly motivated. Thank you for your patience and encouragement throughout this journey. I am so glad that I have a family that is passionate about the achievement of my goals and pursuits. Most importantly, thank you Jesus for being my strength, succor and present help in times of need.

I am indeed grateful to you all!

Abstract

In recent times, malware detection and analysis are becoming key issues. A dangerous class of malware is *metamorphic malware* which is capable of modifying its own code and hiding malicious instructions within normal program code. Current malware detectors are susceptible to metamorphic malware as they are pre-trained to recognize only *predicted* versions of code. However, if detectors could be trained on a larger set of data that included potential mutant variants, they could be more accurate. The task of finding new evasive variants is challenging - many variants might exist.

In this research, a two-phase system is proposed. First, a mutation only Evolutionary Algorithm (EA) is used to search for a diverse set of new, malicious mutants, that evade detection by existing detection algorithms. While this is shown to be successful, it requires multiple runs of the algorithm to produce multiple variants without explicit guarantee of diversity. To address this, a Quality Diversity (QD) algorithm — MAP-Elites, that traverses a high-dimensional search space in search of the best solution at every point of a feature space with low dimension, is then developed to return a large and diverse repertoire of solutions in a single run. This method produces a larger and more diverse archive of solutions than the mutation only Evolutionary Algorithm (EA) and sheds insight into the properties of a sample that lead to them being undetectable by a suite of existing detection engines.

Having created a set of evasive and diverse variants, detectors are then trained using a set of classical classification methods (feature-based and sequence-based models) with results showing that classification of metamorphic malware can be improved by augmenting training data with the diverse set of evolved variant samples. This also includes the use of a pretrained Natural Language Processing (NLP) model in a transfer learning setting to show improved classification of metamorphic malware, using the evolved variants as part of the training data.

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Malicious attacks continue posing serious security threats to most information assets. They also constitute one of the commonly found attack vectors. The recent 2020 Security Threat Report by McAfee Labs revealed that there has been an increase in malicious attacks particularly aimed at employees working from home due to the Covid-19 pandemic, with approximately 981,887 Covid-19 related malicious file detections affecting about 4,355 organisations. They observed that in the first quarter of 2020 alone, there has been about 375 threats per minute with a 71% increase in mobile malware as compared to the last quarter of 2019.

To prevent detection and elimination of malicious binaries, obfuscation techniques are employed by sophisticated malware creators. These techniques often involve either packing the malware (also known as malware packing), transforming its static binary code (polymorphism) or transforming the dynamic binary code of the malware (metamorphism).

Amongst these sophisticated malware families, metamorphic malware is particularly complex and dangerous, presenting security threats to many endpoint devices, including desktops, servers, laptops, as well as kiosks or mobile devices with Internet connection. Its danger arises from its ability to transform its program code between generations using various means, including instruction substitution (substituting a given instruction sequence with its equivalent); garbage code insertion (inserting junk code to the original program code); control-flow alteration (distorting the flow of control within the original program code using loops) and register reordering (reordering the variables in the original program code).

In a bid to curb the impacts of metamorphic malware, several detection strategies have been adopted: Alam et al. [8] provided a detailed overview, including Opcode-Based Analysis (OBA), Control Flow Analysis (CFA) and Information Flow Analysis (IFA) which differ depending on the type of information being used in the analysis. These malware analysis approaches often use either signature based, heuristic based or malware

normalisation techniques for classification and detection of malware instances. The challenges with these techniques are that they are mostly reactive and therefore cannot detect new attacks. Also, for heuristics based detection techniques due to the lack of sufficient data, training ML models to detect this class of malware is often hard.

Regardless of the detection technique considered, a general challenge is to create better data representing future variants of metamorphic malware to train better ML models. The malicious samples that will serve as training data are software, and there are techniques to generate code in other fields such as Evolutionary Algorithm (EA) which treats it as a search problem. This perhaps could be applied to the aforementioned issue. The term EA refers to a class of problem-solving techniques inspired by Darwin's theory of evolution [48], in which the quality (fitness) of a population increases over time due to the pressure caused by natural selection. Given a quality function that needs to be optimised, a population of randomly generated potential solutions to the problem is first created. Solutions are selected for reproduction in a manner which is biased by their fitness; a reproduction operator generates new offspring from selected solutions by applying processes which mix information from two or more solutions (which is termed "crossover" in EA) and/or by a mutation process that makes small, random changes to solutions. As new fitter solutions replace poorer quality ones in the population, the population as a whole becomes fitter as the process is iterated.

The flexible nature of EA allows it to be applied to any task that can be expressed as a function optimisation task. Although a significant amount of literature focuses on its use in combinatorial or continuous optimisation domains, it has more recently been applied to malware analysis and detection, such as malware feature extraction [146], classification problems [156] among others . [47] describes a proof-of-concept that an EA could evolve a detector to recognise a virus signature represented as a bit-string, although this was only tested on 8 arbitrary functions designed to show its ability to cope in complex landscapes, rather than on real viruses. It offers the following advantages:

- exploration of a huge search space, which is one of the challenges to be solved in searching for code variants;
- provision of operators that enable easy manipulation of code;
- proven ability in transformation, optimisation and improvement of software code. [37], [90] and [153]

The creation of malicious variants using EA provides training samples that can be used to improve ML models, such as feature based models - Naive Bayes and sequential models - Long Short-Term Memory (LSTM) among others. The malicious variants will take on different structural and behavioural forms hence providing insight as to where

metamorphic malware will morph to. This will help the classification performance of ML models.

1.1 Research Questions

After thoroughly reviewing related literature, research gaps were identified. This research work aims to apply Evolutionary Machine Learning to improve Metamorphic malware analysis and detection. The research objectives include (1) using an Evolutionary Algorithm (EA) to create new malware mutants that are (a) as evasive as possible (b) behaviourally and structurally dissimilar to the original malware (c) as diverse as possible, (2) improving classification with these large and diverse set of mutants, (3) understanding what types of Machine Learning (ML) models benefit more from these new mutants. To address the aim and objectives, four research questions are formed and given as follows:

- **Q1** - To what extent can evasive and diverse variants of malware be generated, using an EA whose fitness function evolves for specific characteristics of solutions by running the EA multiple times to get a repertoire of solutions?
- **Q2** - How can a Quality Diversity (QD) algorithm (MAP-Elites) be used to generate variants of malware which are both diverse, with respect to multiple characteristics and evasive in a single run?
- **Q3** - How does augmenting training data with the novel variants influence classic classification algorithms in Machine Learning? Which Machine Learning approach benefits most from augmenting with data from the novel variants?
- **Q4** - How can one benefit from the use of a transformer — BERT (that has been trained on large Natural Language Processing (NLP) datasets), in a transfer learning setting to improve classification of metamorphic malware using the novel variants?

1.2 Methodology

An EA was employed in this research for creation and analysis of evasive metamorphic malware. Firstly, a standard EA was employed to generate a new set of malware variants that evade current detection engines, and are diverse with respect to their behavioural and structural similarity. Then, a Quality-Diversity (QD) EA — MAP-Elites [111] was used to generate a set of diverse variants that are optimised with respect to their ability to evade a large set of well-known detection engines. Afterwards, the evolved sets of malware from the two EAs were used as training data for ML models to improve classification of

metamorphic malware. There was an exploration of the possibility of transferring learning from a transformer trained on large NLP datasets, to create even better classification models for metamorphic malware, using the evolved malware as part of the training data.

A quantitative research methodology was adopted in this thesis. This is due to the investigation and analysis making use of both computational and statistical tools/techniques. Experiments were carried out using tools such as Android Emulator, which is used to run the Android files (malware and benign samples) and has been used severally in literature for this purpose as seen in the works of [15] and [155]. For reverse engineering the Android files, tools such as Apktool[4], Zipalign[5] and Apksigner[6] were used. For analysing the samples, tools like Strace[7], Monkeyrunner[8], Virustotal[9] and Droidbox[10] were used. When experimentation was carried out, some statistical tests were carried out to measure the significance of the results. It was achieved by carrying out a number of independent runs of a given experiment as can be seen in Chapters 4, 5 and 6. Experimental work was done using a number of programming languages which include Python, Bash, C and Perl.

1.3 Thesis Contributions

In this thesis, a number of contributions have been made to the field of metamorphic malware analysis and detection by studying ways of creating malware mutants that are evasive, diverse and serve as rich source of dataset for training machine learning models. The contributions are highlighted below:

- Introduced an extended fitness function into a mutation only EA based on malware characteristics (its structure, behaviour and ability to evade existing detection engines), that was tested on three malware families — Dougalek, Droidkungfu and GGtracker. The results showed that using this fitness function, a suite of malicious and diverse malware mutants was created that evade detection by a significantly higher proportion of AV detectors than the original malware. It is also shown that rewarding the EA for generating mutants that are structurally or behaviourally dissimilar to the original malware, also generates evasive mutants (i.e. it is not necessary to explicitly evolve for evasiveness, but that this is obtained indirectly).

[4]APKTOOL - http://ibotpeaches.github.io/Apktool
[5]ZIPALIGN - https://developer.android.com/studio/command-line/zipalign
[6]APKSIGNER - https://developer.android.com/studio/command-line/apksigner
[7]Strace - https://linux.die.net/man/1/strace
[8]Monkeyrunner - https://developer.android.com/studio/test/monkey
[9]Virustotal - https://developers.virustotal.com/reference#getting-started
[10]Droidbox - https://www.honeynet.org/taxonomy/term/191

- Described the first application of a QD algorithm (MAP-Elites) to evolve a repertoire of diverse but evasive solutions in a single run, that was tested on the aforementioned three malware families — Dougalek, Droidkungfu and GGtracker. The results showed that a QD algorithm is able to generate high performing, larger and more diverse repertoire of solutions than the previous mutation only EA for the three families considered. Furthermore, it provides an illumination of the feature space, illustrating the relationship between the characteristics of a mutant malware solution and its ability to evade a large set of detectors.

- Demonstrated improved classification of metamorphic malware by augmenting training data with the novel, diverse variants created using the aforementioned EAs and applying both feature-based and sequence-based classifiers. The results also show that the classifiers do not over-fit to the novel mutants as they are still able to detect existing malware.

- Presented the use of a transformer — BERT (that had been trained on large Natural Language Processing (NLP) datasets), in a transfer learning setting, to improve classification of metamorphic malware using the evolved malware as part of the training data. The result showed that the use of BERT improved the classification performance in some instances and given that the model had been pretrained on large NLP datasets, that knowledge was transferred for the classification of metamorphic malware without the need for large training data.

1.4 Publications

During the course of the PhD research, a number of peer-reviewed publications presenting results that answer our aforementioned research questions have been made and they include:

- K. O. Babaagba, Z. Tan, and E. Hart, “Improving Classification of Metamorphic Malware by Augmenting Training Data with a Diverse Set of Evolved Mutant Samples,” in 2020 IEEE World Congress on Computational Intelligence, Glasgow, UK, 2020.

- K. O. Babaagba, Z. Tan, and E. Hart, “Automatic Generation of Adversarial Metamorphic Malware Using MAP-Elites,” in 23rd European Conference on the Applications of Evolutionary and bio-inspired Computation, P.A. Castillo et al, Ed. Seville: Springer-Verlag New York, Inc., 2020, pp. 117-132.

- K. O. Babaagba, Z. Tan, and E. Hart, "Nowhere metamorphic malware can hide - a biological evolution inspired detection scheme," in Dependability in Sensor, Cloud, and Big Data Systems and Applications, G. Wang, M. Z. A. Bhuiyan, S. De Capitani di Vimercati, and Y. Ren, Eds. Singapore: Springer Singapore, 2019, pp. 369–382.

A copy of each publication can be found in the appendix.

1.5 Thesis Organisation

The rest of the thesis is structured as follows:

- **Chapter 2 - Literature Review**: The second chapter of this thesis introduces malicious software. It then explains metamorphic malware, its creation process, mutation techniques as well as its layers of mutation. Then, it discusses previous detection techniques of metamorphic malware and their challenges. This includes techniques in machine learning, transfer learning, evolutionary computing, among others.

- **Chapter 3 - Research Framework**: This chapter explains the research framework and all software components, tools as well as other metrics employed in the research.

- **Chapter 4 - Using an EA to Evolve New Test Data**: The fourth chapter discusses an adversarial approach to defeating metamorphic malware. This is done by using an EA to create malware that are evasive, yet as behaviourally and structurally dissimilar to the original malware as possible. These mutant samples serve as training data to ML models.

- **Chapter 5 - Using MAP-Elites to Evolve a Repertoire of Diverse Solutions**: In the previous chapter of the thesis, evasive mutants were generated. However, the diversity of the samples could not be guaranteed. Here, a quality diversity algorithm is used - MAP-Elites to drive diversity in the mutants created.

- **Chapter 6 - Improving Classification of Metamorphic Malware**: In this chapter, the classification of metamorphic malware is improved using the previously evolved mutant samples to augment training data. A comparison is also made between feature based and sequential classification for classification performance as well as binary and multi-class classification. Then, learning from a pretrained NLP model is also done in order to see if there is an improvement in the classification performance of the previously built ML models.

- **Chapter 7 - Conclusions and Future Work**: The final chapter summarises and concludes the thesis. The limitations encountered while carrying out the studies are highlighted. Also, areas of future work are brought to the fore to improve the aforementioned solutions proposed.

Chapter 2

Literature Review

In this chapter of the thesis, a background and thorough review of metamorphic malware analysis and detection techniques, is provided. This chapter begins with a description of malicious software, its life cycle, types and characteristics. Then, the platforms in which malware can be activated are discussed. Metamorphic malware is then introduced and discussed including its creation process, mutation techniques as well as its layers of mutation. Furthermore, a brief history of metamorphic malware detection and analysis techniques, is also provided. This includes signature based, heuristics based, malware normalisation and similarity-based techniques. The limitations of current solutions to metamorphic malware detection are also explained. Then, evolutionary based malware detection is discussed including techniques such as the use of quality diversity algorithms.

2.1 Malware (Malicious Software)

A number of definitions have been given to malware by different researchers at various points in time. Malware can be described generically as a type of malevolent code which can also be termed malicious code, malcode, malicious software, to name a few. [107] defines malware as any alteration to a software system in form of addition or deletion of code that is done deliberately to distort the purpose of the system or cause any form of damage.

[145] points out the fact that malware represents a word that basically describes all forms of malicious code such as viruses, spyware among others. Also, [36] describes malware from the angle of its purpose. This means that any program purposefully intended to cause an intrusion can be termed malware. It is noteworthy that these definitions all agree on the fact that malware aims to violate the confidentiality, accuracy or availability of computer or network resources.

2.1.1 Malware Life Cycle

This refers to the stages that are usually followed from the process of creating a malware to its eradication. According to [115], this usually involves processes such as creation, replication and propagation, activation, discovery, assimilation , and eradication. This is illustrated in Fig. 2.1.

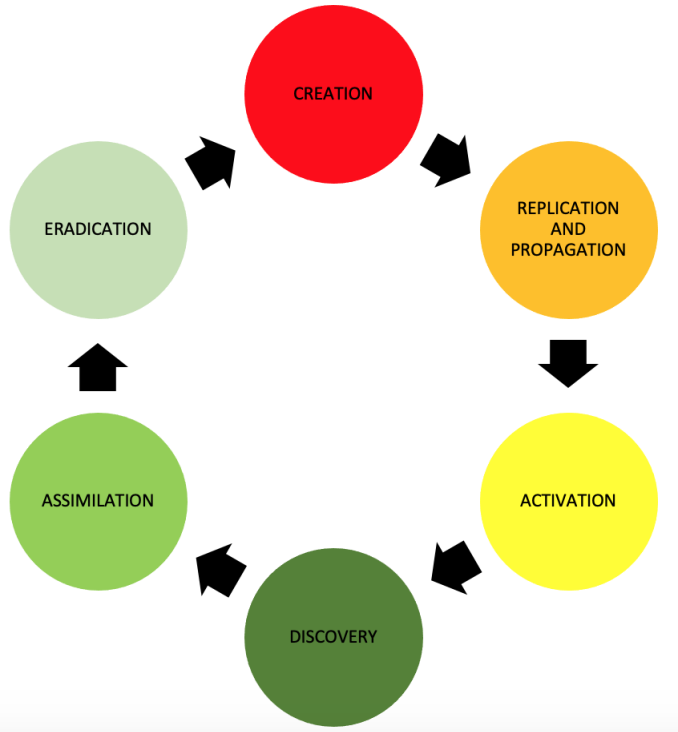


Fig. 2.1 Malware Life Cycle

The malware life cycle begins firstly with the creation phase. It is in this phase that the malware is generated and built. An in-depth understanding of computer networks and programming was required in time past to create malware; presently, the process of malware generation has been automated. Without any serious background in computing, malware can be created. Also, self-help materials are readily available on the Internet as well as tools that automate the task of creating these malware.

The second phase involves replicating and propagating the malware. Once the malware has been created, it is now up to the creators to decide how the malware will be spread. Different malware have different methods of propagation and replication. One common means of propagation is via a computer system as seen in computer viruses. Another means is through emails and instant messaging which is often the way worms are propagated. Once these malware have been propagated, the next phase focuses on their activation. In the activation phase, the malware launches its attack when it is executed. The activation

process is carried out via a number of payloads usually installed on the compromised system. This process might be based on a given condition or could be automatic.

The fourth stage is the discovery stage. This is when the malware analysts detect the malware and attempts to isolate the malware. This phase can happen at any point in the malware life cycle. However, it occurs more frequently after the malware has been activated. In order to keep track of discovered malware they are usually sent to ICSA (International Computer Security Association), where they are published for all malware analysts and builders of antivirus engines. There is still a massive need to keep our systems safe from malicious attacks as the generators of malware will usually ensure that they are able to release the malware before they are detected.

The fifth stage is the assimilation stage which simply refers to the efforts made by malware analysts and antivirus engine creators to change their programs so that they can discover the newly designed malware. This might take quite some time depending on the nature of the malware as well as the developers involved. The final stage is the eradication stage. As the name implies, it is the stage where the malware is eradicated. The process of malware eradication is enhanced when the users install the latest versions of antivirus software. Like the assimilation stage, this process can take a while, hence the need to avoid the propagation of the malware in the first instance.

2.1.2 Malware Types and Characteristics

Malicious software often take various forms and possess varying characteristics. The authors of [115] outline at least seven types of malware which are given below:

Virus: A code that replicates itself is usually termed a computer virus. It does the replication by inserting itself into other programs after which it is copied and transferred to other systems. Viruses are activated upon the execution of a compromised file. A common example of a virus is Michelangelo [142].

Worm: Worms are similar to viruses but carry out replication by executing their own code independent of any other program. They are able to spread to other systems usually via emails as well as networks connected to each other. Any loophole they find in a vulnerable system, they take advantage of, mostly without any need for human intervention. Typical examples of worms include Code Red [45], SQL Slammer [110] among others.

Trojan Horse: This is a malware embedded by its authors in an application or system. The application or system it is embedded in seems to perform some useful task, but in reality, it is performing some unauthorized action. One thing to note about Trojan

horses is that unlike viruses and worms, they cannot replicate themselves. However, they often have a payload and they often result in system as well as data integrity breaches. Common examples include Hydan [27] and Setiri [137].

Malicious Mobile Code: This refers to a software which can be downloaded and locally activated remotely with or without little human interaction. They are usually lightweight and are able to take control of the compromised system to carry out malicious action. They often serve as transmission media for viruses, Trojan horses as well as worms. Similar to Trojan horses, they do not replicate themselves; they rather seek system loopholes and make the best use of the basic privileges that come with mobile programs. These malicious codes are often written in languages such as JavaScript, VBScript, ActiveX among others. One popular malicious mobile code which was written in JavaScript is Nimda [102].

Blended Threat: This refers to a combination of malicious behaviour, all in one attack. Some of these behaviours are from malware such as viruses, worms, malicious mobile code and Trojan horses. The entire attack life cycle of blended threats is dependent on some of the loopholes in servers as well as the Internet. The fact that blended threat is a blend of malicious behaviours obviously suggests its higher attack surface and spread of propagation. A blended threat attack transmits several attacks in a single payload which could be a combination of say a backdoor attack and a denial of service attack. It often causes multiple levels of harm to the affected system and so are considered a serious security threat. Lion and Bugbear are two of the most notable blended threats [41].

Attacker Tools: These are often considered as some of the baggage that come with being a victim of malicious attacks. They refer to tools which comprise of a number of malware that an infected system gets by reason of a malware attack. They take various forms; some of which try to get important data or even get access to a system. The transfer of the attackers can take place at different times depending on the malware involved. Some common attacker tools include:

- Backdoor: This refers to malware that allows the attacker access to a computer by installing itself on the computer. They pay attention to TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) port instructions, and once a connection is established, they take over the compromised system. Examples of backdoors include Zombies [35] often referred to as Bots and Remote Administration Tool (RAT).
- Rootkit: These are malicious codes designed to hide the existence of other malicious codes. This tool is often used by cybercriminals when they have

gained access to an infected system during which they install other malicious content to it. There are two types of rootkits depending on the type and level of access required. They are: user-level and kernel-level rootkit.

- Keystroke loggers: These types of attacker tools keep track of how users make use of the keyboard. They monitor letters typed and so they can gather confidential user information such as passwords, credit card numbers, among others. Some common examples are Spyster [13] and KeySnatch [118].
- Tracking Cookies: Several websites often keep user information that relates to a given website usage. They do this through a file called cookie. They can either be temporal also known as session cookies or long term also known as persistent cookies. They are used sometimes by cybercriminals as spyware to monitor the activities of users on the web.
- Web Browser Plug-Ins: These are used to execute or allow a given action or content via a web browser. Some cybercriminals build malicious ones to spy on users.
- Email Generators: These are attacker tools that are capable of transferring software that can create emails to computers. These computers then are able to generate as well as send unauthorized bulk emails to other computers.
- Attacker Toolkit: These are programs and utilities that are capable of causing harm to a system. They include packet sniffers, port scanners, password crackers, remote login programs to mention a few.

Spyware: These are manipulative malware that attempt to gain unauthorized access to user data or an entire system. Like the name suggests, they do this by spying on users' activities. Some of them seem like legitimate systems such as a security monitor that might just be a spyware. Others are in form of advertising programs also called adware. Some are information stealing malware or computer configuration changing malware. They often take different forms but with the motive of gaining unauthorized access to the user's information.

2.1.3 Malware Platforms

Malware platforms refer to the environment in which malware can be activated. Without an enabling environment, malware cannot be executed. For the purpose of this research work, two major platforms for the execution of malware are considered and they are Desktop based platforms and Mobile platforms. They are explained below [46]:

Table 2.1 Malware Platforms [46]

Malware Platforms	Categories
Desktop-based	Windows based, Mac based and Linux based
Mobile-based	Android OS, iOS, Bada and BlackBerry OS etc.

Desktop based malware platforms: These refer to desktop computers that malware can be run. These types of platforms are often categorized according to the Operating System (OS) that run on them. Examples of desktop based malware platforms include Windows based malware platforms, Mackintosh (Mac) based malware platforms and Linux based platforms.

Mobile based malware platforms: These refer to mobile devices in which malware can be executed. Examples include Android OS (Google Inc.), Bada (Samsung Electronics), BlackBerry OS (Research In Motion), iPhone OS/iOS (Apple), MeeGo OS (Nokia and Intel), Palm OS (Garnet OS), Symbian OS (Nokia), webOS (Palm/HP), among others.

2.2 Metamorphic Malware

This refers to malware that changes its entire code between generations. Usually, this involves the malware mutating itself and hiding its instruction within the normal program code of the host machine [32]. This malware transforms its code using several obfuscation techniques. Some of these techniques include; garbage code insertion, variable substitution, control-flow alteration, among others.

Metamorphic malware pose a lot of threat to computer resources as they usually go undetected by most traditional signature based detection systems. This is because there are too many variants of the malware to keep track of and thus, they often go undetected. These variants are very dissimilar. The difference between metamorphic malware and polymorphic malware is that metamorphic malware lack a decryptor and have a single body carrying data as code. Fig. 2.2 shows how its structure changes between generations.

A formal definition of a metamorphic malware is given as follows [159]: A metamorphic malware (*m, m´*) is a combination of two distinct malware namely *m* and *m´*. The difference between a metamorphic malware and any other malware is that a system attacked by *m* is also attacked by *m´* and vice versa. The same definition goes for metamorphic malware $m_1, m_2, \ldots, m_n$.

Given a function $\Phi_M(d,m)$ for a program M in a computer system located in an environment *(d,m)* where *d* represents data and *m* represents computer programs. *D(d,m)*

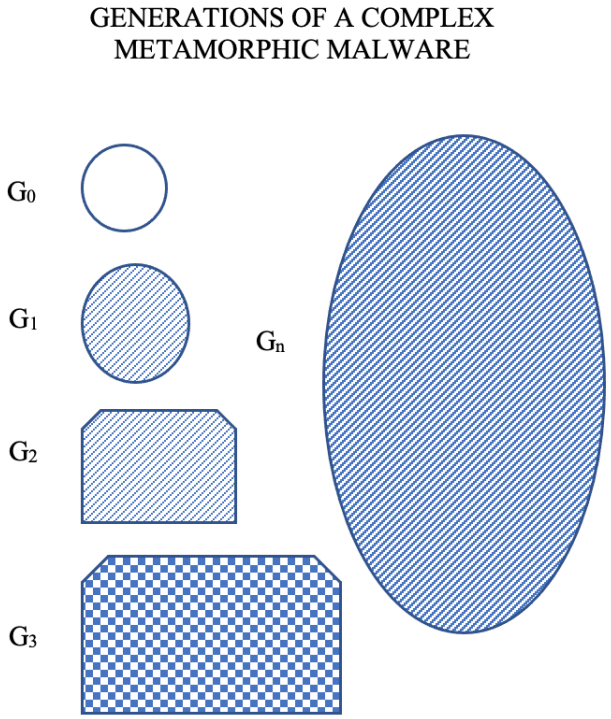


Fig. 2.2 Metamorphic Malware Generations [135]

and *S(m)* represent two recursive functions, where *D(d,m)* is the injury function and *S(m)* is the selection function. *T(d,m)* and *I(d,m)* are two recursive predicates in which a pair of *(d,m)* at the very least exists for which *I(d,m)* holds. Also, no *(d,m)* exists whereby both *T(d,m)* and *I(d,m)* concurrently hold.

T(d,m) is called trigger which is an injury condition, and *I(d,m)* is the infection condition. When the injury condition *T(d,m)* holds, the malware runs the injury function *D(d,m)*. However, when the infection condition *I(d,m)* holds, the malware uses the selection function *S(m)* to select a program, and infects it before running the initial program x. The aforementioned conditions, *T(d,m)* and *I(d,m)* as well as the functions *D(d,m)* and *S(m)*, define the kernel of a non-resident malware [159].

Then, it can be assumed that two different functions *mal* and mal' are metamorphic if for all x, (mal, mal') satisfies;

$$\Phi_{mal(\chi)}(d,m) = \begin{cases} D(d,m), & \text{if } T(d,m) \\ \Phi_{\chi}(d,m[mal'(S(m))]), & \text{if } I(d,m) \\ Phi_{\chi}(d,m), & \text{otherwise} \end{cases} \tag{2.1}$$

and

$$\Phi_{mal'(\chi)}(d,m) = \begin{cases} D'(d,m), & \text{if } T'(d,m) \\ \Phi_{\chi}(d,m[mal'(S'(m))]), & \text{if } I'(d,m) \\ Phi_{\chi}(d,m), & \text{otherwise} \end{cases} \tag{2.2}$$

where T(d,m) (resp., I(d,m), D(d,m), S(m)) is different from T$\acute{}$(d,m) (resp., I$\acute{}$(d,m), D$\acute{}$(d,m), S$\acute{}$(m)).

2.2.1 Creation Process

The process of creating a metamorphic malware is often considered a complex task by most malware writers. The metamorphic malware engine makes use of the following components in the creation process, namely, internal dissembler, opcode shrinker, opcode expander, opcode swapper, relocator, garbager and cleaner [85]. The function of each of the aforementioned components are outlined below:

Internal disassembler: This component is responsible for the code disassembly taking each instruction at a time.

Opcode Shrinker: This component serves as an optimizer and as the name connotes, is responsible for shrinking two or more instructions into a single instruction.

Opcode Expander: This component does the opposite of the opcode shrinker. It expands a single instruction into multiple instructions.

Opcode Swapper: This component is responsible for swapping two or multiple instructions.

Relocator: They are used for relocating every relative reference. Some of which include calls, jumps and pointers.

Garbager: These are used to insert into the original code one or several junk instructions often referred to as do-nothing instructions.

Cleaner: They are responsible for the cleaning up of the garbage code the garbager inserts.

The anatomy of a metamorphic malware is illustrated in Fig. 2.3. The metamorphic malware engine according to [151] has the following functions:

Firstly, whenever the metamorphic malware engine is invoked, it must be able to find the code it needs to transform. Those metamorphic malware that change not only their

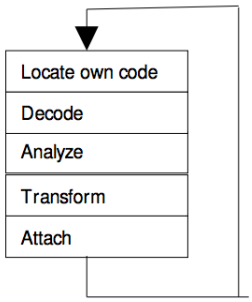


Fig. 2.3 Anatomy of Metamorphic Malware[151]

code but those of their host, have to find their own code in the new mutants. Also, in order for it to be able to carry out the necessary changes, it must be able to decode information.

The metamorphic malware needs some information about its representation in order to mutate itself. This usually means a process of disassembly is required. Occasionally, the engine needs to decode several other information in order to analyze or change the code. This information can either be encoded in the data segments or code of the malware. After decoding the required information, analysis of these information needs to be carried out.

The analysis stage will require that some pieces of information are readily available. This is to ensure that code changes are done appropriately. For instance, some changes only retain the semantics of the original code if a particular register is seen as dead. In this case, information about the register liveness must be readily available.

The malicious code has to be transformed to an equivalent code after the analysis has been done. This often involves instruction blocks swapping among other techniques. Some of the transformation techniques include code substitution, garbage insertion, register renaming, among others.

Finally, the metamorphic malware attaches its mutated version into a host file. The way the organs are ordered indicates the direction of the information flow. The malware uses a feedback loop to determine its input and output, in which case, the input in one generation might become the output in another generation. This is often necessary in making design decisions with regards to the metamorphic engine.

2.2.2 Mutation Techniques

Metamorphic malware transform their codes using the following mutation techniques: instruction substitution, garbage insertion, variable substitution and control-flow alterations [32].

Instruction Substitution: This involves the replacement of an instruction set with a valid and distinct set which serves the same function. An example of this can be

seen in a malware that substitutes an instruction “xor eax, eax” with an instruction “sub eax, eax”. In the above example, the two instructions are equivalent, hence no functional difference is noticed. However, a difference with respect to their opcodes is noticed.

Garbage Code Insertion: This is a very common mutation technique which involves making the original code as unidentifiable as possible, while preserving its original functionality. It is often referred to as dead-code or do-nothing code insertion. It works by inserting junk instructions in the original program code. This garbage code has no effect on the functionality of the code, but changes the code’s structure. This approach can particularly distort opcode based analysis systems. An example is illustrated below and is taken from an Android malware. The description of an Android malware is given in Section 3.3 of the thesis. The lines labelled garbage code in Fig. 2.4, are line numbers added to the Android program that do not affect the functionality of the program code.

```
# direct methods
.method public constructor <init>()V
   .locals 1

   .line 0        garbage code

   .line 0        garbage code

   .line 0        garbage code

   .line 0        garbage code

   .line 0        garbage code

   .line 0        garbage code

   invoke-direct {p0}, Landroid/app/Activity;-><init>()Via

   const-string v0, "http://depot.bulks.jp/get41.php"

   input-object v0, p0, Ljp/oomosirodougamatome/MainActivity;->d:Ljava/lang/String;
```

Fig. 2.4 Garbage Code Insertion

Variable Substitution: This is also known as register substitution. This is when the malware replaces a given memory address known as a variable with a different but valid variable without altering the code’s function. This can be seen in Fig. 2.5 and is also taken from an Android malware that uses variable substitution mutation technique. In the example given below, a variable v0 is replaced with another valid variable without changing the functionality of the code.

Control-flow Alteration: This is another common mutation technique. In this technique, the flow of instructions and the code’s structure are distorted by the malware. This is

```
.method private a(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V
   .locals 3
   const-string v0, "check"
   new-instance v2, Lb;
   const-string v0, "http://depot.bulks.jp/get41.php"
   new-instance v1, Ld;
   invoke-direct {v1, p0}, Ld;-><init>(Ljp/oomosirodougamatome/MainActivity;)V
   invoke-direct {v2, p0, v0, v1}, Lb;
><init>(Landroid/app/Activity;Ljava/lang/String;Landroid/os/Handler;)V
   const-string v0, "id"
   invoke-virtual {v2, v0, p1}, Lb;->a(Ljava/lang/String;Ljava/lang/String;)V
   const-string v0, "tel"
   invoke-virtual {v2, v0, p2}, Lb;->a(Ljava/lang/String;Ljava/lang/String;)V
   const-string v0, "data"
```

Fig. 2.5 Variable Substitution

done through the insertion of meaningless loop instructions which do not distort the order of instruction execution at runtime. The module re-ordering is given below:

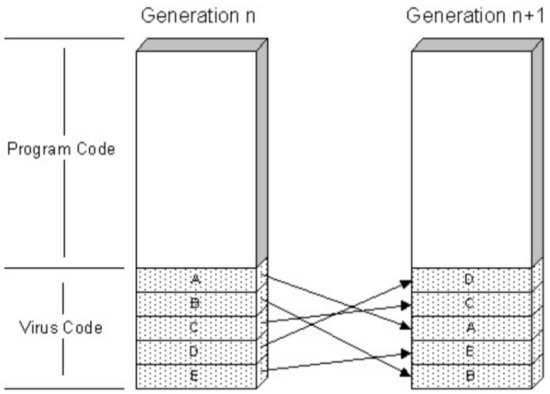


Fig. 2.6 Module Re-ordering [76]

The malware also converts function calls as well as direct jumps into indirect ones through the use of fake destination addresses, so that it is difficult to reconstruct the control flow. An example showing how jump instructions can be inserted into a program is seen in the Win95/Zperm virus [135]. The first generation of the virus is shown below:

```
instruction 1       ; entry point
instruction 2
.
.
.
instruction n
```

A latter generation after some jump instructions have been added to it is shown below:

```
instruction 2
jmp instruction 3
instruction 1              ; entry point
jmp instruction 2
instruction 3
jmp instruction n
```

A sample of the Zperm virus illustrating how it inserts jumps into its own code is given below:

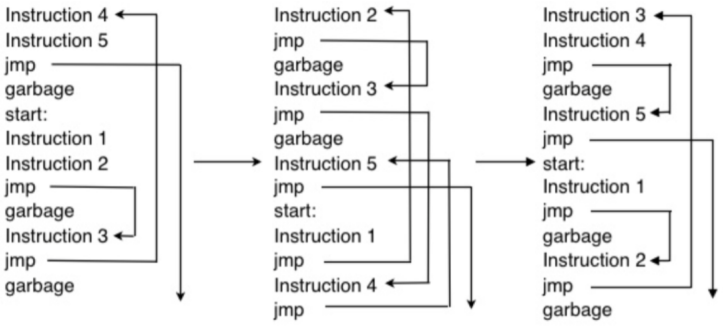


Fig. 2.7 Zperm Jump Insertion [135]

2.2.3 Layers of Mutation

Metamorphic malware binaries have various layers in which they mutate. According to [148], some of these layers include:

Network Layer Mutation: The network layer of the OSI model is responsible for determining paths as well as logical addressing. This layer presents an attack surface for mutable malware; it keeps posing challenges for intrusion detection systems as mutations on this layer are difficult to detect. Two major examples of this network layer mutation are IPv6 and IP Packet Splitting [148].

- IPv6: IPv6 stands for Internet Protocol version 6 and it is the new age addressing model. It provides significantly more address space, a simpler header, automatic configuration, end-to-end connectivity, speedy routing, security of the protocol, extensibility, no requirement for broadcast, among others. The IPv6 is the Internet protocol used by most Internet networks. A number of reasons, ranging from slow deployment, loopholes allowed by most vendors, make it difficult for lots of network based IDS to deal with packets sent through IPv6 networks. Hence, attacks sent via IPv6 go undetected.
- IP Packet Splitting: Packet splitting involves delivering network packets in smaller fragments; this creates a greater attack surface for the delivery of attacks. Usually, network-based IDS try to discover attacks by going through the entire packet length. The attackers will attempt to evade detection via IP packet splitting. Some techniques such as reassembly of IP packets make attacks less likely to go undetected, but the process is computationally expensive, ergo, are not used by a number of network based IDS.

Application Layer Mutation: Another potential mutation layer is the application layer. Some of the main techniques employed in this layer include protocol rounds, evasion of SSL, HTTP, FTP protocols.

- Protocol Rounds: This is a process that enables multi-processing of sessions in the application layer over one connection in the network. This comes with several advantages such as improved computational speed, efficiency, and cost reduction. In protocol rounds however, only the first rounds are often monitored by network based IDS; subsequent rounds are often not monitored. The snag with this is that attackers can take advantage of this loophole in order to introduce malicious protocol rounds. They can trick the IDS by first introducing normal protocol rounds, followed by series of attacks.
- FTP/HTTP Evasion: In order to go undetected by network based IDS, a number of attackers try to add malicious control sequences to FTP commands or attempt to modify HTTP protocol to enable malicious requests to be received by web servers. Even if some IDS use SideStep to discard irrelevant FTP control sequences, some attackers still find ways to evade detection. Also, lots of IDS try to detect intrusive HTTP streams but ignore some invalid character insertion, carriage returns as well as white spaces.
- SSL NULL Record Evasion: Secure Sockets Layer also known as SSL is a protocol which has the responsibility of ensuring that network communications are secure and not subject to interception by any third party. SSL messages

consist of a body and in some cases, include null records. Attackers attempt to insert malicious content into these null records. Also, a handshake could be changed by the attacker to include some illegal values and this can go undetected by the IDS.

Exploit Layer Mutation: This type of mutation is applied to the attacks rather than the network session. They range from polymorphic shell-code to alternative encodings, among others [148].

- Polymorphic Shell-code: A number of polymorphic shell-code engines exist primarily to create malicious payloads that decrypt themselves. Some of these engines evade IDS using attributes such as creation of polymorphic payload decoding schemes, illegal characters support, XOR-encoded payloads among others. They also allow junk code insertion, code substitution among others.
- Alternate Encodings: In order to ensure that data integrity is not breached during communication within the network, data is often encoded severally during network communication. These encodings create an attack surface which attackers often exploit. URL-encoding Obfuscations are also attacks a number of IDS fall victim of.

2.3 A brief history of Metamorphic Malware Detection

A number of techniques have been developed over the years to combat the threat of malicious software. The strategies that have been used can be grouped into three, namely

- Signature based detection,
- Heuristic based detection,
- Malware normalisation and similarity-based detection.

2.3.1 Signature-based Detection

This involves the extraction of unique byte streams which define the malware's signature. It involves scanning files in the host machine in order to find a given malicious signature. The work of [98] was one of the first signature-based malware detection scheme and has subsequently been referenced by other research works such as [136]. The authors of [98] proposed a system called Malicious Code Filter (MCF) which was a static analysis tool used for malware classification. Their scheme looked for tell-tale signs in malicious code. These signs refer to attributes of a piece of program code that can be used in

determining if a piece of program code is malicious or not, without the need for expert coding knowledge. Their system was successful in proving that tell-tale signs are useful in identifying malicious code.

Several signature-based methods have been used for detecting metamorphic malware in particular. The authors of [64] used string signatures for metamorphic malware detection and they achieved a false alarm rate of less than 0.1%. The work, presented in [140], introduces Aho-Corasick (AC), which is a string matching algorithm for detecting metamorphic malware. [134] employs a static scanner for detecting metamorphic malware with high detection rate. More signature-based metamorphic malware detection techniques such as string scanning with special cases like wild-cards or mismatches, bookmarks, speedup search algorithms [86], are found in other related work.

Signature-based methods of malware detection provide a fast and easy means of detecting malware. However, they are often not efficient in detecting advanced malware such as malware that employ obfuscation techniques in masking its code structure. This is because they are only able to recognise specific code versions. It will therefore be useful if signature-based systems are fed with new data that represent potential variants.

2.3.2 Heuristic based Detection

A detection technique that involves the analysis of the behaviour, functionality, and characteristics of a suspicious file without relying on a signature is referred to as heuristic detection. Unlike signature-based detection, its goal is not to discover a given signature but to detect malicious functionalities like a malware's payload or distribution routine. This method employs data mining and machine learning techniques in detecting malware. These include supervised learning (learning with a guide), semi-supervised learning (learning with a partial guide), and unsupervised learning (learning without a guide). Some of these machine learning techniques include Decision Trees (DT) [21], Hidden Markov Models (HMM) [139], and Support Vector Machines (SVM) [129].

Machine learning based malware detection is data driven; it discovers relationships between the underlying structure of data, collected either before or after the execution of the malware and its classification as malicious or non-malicious. Data collected prior to the execution of the malware, includes information about the file derived from it without running. These include its code characteristics and its file format among others. On the other hand, data collected after the malware executes derives from the artefacts left behind by the executed malicious code. These include behavioural descriptions of the malicious code such as process related activities, registry related activities, among others.

In unsupervised machine learning, the aim is to obtain previously unknown structural descriptions of data without a guide, for instance, pre-existing labels. This can be done in a

number of ways, in which clustering analysis is commonly involved and helps segmenting a dataset. Unsupervised learning based malware detection does not require the datasets used to be labelled as either clean or malicious. This makes unsupervised learning useful to cybersecurity experts as a wide variety of unlabelled datasets are readily available.

In supervised machine learning the data has to be labelled as either malicious or clean. This helps the model in determining the labels for new instances. Supervised learning models have to be trained with sufficient data. The trained model is then fed with new samples for prediction. The model needs to be appropriately trained with enough data for better predictions. A pioneering work in heuristic based malware detection is that of [132] which uses Naïve Bayes (NB) in automatically identifying malicious patterns in malware. Their NB approach, calculated from the program's feature set, the probability that the program is malicious. Their heuristic based approach was better than other previously employed signature-based approaches in terms of detection accuracy. Since then, a number of research works such as [130], have used heuristic methods for malware detection.

In metamorphic malware detection, heuristic based methods such as DT, HMM, and SVM have been used. For instance, a combination of statistical chi-squared test and HMM is used by [139] in detecting metamorphic viruses. [21] also uses a statistical-based classifier that employs a DT in metamorphic malware detection. A single class SVM is used by [129] in Android based metamorphic malware classification. These works led to increased metamorphic malware detection rate.

Generative Adversarial Network (GAN)

A number of the previously described ML techniques are designed as a black box so as to improve network security. These approaches assume that provided the attackers do not have access to the underlying ML algorithm, they will struggle in attacking them. However, this assumption is flawed by the fact that a number of attackers design strategies to probe the network, and consequently discover the design features that can be identified as being malicious so as to create evasive software that go undetected by the ML models.

The previously described approach is adversarial in nature and is designed to generate malware that takes advantage of the loopholes in ML models. This method uses deep convolutional neural network to collect data employed in the analysis of malicious software, particularly in the categorization of samples as either clean or malicious. A different network is designed to generate malicious samples to be identified by the initial network as benign. At first, the network performs poorly but with more iterations, this leads to an increased ability of the malware created to go undetected. This is referred to as gradient based adversarial generation and this approach has proven that with very little modification to malware, they can evade detection.

Following from the traditional gradient-based adversarial generation previously explained, the Generative Adversarial Network (GAN) [61] was introduced by Ian Goodfellow in 2014. It differs from the gradient-based adversarial generation method in that rather than training a network to create malicious code against a pre-trained detector, both networks learn through competition with each other. The GAN framework was designed to build generative models that use an adversarial approach that trains two distinct models concurrently. The idea behind it is that deep learning models come in hierarchical structures characterising varying probability distributions over most conventional data that have been very successful in the generation of discriminative models [124]. This is due to the employment of back propagation and dropout algorithms yielding positive training gradient [11]. The GAN models work by using two networks: one which is called a generator trained to learn data specific attributes such as malicious code, images, among others while trying to create a believable sample of this data; the other termed the discriminator determining the probability that a given sample is a member of the training set rather than the generative network. It is noteworthy that the generator maximizes the probability that the discriminator will fall into error [61]. The intrinsic competition between both networks leads to steady improvement of both networks to the point where the generator's samples cannot be distinguished from the testing set of the discriminator [91].

The general idea behind such adversarial models is to create adversarial samples that are specially designed to fool detectors and machine learning models through perturbations that make the ML models misclassify them as benign instead of malicious. The goal of the attacker according to [26] is one or a combination of three things: security violation, attacker specificity, and error specificity. If the attack aim is security violation, then the attack may lead to either an integrity, availability or privacy violation. Depending on whether or not the attack is targeted at causing a classifier to misclassify either a given sample set or any sample, the attack specificity might be targeted or indiscriminate. If the attacker wants a sample misclassified as a particular class then the error is specific. However, if the attack is aimed at misclassifying a sample to any class other than its correct class, then the error is generic.

Several examples abound where GANs have been applied. The researchers in [30], for instance, proposed a model in the domain of spam email filtering that demonstrates how the learner and the data generator interact as a static game. They analyse the adversarial setting, strategy, and conditions with which the prediction has a Nash equilibrium and design algorithms that discover the equilibrial prediction model. The authors in [72] proposed MalGAN, which comprises of a generator that generates a suite of adversarial samples against current detectors which also have the ability to go undetected by future malware detectors. It trains the generator to trick the detector in a hybrid system which

consists of a stacked generator as well as a substitute detector. The work of [104] also describes the use of a Least Squares Generative Adversarial Networks (LSGANs). Their system employs the least squares loss function for the discriminator with experimental results showing two advantages over usual GANs, the first being its ability to create improved images than the typical GANs. Also, the learning process of the LSGANs is more stable than the regular GANs. The use of GANs however, has not been explored in metamorphic malware detection.

Transfer Learning

Although this has also not been explored in metamorphic malware detection, due to the fact that metamorphic malware keep changing their code, there is usually insufficient training data, hence impeding ML model generality. Transfer learning is a good technique to address this problem as it is a machine learning technique wherein the knowledge generated from a task is stored and reused in another task, often a related task [117].

Transfer learning as a technique for use in instances of small training set or small labelled data has received a lot of research attention. It has found application in a number of domains such as in medical application as seen in the work of [105] that modified the AlexNet [88] in order to detect Alzheimer's disease. It has also been used in bionformatics such as in [120] that used it for the study and prediction of associations in genotype-phenotype using Label Propagation Algorithm (LPA) [73]. In the transportation domain, it was applied in [44] to process similar images derived during varying conditions. It has also gained attention in NLP as seen in the work of [128].

Transfer learning has increasingly been used in malware detection. It has been used in computer vision for instance as in [34] that introduced a computer vision-based deep transfer learning for classifying static malware, using knowledge from objects appearing in nature. Also, [127] used Deep Neural Network (DNN) built from the ResNet-50 architecture in classifying malicious software. The malware were converted to grayscale images. Then, the DNN that had been trained previously on the ImageNet dataset, is used in classifying the malicious samples. Similarly, the work of [24] built a deep learning model pre-trained on a large set of image data to improve the classification of malware.

Transfer learning has also been employed in Generative Adversarial Network (GAN) settings as seen in the works of [81] and [82]. As in [82], their model comprised of a generator that created adversarial samples. The detector learns the characteristics of the malicious samples using a deep autoencoder (DAE). Prior to training the GAN, the DAE uses the learned features of malware to create data and transfers the learned information to improve the training of the GAN generator. Their method was shown to outperform other models designed for the same application.

2.3.3 Malware Normalisation and Similarity-based Detection

An attempt to transform metamorphic malware to its original form is termed malware normalisation. The level of code obfuscation determines the effort that will be put into normalising the metamorphic malware. This technique was first introduced by Periot [119] whose approach took advantage of various code optimisation strategies in enhancing the detection of malware. [150] also uses term rewriting as a means of normalising metamorphic malicious code. The various mutations of the metamorphic malware are modelled as rewrite rules which are then changed to form a rule set for normalising the metamorphic malware. This approach was applied to the metamorphic engines' rule set and was used in the normalisation of variants produced from the mutation engine. A comprehensive list of techniques for code normalisation is given in [32].

In addition, similarity-based approaches, for instance, structural entropy and compression-based techniques, were applied in [22] and [92] to detect metamorphic malware. While structural entropy involves the examination of the raw bytes of the mutated file, compression-based detection involves the use of compression ratios of the mutated file in a bid to create sequences that represent the file. In the work of [22], structural entropy was used for metamorphic malware detection. Their approach involved segmenting the binaries and then finding the similarity between their segments. The authors in [92] used a compression based technique in detecting metamorphic malware. Their approach used compression ratios in defining the files. Then, they compared the file sequences against one another, and then used a scoring system to classify the files as either malicious or clean.

2.4 Limitations of Current Solutions to Metamorphic Malware Detection

Metamorphic malware is a class of highly sophisticated malicious software that commonly involves complex transformations of its code during each propagation. Metamorphic malware writers usually attempt to attack the different phases in the malware analysis life-cycle [89]. These attacks include the following:

- **Attacks on disassembly**: Disassembly is done in malware analysis in order to generate a form of representation for the malicious code. The aim is usually to separate the code and data in a malicious binary. One popular method of disassembly is termed the linear sweep method. In this method, the initial byte streams or byte streams from a given start location are assumed to be instructions. The code sequences in these streams are then disassembled. One disadvantage of this method is that it is not suited for automated analysis. The recursive traversal method

overcomes this challenge by carrying out the disassembly process by following the program's flow of control. Attack on disassembly works by fooling either of these methods. The linear sweep method can be fooled by inserting junk code right after an unconditional jump instruction, and the recursive traversal can likewise be fooled by introducing a junk code immediately after a conditional jump instruction while ensuring that the jump condition is always true. The disassembly is thrown off, provided that the junk code matches a legal instruction code. Consequently, any heuristic that validates the instruction succeeding the jump instruction is defeated.

- **Attacks on procedure abstraction**: Procedure abstraction involves grouping the malware's instructions into segments called procedures. A procedure forms the core of most algorithms used for malware analysis. An attack on procedure abstraction involves increasing the difficulty involved in finding a procedure in the malicious code. This can be done through several means like hiding the call instruction through code obfuscation, among others.

- **Attacks on control flow graph generation**: Each of the program codes' procedures have Control Flow Graphs (CFGs). CFGs are directed graphs with nodes representing statements, and edges representing the flow of control between the statements. The creation of these CFGs can be attacked through deceiving the CFG generation algorithm into producing edges that are redundant. An attack on the CFG generation can also be done through the obfuscation of assembly code, making it difficult to determine for instance, a jump instruction's target.

- **Attacks on data flow analysis**: Data flow analysis comprises of several analysis such as data dependence analysis, analysis done on obfuscation optimization, among others. Data flow analysis works with the assumption that each program has data divided into variables each with a unique identifier, type and size. In reality, each program's data is just a continuous byte sequence. In a CFG, data usually flows into a new node from two different preceding nodes, each of the preceding node combining to form a single set. This combination often leads to information loss and consequently inaccurate analysis. An attack on data flow analysis can involve the movement of some parts of a CFG into the nodes it precedes, and obfuscating the computation along each of the CFG's path. Another attack can result from the usage of data outwit the program's scope.

- **Attacks on property verification**: Property verification checks if a given property exists within the malicious code. This process usually involves both a formal definition of the property deemed suspicious as well as the malicious code's CFG and data flow information. Property verification uses theory provers that rely

on human intervention to prevent them from looping infinitely. Most of these theory provers, such as model checkers which work on problem abstractions, are computationally demanding. Theory provers are usually used in antivirus engines in the determination of suspicious properties of malware. An attack on property verification can come from malware writers who know how these theory provers work, and how they transform the suspicious properties into more complex ones.

Due to sophisticated means by which metamorphic malware can change its code, many existing detection approaches perform poorly. Signature-based detection approaches, for instance, are not efficient when faced with novel metamorphic malware. They are not only very time-consuming since they require new signatures to be compared against large databases of malicious signatures, but are also required to periodically update their databases. Moreover, signature-based approaches are often reactive and therefore cannot detect new attacks.

The file scanning process in heuristic based detection is usually only based on the attack name/label, leading to limited information derived. This method sometimes uses statistics for its predictive analysis, which is prone to diagnostic errors emanating from the initial learning process being corrupted. If the algorithm is not trained appropriately, the resulting predictions may be inaccurate.

Most malware normalisation and similarity-based approaches still cannot detect advanced metamorphic malware with complex levels of obfuscation. Consequently, low detection rates are derived when they are used on such malware. In the case of malware normalisation using control flow graphs, the normalisation process can be hampered by code streams that cannot be reached, which can lead to control-flow graph fall-through edges. The complex code streams (such as those that employ opaque predicates and branch functions) are often difficult to be detected. Similarity based techniques are often prone to false alarms, and are susceptible to mutations that employ a lot of packing or compression. The compression ratio is very important in the segmentation phase of compression-based similarity detection. Consequently, previously compressed code makes this detection inefficient.

2.5 Evolutionary Computing

Evolutionary Computing (EC) is inspired by Darwinian principles of natural selection and provides highly time efficient, flexible, adaptive solutions for various problems, such as optimization, modeling, simulation among others. Significant efforts have been made over the past decades by research communities towards advancing EC. The simulation of the evolutionary process on computers can be done seamlessly, speedily and effectively

[48]. However, one major area of concern in EC is the interpretation of experimental results, which is heavily dependent on the simulation medium used and the fidelity of the computing models employed [48]. The following subsections rewind the road map of EC.

2.5.1 Evolutionary Computing Problem Areas

EC has found its real-world applications in various types of problems including optimization, modeling, simulation and search. Based on the black box theory in computing [31] (which assumes that a computer is a system requiring and processing an input, then producing an output), these problems are defined formally as follows [48].

- **Optimization Problem**: In such a case, the model and expected outcome of a problem are known. EC is tasked with finding the sets of inputs that result in the expected outcome. The traveling salesman problem [71], is a typical example, in which the salesman aims to find the shortest journey to visit each city only once. With the expected output in mind, one needs to figure out the candidate inputs that satisfy the given requirements. The optimization problem can be formulated as follows. Given a function p in a set E of real numbers with a member y_0. The problem, $p(y_0) <= p(y)$ *or* $p(y_0) >= p(y)$, is optimized for all y in E. In optimization problems, the terms, objective function and constraints, are often referred in its description. An objective function is a numeric representation of the weights of a solution, while constraints refer to conditions that must be met to provide an optimal solution. As shown in Fig. 2.8, problems that have both constraints and objective function are called Constrained Optimization Problems (COP); those that have constraints but no objective function are Constraint Satisfaction Problems (CSP); and those with no constraints but an object function are referred as Free Optimization Problems (FOP). The optimisation problem is also illustrated in Fig. 2.9.

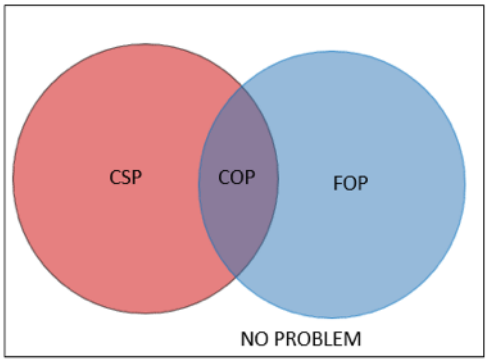


Fig. 2.8 Optimization Problem with constraints and Objective Function

- **Modeling Problem**: In this type of problem, the inputs and expected outputs are given, while the process involved in the transformation of the inputs to the outputs is unknown. This can be seen in Fig. 2.9. A typical example is the modeling of stock exchange, where the inputs are known metrics (such as the conditions of the economy and society) and the expected output is the Dow Jones index [10] for instance. The problem here is to derive a formula that relates the metrics to the Dow Jones index.

- **Simulation Problem**: One is required to generate the corresponding output from a given model and input(s) in this type of problem, in which an output can always be generated from any set of given input(s) as shown in Fig. 2.9. An example is a system to predict a user's behaviour given specific conditions, where the inputs include a series of actions taken by the user and using a sociological or psychological model the behaviour (output) can be determined.

- **Search Problem**: In the aforementioned problems, it is assumed that a computational model solves a problem from the input to the output and this is not reversible. However, the solution to a problem can be derived from different outputs. The set of inputs to a given model forms a search space and these inputs could exist for all kinds of problems, namely optimization, modeling and simulation problems.

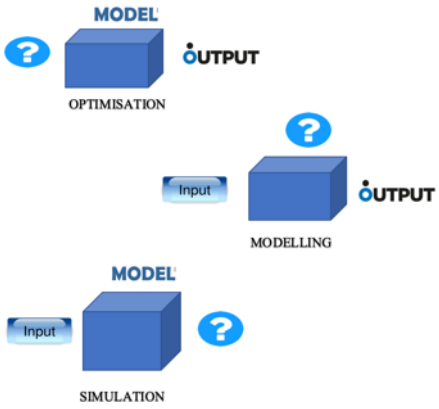


Fig. 2.9 Optimization, modeling and simulation using computing black box model

2.5.2 Components of an Evolutionary Computing Algorithm

A simple illustration of an evolutionary algorithm is given in this subsection. Important components of evolutionary algorithms are as follows [48].

- **Definition of Individuals** involves establishing a relationship between the initial problem setting and the context of the problem to be solved in which evolution

Algorithm 1 Evolutionary Algorithm [48]

1: Initialize the population
2: **while** Termination condition is not met **do**
3: Select parents
4: Vary parents either using crossover, mutation or replacement operators
5: Evaluate the fitness of the parents
6: Based on the evaluation, select new parents
7: **end while**
8: **return** The fittest solutions

occurs. In this stage, a definition is made with regards to the link between phenotypes (predicted solutions that emanate from the initial problem context) and genotypes (encodings within the evolutionary algorithm). Given an integer optimisation task, the integers in the problem space represent the phenotypes. If we want to define the integers in their binary form, the binary representation then becomes the genotype. The phenotype 10 which is the integer representation then becomes genotype 1010 in its binary form.

- **Population** is comprised of several sets of genotypes. It contains several definitions of possible solutions. The population is a fixed constant in evolutionary algorithms, as it does not change throughout the evolutionary process.

- **Fitness Function** defines the conditions that must be adapted by the population. It is used in the selection process. In a problem space, a fitness function takes a solution as input and produces the optimality of the solution as output. The fitness of the individuals is calculated before new parents are selected as seen in step 5 of Algorithm 1.

- **Parent Selection Mechanism** is the process involved in selecting individuals and is done based on the quality of the individuals. The individuals that are selected are called parents. They are recombined using the variation operators discussed below, to produce children. The selection of parents is probabilistic. Consequently, fitter individuals are more likely to be selected as parents. This process is essential to ensure that better quality solutions are produced. This is done in step 3 of Algorithm 1.

- **Variation (Recombination and Mutation) Operators** are the operators employed in generating new individuals from old individuals. One example of such an operator is the mutation operator. The mutation operator is used to trigger a random and unbiased change in the solution. This allows for a bit of variation in the solutions produced. Another common variation operator is the crossover operator. This

operator generates one or two children genotypes by combining information from two parent genotypes. Crossover aims to produce better quality off springs by merging parents with highly desired qualities as seen in step 4 of Algorithm 1.

- **Survivor Selection Mechanism** is similar to selection which aims at separating individuals on the basis on their quality. It involves replacing bad quality solutions with better quality solutions. This replacement is usually based on either the fitness of the offspring or age of the offspring.

The evolutionary process is illustrated in Fig. 2.10. It starts with an initial population of individuals which have a given representation as explained previously. Then, the individuals are varied using either mutation, crossover or replacement operators to produce new ones. Furthermore, the new individuals are evaluated using a fitness function to determine their quality. Then, the fitter individuals are selected to form a new population, and the process is repeated until a termination condition is met. This termination condition can be met for instance, when the best solution is found or when a maximum number of generations is reached.

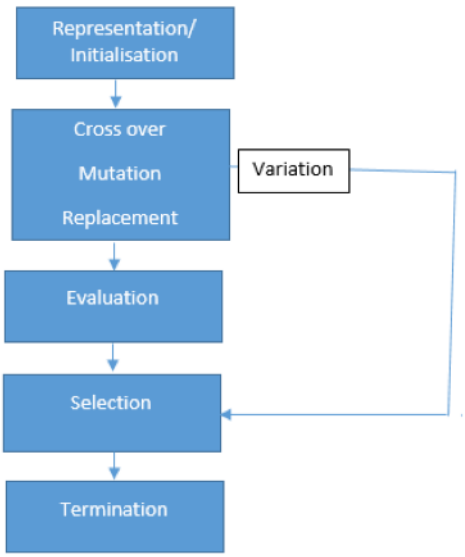


Fig. 2.10 Evolutionary Process

2.5.3 Popular Evolutionary Computing Algorithms

This subsection presents a concise introduction to two popular EC algorithms used in cybersecurity research. They are discussed in detail below.

Genetic Algorithm (GA)

Genetic algorithms [70] are one of the most commonly used algorithm families in EC. They are often applied to solve optimization problems. Uniquely, GAs can solve a given problem without requiring any additional information of the problem itself, which facilitates GAs outperforming most search algorithms [55] and other optimization techniques [109].

A GA has all the components of EC, including selection, population of chromosomes, and fitness function among others. In GAs, the population of chromosomes are usually represented as bit strings. The bit strings take the values of 0s and 1s. A GA uses variation operators, such as mutation and crossover operators, to cause a level of change in the current population. For example, assuming that we have a string 10000110, it can be mutated at its fourth position to give a new string 10010110.

Similarly, applying a crossover operator on two distinct strings, such as 10001111 and 11111000, to swap the sub-strings after the fifth positions of both strings produces offspring 10001000 and 1111111. In other words, the GA population evolves by replacing the current population with another population. Then, the GA uses a fitness function to determine which chromosomes in the present population will survive. The fitness function provides a score to every chromosome in the population. The score explains how fit the chromosome is in solving the prevalent problem [109].

The flowchart shown in Fig. 2.11 describes the process undertaken by a GA. As can be seen in Fig. 2.11, GAs start with a group of candidate solutions which form the population. Each of the individuals in the population, represent a possible solution to the problem at hand. The fitness of each individual is then calculated and the population is sorted according to the fitness of the individuals. The top individuals which are those with better fitness values are then selected for mutation or crossover, and a new population emerges. This process is repeated for the new population until a stopping criteria is reached.

Genetic Programming (GP)

Genetic Programming is another family of EC algorithms. They are often considered a variant of GA [69]. They consist of a population of individuals, that are evolved using genetic operators such as selection, mutation and crossover operators. Similar to other EC algorithms, high quality individuals are selected using a fitness function, to ensure the survival of these individuals in future generations. GP finds its applications in a number of areas such as feature selection and construction, classification problems among others, as they can be used to evolve sets of rules, knowledge abstractions, mathematical expressions among others [49]. GP usually takes different forms to evolve computer programs. Originally, GPs took the form of trees written in List Processing (LISP) programming language [87], [116]. More recently, GPs represent the individuals as

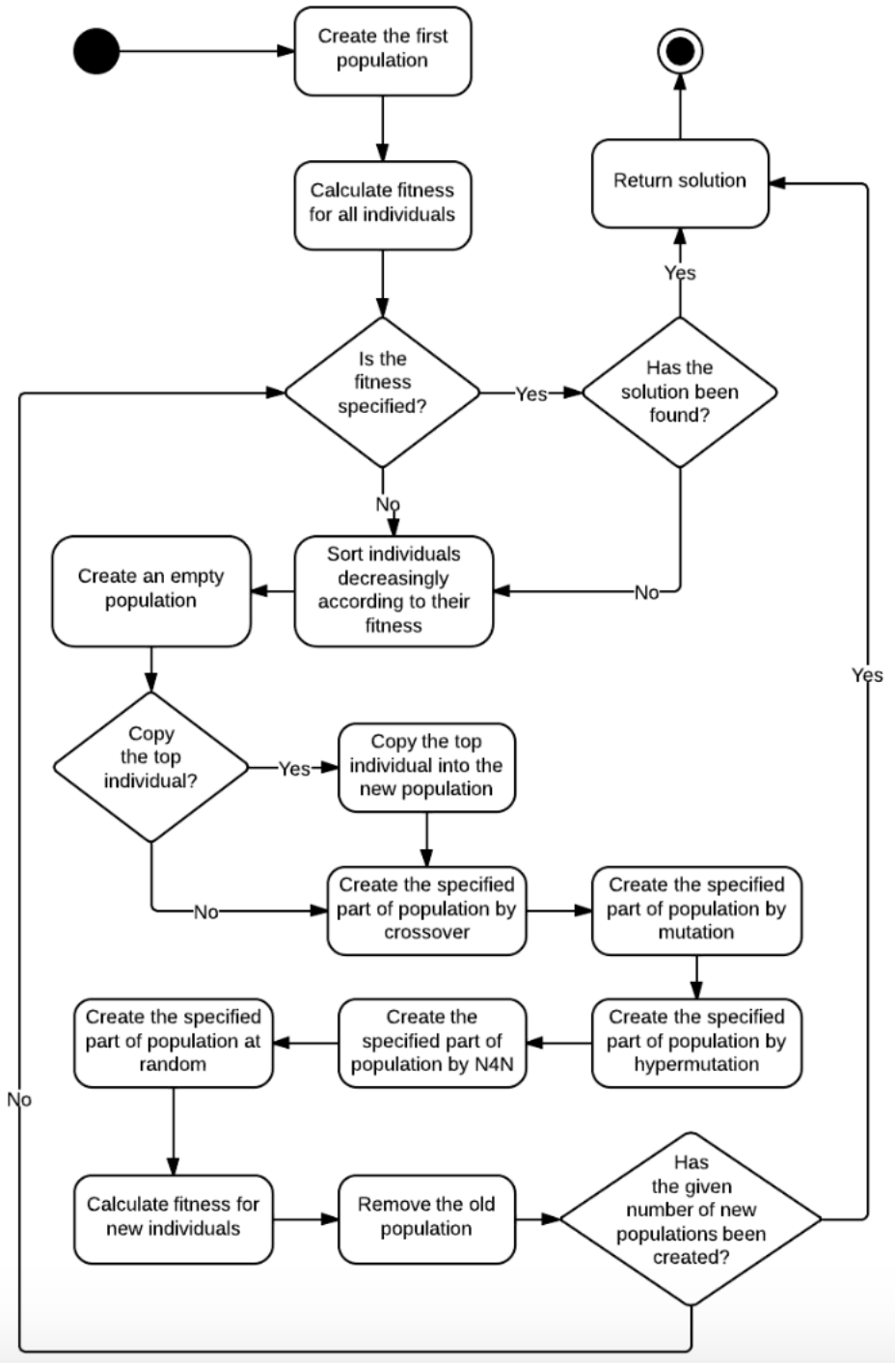


Fig. 2.11 A Genetic Algorithm [113]

either linear sequences [28] or binary sequences [20] of instructions. Cartesian GPs use a set of whole numbers to present the genotypes which are mapped to a Cartesian graph [108]. Parallel distributed GPs are used in computing applications with a high degree of parallelism [121].

2.5.4 Evolutionary based Malware Detection

The idea of using EA based techniques for malware analysis and detection is not a new concept. For example, [106] use EAs to improve classifier selection and performance for malware detection. In [78], the authors use GP to evolve variants of a buffer-overflow attack with the objective of providing better detectors, showing that GP could effectively evolve programs that "hid" malicious code, evading detection by Snort in 2011 instances. Here, the focus was on buffer-overflow attacks rather than metamorphic malware.

Using EAs to generate Test Data (Malware Variants) to improve Malware Detection

EAs like GP have been used by authors like [153] in improving program codes. This includes how to boost up the non-functional features of software that already exist so that they work together with the compiler and target platform simulation. Other authors like [57] have shown how GP can be employed in bug fixing in software. Authors like [90] and [37] also show how GP can be used to optimize the performance of existing software or code. All of the aforementioned examples show how EAs have been used in modifying and improving software code.

There are also examples of research work that use EAs to create test data, which are also malicious code variants to be used to improve the detection of metamorphic malware. In [15], the authors used GP to create new mutant samples, applying their approach to Android based metamorphic malware. New malware samples were created using mutation techniques, and evaluated on their ability to evade detection by eight antivirus systems. Similarly, [155] creates metamorphic pdf malware using GP. They also test the instances of the evolved malware to determine if they evade detection by pdf detectors. The mutants were generated by employing mutation operators like junk code insertion, code reordering, to mention a few for [15], and operators like deletion, insertion and replacement for [155]. All of the above approaches generate new malware that can be used to train malware detectors in order to achieve greater detection rates by the detectors.

As stated above, [155] focus on pdf malware. Although [15] focus on metamorphic malware, they only consider 8 anti-virus engines (Eset, GData, Ikarus, Kaspersky, Avast, TrendMicro, BitDefender and Norton) when evaluating whether their evolved malware is able to evade detection. It is also unclear whether they test whether the new mutants that are able to evade detection are still malicious. The fitness function that guides evolution

scores each solution with a discrete value between 0 and 8, depending on how many engines it evades. This provides very little information to guide the evolutionary algorithm through the search-space to find evasive solutions.

Quality diversity Algorithms

Quality diversity algorithms are a form of EA. However, while EAs tend to converge to a single quality solution, quality diversity algorithms attempt to maintain both diversity and solution quality. They generally provide ways of finding good solutions in a highly dimensional search space [39]. These algorithms aim to improve both the diversity as well as the performance of solutions. In some instances, the generation of diverse solutions can be beneficial as they can guide the search towards high quality solutions in the search space. Given various features of a solution represented as the behaviour space of the solution, a quality diversity algorithm seeks to find several diverse behavioural spaces of that solution while ensuring that each behavioural space has a solution with best performance for the given behavioural space [122].

Two commonly referenced quality diversity algorithms in literature are Multi- dimensional Archive of Phenotypic elites (MAP-Elites) [111] and Novelty Search with Local Competition (NSLC) [94]. While MAP-Elites discovers high quality instances with varying behavioural characteristics within separate cells in a behavioural grid, novelty search with local competition combines local fitness competitions and novelty search for solutions that have similar features. This results in a wide range of local competitions happening concurrently in behavioural spaces that are very diverse [122].

MAP-Elites

MAP-Elites discovers high quality instances with varying behavioural characteristics (defined by the user) within separate cells in a behavioural grid [111]. Given various features of a solution represented as the behaviour space of the solution, the algorithm seeks to find several diverse behavioural spaces of that solution while ensuring that each behavioural space has a solution with best performance for the given behavioural space [122]. When using MAP-Elites, a single quality objective also referred to as performance criterion or objective function, used to evaluate the fitness of the solutions, is defined. Features of interest also known as feature descriptors mapped to a discretized grid, are also defined. For each descriptor $< f_1, f_2 >$, MAP-Elites optimises the quality objective associated with the descriptor. Then, multiple solutions are generated, one for each pair of features $< f_1, f_2 >$. Finally, the best solution for each cell, is found.

The algorithm begins with the generation of initial solutions at random. The feature vectors $< f_1, f_2 >$ of the solutions are calculated and mapped to an archive. Then, only the

most elite solution, that is, the best solution found so far based on the objective function, is stored per cell. Then until the algorithm finishes, a random genome is selected. A variation operator is applied to the random genome which is evaluated and placed in the grid only if the cell is empty, or if it is better than the current occupant, else it is discarded. This algorithm taken from [111], is illustrated in Alg. 2.

Algorithm 2 MAP-Elites algorithm [111]

procedure MAP-ELITES ALGORITHM (SIMPLE, DEFAULT VERSION)(I, G)
  ($\mathscr{P} \leftarrow \phi, \mathscr{X} \leftarrow \phi$) ▷ Create an empty, N-dimensional map of elites: solutions $\mathscr{X}$ and their performances $\mathscr{P}$
  for iter $= 1 \rightarrow I$ **do** ▷ Repeat for I iterations
    if iter $> G$ **then** ▷ Initialize by generating G random solutions
      $x' \leftarrow random\_solution()$
    else ▷ All subsequent solutions are generated from elites in the map
      $x \leftarrow random\_selection(\mathscr{X})$ ▷ Randomly select an elite x from the map $\mathscr{X}$
      $x' \leftarrow random\_variation(x)$ ▷ Create x, a randomly modified copy of x (via mutation and/or crossover)
    end if
    $b' \leftarrow feature\_descriptor(x')$ ▷ Simulate the candidate solution x' and record its feature descriptor b'
    $p' \leftarrow performance(x')$ ▷ Record the performance p' of x'
    if $\mathscr{P}(b') = \phi$ or $\mathscr{P}(b') > p'$ **then** ▷ If the appropriate cell is empty or its occupants's performance is $<=$ p', then
      $\mathscr{P}(b') \leftarrow p'$ ▷ store the performance of x' in the map of elites according to its feature descriptor b'
      $\mathscr{X}(b') \leftarrow x'$ ▷ store the solution x' in the map of elites according to its feature descriptor b'
    end if
  end for
  return feature-performance map ($\mathscr{P}$ and $\mathscr{X}$)
end procedure

The use of MAP-Elites algorithm as a quality diversity algorithm started with [111], who first introduced it as an algorithm for the illumination of search spaces. They applied the algorithm to three different problem spaces namely, the production of modular neural networks as well as the design of real and simulated soft robots. They showed the

algorithm's efficiency in describing the close links between performance and interesting features in the search space. They also show how MAP-Elites helps in creating diverse and high quality solutions.

Following their work, the algorithm has found application in other domains such as video games, with the introduction of MAP-Elites with Sliding Boundaries (MESB) [56], an extension of the MAP-Elites algorithm which was applied to a card game comprising of several features of interest all in multiple dimensions. The experimental work done in [56] shows MAP-Elites ability to discover varying and diverse styles of playing the game. Other authors like [77] explore ways to improve the quality of solutions produced by MAP-Elites in noisy settings mainly through adaptive sampling by slowly yet incrementally choosing the samples employed in the assessment of the performance of solutions.

These quality diversity algorithms have not been used in the area of malware analysis and it is clear that they hold a lot of promise in providing diverse variants of malware and giving information as to highly performing areas of the search space, depending on the features of interest. Unlike a standard EA, they do not converge to a single solution but maintain an archive of diverse solutions. These diverse solutions i.e. diverse malicious variants can be used to train machine learning models leading to higher detection rates.

2.6 Summary

Malicious code of varying forms still continue to pose serious security threats to information assets particularly malware such as metamorphic malware that stochastically changes its form between generations. In this chapter of the thesis, an extensive study has been carried out on this class of malware, their creation process, mutation techniques as well as their layers of mutation. A review of existing literature in metamorphic malware analysis and detection including signature based, heuristic and malware normalisation and similarity based detection approaches, has also been conducted.

Furthermore, some of the challenges to current solutions to metamorphic malware with one of the main points being the lack of training data for ML models to learn from, hence impeding model generality, have been highlighted. Also, an analysis of the use of evolutionary based malware detection, particularly how they have been used to create training data to improve malware detection, has been made. In addition, a review of quality diversity algorithms as an EA technique for creating diverse data has been done.

In this thesis, EAs have been used to create data (mutant variants of malware) to augment data-sets with new training data in order to improve the classification of metamorphic malware. Specifically, this thesis does this by

- Using two types of EAs (a standard mutation only EA and a QD algorithm — MAP-Elites) to create mutant variants of malware that are diverse and are more evasive than their parent malware.
- Improving ML models by augmenting training data with the diverse set of evolved mutant samples created using the two EAs.

The highlighted points are discussed in much more detail in the following chapters of the thesis by answering the research questions raised in Chapter 1 of the thesis.

Chapter 3

Research Framework

3.1 Metamorphic Malware Detection Framework

This section proposes a framework of metamorphic malware detection for mobile malware, in order to address some of the challenges raised in Chapter 2 of this thesis. Recall that the challenges involved in the detection of metamorphic malware emanate from the fact that they continuously change their form between generations and so, it is difficult to know where these malware will morph to. Hence, in this research work, the aim is to address this using a framework that utilises an EA to create evasive and diverse mutant variants of malware, then improves the classification of metamorphic malware using these large and diverse set of mutants. The framework is comprised of five functional modules, namely, a data source (i.e., a mobile malware dump), a disassembly tool (i.e., apktool), a mutation engine, a data store for APK variants, and a malware detector. The conceptual overview of the proposed framework is shown in Fig. 3.1.

The framework includes a data source where the mobile malware is collected from. Then, it uses a disassembly tool to disassemble the mobile malware from APK to smali. The smali files can then be fed to the mutation engine module. This module generates novel malware mutants, representing potential future variants of existing malware. The new mutants are then stored in a data store. The data stored in the data store module is then used to train a detection module that offers improved protection against future mutants. If the system is used in an adversarial context, then improvements in the detection module drive the generation of more diverse mutants, hence, driving further improvement in the detection system.

The malware generation module was implemented using the two EAs introduced in Chapters 4 and 5, respectively. EA is a well-known technique for its ability to search vast spaces of potential solutions [58]; it is used to efficiently search for unseen mutants

that represent potential states that existing malware might morph into, thereby providing improved training data for a detection module.

The mutation engine module is further described in Fig. 3.2, whose EA component is where the mutant samples are generated, optimised to minimise detection rate, structural and behavioural similarity to the original malware (this section only describes the framework and tools involved in a nutshell. The details of the EA are given in Sections 4.2 and 5.2.1). The best APKs generated are stored in a data store after they have undergone a maliciousness test to see that they retain their maliciousness. Both variants that retain their maliciousness and those that do not, are saved as they both serve as training data for machine learning models.

The detector then employs machine-learning using an appropriate learner in detecting the generated mutants. The machine-learning based detector receives the newly created mutants from the data store as training data.

3.2 Why Mobile Malware?

Mobile platforms are becoming main targets of malicious attacks. A number of malware writers are gearing their efforts towards the creation of mobile malware. This is because

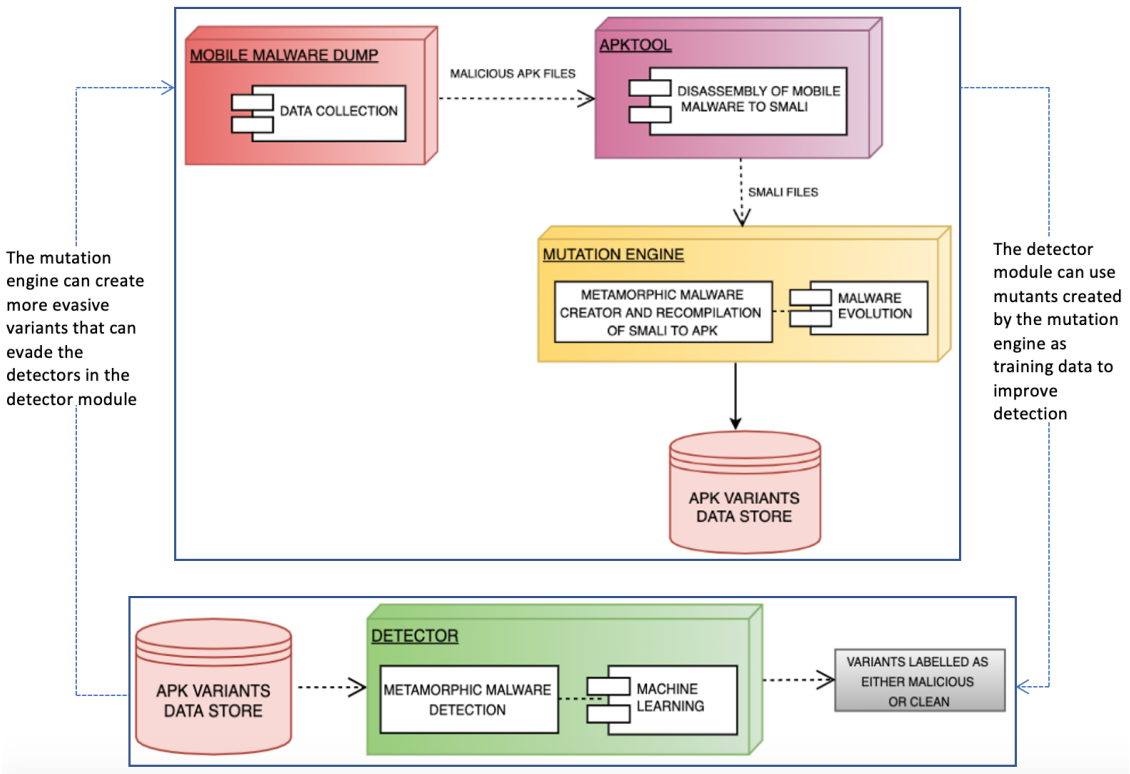


Fig. 3.1 A conceptual overview of our detection framework

there is an increased usage of mobile devices due to their portability, among other reasons. As most of these mobile devices run an Android Operating System (OS), this mobile computing platform is more susceptible to be infected by malicious software. Statistical reports [50] indicate that nearly 100% new malicious programs discovered in recent years emanate from Android platforms.

Android platforms represent an easy target for malicious attacks partially due to a reason that Android platforms are open source [103]. Consequently, they are freely accessible and are subject to easy modification. Writers of malicious software can therefore, extensively study the Android OS and discover an attack surface.

Despite the fact that Google has some defending mechanisms to protect its Android smartphone users from being compromised by malware, these mechanisms have limited functionalities. Google Play, for instance, is a well known official Android market. This market provides a platform in which Android applications can be downloaded. Google play has a tool for detecting malicious software termed Google Bouncer. However, several malware are able to go undetected by the detection tool. Google also has an application signing tool which is a security tool intended to be used in signing applications when

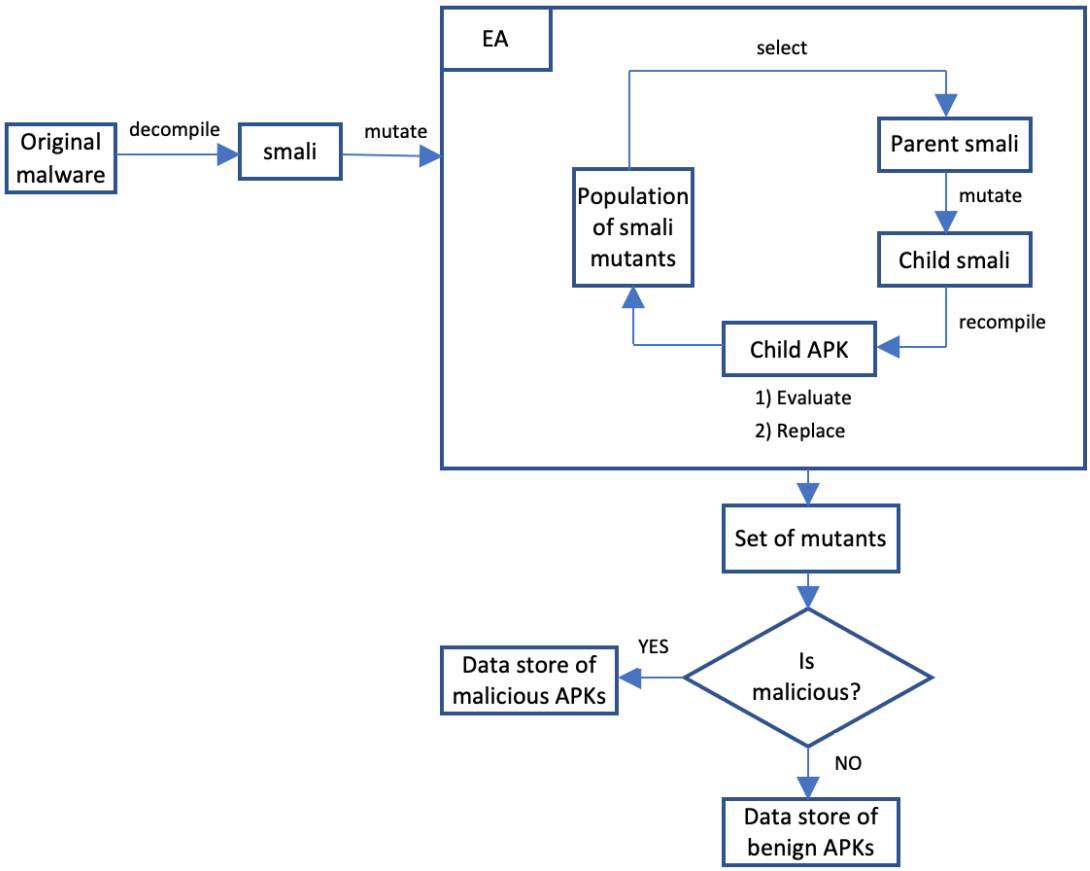


Fig. 3.2 The Mutation Engine module that uses an EA

they are uploaded or installed. However, because the digital signatures derived need to be managed effectively, and Google Play Store struggles with the signature management, this security measure is not really efficient.

Other Android markets are available for downloading applications. Some examples include the application stores from Amazon, Samsung, among others. Some of these alternatives are limited because the user does not have as much control when it comes to the application security. Attackers therefore, leverage on this to detect loopholes in these alternatives.

3.3 What an Android Malware Looks Like?

The mutant samples of malware created in this research are mobile malware. These malware are infected Android binaries. Android applications are archived as apk files, and they consist of the following files as illustrated in Fig. 3.3 [59].

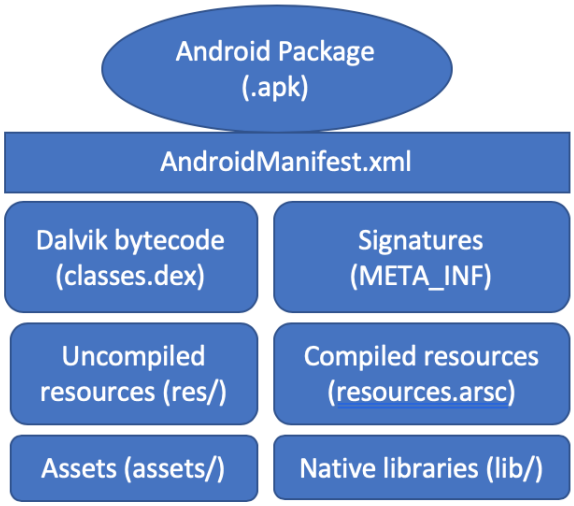


Fig. 3.3 Structure of an APK

- AndroidManifest.xml file - All the essential components of an Android application are contained in the AndroidManifest.xml file. It consists of all the classes required to execute the application. The permissions used and actions utilized for activating every specific Android component, is included in the AndroidManifest.xml file.

  The AndroidManifest.xml file is initially accessed at the commencement of an application launch because it contains every entry-point class needed by the application. The file is also used to read all the permissions used to run the application. Without this file, the application will fail to function as it should.

- Dalvik bytecode (classes.dex file) - The classes.dex file is a dalvik virtual machine executable obtained while compiling the .java files which consists of all the classes used by the application, or used while converting the apk files from java virtual machine into dalvik byte code. In order to aid the readability of the classes.dex file, they can be easily converted to smali file using tools such as apktool. This file is built to optimize mobile apps. The smali file is used for our analysis as they can be easily modified by the EA. The structure of a smali file is illustrated in Fig.3.4. It comprises instance fields and methods. The methods in a smali file comprise of instructions which include direct invocations as seen in Fig.3.4. The classes.dex file

```
.class public Ljp/oomosirodougamatome/MainActivity;
.super Landroid/app/Activity;


# instance fields
.field a:Ljava/lang/String;

.field b:Ljava/lang/String;

.field c:Ljava/lang/String;

.field final d:Ljava/lang/String;

.field private e:Ljava/lang/String;

.field private f:Ljava/lang/String;

.field private g:Ljava/lang/String;


# direct methods
.method public constructor <init>()V
    .locals 1

    invoke-direct {p0}, Landroid/app/Activity;-><init>()Via

    const-string v0, "http://depot.bulks.jp/get41.php"

    iput-object v0, p0, Ljp/oomosirodougamatome/MainActivity;->d:Ljava/lang/String;
```

Fig. 3.4 Structure of a Smali file

comprises a header, string IDs, type IDs, proto IDs, field IDs, method IDs, class definitions, data and debug section. The header is a byte stream with a file checksum number, 20 bytes SHA-1 hash of the whole .dex file excluding the magic number, the file checksum, and the hash. The classes.dex also comprises header and file size, an endian tag, data segments that include specific addresses, the sizes and positions of the different IDs consecutively.

The data segment (where related strings are ordered and stored) is represented by the string IDs. References to parallel string types are contained in the type ID. The proto IDs represent the method prototype. Information about the class' fields is

given by the field ID. Method information is defined by the method ID. All the arguments used by a given class are represented by the class definition.

The most important segment of a .dex file is the data section and it has both class data and code item. The class data item is the foremost part and contains data on the size of static and instance fields, offsets and the virtual and direct methods' size with annotations. The code item consists of the bytecode of all the class methods. All important information about the name of variables, line numbers and source file data, is contained in the debug section.

- Signatures (META_INF) - This directory contains the MANIFEST.MF file which holds the APK metadata. The folder also stores the APK signature.
- Resources folder - All resources are stored in either the "res" directory or the resources.arsc file. The resources folder consists of all the .xml files required in designing layouts, menus, dialogs, and lots more. While the res/ directory contains the uncompiled resources, the resources.arsc file contains the compiled resources.
- Assets folder - Assets refer to external resources, such as audio/video files and images, which are needed for the execution of an application. These files are stored in the assets folder that may contain infected binaries too.
- Native libraries (lib) - This folder contains all native libraries that have been compiled and are employed by the application. This directory has several sub-directories, each corresponding to the CPU architecture supported by the application.

3.4 Analysis and Reverse Engineering of Android APK Files

In order to analyse the Android files, the following tools were employed:

- AV engine: Virustotal [149] which is an online based malware detection service that contains 63 updated antivirus engines, is used to check the evasiveness of the mutants. It was selected because it is open source and comprises of a large number of detection engines that are periodically updated. Virustotal returns the percentage of the 63 engines that can detect the malicious mutants.
- Collecting the behavioural trace: Strace [95] is used to monitor the behaviour of the variants by keeping track of their system calls. It is a well known tool for studying the behaviour of apk files. The variants' main activity is executed using

MonkeyRunner [12] in order to simulate user interaction. This allows us to capture the execution behaviours of the apk variants at runtime. The output of Strace is a file comprised of the variant's process ID, along with its system calls and parameters. Thus, the file is transformed to a fixed sized vector whose elements correspond to the frequency of the system call used. Since 251 systems calls are considered, the vector consists of 251 elements.

- Maliciousness and Executability: To assess the executability of a mutated APK file, tests against its compilation and execution are conducted after mutation. The tools that are used for the assessment of the executability of the mutated APK files are listed as follows:
  - To check for the compilation of an APK file: apktool [2], apksigner [1] and zipalign [6]
  - To check that the file runs: Android emulator [3].

  Android emulator [3] was employed because it allows for the simulation of Android devices on a computer system for testing applications without the requirement of a physical device. Android studio [4] is specifically employed for this purpose and was selected due to its numerous advantages. These include quicker coding with its intelligent code editor and great emulator. It also provides various templates, testing tools and frameworks among other benefits [5]. Then apksigner and zipalign tools are used which are Android studio's build and alignment tools for apk files [1], [6]. As a result of its documented success in reverse engineering apk files, apktool is employed in reverse engineering the mutated apk files [14].

  The functions that check to see that the APK variants run and compile properly are wrapped in a bash script. Finally, in order to ensure that the evolved variants retain their maliciousness, Droidbox [138] is used. Droidbox is a sandbox designed for Android that can be used in the dynamic analysis of apk files. The dynamic analysis involves studying the behaviour of the apk while running it. Once an apk is submitted to Droidbox, it executes the apk and collects information relating to the files created, deleted and downloaded by the apk, connections opened, traces of API calls, among others.

The aforementioned analysis tools capture all the software and tools employed in this research with the machine learning modules explained in Section 3.6 of this chapter. In terms of hardware employed, all experiments were carried out on a VMware Workstation running Ubuntu 16.04 LTS on an Intel Xeon CPU. This research work employs PyCharm Integrated Development Environment (IDE) and uses a number of programming languages which include Python, Bash, C and Perl.

3.5 Similarity Measurement

Both text based and behaviour based similarity between the original malware and its variants, is measured. The text based similarity metrics also include source code similarity metrics.

3.5.1 Text based Similarity Metrics

In measuring the text based similarity between the original malware families and their variants, the following metrics are used:

- Cosine similarity: This measures the cosine angle between two non-zero vectors.
- Levenshtein: This similarity metric seeks to find the number of deletions, insertions, or substitutions needed to transform a file A to a file B [67].
- FuzzyWuzzy: This measure does string matching to find strings that match a specific pattern approximately [43].

The source-code level similarity between the original malware and the variants, is also measured using the following similarity metrics:

- Jplag: This measures the similarity between two source codes by first converting the source codes to tokens, and then applying an algorithm to deduce the matches between their token lists from the biggest to the least matches [66].
- Sherlock: It sorts two programs based on their similarity by converting the programs to signatures, and calculating the similarity between them [66].
- Normalized Compression Distance: This measures the distance between two files based on compression. Given the original malware as m and its variants as v, the compression distance is given as [125],

$$NCDz(m,v) = \frac{Z(mv) - minZ(m), Z(v)}{maxZ(m), Z(v)} \tag{3.1}$$

  where Z(m) is the compressed file m's length compressed using a compressor Z. The compressor used in this work is 7-zip.

Each of the similarity metrics produces a value between 0 and 1 where a score of 1 means the original malware and the variant are identical, and 0 means the original malware and the variant are completely different. The average of these metrics is computed, and that represents the structural similarity. These metrics are implemented using various Python and C libraries and were chosen following previous work by [125].

3.5.2 Behaviour based Similarity Metrics

As explained in Section 3.4, the behavioural trace of the mutants, is collected and converted into a fixed sized vector composed of 251 elements. The behavioural based similarity between the original malware and its mutants is then calculated as the cosine similarity between their system call vectors. The cosine similarity metric is explained in Section 3.5.1 of this chapter, and it was chosen following previous work in literature [125].

3.6 Machine Learning Modules

In this research, a number of feature based ML models are employed such as Naive Bayes (NB), among others, implemented using Scikit-learn[1] libraries for Python, an easy to use efficient open source ML tool for predictive data analysis. It is built on NumPy[2], SciPy[3], and Matplotlib[4]. Keras library[5] implementation in python is also used for the sequence based machine learning. Keras is a deep learning library built on TensorFlow 2.0[6]. It is also an easy-to-use framework that is well developed and scalable to large GPU clusters. A detailed implementation of the ML modules is explained in Section 6.3 of this thesis.

3.7 Data Description

The malware utilised in this work are from the Contagio Minidump [147] and MalGenome [158]. Contagio Minidump comprises 237 malware samples collected between December 2011 and March 2013. MalGenome on the other hand, consists of 1,260 Android samples from 49 different families. The selection of samples from these dumps was done based on their malicious payload. In categorizing the samples, four groups are used, categorized by [157], representing the groups most malware fall into. These include privilege escalation, remote control, financial charges, and personal information stealing. The parent malware used to initialise the EA were randomly selected from the dump using the aforementioned malicious payload. The three specific malicious families utilised in this research include Dougalek (personal information stealer), Droidkungfu (privilege escalator and remote controller) and GGtracker (financial charger and personal information stealer).

1. Privilege Escalation: The complexity of the Android platform, owing to the fact that it comprises both Linux kernel and Android framework which have over 90

[1]Scikit-learn - https://scikit-learn.org/stable/
[2]Numpy - https://numpy.org/
[3]SciPy - https://www.scipy.org/
[4]Matplotlib - https://matplotlib.org/
[5]Keras - https://keras.io/
[6]TensorFlow 2.0 - https://www.tensorflow.org/install

libraries, makes it prone to attacks in the form of privilege escalation. Droidkungfu [52], an Android malware family first discovered in May 2011, is an example of a malware family that uses privilege escalation. It is one of the families gotten from the MalGenome dump. It uses encryption to obfuscate its code in order to go undetected by detectors. It includes encrypted root exploits and malicious payloads that are in touch with C&C servers, from which they get instructions to be executed. This family of malware is considered in our analysis.

2. Remote control: This feature allows mobile malicious attackers gain remote control of the phone. Malware families that have this functionality are in communication with remote C&C servers. Droidkungfu is also an example of a malware family that uses remote control malicious payload and is considered in our analysis.

3. Financial charges: Some malicious attacks are launched to deliberately extort money from the users infected in form of financial charges. They subscribe users to premium services without proper authorisation, and in most cases the infected parties are unaware of such services. GGtracker [51] is an example of such a family of malware. It is one of the families in the MalGenome dump, and subscribes the infected users to various US premium services without their consent. It is also one of the families of malware studied.

4. Personal information stealing: There are also other malware families whose major goal is to collect information. This information could be on the infected user's account, contact list, text messages, among others. Malware families such as Dougalek [141] from the Contagio minidump and GGtracker, fall into this category and are analysed in our study.

3.8 Summary and Conclusion

In this chapter, the research methodology has been elucidated. A summary of the framework of the research (i.e. highlighting and explaining the main components of the framework which include the mobile malware dump, the disassembly module (apktool), the mutation engine, the data store, and the detector module), has been made. In this project, the malicious samples employed are mobile malware, and the structure of an Android malware is described, with its components explained. A case is also made for the choice of *mobile* malware as the malicious class studied in this research work.

Furthermore, an explanation is made for the analysis and reverse engineering of the Android files used in this work. The various tools used including the AV engine, the tools used in collecting the behavioural trace of the samples as well as the tools for testing the

maliciousness and executability of samples, are explained. Also, the libraries used for measuring the similarity between the original malware and its variants, are described. This includes both text and code level similarity measurement tools.

The final section describes the data source and samples employed in this research. This includes three families, namely, Dougalek, Droidkungfu and GGtracker, and they fall into four categories which comprise those that escalate privilege in mobile phones, those that try to gain remote control of mobile phones, those that incur financial charges to the mobile phone users, and those that attempt to steal the personal information of the mobile phone users. All of the information in this chapter provides the software components, relevant metrics, and analysis details used in all the experimental chapters (i.e. Chapters 4, 5 and 6).

Chapter 4

Using an EA to Evolve New Test Data

The work presented in this chapter of the thesis was published in the proceedings of Dependability in Sensor, Cloud, and Big Data Systems and Applications (DependSys 2019) [16]. A copy of the publication is given in appendix A.A.1.

Antivirus (AV) engines provide a standard security measure to combat malicious attacks. However, as has previously been established in Chapter 2 of the thesis, most engines fail to generalise over metamorphic malware that changes its form over time in unpredictable ways. To combat this, a proposal of an adversarial solution which generates a suite of new malicious mutants of existing malware that are undetectable by many existing AV engines, and can therefore be used to develop new countermeasures, has been made. Specifically, an Evolutionary Algorithm (EA) which we refer to as MAL_EA, is utilised to discover undetectable mutants, guided by three different fitness measures that include the evasiveness of the variants, and their behavioural and structural similarity to the original malware. The experiments conducted in this chapter are done over three classes of malware to evaluate the effectiveness of the MAL_EA variants. Results show that MAL_EA is capable of generating a diverse set of mutants that evade detection by a significantly higher proportion of AV detectors than the original malware. In addition, the resulting sets of new mutants are shown to be diverse in terms of their detection, behavioural and structural signatures.

4.1 Introduction

A number of solutions have been presented over recent years to tackle the problems posed by malware, for example, in the form of viruses or Trojans as extensively discussed in Chapter 2 of the thesis. However, malicious attacks continue to wreak havoc in computer systems and the internet at large. Metamorphic malware in particular, have been

extensively studied because they represent one of the most complex and dangerous groups of malware.

Metamorphic viruses as previously explained, transform their code as they propagate, thus, evading detection by static signature-based virus scanners, and have the potential to lead to a breed of malicious programs that are virtually undetectable statistically [154]. This dangerous group of malware attacks various computing platforms but particularly mobile platforms for reasons previously described.

A recent approach to detecting malware on Android platform is through the use of adversarial learning techniques [99] previously described in Section 2.3.2. This method is designed to challenge systems by deliberately feeding them with malicious inputs (*adversarial samples*) in order to discover loopholes and subsequently improve detection models to make them more robust to attack. However, a challenge in this approach is to create the adversarial samples required for training.

It is not enough just to evolve new evasive variants of malware but that they should also be diverse so that they can represent a wide variety of potential states the malware can morph to. Diversity of the malicious variants can mean a number of things — for example, structural diversity (this means that the variants take different code level and semantic structures as compared to their parent malware) or behavioural diversity (this means that the behavioural characteristics of the variants are as dissimilar as possible from those of the parent malware). Both structural and behavioural diversity of variants are considered in this work. This is because mutant variants of malware can behave similarly but look different as well as look similar but exhibit different behavioural characteristics. Our idea is to use these characteristics to evolve diverse variants (either one at a time - or using all at once).

In this chapter, a method of generating a suite of malicious, novel mutants is introduced, using an Evolutionary Algorithm [48] — a population-based meta-heuristic search process inspired by processes occurring in biological evolution that guides a population to adapt towards a desired goal — in this case, to discover novel malicious variants of existing malware that evade current detection systems. The approach used in this chapter is a precursor to a true adversarial system — it generates new samples. However, in this chapter we do not go on to test whether this actually improves ML models (the testing is done in Chapter 6 of the thesis). Hence, it is a step towards an adversarial system.

4.1.1 Research Questions

The specific research questions addressed in this chapter are listed below:

- Does modifying the fitness function of MAL_EA to include solution characteristics enable us to obtain samples that are evasive but also diverse with respect to the original malware?
- How do the 63 detection engines vary in their ability to detect the new malware variants for each family of malware tested?

4.1.2 Contribution

The key contribution explained in this chapter is introduction of a novel Evolutionary Algorithm, MAL_EA to generate a suite of malware variants that evade existing detection systems, while remaining malicious and executable. Specifically, this chapter demonstrates that while a fitness that directly rewards evasiveness is capable of generating sets of new mutant malware, rewarding behavioural or structural dissimilarity indirectly leads to evasive variants. In addition, it is showed that the variants produced exhibit diversity with respect to their detection signatures, their behavioural signatures, and the structural signatures, when compared to the original malware and to each other. This extends existing work in

- Developing a suite of new fitness functions to guide the evolutionary search for adversarial mutants.
- Application of the technique to a broad range of malware families.
- Including a large set of 63 detection algorithms in the evolutionary process to generate mutants that evade a wide range of detectors.
- Generating a diverse set of mutants that provide rich training data for ML algorithms.

4.2 Evolutionary Algorithm

In this chapter, MAL_EA is used in the generation of mutant variants of malicious code. An EA is used because it has proven ability in transformation, optimisation, and improvement of software code [37], [90] and [153]. This work employs a canonical model of an EA and customises the operators. MAL_EA returns the single best mutant found (according to the fitness function) at the end, so it is run multiple times to get multiple variants. MAL_EA is given in Alg.3 below:

MAL_EA begins with an initial population of random malicious mutants. In a repeating cycle, parents are selected from this population and mutated to create offspring. The specific operators used within MAL_EA are now described:

Algorithm 3 Evolutionary Algorithm — MAL_EA

```
1:  𝒫 ← initalise_random_population()        ▷ created by mutating original malware
2:  evaluate(𝒫)                              ▷ score each malware according to chosen fitness metric
3:  f_w = min(𝒫)                             ▷ worst member of population
4:  while Maximum number of iterations not reached do
5:      p ← select_parent(P)
6:      c ← mutate(p)
7:      f_c ← evaluate(c)
8:      if f_c > f_w then
9:          𝒫 ← 𝒫 ∪ c                        ▷ add child to population
10:         remove_worst()                   ▷ remove least fit mutant
11:         f_w = min(𝒫)
12:     end if
13: end while
14: return 𝒫
```

4.2.1 Initialisation

To create an initial population of mutants, a single mutation operator (as described in the following section) is applied to the *original* malware. The mutation operator is chosen at random and applied exactly once. This process is repeated to create p mutants each time, starting from the original malware. Recall from Chapter 3, that the original malware is an apk file which is reverse engineered using apktool to obtain a smali file. Also recall the executability of the mutants is checked using tools like apktool, apksigner, zipalign and an Android emulator. The maliciousness of the variants is also analysed using Droidbox.

4.2.2 Selection

The selection operator is aimed at biasing the search towards selecting fitter parents to create offspring from them. In this work, a standard operator, tournament selection [53] has been employed. This selection operator has been chosen because it can be easily tuned to vary selection pressure. In tournament selection, the population is sorted and k potential parents are selected at random. The best parent among the k prospective parents is selected to be recombined to form children in the next iteration [133].

4.2.3 Mutation Operators

To create a mutant from a malware m, a mutation operator is applied to the smali file obtained from decompiling m. Three mutation operators are defined; a mutation operator is selected at uniform random to perform the mutation. The description of the three mutation operators is as follows:

- M1: Instruction reordering - this introduces a goto statement that jumps to a label that does nothing.
- M2: Garbage code insertion - inserts junk codes like line numbers that do not affect the functionality of the code.
- M3: Variable swapping - this replaces variables with other valid variables in the program code while ensuring consistency with the variable names.

4.2.4 Fitness Functions

Three different fitness functions are evaluated as methods of guiding the evolutionary search towards novel variants. The fitness metrics are selected based on the characteristics of malware that is, their structure, behaviour and ability to evade a large set of detectors. In order to create adversarial mutants, each of the fitness values is *minimised*.

- *DR* (Detection Rate): proportion of detectors within a specific AV engine that detect the mutant. This returns a value between 0 and 1, with 0 signifying that no detector recognised the mutant. This uses *Virustotal* as described in Section 3.4, which is an online based malware detection service that contains 63 updated antivirus engines to provide the suite of AV engines.
- *BS* (Behavioural Similarity): returns a value between 0 and 1 which compares the behavioural signature of the mutant to the original malware, with 0 indicating maximum *dissimilarity*. This uses *Strace* to monitor the behaviour of the mutants by keeping track of their system calls. The mutants' main activity is executed using *MonkeyRunner* in order to simulate user interaction. The output of Strace is a file comprised of the variant's process ID, along with its system calls and parameters. Thus, the file is transformed to a fixed sized vector whose elements correspond to the frequency of the system call used. Since 251 systems calls are considered, the vector consists of 251 elements. The behavioural similarity between the original malware and its mutants is then calculated as the cosine similarity between their system call vectors.
- *SS* (Structural Similarity): a value between 0 and 1 that measures the semantic similarity between the mutant and original value by comparing lines of code (where 0 represents the most dissimilarity). This uses several similarity metrics including text (such as cosine similarity) and source code (such as JPlag - a plagiarism detector) similarity metrics as described in Section 3.5 of the thesis. Each of the similarity metrics produces a value between 0 and 1 where a score of 1 means the

original malware and the variant are identical, and 0 means the original malware and the variant are completely different. The average of these metrics is computed, and that represents the structural similarity.

If a mutant is deemed non-executable, then it is assigned a fitness value of 1 (i.e. the worst possible fitness), regardless of the measured metric or the fitness function used.

Note that only DR directly rewards mutants for evading detection. Functions BS and SS reward behavioural and structural dissimilarity, which implicitly might lead to mutants that evade detection. The idea of rewarding novelty rather than objective fitness has recently received a great deal of attention in the EA community, with a significant body of work showing that it leads to evolution finding stepping stones that enable very high-quality solutions to be reached [93], [60].

4.3 Experiments

Experiments are conducted on the three chosen families (Dougalek, Droidkungfu and GGtracker) explained in Section 3.7. For each family, MAL_EA is run using each of the three defined fitness functions, resulting in 9 different treatments. Given the stochastic nature of MAL_EA, each treatment is repeated 10 times. The best fitness obtained in each run is recorded.

4.3.1 Evolutionary based Parameters

The parameters used in creating the evasive malware are summarised in the Table 4.1 and they are derived following empirical analysis. The evolutionary program uses uniform mutation with a mutation rate of 1; this value is chosen to ensure that mutation always occurs, given that this is the only variation operator used, as crossover is not used in the experiments. The crossover operator is not used, as it leads to the generation of non-executable variants. MAL_EA uses tournament selection (described in Section 4.2.2 of this chapter) with a tournament size of 5 for a fair level of selection pressure. As a result of the computational cost involved in running the experiments, the number of iterations is limited, and the evolutionary program is run 100 times. The size of the population is then set proportionally to 20.

Furthermore, to justify the use of an EA, the results derived while using MAL_EA is compared against a Random Search (RS): 120 random variants are generated and evaluated. This number represents the total number of points visited in the search space by MAL_EA. Comparison of these results will indicate whether the evolutionary operators are useful in locating high-fitness variants.

Table 4.1 Parameter settings for Evolutionary Algorithm

Parameters	Values
Selection	Tournament Selection, k=5
Population Size	20
Iterations	100

Table 4.2 Count of malicious variants returned from 10 runs of the EA under each of the 3 fitness functions

Fitness Function	Dougalek	Droidkungfu	GGtracker
DR(x)	7	10	9
BS(x)	7	9	8
SS(x)	10	9	7

4.3.2 Influence of Fitness Function on Evasiveness of Evolved Mutants

In this section, the influence of each of the three fitness functions in terms of their ability to locate novel evasive mutants, is investigated.

Following 10 runs using each of the fitness functions in Section 4.2.4, a test of each of the mutants is done, to ensure they are still malicious using Droidbox, as explained in Section 3.4. Note that each run returns the single solution that minimises the fitness function. Consequently, 10 solutions are generated from 10 runs. Table 4.2 shows how many variants retain their maliciousness.

From Table 4.2, it can be observed that a minimum of 7 out of the 10 runs result in malicious variants, regardless of the fitness function used or malware family. MAL_EA finds malicious runs for 100% of the runs using the evasiveness metric (DR(x)) with Droidkungfu, and also in the Dougalek family using the structural similarity metric (SS(x)).

The percentage of detectors that fail to recognise the novel variants over the x runs that are malicious, is plotted. This is repeated for each malware family using each of the fitness functions. In Figs. 4.1, 4.2 and 4.3, the red line shows the percentage detectors that fail to recognise the original malware. Additionally, the performance of the EA in terms of its ability to find evasive variants is compared to a random search for evasive variants.

- Dougalek: It can be noted from the boxplot in Fig. 4.1, that for each of the functions in the fitness function, the resulting variants are more evasive than the original malware. This means a higher percentage of detectors failed to identify the new variants, as compared to the original malware. 40.3% of the detection engines failed

to detect the original malware while for the best evolved variants of Dougalek, 72%, 66.7% and 67.3% of engines failed to detect the new variants created using functions DR(x), BS(x) and SS(x) respectively. It is particularly interesting to note that even when evasive variants are not directly evolved (i.e. using fitness function DR(x)), evasive variants can still be created, as seen from the boxplots of BS(x) and SS(x). It can also be noted that for the Dougalek family, the plots of evasiveness of the newly created variants for both DR(x) and SS(x) are similar, as seen in Fig. 4.1. This implies that even when one is evolving for structurally dissimilar variants, variants are still obtained, that are almost as evasive as when one is evolving for evasive variants.

Furthermore, from Fig. 4.1, it can be observed that the median percentage of engines that failed to detect the new mutants for the Dougalek family for the EA ranges from 62.1% to 69.4% depending on the fitness function employed. However, for a random search (RS(x)) for evasive variants, the median percentage of engines that failed to detect the new mutants is 58.5%. Therefore, it can be said that an improved evasive performance is noticed when MAL_EA is used as compared to using a random search.

- Droidkungfu: Similarly from Fig. 4.2, it can be observed that for each of the functions in the fitness function, the resulting variants are also more evasive than the original malware. It can also be noted that 65% of the detection engines failed to detect the original malware, while for the best evolved variants, 94%, 82.1% and 83% of engines failed to detect the new variants created using functions DR(x), BS(x) and SS(x), respectively. This also shows that even when there is no evolving for evasive variants (DR(x)), evasive variants for functions BS(x) and SS(x) can still be created.

  It can also be seen from Fig. 4.2, that the median percentage of engines that failed to detect the new mutants for the Droidkungfu family for the EA ranges from 73.2% to 87.3% depending on the fitness function employed. However, for a random search (RS(x)) for evasive variants, the median percentage of engines that failed to detect the new mutants is 63.1%. Again, this shows that an improved evasive performance is noticed when MAL_EA is used as compared to using a random search.

- GGtracker: Also in Fig. 4.3, each of the functions in the fitness function produce variants that are more evasive than the original malware. From Fig. 4.3, it is apparent that only 38.3% of the detection engines failed to detect the original malware while for the best evolved variants, 73.3%, 62.1% and 62.1% of engines failed to detect the new variants created using functions DR(x), BS(x) and SS(x) respectively. It can

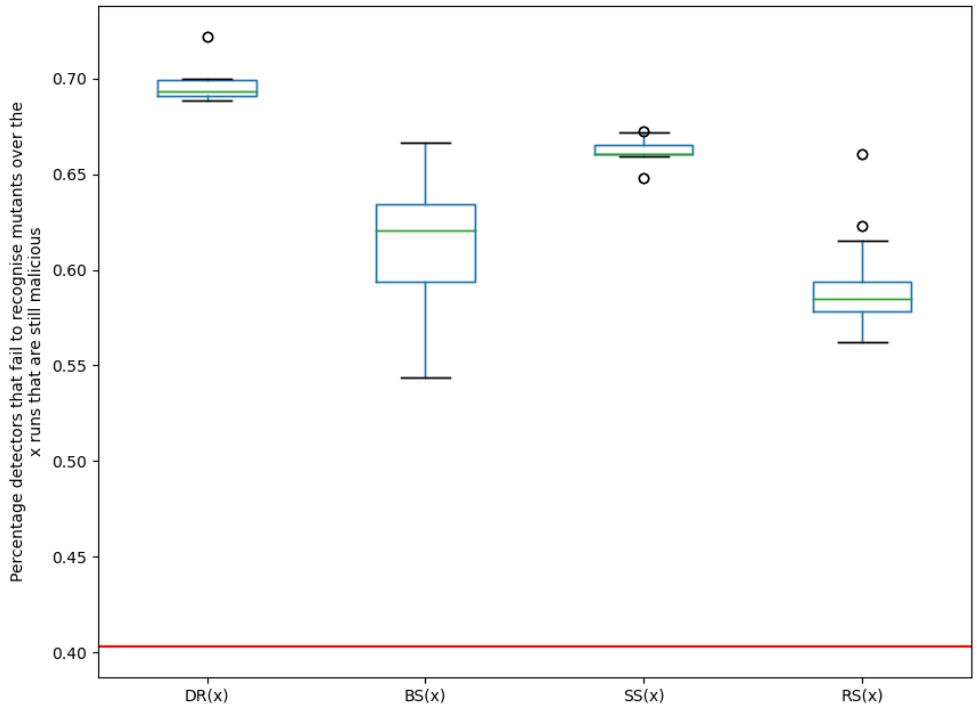


Fig. 4.1 Percentage detectors that fail to recognise the novel variants over the x runs that are still malicious, using each of the fitness functions for Dougalek family, with the red line showing the percentage detectors that fail to recognise the original malware

also be observed from Fig. 4.3, that the plots of evasiveness of the newly created variants for DR(x), BS(x) and SS(x) are also similar. This means that even when evolving for behaviourally and structurally dissimilar variants, variants that are almost as evasive as when we are evolving directly for evasive variants, are still obtained.

From Fig. 4.3, it can be observed that the median percentage of engines that failed to detect the new mutants for the GGtracker family for the EA ranges from 61.4% to 66% depending on the fitness function employed. For the random search (RS(x)) for evasive variants however, the median percentage of engines that failed to detect the new mutants is 61.4%. RS(x) obtains a similar median to BS(x) and SS(x), showing that these two metrics using an evolutionary search do not outperform RS(x). However, Fig. 4.3 indicates that the DR(x) metric used with the EA outperforms RS(x).

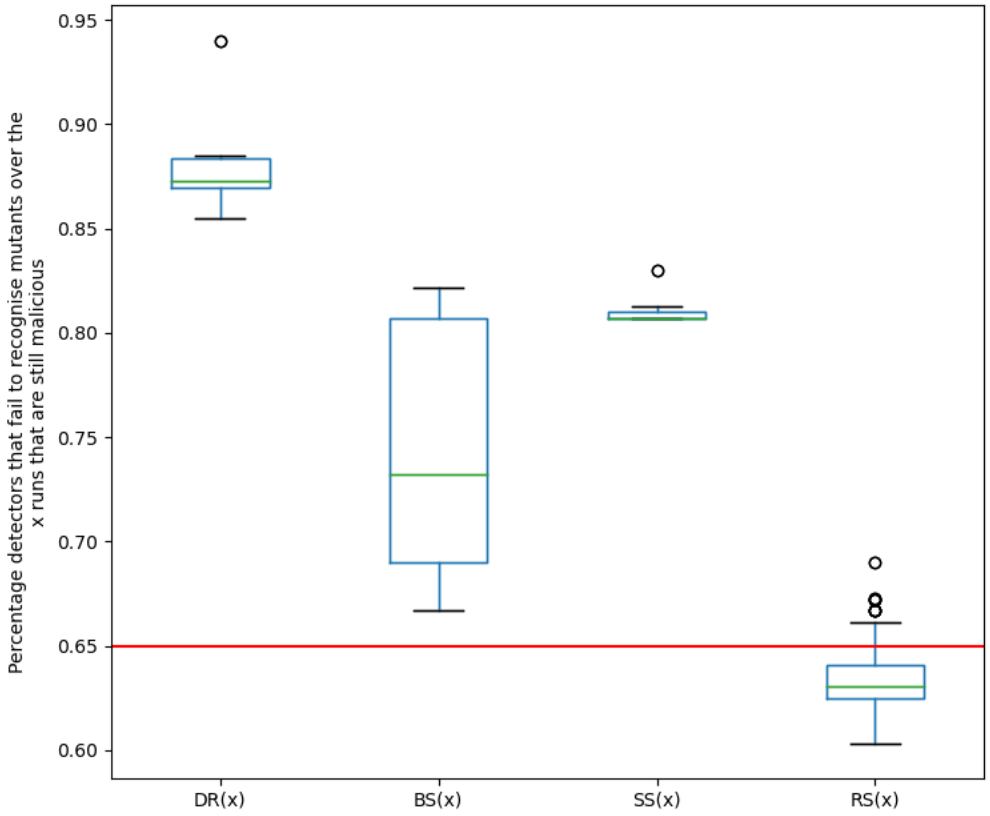


Fig. 4.2 Percentage detectors that fail to recognise the novel variants over the x runs that are still malicious, using each of the fitness functions for Droidkungfu family, with the red line showing the percentage detectors that fail to recognise the original malware

The estimated cost is calculated in terms of time taken for a complete run of 100 iterations, using each of the functions in the fitness function, and this is summarised in Fig. 4.4. The function SS(x) takes the most time; this is explained by the fact that it uses more tools in the similarity matching.

4.3.3 Analysis of Evasion Characteristics of New Mutants

In this section, an analysis of which of the detection engines are fooled by the novel mutants evolved, which were not fooled by the original malware, is conducted. For each of the m malicious variants produced by 10 runs of MAL_EA for a given fitness function, the frequency f which a detector d failed to detect the malware ($0 < f < 10$), is counted. In Figs. 4.5, 4.6 and 4.7, the blue histograms represent results for function DR(x), the

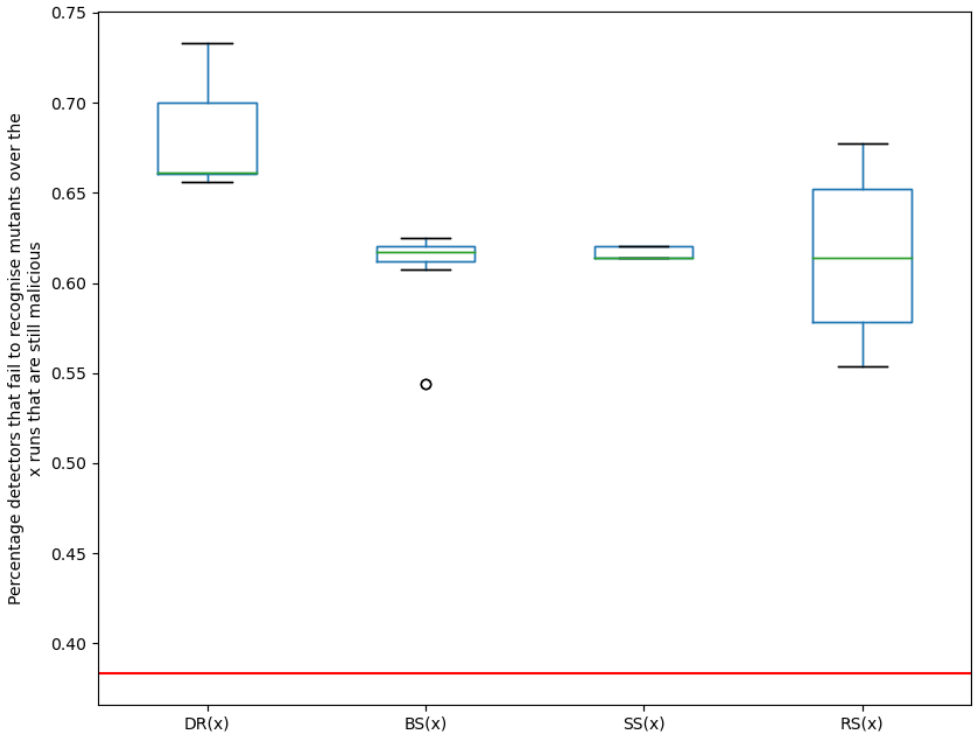


Fig. 4.3 Percentage of detectors that fail to recognise the novel variants over the x runs that are still malicious, using each of the fitness functions for GGtracker family, with the red line showing the percentage detectors that fail to recognise the original malware

orange ones represent results for function BS(x) and the green histograms represent the results for function SS(x).

- Dougalek: The plot in Fig. 4.5, shows that for fitness function DR(x), 14 engines were not fooled by any of the mutants; some of such engines include AVG and Tencent. 17 engines on the other hand, were fooled by all the mutants with examples like AVware and McAfee. The number of engines not fooled by any variant when fitness function BS(x) is used, is 19. Examples include AVG and Fortinet. 11 engines however, were fooled by all the mutants for function BS(x) with examples like GData and McAfee. Also, 16 engines were not fooled by any variants for the function SS(x). Examples include Symantec Mobile and Fortinet. 17 engines however, were fooled 100% of the time for function SS(x). Examples include McAfee-GW and BitDefender.

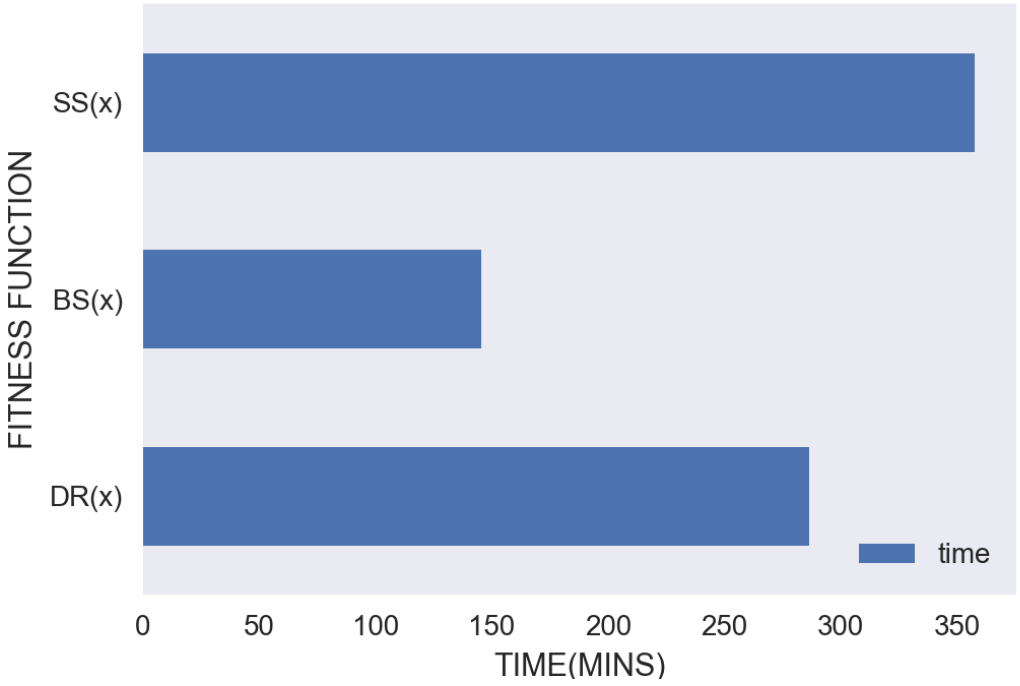


Fig. 4.4 Average time taken for a complete run of 100 iterations, using each of the functions in the fitness function

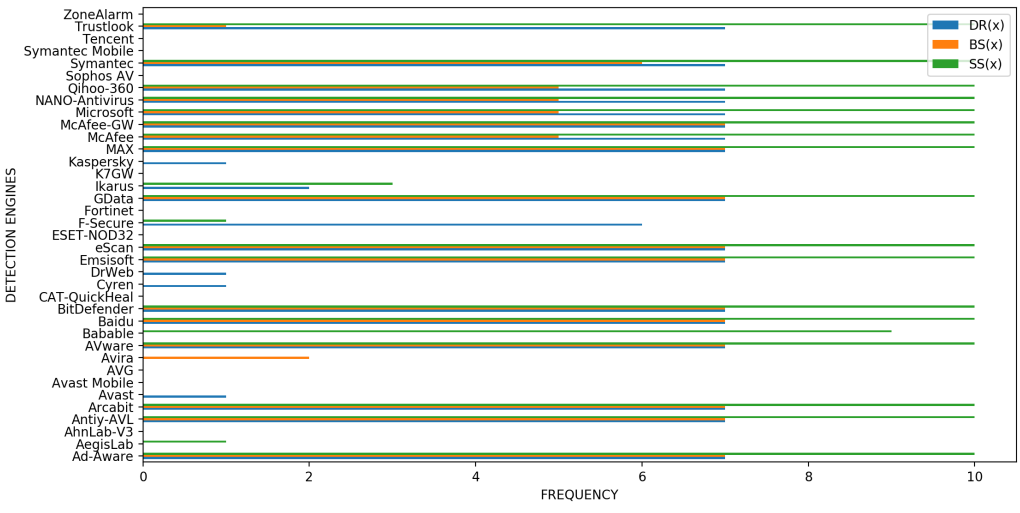


Fig. 4.5 Frequency f of detectors d that failed to detect the malware ($0 < f < 10$) for the fitness functions (DR(x), BS(x) and SS(x)) for Dougalek family

- Droidkungfu: From Fig. 4.6, it is apparent that 3 engines were not fooled by any of the mutants for function DR(x). Examples include Avast Mobile and NANO

Antivirus. 12 engines however, were fooled by all the mutants 100% of the time for function DR(x) - examples include Fortinet and Kaspersky. For fitness function BS(x), 6 engines were not fooled by any of the mutants; some of such engines include Avast Mobile and NANO Antivirus. 7 engines on the other hand, were fooled by all the mutants with examples like Symantec Mobile and Tencent. Furthermore, 7 engines were not fooled by any mutant for function SS(x), with examples such as AVG and Cyren. On the other hand, 11 engines were fooled by all the mutants for function SS(x). Examples include Kaspersky and ZoneAlarm.

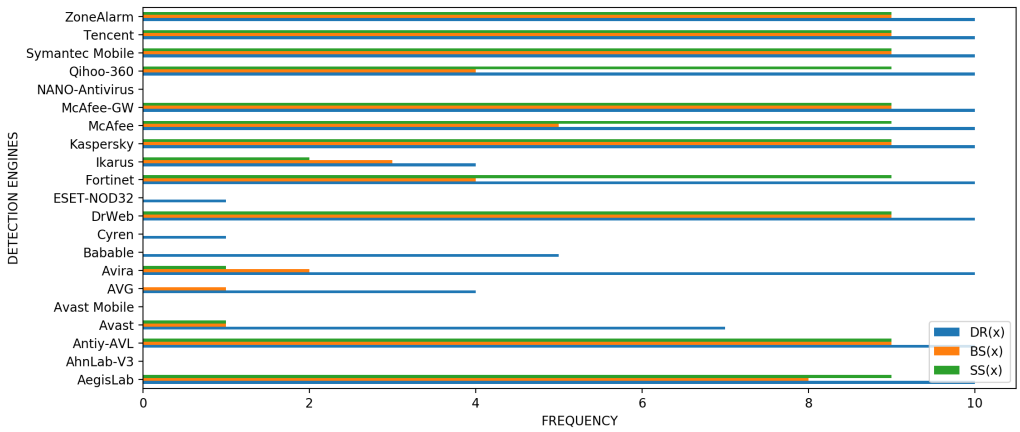


Fig. 4.6 Frequency f of detectors d that failed to detect the malware ($0 < f < 10$) for the fitness functions (DR(x), BS(x) and SS(x)) for Droidkungfu family

- GGtracker: In Fig. 4.7, the number of engines not fooled by any variant when fitness function DR(x)is used, is 9. Examples include CAT-QuickHeal and DrWeb. 13 engines however, were fooled by all the mutants for function DR(x), with examples like Arcabit and BitDefender. Also, 15 engines were not fooled by any variants for function BS(x). Examples include K7GW and Kaspersky. 16 engines however, were fooled all the time for function BS(x) - examples include AVware and TrendMicro. In addition, 16 engines were not fooled by any mutant for function SS(x), with examples such as Avast and AVG. On the other hand, 18 engines were fooled by all the mutants for function SS(x). Examples include Cyren and McAfee.

4.3.4 Diversity

Recall from Section 4.1, that diversity of the samples can mean several things such as behavioural diversity which means the behavioural characteristics of the variants are as dissimilar as possible to those of the parent malware and to each other. It could also mean structural diversity that is, the variants take varying code level and semantic structures

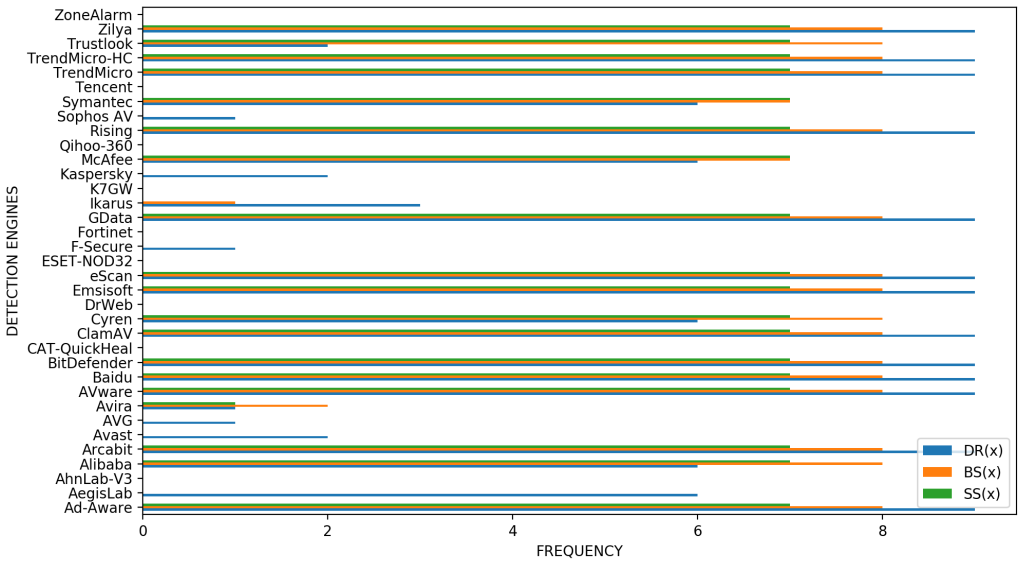


Fig. 4.7 Frequency f of detectors d that failed to detect the malware ($0 < f < 10$) for the fitness functions (DR(x), BS(x) and SS(x)) for GGtracker family

as compared to their parent malware as well as themselves. Also, it could mean that the mutants differ in terms of how they evade detection by existing detection engines.

In the previous section, a fitness function is used that tries to maximise one of three metrics namely structural dissimilarity, behavioural dissimilarity and evasiveness of the variants as compared to their parent malware. Each of these also maximise a diversity measure. Hence, they are used to guide the search towards a variant that is most dissimilar to the original malware. In this section, the diversity of the set of variants produced from multiple runs of the algorithm is studied to see whether the EA runs produce different variants, and to analyse whether they are more or less diverse according to each of the three metrics.

Diversity of the variants produced for each of the malware classes, is measured with respect to

- A detection signature
- A behavioural signature
- Structural similarity

The terms are explained below.

Diversity with respect to Detection Signature

The first attempt is to understand whether the evolved mutants are diverse with respect to their "detection" signatures (i.e. the behaviour of the 63 detection engines when presented with the mutant). A *detection vector d* is defined for each evolved mutant as a vector of 63 elements, corresponding to each of the 63 detection engines used for testing. The value of an element d_i is 1 if the corresponding engine detected the mutant and 0 otherwise. For each subset of n variants denoted by (m, f), (i.e. from a malware family m evolved using fitness function f), the percentage of *unique* detection signatures within the set of size n to obtain an indication of variant diversity, with respect to the ability of the detection engine to recognise the variant, is noted. Results are shown in Table 4.3.

From Table 4.3, it is evident that in terms of uniqueness of the detection signatures, the highest percentage of unique variants is seen in the Droidkungfu family when evolving for evasive (with 90% uniqueness) and behaviourally dissimilar (with 89% uniqueness) variants. However, when evolving for structurally dissimilar variants, the most unique variants are found in the Dougalek family (with a 50% uniqueness score).

The 63-dimensional vectors obtained from each of the evolved mutants, are mapped to a 2-dimensional space using t-SNE (t-Distributed Stochastic Neighbor Embedding) [144]; this is a non-linear dimensionality reduction algorithm commonly used for exploring high-dimensional data. It seeks to retain both the local and global structure of the data at the same time, (i.e. mapping nearby points on the manifold to nearby points in the low-dimensional representation (preserving local structure) while also preserving geometry at all scales, mapping far away points to far away points (global structure)). 76 new malicious mutants are evolved in total (considering all classes and fitness functions). After calculating the 76 detection vectors, any duplicates are removed (i.e. cases where variants within a subset defined by a malware family *and* fitness function have identical vectors). This reduces the total number of vectors to 46.

Figure 4.8 shows a plot obtained from t-SNE of the evolved malicious variants (after removing duplicates) in order to visualise diversity within the detection signatures. It can be noted that a distinct cluster is formed by variants of the Droidkungfu class as seen in the top left region of Fig. 4.8, with the other two malware families forming a separate cluster. Within each class cluster, further clusters can be identified according to the method used to generate the mutant. Within each of these sub-clusters, there is also diversity with respect to the detection signature.

Therefore, it is concluded that although not all the clusters are clearly differentiated by t-SNE, there is some evidence of clustering as seen in the Droidkungfu class. Consequently it can be said that the evolutionary approach used, is capable of generating distinct and diverse malware variants, providing useful data to develop new detection engines.

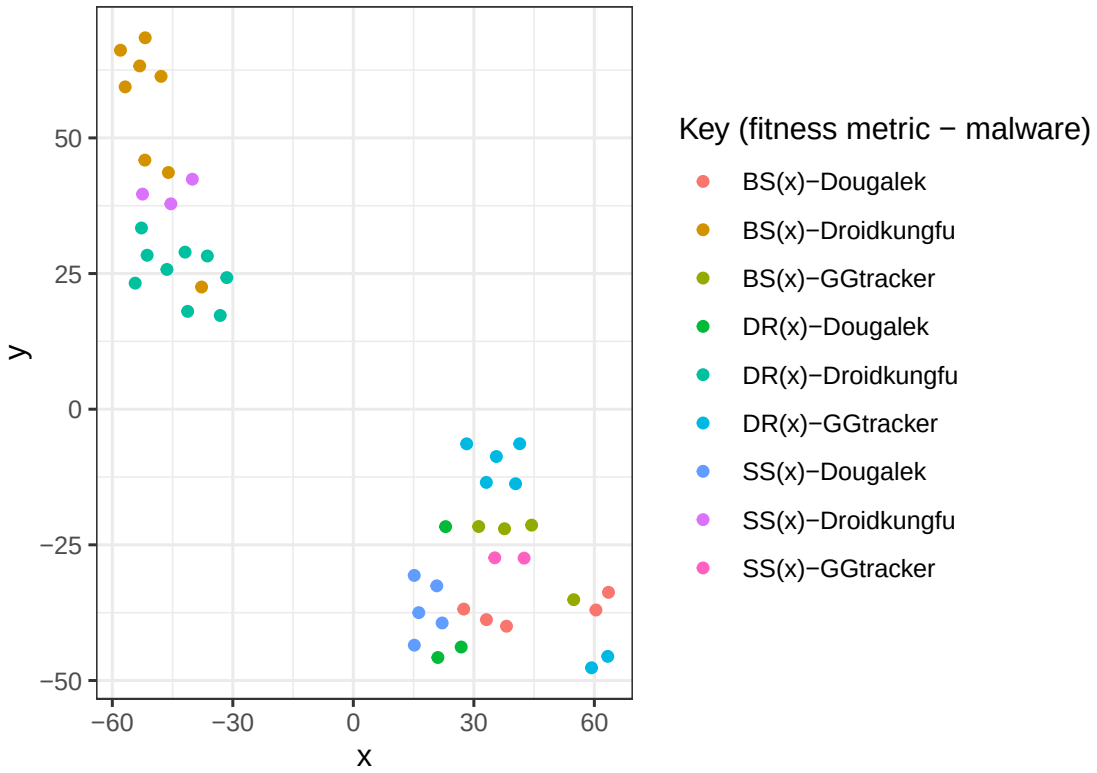


Fig. 4.8 Clustering of the detection vector associated with each evolved mutant, shown according to the fitness function used to evolve the mutant. Dimensionality reduction and clustering performed using t-SNE

Table 4.3 Percentage of evolved malicious variants that have a unique *detection signature*, shown by fitness function used to evolve

	Dougalek	**Droidkungfu**	**GGtracker**
Detection	43	90	78
Behavioural Similarity	71	89	50
Structural Similarity	50	33	29

Table 4.4 Percentage of evolved malicious variants that have a unique *behavioural signature*, shown by fitness function used to evolve

	Dougalek	**Droidkungfu**	**GGtracker**
Detection	100	70	89
Behavioural Similarity	100	78	78
Structural Similarity	80	75	100

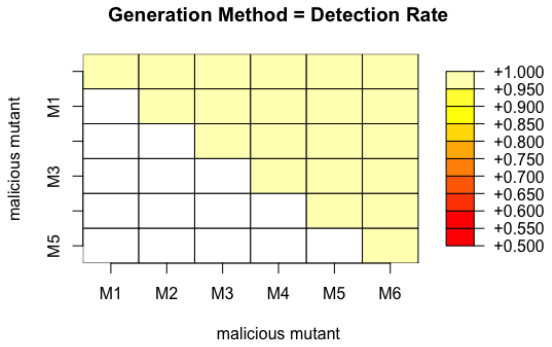
Diversity with respect to Behavioural Signature

The *behavioural signature* of a variant is defined as the vector b comprising of 251 elements with each element representing the system calls used by the variants. Each element b_i has a value that equals the frequency of the corresponding system call used. In the same manner as previously described with respect to diversity in the detection signatures, the number of unique *behavioural vectors* for each subset of evolved variants (m, f), is counted. Results are shown in Table 4.4. In terms of the uniqueness of the behaviour signature as seen in Table 4.4, Dougalek is the most unique family with 100% uniqueness when evolving for evasive and behaviourally dissimilar variants. GGtracker on the other hand, is the most unique family in terms of the evolution for structurally dissimilar variants. It is also interesting to note that out of the 251 behavioural tests performed (length of the behaviour vector), only 33 behaviours are actually ever recorded (frequency $>= 1$). The remaining behaviours have frequency 0 in **all** of the tests, regardless of malware class or fitness function used.

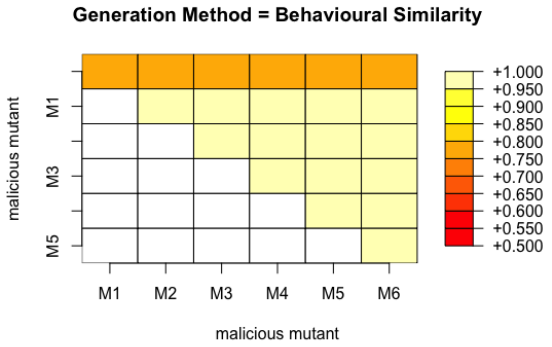
When comparing the diversity obtained in terms of behavioural and detection signatures of the evolved mutants, it is observed that there is more diversity in terms of the behavioural signatures than detection signatures, but that diversity is apparent with respect to both signatures.

Diversity with respect to Structural Similarity

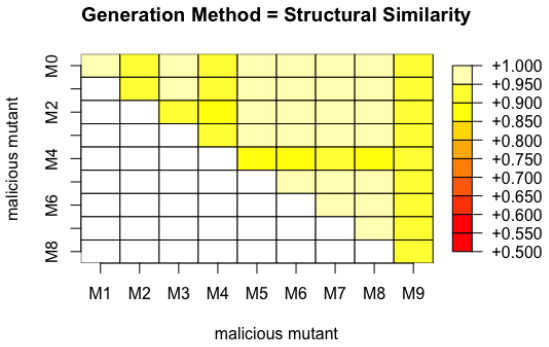
An examination of how much diversity is present in a set of evolved mutants, with respect to their structural similarity, is carried out. While evolving using the fitness function SS(x) produces mutants that are dissimilar to the original malware, it is advantageous to



(a) Structural Diversity Dougalek for function DR(x)



(b) Structural Diversity Dougalek for function BS(x)

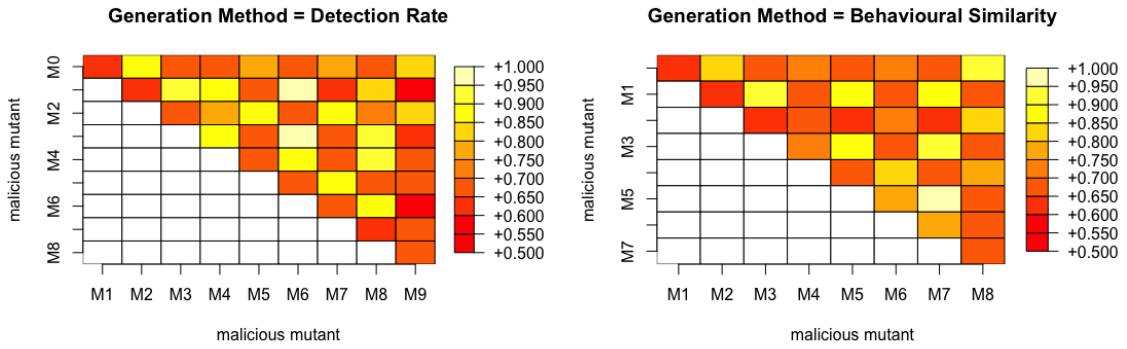


(c) Structural Diversity Dougalek for function SS(x)

Fig. 4.9 Analysis of Structural Diversity for Dougalek - For each of the *M* **malicious** mutants produced by 10 runs of MAL_EA using each fitness function (DR(x), BS(x) and SS(x)), a pairwise similarity between each pair of mutants is calculated

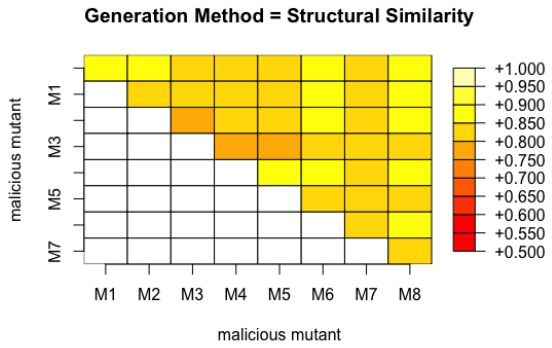
also generate mutants that are structurally dissimilar to each other in order to maximise diversity in a training set.

The pairwise similarity between each pair of mutants produced by a fitness function for all the malware families, is calculated. This gives a score between 0 and 1, where 0 means the pair is completely different, and 1 means they are exactly the same. A colour grid is then plotted representing the similarity scores of each pair as seen in Figs. 4.9, 4.10 and 4.11. It can be noted from these figures that the fitness function that produces the most structurally diverse variants is function SS(x) for GGtracker family, function BS(x) for Droidkungfu family and function SS(x) for Dougalek family. It is interesting to note that the least structurally diverse variants are produced by the Dougalek family, and the most structurally diverse variants are produced by the Droidkungfu family.



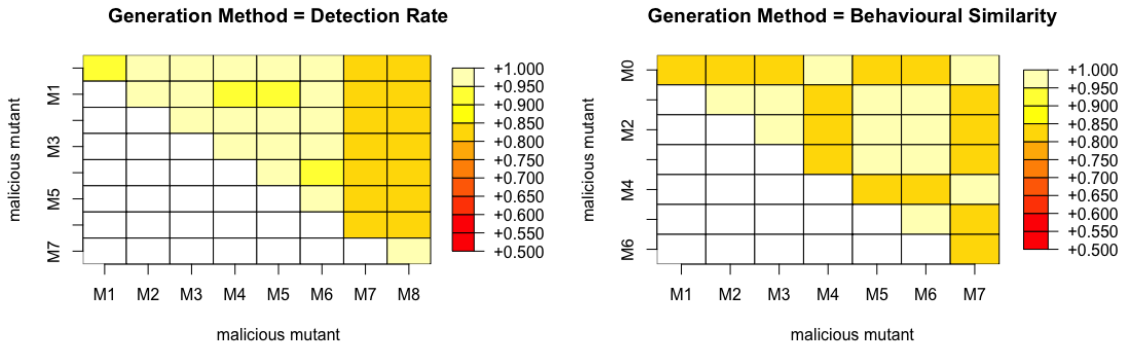
(a) Structural Diversity Droidkungfu for function DR(x)

(b) Structural Diversity Droidkungfu for function BS(x)



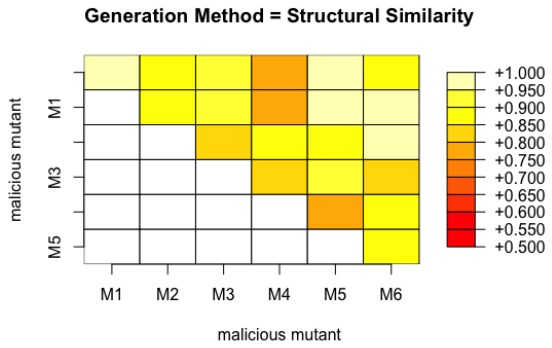
(c) Structural Diversity Droidkungfu for function SS(x)

Fig. 4.10 Analysis of Structural Diversity for Droidkungfu - For each of the *M* **malicious** mutants produced by 10 runs of MAL_EA using each fitness function (DR(x), BS(x) and SS(x)), a pairwise similarity between each pair of mutants is calculated



(a) Structural Diversity GGtracker for function DR(x)

(b) Structural Diversity GGtracker for function BS(x)



(c) Structural Diversity GGtracker for function SS(x)

Fig. 4.11 Analysis of Structural Diversity for GGtracker - For each of the *M* **malicious** mutants produced by 10 runs of MAL_EA using each fitness function (DR(x), BS(x) and SS(x)), a pairwise similarity between each pair of mutants is calculated

4.4 Summary and Conclusion

4.4.1 Summary

The following comments are provided to summarise all the results obtained in this chapter of the thesis.

- In terms of the influence of fitness function on evasiveness of the evolved mutants, it is apparent that for each of the functions (DR(x), BS(x) and SS(x)) in the fitness function, the resulting variants are more evasive than the original malware. This can be seen for all three malware classes. It was also show that for the majority of cases,
- The analysis of evasion characteristics for the three classes of malware show that for all malware families, there are a number of engines which are fooled by the mutants, but not fooled by the original malware. However, some engines like AhnLab-v3 are neither fooled by the mutants nor the original malware for the three malware families considered.
- Diversity is noticeable for the three classes of malware, with respect to the three fitness function metrics. In terms of the detection signatures, a level of diversity within the samples was observed. The same holds true for diversity, with respect to behavioural signature and structural similarity of the malware classes.

Answering the Research Questions

Recall from Section 4.1.1, that the first research question seeks to find if modifying the fitness function of MAL_EA to include solution characteristics enable us to obtain samples that are evasive but also diverse with respect to the original malware. Using three different fitness functions based on malware characteristics (behaviour, structure and ability to evade existing detection engines) for three malicious families, the evolutionary search was guided towards new variants that are more evasive than the original malware with experiments showing that the EA itself is effective as it outperforms a random search. Also, there is diversity within the malware mutants for each fitness function metrics. The diversity within the samples however, could not be guaranteed.

In answering the second research question, “How do the 63 detection engines vary in their ability to detect the new malware variants for each family of malware tested?”, tests were done to analyse which detection engines were more susceptible to the new mutants for the malware families considered. The results show that the mutants are able to fool detection engines that their parent malware were unable to fool. Engines like AhnLab-v3 however, are neither fooled by the mutants nor the original malware for the three malware families considered.

4.4.2 Conclusion

In this chapter, a new method has been proposed, to generate sets of mutant variants from already existing malware, using an Evolutionary Algorithm — MAL_EA. The goal was to generate a diverse set of new, malicious, mutants that evade detection by existing detection algorithms, where diversity is measured with respect to the structural and behavioural similarity of the mutants to the original malware. Using examples from three different malware families as a test-bed, selected based on their malicious payload described in Section 3.7 of the thesis, the new variants generated have been shown to be significantly more evasive than the original malware they were derived from, and display a diverse range of behaviours. Interestingly, it is shown that rewarding the evolutionary algorithm for generating mutants that are structurally or behaviourally dissimilar to the original malware, also generates *evasive* mutants (i.e. it is not necessary to explicitly evolve for evasiveness, but that this is obtained indirectly). Furthermore, the results have shown that the use of an EA is beneficial in the majority of cases, i.e. it outperforms a random search.

The proposed approach is generic enough to be applied in generating new adversarial samples from any class of malware. There is scope for improving the EA used, as earlier mentioned, as the diversity within the samples could not be guaranteed. Diversity is important because the evolved mutant samples is to be used as training data for ML models. Hence, diversity improves model generality.

Quality Diversity (QD) algorithms explicitly account for diversity, and also return a repertoire of solutions in a single run; this might be better than the approach used in this chapter, as multiple runs might all return the same variant. In the next chapter, a Quality Diversity (QD) algorithm is used, specifically MAP-Elites to drive diversity. This algorithm is designed to generate a repertoire that is diverse, with respect to some features, in a single run.

Chapter 5

Using MAP-Elites to Evolve a Repertoire of Diverse Solutions

The work presented in this chapter of the thesis was published in the proceedings of the 23rd European Conference on the Applications of Evolutionary and bio-inspired Computation (Evostar 2020) [17]. A copy of the publication is given in appendix A.A.2.

In this chapter, a Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) algorithm is utilised to generate a large set of novel, malicious mutants that are diverse with respect to their behavioural and structural similarity to the original mutant. Using the previously described classes of malware as a test-bed, it is shown that the MAP-Elites algorithm produces a large and diverse set of mutants, that evade between 66% to 72% of the 63 detection engines tested. When compared to results obtained using repeated runs of an Evolutionary Algorithm — MAL_EA that converges to a single solution result previously described in Section 4.2, the MAP-Elites approach is shown to produce a significantly more diverse range of solutions, while providing equal or improved results in terms of evasiveness, depending on the dataset in question. In addition, the archive produced by MAP-Elites sheds insight into the properties of a sample that lead to them being undetectable by a suite of existing detection engines.

5.1 Introduction

As has been previously established in Chapter 4, the search for mutant variants of malware can be a difficult task as it involves the traversal of a large search space of potential malicious variants. In addition, in order to drive improvements in detection models, it is advisable to create as many new samples as possible for a model to learn from, and furthermore, that the samples are as *diverse* as possible to improve model generality. A number of evolutionary algorithms (EA) have been proposed in the past to generate

variants of malware samples, for example, in the domains of pdf-malware [155] and in Android malware [15, 16]. However, these algorithms generate only a single new sample with each run of the algorithm. Given that most machine-learning approaches require large amounts of training data, not only is it time-consuming to generate multiple samples in this way, but there is no guarantee that the samples will be diverse. Furthermore, existing methods do not provide much insight into the properties of the generated samples.

To address this, a solution is proposed in this chapter that generates a *set* of variants that are *diverse* with respect to two features, the Structural Similarity (SS(x)) and Behavioural Similarity (BS(x)) of the variants, with respect to the original malware. Specifically, a Quality-Diversity algorithm is applied — MAP-Elites [111] — to generate a set of diverse variants that are optimised with respect to their ability to evade a large set of well-known detection engines. Recall from Section 2.5.4, that MAP-Elites algorithm traverses a high-dimensional search space in search of the best solution at every point of a feature space with low dimension defined by the user, and is one of a new raft of Quality-Diversity optimisation algorithms [123] that aim to return an archive of diverse, high-quality behaviours in a single run. The algorithm has multiple documented successes in evolutionary robotics [111], but also in design applications, car wing-mirror design [65].

5.1.1 Research Questions

Two questions are addressed in this chapter:

- How does diversity of samples produced by running MAP-Elites algorithm compare to repeated executions of the standard Evolutionary algorithm, MAL_EA, described in [16] (the algorithm described in Section 4.2)?
- How does the evasiveness of samples produced by MAP-Elites algorithm compare to repeated executions of a standard Evolutionary algorithm, MAL_EA (described in Section 4.2)?

5.1.2 Contribution

The contributions of the chapter is three-fold. Presumably, this is the first use of an illumination algorithm to generate a diverse set of mutant variant of malware. The approach is rigorously evaluated in terms of the number of samples generated, their evasiveness, and their diversity with respect to two features that measure the behavioural and structural similarity to the original malware. Secondly, a comparison is provided, to results obtained by running a single evolutionary algorithm (MAL_EA) multiple times in

order to generate a set of variants [16] (the algorithm described in Section 4.2), comparing the same metrics as above. Results show that MAP-Elites generates larger, more diverse sets of variants than the EA, while retaining approximately the same levels of performance (in terms of the evasiveness of the samples generated). Finally, novel insights are provided, into the factors that contribute to evasiveness, based on the results obtained from the illumination algorithm.

5.2 Methodology

In this section, the malware mutant generator that uses MAP-Elites algorithm to generate an archive of highly evasive but diverse mutants that can be used as future training data by a machine-learning model, is described.

5.2.1 Modification of MAP-Elites Algorithm for Malware Mutant Generation

The modification of MAP-Elites algorithm for malware mutant generation, proposed by us is given in Alg. 4. The highlighted portions of Alg. 4, show the difference between the original MAP-Elites algorithm given in [111] (described in Section 2.5.4 of the thesis) and the modified version presented by us. First, an empty archive is created as a two-dimensional grid defined by two features: the behavioural similarity and the structural similarity of a solution to the original malware. The grid is divided in 20x20 equally sized cells; these are created by equally "binning" the range of each feature (which take values between 0 and 1), thereby creating a potential archive of 400 solutions. The algorithm is then initialised with a random population of mutants, each created by applying a single mutation to the original malware. After calculating the feature descriptor for the mutant (see Section 5.2.2), the mutant is mapped to the corresponding cell in the archive.

Mutants are generated by applying a single mutation to an existing malware by selecting a mutation operator at random from the following list:

- Garbage Code Insertion (GCI) - This inserts a piece of junk code, e.g. a line number into the original program code.
- Instructional Reordering (IR) - This adds a go-to statement in the original program code which jumps to a label that does nothing.
- Variable Renaming (VR) - This renames a variable with another valid variable name in the original program code.

As described in Section 3.3, the original malware is in the form of an executable *apk file*. To create mutants, the apk must first be reverse engineered by converting it to a *smali* format using apktool. Thereafter, the following steps are executed:

1. Apply a mutation operator to the smali code.
2. Recompile the smali to apk in order to test that the variant created is executable.
3. Sign the recompiled apk using apksigner and align using zipalign.
4. Calculate the feature descriptor of the mutant.
5. Calculate the fitness of the mutant (detection-rate).

Subsequent solutions are created by random selection from the elites in the map. Upon selecting each random elite, they are also mutated by applying a randomly selected mutation operator, from the list given above. New mutants are placed in the archive if the corresponding cell is empty *or* replace an existing solution in a cell if their fitness is better than the existing solution.

Algorithm 4 MAP-Elites algorithm for mutant generation, modified from [111]

procedure MAP-ELITES(I, G)
  ($\mathscr{E} \leftarrow \phi, \mathscr{X} \leftarrow \phi$) ▷ N-dimensional map of elites: mutants $\mathscr{X}$ and their evasiveness $\mathscr{E}$
  for iter $= 1 \rightarrow I$ **do** ▷ Repeat for I iterations
    if iter $> G$ **then** ▷ Initialize by generating G random solutions by mutating original malware
      $x' \leftarrow random\_solution()$
    else ▷ Subsequent solutions are generated from elites in the map
      $x \leftarrow random\_selection(\mathscr{X})$ ▷ Randomly select an elite x from the map $\mathscr{X}$
      $x' \leftarrow random\_mutation(x)$ ▷ Create a mutant of x
    end if
    if $executable(x')$ **then** ▷ Confirm that mutated solution compiles and executes
      $b' \leftarrow feature\_descriptor(x')$ ▷ Calculate and record the behavioural and structural similarity between x' and the original malware
      $e' \leftarrow evasiveness(x')$ ▷ Record the evasiveness e' of x'
      if $\mathscr{E}(b') = \phi$ or $\mathscr{E}(b') > e'$ **then** ▷ If the appropriate cell is empty or its occupants's evasiveness is >= e', then
        $\mathscr{E}(b') \leftarrow e'$ ▷ store the value for evasiveness of x' in the map of elites according to its feature descriptor b'
        $\mathscr{X}(b') \leftarrow x'$ ▷ store the solution x' in the map of elites according to its feature descriptor b'
      end if
    else
      delete x'
    end if
  end for
  return feature-evasiveness map ($\mathscr{E}$ and $\mathscr{X}$)
end procedure

5.2.2 Feature Descriptor

Two feature descriptors (defining a 2D grid) of the mutants is defined by the behavioural and structural similarity between the original malware and a mutant. A *behavioural signature* of the mutant is derived from monitoring its system calls using Strace, using Monkeyrunner to simulate user interaction. This creates a behavioural signature represented as a vector of 251 elements, each corresponding to the frequency of 251 possible

system calls made by the mutant. The behavioural similarity between the original malware and the mutant is calculated as the cosine similarity between the two system call vectors, returning a value between 0 and 1, where the former indicates that the original malware and the mutant share no behavioural similarity, while 1 means the original malware and the mutant have equivalent behaviour.

The *structural similarity* between the mutants and the original malware is measured using both text based and code level similarity metrics as described in Section 3.5. The similarity metrics are then *averaged*, returning a value from 0 to 1 where 0 means the original malware and the mutant are completely dissimilar and 1 means they are the same. These metrics are chosen following previous work by [125].

5.2.3 Fitness Evaluation

The fitness of a mutant is measured in terms of its ability to evade a set of well known detection engines. Mutants are evaluated using Virustotal as described in Section 3.4. The fitness measures the *detection-rate* (i.e. the percentage of the antivirus engines that *fail* to detect a mutant). A detection-rate of 0 denotes that no engine detected the variant, while a value of 1 denotes all the engines detected the mutant. The problem is thus treated as a *minimisation* problem.

5.3 Experimental Settings

The parent malware chosen belong to Dougalek, Droidkungfu and GGtracker families, explained in Section 3.7. Experiments are conducted separately on each family.

To assess the quality of the MAP-Elites based mutant generator, its performance is compared against that of an EA — MAL_EA (the algorithm described in Section 4.2) designed for the same task. Recall that MAL_EA is a classical EA that uses a single objective performance-based fitness function to drive evolution, with no regard to features of the resulting solutions. MAL_EA is a steady-state EA that uses same mutation operators as the MAP-Elites based mutant generator and no crossover. It uses tournament selection and replaces the worst solution in the population with the best solution produced in the tournament, provided that it is better than the worst solution. The parameters used are given in Table 5.1. The parameters used in MAL_EA were derived following empirical analysis and are described in Section 4.3.1 of the thesis. For MAP-Elites on the other hand, it uses random selection for parent selection. Most MAP-Elites algorithms only use mutation [54] not crossover (although some later ones do use crossover [9]) and therefore the default mutation rate is 1. The crossover operator is not used in our experimentation, as it leads to the generation of non-executable variants. Furthermore, as a result of the

Table 5.1 Evolutionary based Parameter Settings

MAL_EA	
Parameter	**Setting**
Selection	Tournament
Population size	20
Iterations	120
Mutation rate	1

MAP-Elites	
Parameter	**Setting**
Selection	Random Selection
Bootstrap	20
Iterations	120
Mutation rate	1

computational cost involved in running the experiments, the number of iterations is limited to 120. The number of bootstrap iterations is then set proportionally.

To compare between the MAL_EA method and the new MAP-Elites approach proposed here, four standard metrics for measuring algorithm performance are used, which include global performance, coverage, reliability and precision, taken from [111].

Global performance (Equation 5.1) computes for each run, the fitness of the single best performing solution s divided by the fitness of the best solution S possible. Following the approach described in previous literature, as the theoretical optimum is unknown, the value for the best solution possible is taken to be the single best solution obtained from any run of either algorithm. This is given as:

$$Global\ Performance = \frac{s}{S} \tag{5.1}$$

Coverage (Equation 5.2) measures how many cells of the feature space a run of an algorithm is able to fill of the total number that are possible to fill. For a single map, it is defined as number of filled cells $n(f_c)$ in the map divided by the total number of cells that theoretically could be filled. As this number is generally unknown, it is approximated by counting the total number of unique cells that have been filled considering all runs of all algorithms $n(F_c)$. Coverage is therefore an indicator of diversity.

$$Coverage = \frac{n(f_c)}{n(F_c)} \tag{5.2}$$

The *reliability* metric measures for each run, the closeness of the best solution found for each cell to the best possible performance for that cell, averaged over the whole map. As above, as the best possible performance is unknown, the value is approximated as the best solution obtained for the cell from any run of any algorithm, supposing that $M_{x,y}$ represents the highest performing solution found from all runs of the algorithm for both MAP-Elites and MAL_EA at coordinate x, y, and assuming that $M = m_1 ... m_k$ is a vector which contains the final performance map derived from every run of the algorithm for

both MAP-Elites and MAL_EA. Then,

$$M_{x,y} = \max_{\{i \in [1,\ldots,k]\}} m_i(x,y) \tag{5.3}$$

The reliability of a performance map m is given as:

$$Reliability = \frac{1}{n(M)} \sum_{x,y} \frac{m_{x,y}}{M_{x,y}} \tag{5.4}$$

where $x,y \in \{[x_{min},\ldots,x_{max};y_{min},\ldots,y_{max}]\}$, and $n(M)$ is count of unique cells filled by any run of the algorithm for both MAP-Elites and MAL_EA.

Precision is similar to reliability but for a single run; it averages only the performance of cells that were filled for that run, and provides an indication of how high-performing a solution is relative to what is possible for that cell.

The precision of a performance map m is given as:

$$Precision = \frac{1}{n(M)} \sum_{x,y} \frac{m_{x,y}}{M_{x,y}} \tag{5.5}$$

for $x,y \in \{[x_{min},\ldots,x_{max};y_{min},\ldots,y_{max}]\} | filled_m(x,y) = 1$, where $filled_m(x,y) = 1$ is a matrix that takes on either value 1 in an (x,y) cell if the algorithm generated a solution in that cell or 0 otherwise and where $n(M)$ is count of unique cells filled by any run of the algorithm for both MAP-Elites and MAL_EA.

Note that the standard definition of these metrics assumes a maximisation problem, therefore, in order to calculate these values, performance is defined as (1-*detection_rate*), using the definition of *detection_rate* given in Section 5.2.3 (i.e. a solution that was not detected by any of the antivirus engines has a performance of 1).

To ensure a fair comparison, each algorithm is run for exactly the same number of fitness evaluations (i.e. 120). In the case of MAP-Elites, this includes bootstrapping with $\mathscr{G} = 20$ iterations (step 4 of the Alg. 4), and then running for 100 more iterations (step 6 on-wards). 10 repeated runs are performed, each returning an archive of solutions.

For the EA comparison experiments, as in [16], a population size of 20 is used. The EA was run 10 times with each of the three fitness functions defined in [16], the first optimising directly for evasiveness, the second optimising for behavioural similarity, and the third for structural similarity. Each run results in a single solution, giving 30 variants in total, which are then combined into a single archive. The feature descriptor b is calculated for each of these 30 variants as described above, so that the results can be directly compared with MAP-Elites.

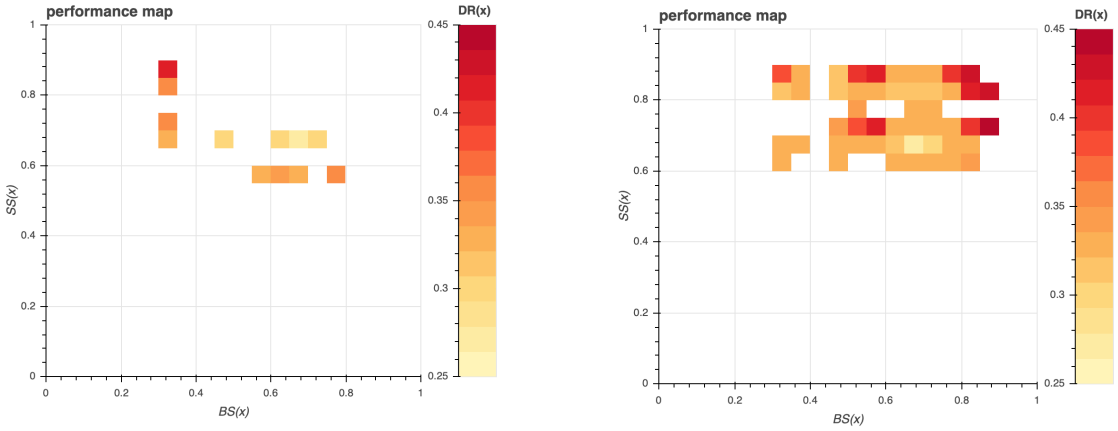


Fig. 5.1 Performance map of MAL_EA and MAP-Elites for Dougalek family

5.4 Results and Analysis

Here, a qualitative comparison between the MAP-Elites based mutant generator and MAL_EA, is provided. Then, using the metrics described in Section 5.3, a quantitative comparison is carried out. Additional analysis is then provided, to gain insights into which of the anti-virus engines prove weakest in failing to detect the evolved variants.

5.4.1 Qualitative Comparison of MAP-Elites and MAL_EA

Figs. 5.1, 5.2 and 5.3 show the maps obtained from merging the repeated runs of (a) the EA and (b) MAP-Elites for Dougalek, Droidkungfu and GGtracker families. The x and y axes are defined by the selected feature descriptors (i.e. the behavioural (BS(x)) and structural similarity (SS(x)) between the original malware and the mutants respectively). A value of 0 for each axis indicates 0% similarity, and 1 represents a 100% similarity between the original malware and the mutants. The color bar represents the detection rates (DR(x)) of the mutants, with a value of 0 meaning 0% of detectors detected the variants, and 1 meaning 100% of detectors detected the variants. For both the x and y axes as well as the detection rates, the lower the values, the better. Hence, the lighter the shade of the filled cells in the map, the better.

It is obvious from Fig. 5.1 that MAP-Elites generates a larger archive of solutions than MAL_EA for the Dougalek family. Although the solutions from both algorithms cover approximately the same *range* for each feature, the MAL_EA map is sparsely occupied with the 30 solutions obtained from the multiple runs, filling only 12 cells; this indicates a lack of diversity with respect to the two features obtained from multiple runs of the EA. In contrast, MAP-Elites finds *50* solutions that are evenly distributed along the range of each feature.

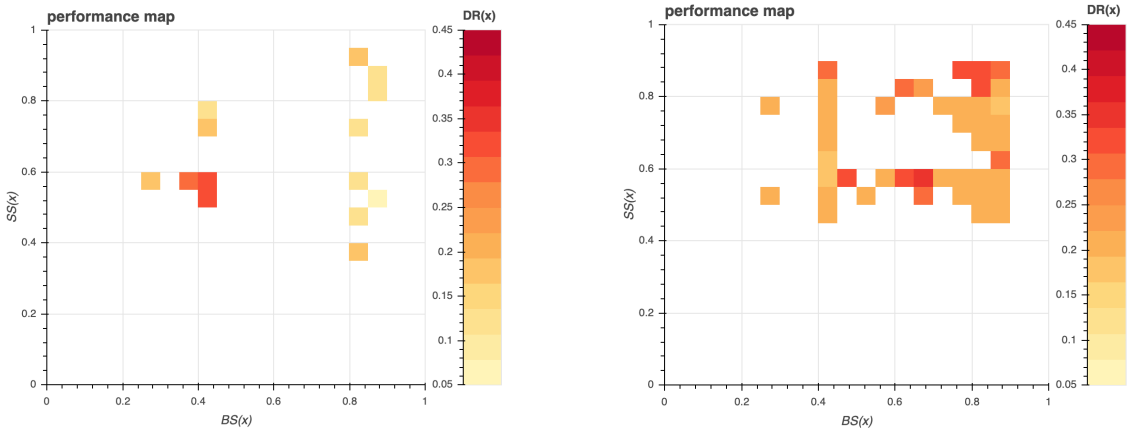


Fig. 5.2 Performance map of MAL_EA and MAP-Elites for Droidkungfu family

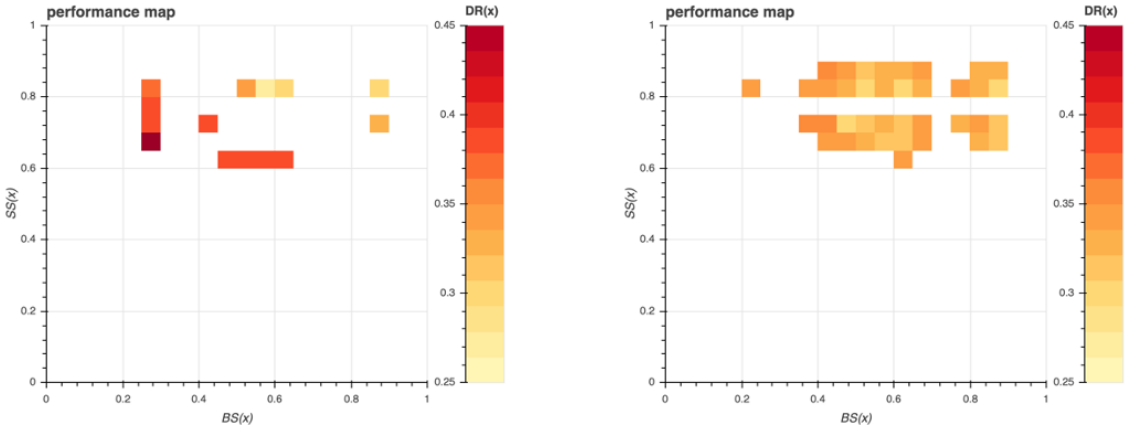


Fig. 5.3 Performance map of MAL_EA and MAP-Elites for GGtracker family

Similar observations apply to the Droidkungfu archives shown in Fig. 5.2. Although the range of the structural similarity feature extends more widely in the MAL_EA archive than that observed in the MAP-Elites archive, MAP-Elites finds solutions that are distributed more consistently across the range, as opposed to the more sparse distribution found by MAL_EA. For this malware, the 30 solutions obtained by MAL_EA occupy only 14 cells, again in stark contrast to that of MAP-Elites which finds a diverse set of solutions occupying 44 cells.

For GGtracker shown in Fig. 5.3, MAP-Elites generates a larger archive of solutions than MAL_EA for this family. The range of both structural and behavioural similarity features extends more widely with MAP-Elites than MAL_EA. Also, it can be seen that the 30 solutions obtained by MAL_EA occupy only 14 cells, which is also in stark contrast to that of MAP-Elites which finds solutions that are diverse and occupy 38 cells.

Table 5.2 The table shows the median results obtained for the 4 metrics on each dataset. Values in bold indicate the best performing algorithm where the difference between the medians is statistically significant (at a level of 0.05).

		Performance	Coverage	Reliability	Precision
Dougalek	MAP-Elites	**0.94**	**0.5**	**0.48**	0.96
	MAL_EA	0.92	0.06	0.06	0.97
Droidkungfu	MAP-Elites	0.85	**0.49**	**0.46**	0.94
	MAL_EA	0.86	0.07	0.07	**0.97**
GGtracker	MAP-Elites	**0.86**	**0.5**	**0.47**	0.93
	MAL_EA	0.7	0.08	0.08	0.95

5.4.2 Quantitative Comparison of MAP-Elites and MAL_EA

In Figs. 5.4, 5.5 and 5.6 the performance of both MAP-Elites and MAL_EA in terms of the global performance, coverage, reliability and precision metrics for Dougalek, Droidkungfu and GGtracker families, is shown. Mann-Whitney U tests (a test used for the comparison of the differences between two independent groups when the dependent variable is either ordinal or continuous, but not normally distributed [112]) with a 95% confidence interval, are used to determine statistical significance. Table 5.2 provides a summary of the median results derived for the 4 metrics on each dataset. Values are given in bold where the p-value indicates significance at a confidence level of 0.05. Where neither values is shown in bold, the significance test failed to reject the null hypothesis that the distributions are different.

It can be seen from Table 5.2 that for Dougalek, MAP-Elites does significantly better than MAL_EA in terms of global performance, reliability and coverage. For *precision* however, the significance test fails to reject the null hypothesis that the distributions are different.

For Droidkungfu, MAP-Elites performs significantly better than MAL_EA for coverage and reliability. In terms of global performance, the significance test failed to reject the null hypothesis that the distributions are different. In terms of the precision metric, MAL_EA outperforms MAP-Elites with the statistical test showing this difference is significant (i.e. when MAL_EA is able to fill a cell, it reliably finds a high-performing solution for the cell).

For GGTracker, MAP-Elites does significantly better than MAL_EA in terms of global performance, reliability and coverage. For *precision* however, the significance test fails to reject the null hypothesis that the distributions are different.

From Table 5.3, it is observable that both MAL_EA and MAP-Elites produce variants that are more evasive than the original malware for Dougalek, Droidkungfu and GGtracker families (i.e their detection rates are lower). For Dougalek, both methods produce a variant

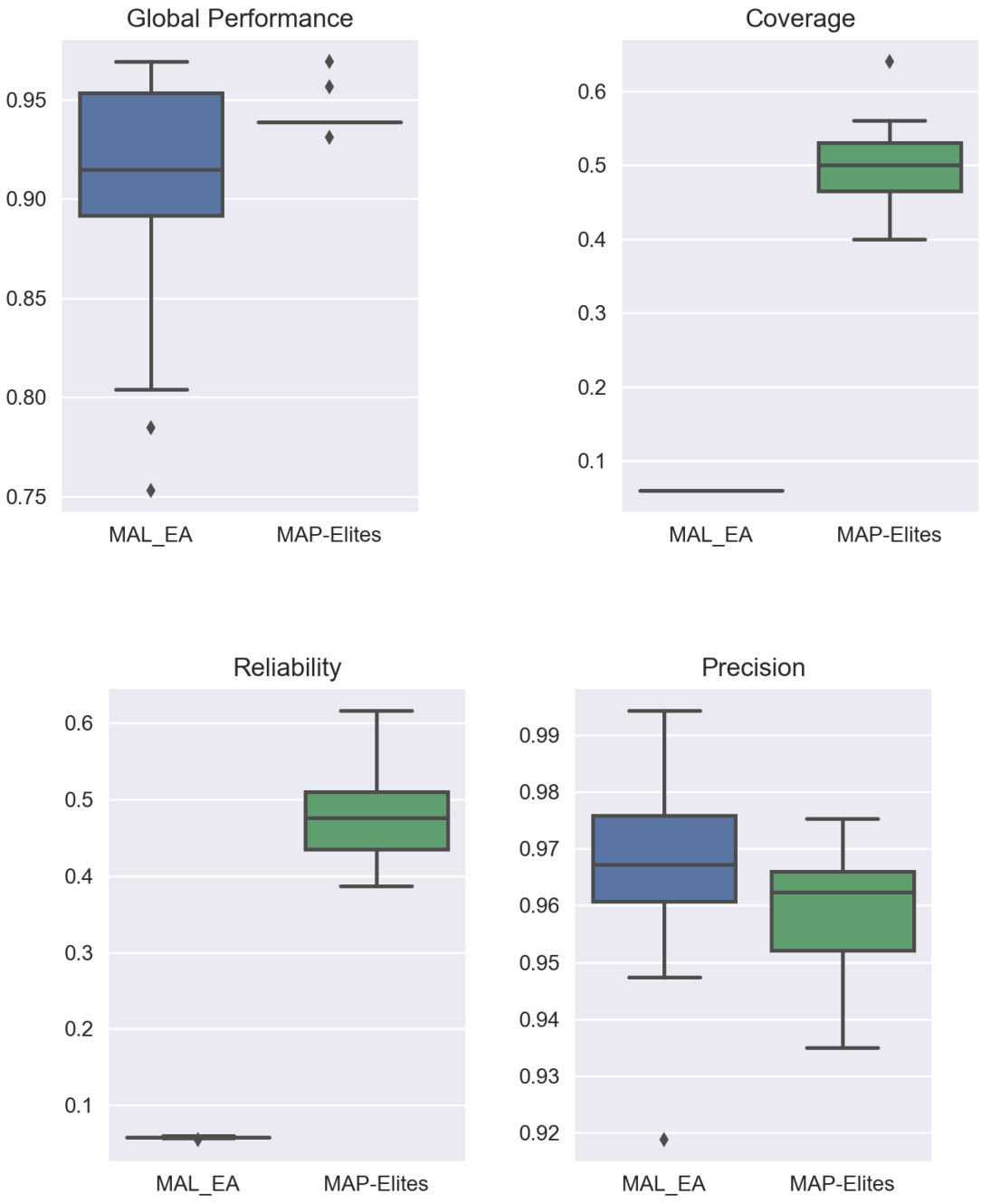


Fig. 5.4 Boxplots of Global Performance, Coverage, Reliability and Precision for MAL_EA and MAP-Elites for Dougalek family

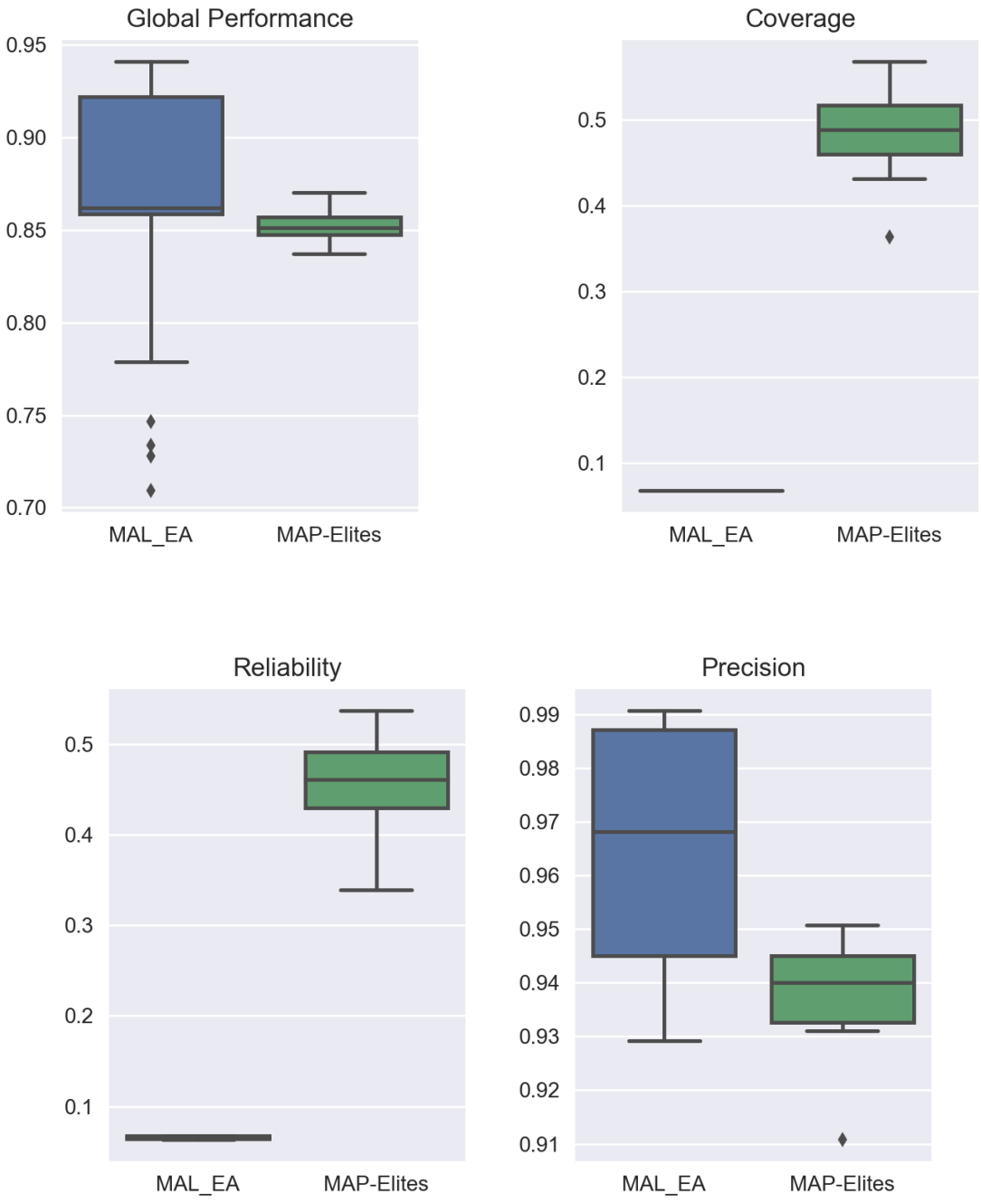


Fig. 5.5 Boxplots of Global Performance, Coverage, Reliability and Precision for MAL_EA and MAP-Elites for Droidkungfu family

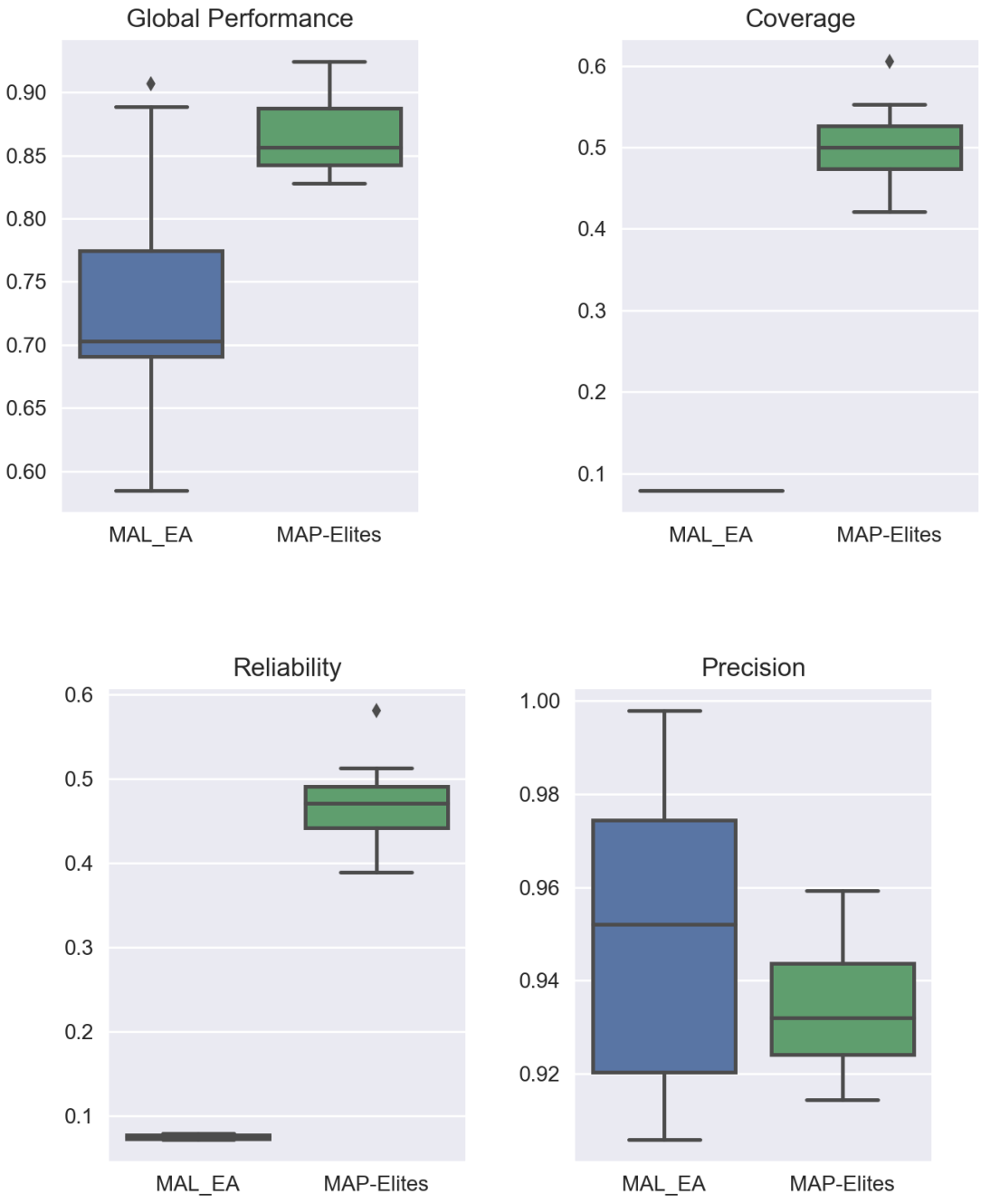


Fig. 5.6 Boxplots of Global Performance, Coverage, Reliability and Precision for MAL_EA and MAP-Elites for GGtracker family

Table 5.3 Fitness of the best, median and worst variants produced by MAP-Elites and MAL_EA, where fitness is defined by the detection-rate, i.e. the percentage of detectors that recognise the variant as malicious. Hence, 0 represents a failure of all 63 detectors.

Parameter	Dougalek		Droidkungfu		GGtracker	
	MAP-Elites	MAL_EA	MAP-Elites	MAL_EA	MAP-Elites	MAL_EA
Best	0.28	0.28	0.18	0.06	0.29	0.27
Median	0.32	0.34	0.2	0.19	0.31	0.38
Worst	0.33	0.46	0.21	0.33	0.32	0.46
Original Malware	0.6		0.35		0.6	

that is only detected by 28% of the detectors (compared to the original malware that was detected by 60% of the detectors). For Droidkungfu, MAL_EA produces a single variant that is only detected by 6% of the detection engines, outperforming MAP-Elites in which the single best variant is detected by 18%. In comparison to the original malware, the two methods do better as the original malware was detected by 35% of the detectors. For GGtracker, MAL_EA produces a single variant that is only detected by 27% of the detection engines, outperforming MAP-Elites in which the single best variant is detected by 29%. Again, both methods outperform the original malware that was detected by 60% of the detectors. Although the median values are similar for both Dougalek and Droidkungfu families, it can be observed that for GGtracker, the median value for MAP-Elites is better than MAL_EA. Also, the worst variant produced by MAP-Elites is more evasive than the worst variant from MAL_EA for Dougalek, Droidkungfu and GGtracker families.

In summary, MAP-Elites consistently outperforms MAL_EA in terms of the *coverage* and *reliability* metrics, while finding solutions that are better or comparable in terms of the *performance* metric. Although for the Droidkungfu and GGtracker families, the *single* most evasive variant is found by MAL_EA; recall that the goal of the study is to produce a *set* of diverse, hard to detect variants to provide an improved training set for a machine-learning algorithm. In this respect, a diverse set of evasive variants is significantly preferable to a small set of highly evasive variants.

5.4.3 Analysis of the Antivirus Engines

In order to gain more insight into which engines are most susceptible to potential mutated versions of the original malware, the percentage of new variants evolved using MAP-Elites that a detector fails to recognise, is determined. The only engines considered are the engines which recognised the original parent malware in this analysis, in order to understand which engines are vulnerable to potential mutants, and which remain capable

of detecting the malware. The results are shown for each malware family in Figs. 5.7, 5.8 and 5.9.

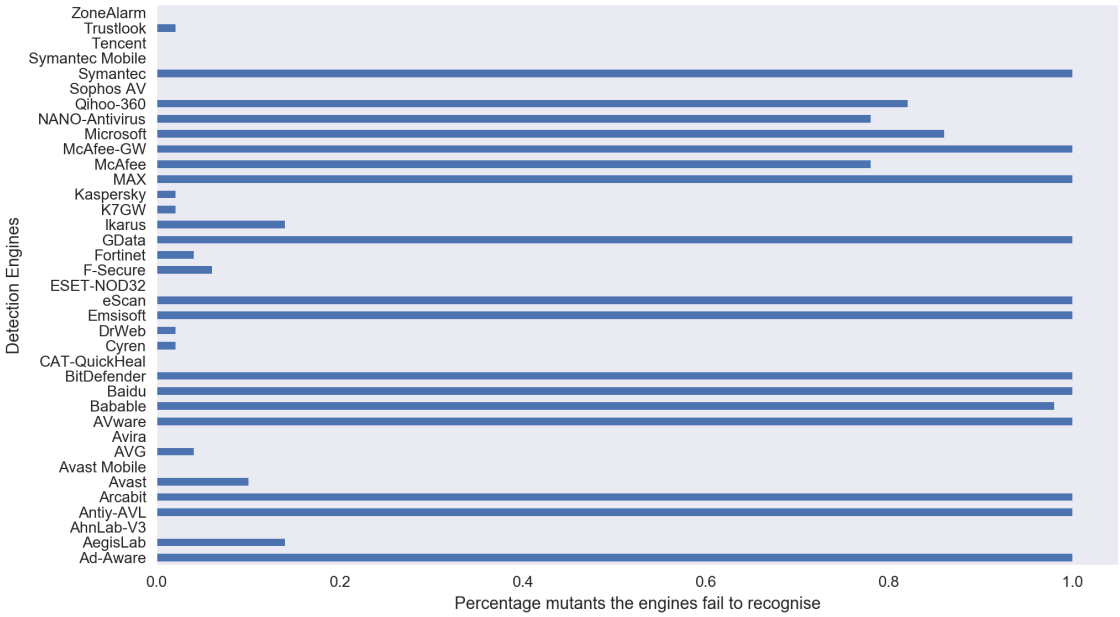


Fig. 5.7 Percentage of the mutants evolved from MAP-Elites that a specific detection engine failed to recognise for Dougalek family

It can be seen from Fig. 5.7, that 9 of the 37 engines that detected the original Dougalek parent malware also recognise all of the mutants evolved by MAP-Elites. Examples include Avast Mobile, Avira and Tencent. At the other extreme, 12 engines failed to detect 100% of the newly generated mutants. Examples include GData, Symantec, BitDefender and McAfeeGW. For the Droidkungfu malware, 4 of the 21 engines that detected the original malware also detect all of the evolved mutants. Examples again include the Avast Mobile and Avira engines that also proved robust to the Dougalek mutants. Three of the 21 engines failed to recognise *all* of the evolved mutants — AegisLab, DrWeb and McAfeeGW. It can be noted that the McAfeeGW engine appears very vulnerable to Dougalek and Droidkungfu families of metamorphic malware. For GGtracker family, 7 of the 37 engines that detected the original malware also detect all of the evolved mutants. Examples include the ZoneAlarm and Avira engines. Fourteen of the 37 engines failed to recognise all of the evolved mutants. Examples include Arcabit, AVware and Ad-Aware. It is noteworthy that the Avira engine proved 100% of the time robust to all families of malware mutants.

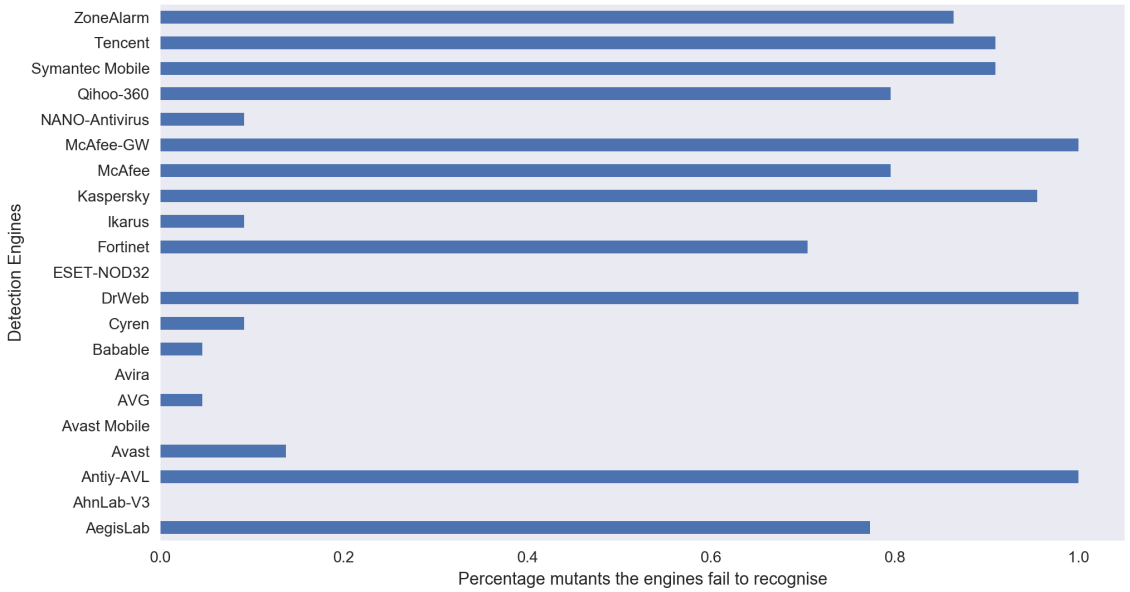


Fig. 5.8 Percentage of the mutants evolved from MAP-Elites that a specific detection engine failed to recognise for Droidkungfu family

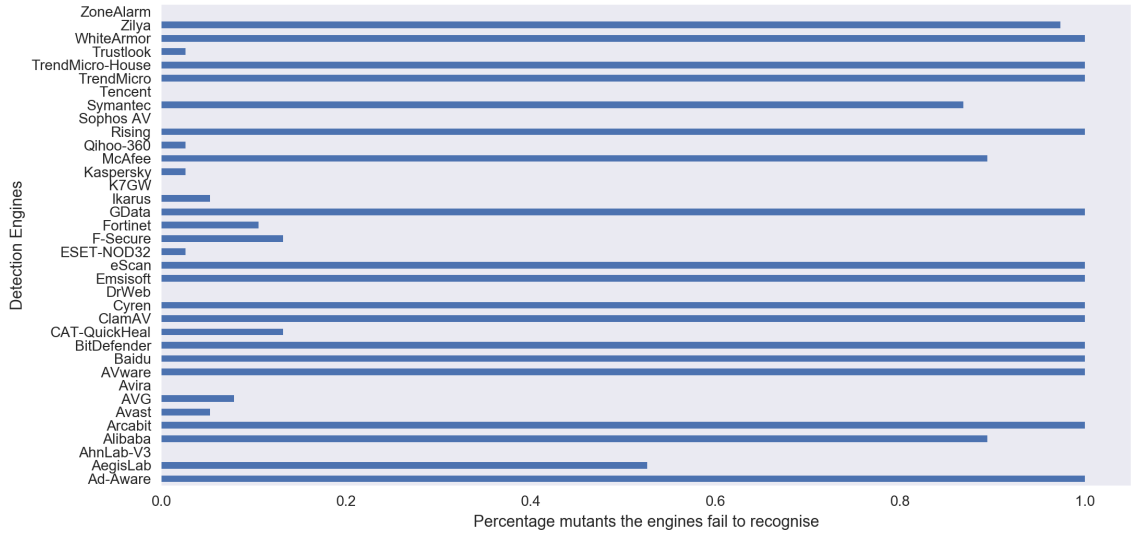


Fig. 5.9 Percentage of the mutants evolved from MAP-Elites that a specific detection engine failed to recognise for GGtracker family

5.4.4 Running MAP-Elites over longer periods to increase coverage

In this section, the results obtained from running MAP-Elites over longer periods to increase coverage, are discussed. Unlike the previous sections where MAP-Elites is run for 120 iterations, and bootstrapping for 20 iterations, here, MAP-Elites is run for 250 and 500 iterations, and bootstrapping for 50 iterations in both instances. Initial results were carried out for the Dougalek family as seen in Figs. 5.10 and 5.11. From Figs. 5.10 and 5.11, it can be seen that as compared to the performance map obtained when 120 iterations which fills between 20 and 32 cells is run, a coverage of 29 and 35 when MAP-Elites is run for 250 and 500 iterations respectively, is observable. However, since the increased coverage is not significant and given the computational cost of running MAP-Elites over longer periods, further experimentation for the other families of malware, was not pursued.

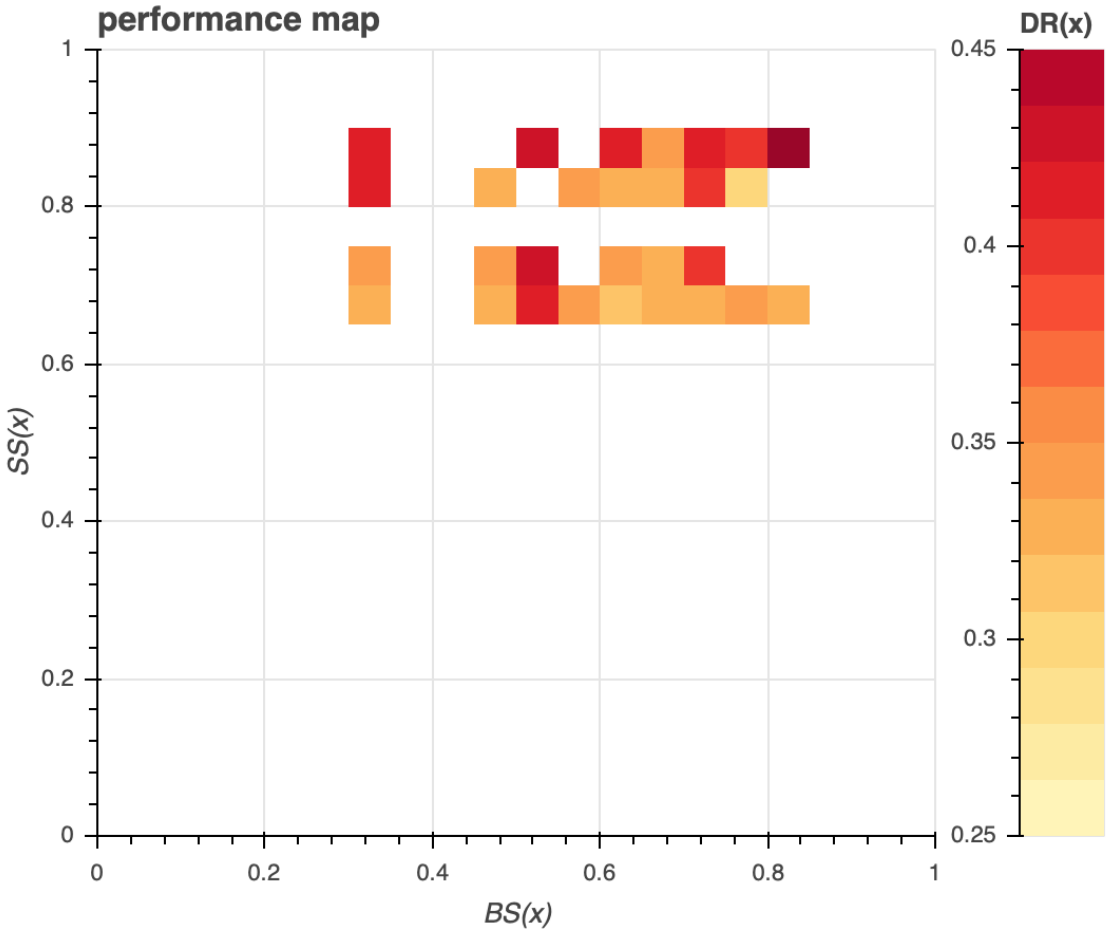


Fig. 5.10 Performance map for running MAP-Elites for 250 iterations for Dougalek family

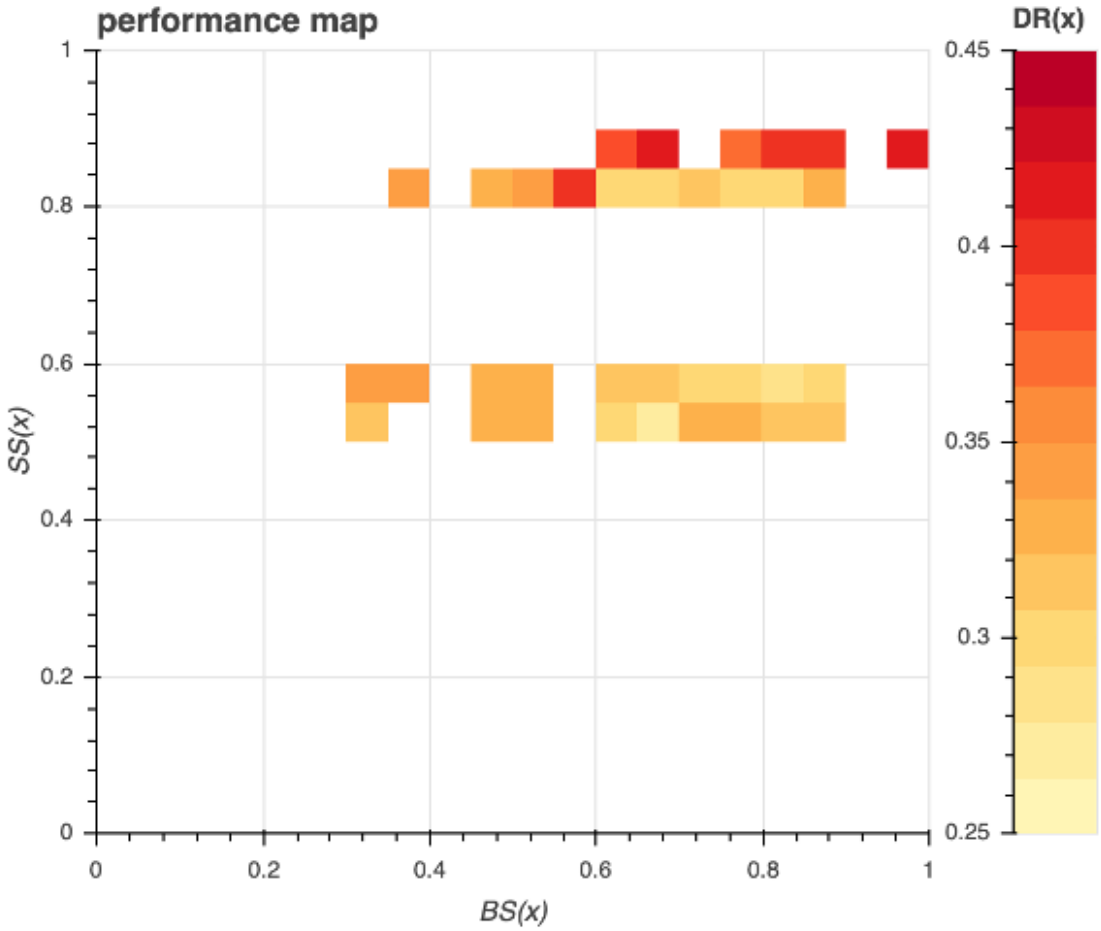


Fig. 5.11 Performance map for running MAP-Elites for 500 iterations for Dougalek family

5.5 Summary and Conclusion

5.5.1 Summary

The following comments are provided to summarise all the results obtained in this chapter of the thesis:

- The qualitative analysis of MAP-Elites and MAL_EA shows that for all malware families studied, MAP-Elites produces a larger archive of solutions that are evenly distributed along the range of each feature.
- The results from the quantitative analysis of MAP-Elites and MAL_EA shows that in all cases observed, MAP-Elites does better than MAL_EA for the coverage and reliability metrics, finds equally good solutions for the performance metrics, and the solutions derived are diverse while being evasive.

- The analysis of the antivirus engines for the mutants created using MAP-Elites revealed that for the three malware classes, the evolved mutants were able to evade more engines than their parent malware. It was noticed that engines such as McAfeeGW were more susceptible to the Dougalek and Droidkungfu families, while engines like Avira, was robust to all families of malware mutants in all of the analysis.

- It was also observed that running MAP-Elites for longer does not significantly lead to increased diversity and evasiveness. Hence, further experiments in this direction were not carried out due to its computational cost.

Answering the Research Questions

From Section 5.1.1, the first research question, "How does diversity of samples produced by running MAP-Elites algorithm compare to repeated executions of the standard Evolutionary algorithm, MAL_EA, described in [16] (the algorithm described in Section 4.2)?" was answered by comparing the diversity of samples generated using both MAP-Elites and MAL_EA with results showing that MAP-Elites produces solutions that are more diverse as compared to using a standard EA (MAL_EA).

The second research question, "How does the evasiveness of samples produced by MAP-Elites algorithm compare to repeated executions of a standard Evolutionary algorithm, MAL_EA (described in Section 4.2)?" was also answered by comparing the evasiveness of samples generated using both MAP-Elites and MAL_EA. The results show that MAP-Elites produces solutions that are as evasiveness as those produced by MAL_EA. Furthermore, like MAL_EA, the mutants created by MAP-Elites are also able to go undetected by more antivirus engines than their parent malware.

5.5.2 Conclusion

In the previous chapters, the ability of metamorphic malware to change its code over time with this posing significant challenges for detection models that are trained on static sets of data, has been highlighted. One approach to tackling this is to train models with data sets that include potential variants of the malware. It is therefore desirable to create new data sets that a) contain large numbers of new malicious samples and b) that those samples are as diverse as possible in order to maximise the performance of the model. In order to challenge the model and drive improvements, it is also desirable to create new samples that are highly evasive with respect to current detection methods.

It is proposed in this chapter that Quality-Diversity (QD) algorithms that produce diverse archives of high-performing solutions offer an obvious solution to this problem.

Although they have proved effective in a range of domains in recent years, it is gathered that this is their first use in generating a diverse set of adversarial malware samples within the metamorphic malware detection domain. It is shown that MAP-Elites — an example of a QD algorithm — is capable of generating high-performing and diverse samples for three malware families.

When compared to an EA (MAL_EA), it produces larger sets of data with more diversity (with 50% (Dougalek), 49% (Droidkungfu) and 50% (GGtracker) coverage for MAP-Elites, as opposed to MAL_EA's coverage of 0.06% (Dougalek), 0.07% (Droidkungfu) and 0.07% (GGtracker)), while still producing comparable performance in terms of minimising detection rates.

It could be argued that Quality-Diversity algorithms are a ripe avenue for exploration in this field, particularly if they can be combined in a setting where the variants generated can be used to augment existing training data, in order to improve the classification and detection of metamorphic malware. In the next chapter, classification of metamorphic malware is improved by augmenting training data with a diverse set of evolved mutant samples. The evolved samples comprise solutions generated using both MAP-Elites and MAL_EA algorithms. Both feature based and sequential models were used in the classification of the mutant variants of malware.

Chapter 6

Improving Classification of Metamorphic Malware by Augmenting Training Data with a Diverse Set of Evolved Mutant Samples

Some of the work presented in this chapter of the thesis was published in the proceedings of the IEEE World Congress on Computational Intelligence (WCCI) 2020 [18]. A copy of the publication is given in appendix A.A.3.

Detecting metamorphic malware provides a challenge to machine-learning models as trained models might not generalise to future mutant variants of the malware. These malware transform their codes between generations and most machine-learning models are only trained to recognise specific code variants. To address this, the possibility of improving machine-learning models by augmenting training data-sets with samples of potential variants, is explored. These variants are generated using two evolutionary algorithms — MAL_EA and MAP-Elites that evolve a structurally and behaviourally diverse set of mutants, optimised to avoid detection by a large set of existing detection-engines. Using features calculated from the behavioural trace of a sample as input, the ability of five non-sequential machine-learning methods (Logistic Regression, Support Vector Machine, Naïve Bayes, Decision Trees, and K-Nearest Neighbour) to detect the new variants, is evaluated. It is also observed that the detection rate is considerably improved by including the new samples as training data, and that the classifiers still generalise over a range of malware. This experiment is then repeated using a sequence-based deep-learning method (LSTM) as the classifier, which is shown to out-perform the feature-based classifiers in some test scenarios. Tests are made using both sequential and non-sequential machine-learning methods as this helps to answer one of our research

questions, “Which machine learning approach benefits most from augmenting with data from the novel variants?” Finally, an analysis is done to see if classification performance can be improved using BERT (a model pretrained on large NLP data sets), in a transfer learning context, and this is shown to improve performance in some test scenarios.

6.1 Introduction

Machine learning (ML) based methods are now common in malware detection [143], trained using samples of known malware. However, this leaves an attack surface for adversaries to explore with newly created malware that is dissimilar from the training data and designed to avoid detection. For metamorphic malware which can continually change its form, this is even more of an issue. The adversarial learning method [100], described in Section 2.5.4, presents a solution to this issue, in which a feedback loop is created in a system that (1) continually searches for malware samples that are misclassified by the machine-learning model used for detection, and (2) retrains the model based on these samples. However, [100] only provides a theoretical framework for the analysis of adversarial examples and their pilot study applied to spam filtering does not ascertain that their search for malicious samples result in executable and malicious samples.

An augmentation to this method is, therefore, proposed in this chapter, which incorporates Evolutionary Algorithms (EAs) to enhance the evolution of a diverse set of malware-mutants that are optimised with respect to their ability to evade detection by a large set of detection engines, while retaining their maliciousness. To achieve that, two EAs, introduced in Sections 4.2 and 5.2.1, are involved in the evolution of these mutants. The former EA — MAL_EA uses a classical approach to evolve an optimised mutant, whereas the latter EA uses a Quality-Diversity algorithm — MAP-Elites to return a set of mutants which are diverse with respect to their behavioural and structural similarity to the original malware. Then, this chapter investigates whether these mutants can be used to train better machine learning models capable of detecting other potential unseen mutants. This includes the use of a pretrained NLP model in a transfer learning setting to show improved classification of metamorphic malware using the evolved variants as part of the training data. The pretrained model is particularly useful in situations where there is lack of sufficient training data, which is why it is suited to our problem instance.

6.1.1 Research Questions

This chapter seeks to answer the following research questions:

- To what extent are models trained only on existing samples of known malware capable of detecting potential mutants?

- Can these models be improved by augmenting the training set with evolved samples, and if so, what is the most appropriate method of combining the evolved samples with existing samples?
- Do models trained on evolved data representing future mutants retain their ability to recognise existing known malware?
- Do multi-class models improve the classification of metamorphic malware than binary models?
- Can a transformer — BERT (that has been trained on large Natural Language Processing (NLP) datasets), be used in a transfer learning context to improve classification of metamorphic malware using the evolved samples?

6.1.2 Contribution

The contributions of this chapter can be summarised as follows. Firstly, evidence is provided to show that models trained only on existing samples of malicious code are vulnerable, failing to detect mutant samples. Secondly, it is demonstrated that the samples evolved using Evolutionary Algorithms provide a rich-source of training data that improves the ability of classifiers to detect new mutants, while retaining their ability to generalise across existing samples. Thirdly, it is shown that sequential models yield a better performance in some instances than non-sequential models. The feature vector data is also made available [19], so the readers can make use of the data. Finally, it is shown that using a transformer that has been pretrained on large NLP data sets in a transfer learning setting, leads to improvements in classification performance in some instances.

6.2 Methodology

The research questions outlined in Section 6.1.1 are answered by conducting a study using two types of ML models, namely, non-sequential and sequential models. Five classical ML models are compared with Long-Short-Term-Memory (LSTM) [63], a deep-learning method. The former methods rely on features describing the frequency of system-calls made by a malware, while the latter uses the ordered sequence of system calls directly as input. The aforementioned models are employed to understand if the classification of metamorphic malware by augmenting training data with evolved samples using both frequency of system calls or time ordered system calls as features, can be improved. Then, an analysis is made of which method yields the best performance. Then, the classification

performance is observed with the use of BERT (a pretrained NLP model), in combination with the evolved samples to see if this results in an improvement.

In this section, the algorithms employed in the generation of the training samples are also discussed. Then, the data-collection and pre-processing steps taken to obtain both normal and malware samples are described. Last but not least, the machine-learning algorithms (sequence and non-sequence based) employed in the detection process and justification for using these algorithms, are introduced.

6.2.1 Generation of Mutant Variants of Existing Malware

In this research, the ultimate focus is on metamorphic malware. However, since it is not easy to collect sufficient samples and their mutants, Android malware from well-known families are used as a proxy to generate mutant samples which represent potential future variants. This is done in order to prove the concept that augmenting training data, with a diverse set of evolved mutant samples, improves classification of metamorphic malware is a viable method.

The mutant samples of malware used as training data and test data in this work were created using the two evolutionary algorithms (MAL_EA and MAP-Elites) described in Sections 4.2 and 5.2.1. Recall that the first algorithm — MAL_EA used a steady state mutation-only EA to generate variants, which were evaluated according to one of three fitness functions: minimises the behavioural similarity between a variant and the original malware, minimises the structural similarity between a variant and the original malware, and minimises the detection rate with respect to 63 detection-engines. Only evolved samples that are both evasive and malicious are retained. The aforementioned methods also ensure that all samples are distinct (more details can be found in Chapters 4 and 5 of the thesis). In order to test that the evolved mutants retain their malicious nature, Droidbox is used (described in Section 3.4).

The second EA uses a Quality-Diversity algorithm — MAP-Elites — to produce a set of adversarial samples that are diverse, with respect to their structural and behavioural similarity to the parent malware, denoted by s and b respectively. For each descriptor $< s,b >$, the algorithm optimises the evasiveness of the variant associated with the descriptor. This method was shown to produce a larger variety of mutants, and to maximise the evasiveness.

Evolved mutants produced using both algorithms are used because, though MAP-Elites produces a more diverse and larger archive of solutions, MAL_EA produces the most evasive variants in a single run.

Table 6.1 Data-sets utilised

Samples	Data Source	Total number of Samples	Description
Benign Samples (B)	Google Playstore	50	Composed of games, beauty, communication and entertainment applications
Malicious Samples from web (M_w)	Contagio Minidump	50	Composed of Dougalek family, Droidkungfu and other families of malware
Evolved malware (EM)	Variants of malware samples from Contagio Minidump and Malgenome dump	50	Samples generated using MAP-Elites and MAL_EA (This set of samples contains three families; 21 from Dougalek, 15 from GGtracker and 14 from Droidkungfu)
Evolved malware - unseen set for testing (EM_u)	Variants of malware samples from Contagio Minidump and Malgenome dump	30	Samples generated using MAP-Elites and MAL_EA (This set of samples contains two families; 21 from Dougalek, and 9 from Droidkungfu)

6.2.2 Data Collection

The Android samples used are archived as APK files (the structure of an Android sample is described in Section 3.3 of the thesis). Benign and malicious data are collected from various sources as shown in Table 6.1. The malicious samples comprise (1) malware mutants generated by both EAs described in Section 6.2.1 and are mutant variants of (a) malware samples from Contagio Minidump and Malgenome, (b) malware samples of Dougalek, GGtracker and Droidkungfu families, (2) Additional malicious samples M_w collected from the web which are also from the Contagio Minidump and consist of families such as Dougalek, Droidkungfu, among other malicious software, and (3) malware samples from the evolved mutants EM generated by the methods previously described in Section 6.2.1, containing mutants generated from 3 malware families.

The malicious samples are selected from the dump on the basis of their malicious payload [157]. They fall into one of the four categories, namely, (a) *those that escalate privilege*, (b) *those that try to gain remote control of phones*, (c) *those that result in financial charges* and (d) *those that steal personal information of users*, as described in Section 3.7 of the thesis.

The clean samples B on the other hand, are sourced from Google play store and downloaded using Apkdownloader. The aforementioned data are all used for training the machine-learning models.

A final dataset EM_u consists of an additional 30 generated mutants and is held out as an unseen set for testing. This dataset only contains mutants from those malware families present in the dataset collected from the web.

The following tests were conducted to answer the research questions highlighted in this chapter of the thesis:

- To answer the first research question, the sequential and non-sequential machine-learning models were trained on a data-set that comprises the benign samples (dataset B, Table 6.1) and the malicious samples collected from the web (dataset M_w, Table 6.1). This was done to determine if models trained on existing malware are capable of detecting potential mutants.

- In order to determine whether adding evolved mutants to the training set results in better models, two further experiments were conducted:

    1. The training set consists of the benign samples B and 50 evolved mutants EM
    2. The training set consists of the benign samples B, 25 evolved mutants randomly selected from EM and 25 samples randomly selected from M_w (the samples from the web)

- The experiments described above were repeated using the LSTM. That is, three methods of training a model are considered. (1) using equal amounts of benign data and malware collected from the web (B and M_w), (2) using equal amounts of benign data and evolved mutants (B and EM), and (3) 50 samples of benign data, and 25 samples each of malware from the web and evolved mutants (B, M_w and EM). A comparison is made of the results of the best non sequence based model (Naïve Bayes) and LSTM on the aforementioned test cases.

- In order to answer the research question that seeks to find if models trained on evolved data representing future mutants retain their ability to recognise existing known malware, additional experiments are conducted where the models trained using evolved samples are tested on a set of unseen malicious samples from M_w (i.e. the samples collected from the web). The model trained with (B, EM) is tested on the unseen set M_w. The model trained with (B, EM, M_w) is tested on the 25 samples from M_w not used in training.

- Furthermore, to answer the research question that seeks to find if multi-class models improve classification of metamorphic malware than binary models, a further

training data set is created, named *6020combo*, which comprises 60 benign samples and 60 malicious samples. The 60 benign samples consist of 20 entertainment applications, 20 security applications and 20 communication application. The 60 malicious samples contain 20 malware from the Dougalek family, 20 malware from the Droidkungfu family and 20 malware from the GGtracker.

In addition, increasing the malicious samples for training is considered by examining 60 benign samples and 157 malicious samples (including 50 from Dougalek family, 55 from the Droidkungfu family and 52 from the GGtracker family). This increased data combination will be referred to as *6050combo* from here on.

For testing, dataset is used comprising 27 benign samples, 23 malicious samples (10 Dougalek family, 5 Droidkungfu family and 8 GGtracker family) for the 6050combo. For the 6020combo, a dataset is used that consists of 27 benign samples, 16 malicious samples (10 Dougalek family, 3 Droidkungfu family and 3 GGtracker family).

- Finally, to answer the last research question that seeks to find out if using a pretrained NLP model (BERT) in a transfer learning context can improve the classification performance, the aforementioned experiments are repeated using the *6020combo* and *6050combo* data sets for the Naïve Bayes, LSTM and BERT models.

6.2.3 Data Processing

Information obtained from the behavioural trace collected from running a sample, is used as an input to the classifiers. Strace is employed to monitor the behaviour of the malware. By running the malware's main activity using MonkeyRunner, user interaction can be simulated with the malware. Both sequential and non-sequential features are then generated from this log as follows:

Non-sequential data processing

For the non-sequential data processing, the logs generated from Strace are converted to a fixed sized vector with each element corresponding to the observed frequency of a potential system call employed. 251 systems calls are considered. An example of the vector is given below:

System Call ID	1	2	3	4	....	251
Frequency	20	74	0	8	....	150

Sequential feature generation

The sequential features are also extracted from the Strace log. However, the feature vector consists of the time-ordered list of system calls extracted from the log. An example of the vector is given below:

System Call ID	1	2	3	4	....	n
Time-ordered System Call	6	52	7	4	....	78

6.2.4 Machine Learning

Five non-sequential models are selected to test as predictors, based on their prevalence in the metamorphic malware detection literature. They are Logistic Regression, Support Vector Machine, Naïve Bayes, Decision Trees, and K-Nearest Neighbour.

For comparison with the non-sequential models, LSTM is chosen for detection based on sequences of system calls. Although deep-learning models, such as LSTM network, have shown their superiority in handling time-ordered information in other problem domains, they are less explored in malware detection [101], [40], [131].

The algorithms employed as well as the justification for using them is briefly explained below:

Logistic Regression (LR)

This algorithm is often used for binary data and when the target variable is categorical, for instance, to predict whether a software code is benign (0) or malicious (1). A logit transformation is employed to force the Y value to take on varying values between 0 and 1. The probability $P = 1/(1+e^{-(c+bX)})$ is first computed, and then X is linearly associated to $log_n^{P/(1-P)}$. The training time complexity of Logistic Regression is $O(nd)$ and the space complexity is $O(d)$ where n is the number of training examples and d is the number of dimensions of data.

One major advantage of Logistic Regression and one of the reasons why it was chosen as one of the considered algorithms, is that it allows the analysis and estimation of different explanatory variables by extending its basic principles. Hence, given the varying features of malicious and benign code, LR is a good algorithm for evaluating such features [68].

Support Vector Machine (SVM)

SVM has as its goal, identifying a hyper plane in a feature space of N dimensions that uniquely classifies data points. The hyper plane discovered is ideally one that maximises the distance between data points of the classes (i.e. has the maximum margin). This results

in greater confidence in future classification attempts. SVM requires little computational power yet produces high accuracies [80]. This algorithm is used because it performs optimally in cases where there is a distinct margin of separation among classes. Also, it has been shown to perform well in highly dimensional data particularly when the number of dimensions exceeds the sample size. It is also memory efficient. The training time complexity of SVM is $O(n^2)$ and its space complexity is $O(k)$, where n is the number of training examples and k is the number of support vectors [33].

Naïve Bayes (NB)

This is a probability based machine learning model employed in classification tasks. It follows the Bayesian theorem summarised in (6.1) below:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{6.1}$$

From the theorem, it is noted that provided that B has occurred, the probability of A occurring can then be computed. This implies that while A is the hypothesis, B is the evidence. In this work, A and B are the elements (or the occurrence frequency of a system call) of a feature vector. Naïve Bayes assumes that features are independent and so, the absence of a feature has no effect on the other [152]. Given n training examples, d dimensions of data and c classes, the training time complexity for Naïve Bayes is $O(ndc)$ and its space complexity is $O(dc)$.

It is good to note that provided the assumption of independent predictors is true, this algorithm outperforms a number of machine-learning models. Also, it is able to learn from small training samples thus, requiring less training time. It is also easy to implement [25], [7].

Decision Trees

A Binary Decision Tree results from the split of a node into two child nodes severally, starting with the root node which comprises the entire learning sample [29]. From Fig. 6.1, a Decision Tree formed from system calls made by both benign and malicious samples, can be seen. From the Decision Tree, a system call sequence beginning with system calls ioctl, epoll_wait and clock_gettime represents a benign sample, and one that begins with ioctl, ioctl and futex represents a malicious sample. Note that the last node also referred to as the leaf node represents the answer to the problem which, in this case, is the class. The training time complexity of a Decision Tree is $O(nlog(n)d)$ where n is the number of points in the training set and d is the number of dimensions of data. Its space complexity is O(nodes).

As compared to other machine-learning models, Decision Trees need reduced data preparation efforts. There is also no need for data normalization or scaling when using decision trees. It is a very intuitive model [84].

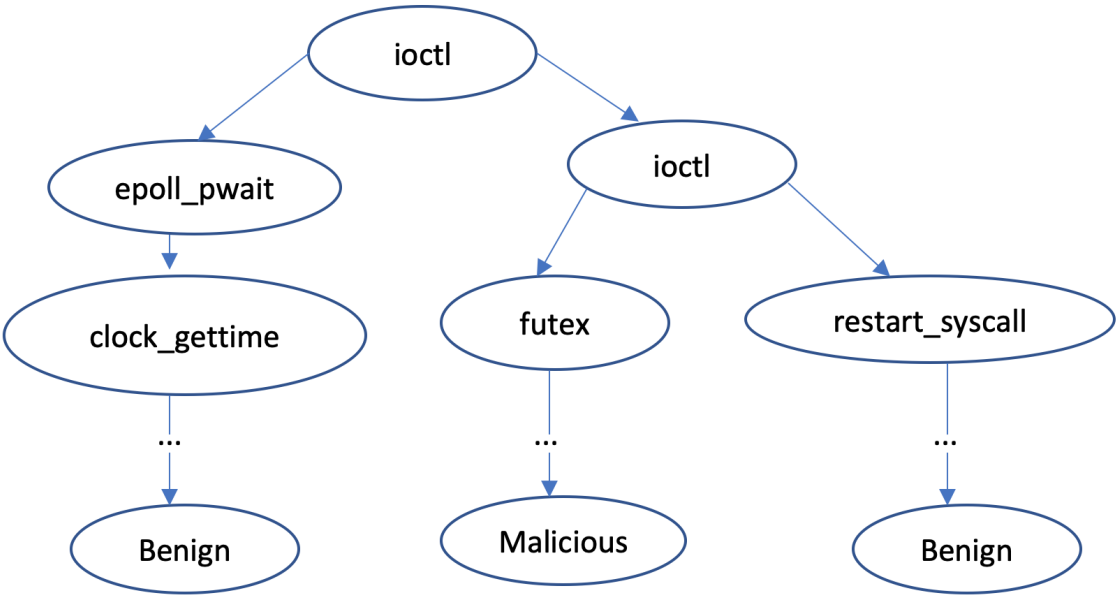


Fig. 6.1 An example Decision Tree created using system calls made by both benign and malicious samples

K Nearest Neighbour (KNN)

This algorithm seeks to find K instances in the training data in a feature space of N dimensions that are closest to the instance to be classified. It assumes that if an instance is close together in the feature space with the instance to be classified, then it is likely to be similar and have the same class as the instance to be classified. Given k neighbours, n training examples and d dimensions of data, the training time complexity for KNN is $O(knd)$ and its space complexity is $O(nd)$. Although it is one of the simplest ML models to implement, it is highly sensitive to the occurrence of parameters that are irrelevant [114]. It was selected as one of the algorithms to be considered because asides from its simplicity, it is also intuitive. As a non-parametric algorithm, there are no assumptions to be met. It also does not require a training step and learns from the training data only at the point of making prediction. Consequently, it is faster than a number of algorithms that need a training period such as SVM, Logistic Regression, among others. It can also be used for both classification and regression problems. Furthermore, it easily adapts to changes in input data in real time scenarios, given that it is an instance based learner [74][23].

Recurrent Neural Network (RNN)

This is a type of machine learning algorithm that is built for learning from time series sequence data. An RNN is a family of deep learning algorithms, which comprise layers connected in such a way that they go from the output of one layer to the input of the next layer, and have feedback loops returning to the preceding layer. An RNN learns from sequences of time series data, particularly, where temporary information is deduced from the sequences and employed in finding associations that exist between data and the expected network output [97].

In order to learn from the sequential data obtained from the samples described in Section 6.2.3 of this chapter, LSTM is used, a neural network specially designed for learning long-term dependencies from such data. It consists of gates that are able to hold, recover and forget information over a long time span. Traditional neurons in the hidden layer are substituted with memory blocks in LSTM, with these blocks accepting inputs from the network via the input node and outputs from the output gate multiplication as seen in Fig. 6.2. LSTM has a time and space complexity per weight of $O(1)$ [63].

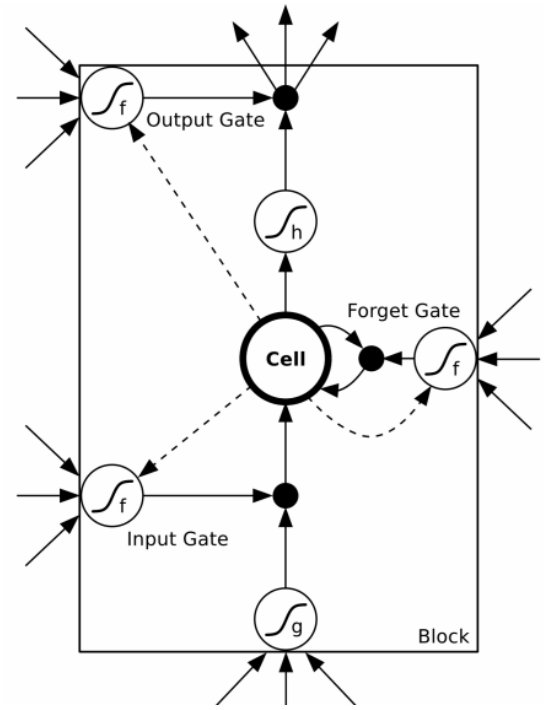


Fig. 6.2 An example LSTM memory block where the connections with weights proceeding from the cells to the gates are illustrated using dashes and the black circles are multiplications [63]

BERT

BERT [42] is an acronym for Bidirectional Encoder Representations from Transformers. It was created for the pre-training of deep representations that are bidirectional, from text that are not labelled by taking into consideration, the contextual information of the text by working out both the left and right context of the token. Consequently, the pre-trained BERT models can be easily adjusted and tuned with only an extra output layer to produce advanced models for a large number of NLP tasks. This model is pre-trained on a massive unlabelled text corpus which includes the whole of Wikipedia (this has about 2.5 billion words) and Book Corpus (this comprises about 800 million words). After being tested on about 11 NLP tasks, it produces novel state-of-the-art results such as improving the GLUE score by 7.7%, the MultiNLI accuracy by 4.6%, the SQuAD v2.0 Test F1 by 5.1%, among others

As illustrated in Fig. 6.3, BERT model comprises both pre-training and fine-tuning steps. The pre-training task occurs with the training of the model on instances that are unlabelled for various distinct pre-training tasks. The fine-tuning process on the other hand, begins with the initialization of the model with pre-trained parameters. Then, using the labelled data derived from the downstream tasks, the parameters are fine-tuned. BERT uses $O(n^2)$ time and space with regard to the length of the sequence.

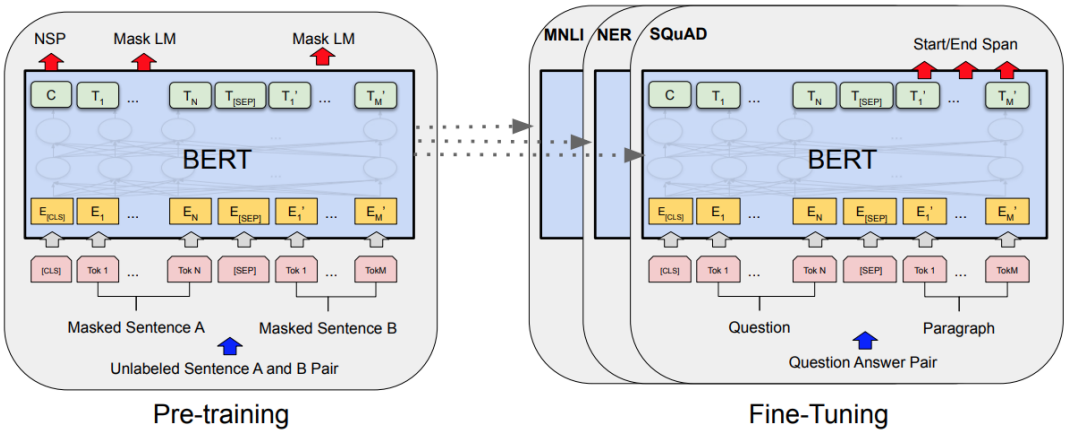


Fig. 6.3 A BERT model illustrating its pre-training and fine-tuning tasks [42]

6.3 Experiments

All classical non-sequential ML models used the implementation of the ML algorithms from Scikit-learn libraries for Python, with the Keras library[1] implementation of LSTM

[1] Keras - https://github.com/fchollet/keras

in python for the sequence based machine learning. 10-fold cross validation was used to train and validate models. An unseen test-set containing evolved mutant malware was used in testing.

The LSTM and its hyper-parameters were empirically tuned. As a result of its documented success in terms of its accuracy and computational power, "Adam" optimiser [83] was employed. Batch sizes between 10 and 500 were used, and experimented using either 1 or 2 layers of LSTM. Following this, LSTM was chosen with 2 layers; each layer has 128 neurons. The binary cross entropy function was used as the loss function for the binary classification and sparse categorical cross entropy was used as the loss function for multi-class classification. Moreover, as this problem is a classification problem, a Dense output layer comprising of one neuron was employed. A sigmoid activation function is used for the binary classification while a softmax activation function is used for multi-class classification. A batch size of 50 was made use of so as to space out the updates of weight. The model was fitted using just three epochs as it speedily over-fitted the problem.

The DecisionTreeClassifier was imported from the tree class of Scikit-learn library. The linear model class of Scikit-learn was used when implementing the Logistic Regression model. KNN on the other hand, was imported from the neighbors class of Scikit-learn. The Gaussian Naïve Bayes was used in this work which was imported from the naïve_bayes class of Scikit-learn. SVM was imported from the svm class of Scikit-learn. As the performance of the five non-sequence based algorithms was essentially compared in order to get the best algorithm of the five which was used to compare with LSTM (sequential algorithm), the default settings of the algorithms was employed.

For the implementation of BERT, ktrain[2] - an interface to Keras - is used. In particular, ktrain's text module is used. Our dataset is loaded using the texts_from_folder function in the text module and is preprocessed using the "bert" model. The pretrained BERT model (based on Google's multilingual cased pretrained model[3]) is loaded using the text_classifier function of the text module and wrapped in ktrain's learner object. The multilingual cased pretrained model is employed because it fixes normalisation problems in several languages and it is the latest model recommended by Google). The model is then trained using ktrain's fit_onecycle. As with the LSTM model, a batch size of 50 was made use of so as to space out the updates of weight. The model was fitted using just three epochs as it speedily over-fitted the problem.

The experiments then were conducted and results are analysed in the subsections below to answer the research questions.

[2]ktrain - https://github.com/amaiya/ktrain

[3]Google's Multi-lingual Bert Model - https://github.com/google-research/bert/blob/master/multilingual.md

Table 6.2 10-fold cross validation with model that uses samples of B and M_w as training data

Model	10-fold Mean Accuracy (std)	Validation Accuracy
LR	0.889881 (0.115217)	0.9
KNN	0.910317 (0.103585)	0.95
CART	0.912103 (0.080542)	1
NB	0.937103 (0.084450)	0.95
SVM	0.899603 (0.093919)	1

6.3.1 Detection of Potential Mutants using Models trained with Known Malware Samples

Recall from Section 6.2, that in answering the first research question, the models are trained using datasets B and M_w, Table 6.1. Table 6.2 shows the results from the cross-validation experiment, indicating that the trained models perform well on this data-set which only contains samples of known malware from the web. The least accuracy observed being 0.89 ($\approx 89\%$) as seen with the Logistic Regression (LR) model.

However, when the best trained model (Naïve Bayes) is tested on the unseen set of evolved mutants EM_u (top line of Table 6.5), the model performs very poorly with an accuracy of 0.4. This shows that the models trained on existing malware samples are not capable of detecting potential mutants which answers the first research question.

6.3.2 Improving Detection using Augmented Training Data with Evolved Mutants

In this section, an attempt is made to answer the second research question, "Can ML models be improved by augmenting the training set with evolved samples, and if so, what is the most appropriate method of combining the evolved samples with existing samples?" The results are shown in Tables 6.3 and 6.4 respectively. It is noted that both approaches achieved good accuracy on the validation sets , with the least performing algorithm for both approaches producing an accuracy of 0.87 ($\approx 87\%$) for the cross validation results. It is noticed as before, the Naïve Bayes method performing well. It can also be observed from Tables 6.2, 6.3 and 6.4 that from the cross validation results, the LR algorithm produces its best accuracy of 0.914881 ($\approx 92\%$) in the experiment that uses samples of B, M_w and EM datasets, the KNN algorithm produces its best accuracy of 0.910317 ($\approx 91\%$) in the experiment that uses samples of B and M_w datasets, CART's best performance of 0.922817 ($\approx 92\%$) can be observed when samples of B and EM datasets are used. SVM produces its best accuracy of 0.905556 ($\approx 91\%$) in the experiment that uses samples of B,

Table 6.3 10-fold cross validation with model that uses samples of B and EM as training data

Model	10-fold Mean Accuracy (std)	Validation Accuracy
LR	0.903770 (0.102321)	0.9
KNN	0.871032 (0.124375)	0.85
CART	0.922817 (0.084685)	0.95
NB	0.949603 (0.062133)	0.95
SVM	0.899206 (0.071488)	075

Table 6.4 10-fold cross validation with model that uses samples of B, M_w and EM as training data

Model	10-fold Mean Accuracy (std)	Validation Accuracy
LR	0.914881 (0.093706)	0.9
KNN	0.897817 (0.099469)	0.9
CART	0.913492 (0.097603)	0.9
NB	0.949603 (0.062133)	0.95
SVM	0.905556 (0.101645)	0.95

M_w and EM datasets. The best performing algorithm — Naïve Bayes produces the best accuracy across all experiments and has as its best accuracy a value of 0.949603 ($\approx 95\%$) obtained in the experiments that use samples of B and EM datasets and samples of B, M_w and EM datasets. It can be observed for the Naïve Bayes method, that its accuracy is improved when it is trained using evolved mutants. It is important to note that in Tables 6.2, 6.3 and 6.4, the training and validation data resulted from a repeated split of the dataset in a process known as cross validation. This was used to calculate the 10 fold mean accuracy. An independent and unseen test data was created and used to compute the validation accuracy.

The trained models (trained using (1) 50 samples of B and 50 of EM (2) 50 samples of B, 25 of EM and 25 of M_w), as explained in Section 6.2.2) were then tested on the unseen set of evolved mutants EM_u, with results shown in the second two lines of Table 6.5, obtained using the Naïve Bayes model. Compared to the results from the same model trained with the malicious samples from the web, a clear difference is observed, with both of the trained models resulting in accuracy of over 80%. This clearly indicates that the evolved mutants provide useful additional data by which to train a model to recognise future variants of metamorphic malware.

Table 6.5 Comparison of accuracy obtained on the unseen test set EM_u using a Naïve Bayes model trained on 3 different training sets

Training Data	EM_u **(Accuracy)**
B **and** M_w	0.4
B **and** EM	0.84
B**,** M_w **and** EM	0.82

Table 6.6 Comparison of accuracy obtained on the unseen test set EM_u using an LSTM model trained on 3 different training sets

Training Data	EM_u **(Accuracy)**
B **and** M_w	0.53
B **and** EM	0.9
B**,** M_w **and** EM	0.62

6.3.3 Sequential Model(LSTM) versus Best Classical Model (Naïve Bayes) in Detection of Potential Mutants

Here, more experiments are conducted to answer the second research question. Particularly, the research question, "Is a sequential model (LSTM) better in terms of accuracy than the best classical model (Naïve Bayes) in detecting potential mutants?" is answered. From Tables 6.5 and 6.6, it can be seen that as in the classical models, the model trained using only malware collected from the web, performs poorly on the mutant samples, although the LSTM has slightly higher accuracy than Naïve Bayes with an accuracy of 53% as opposed to an accuracy of 40% for Naïve Bayes. The LSTM outperforms Naïve Bayes for the model trained purely on benign and evolved samples ((B and EM), with accuracy of 90%). On the other hand, the model trained with equal proportions of malicious samples from the web and the evolved mutants performs less well than the Naïve Bayes model, obtaining accuracy of 62% compared to 82%.

In answering this research question, it is made plain that the sequential model (LSTM) performs better than the best classical model (Naïve Bayes) in two out of the three test scenarios (using training data comprising (B and M_w) and (B and EM)). However, the Naïve Bayes outperforms LSTM when trained with B, M_w and EM.

Table 6.7 Models that use B and EM as well as B, M_w and EM as training data and M_w as test data for both Naïve Bayes and LSTM

Training Data	**Naïve Bayes Accuracy (M_w)**	**LSTM Accuracy (M_w)**
B **and** EM	1	0.73
B, M_w **and** EM	1	0.91

6.3.4 Training Models with Predicted Future Mutants and Evaluating whether they retain their ability to recognise Existing Malware

The previous results show that models trained using evolved mutants are capable of recognising other evolved mutants. However, it is important to evaluate whether these models are fitted to the evolved mutants and therefore, fail to recognise existing malware. This will provide answers to the third research question, "Do models trained on evolved data representing future mutants retain their ability to recognise existing known malware?" Recall from Section 6.2.2, that in answering this research question, training takes place using samples of (B, EM) and (B, EM, M_w) and are tested on unseen malicious samples from the web, M_w. The training and validation data follow a data split of 80:20 ratio (following a Pareto split [96]) and an independent test set is used to compute the accuracy of the models as previously mentioned.

As can be seen from Table 6.7, it is clear that models trained with potential mutants are also able to detect other malicious samples (unseen malicious samples from the web taken from dataset M_w, Table 6.1). An accuracy of 100% is obtained for both Naïve Bayes models, with 73% and 91% respectively for the LSTM model that use (B, EM) and (B, M_w, EM) as training data. It is therefore conclusive that the models are not over-fitting to the new mutants, and losing their ability to generalise.

An analysis is also done that investigates when the sequential (LSTM) and non-sequential (Naïve Bayes) methods agree on which instances are misclassified. For each method, a comparison is made of overlap between the set of mis-classified instances produced by each method in order to understand whether both methods fail on the same instances or not (where *overlap* refers to the number of instances that are mis-classified by *both* methods). The exact instances that are mis-classified are given in [19].

From Table 6.8, it can be seen that using the M_w set during training results in a large number of mis-classified instances by both methods, and the overlap is high (10 instances, representing approximately 56% and 71% of mis-classified instances for Naïve Bayes and LSTM respectively). All the overlapping mis-classified instances come from the Dougalek family. For the EM training set, the overlap is high, in that it represents 40%

Table 6.8 Mis-classified instances on the unseen test set EM_u using both Naïve Bayes (NB) and LSTM models trained on 3 different training sets. The final column notes which families the overlapping instances (samples that are mis-classified by *both* methods) came from. The feature vector data is available in [19]

Training Data	Detection Accuracy of NB	Detection Accuracy of LSTM	# Mis-classified		Overlap	Instances
			NB	LSTM		
B **and** M_w	0.4	0.53	18	14	10	Dougalek(10)
B **and** EM	0.84	0.9	5	3	2	Dougalek(1), Droid-kungfu(1)
B, M_w **and** EM	0.82	0.62	6	11	2	Dougalek(1), Droid-kungfu(1)

of the instances mis-classified by Naïve Bayes and 66% of those mis-classified by the LSTM. Of the 2 overlapping instances, one is from the Dougalek family and the other from Droidkungfu. Finally, for the training set that includes M_w, EM, although the overlap is also 2 instances, this represents a smaller proportion of the instances mis-classified by each method (approximately 33% and 18% respectively). Again, the overlapping instances are from the Dougalek and Droidkungfu families.

It can be noted that the aforementioned overlap shows that using both features describing the frequency of system-calls made by a sample, and using the ordered sequence of system calls directly, the ML models (Naïve Bayes and LSTM) seem to agree often on the mis-classified instances when the M_w and EM data sets are used for training. This does not hold true for the training set that includes M_w, EM, as a much lower overlap is observed between the ML models.

6.3.5 Multi-class versus Binary-class models in Metamorphic Malware Detection

The experiments previously conducted were done using binary classifiers. This means that there were only two classes considered in the dataset, benign or malicious instances. It is noticed from the last line of Table 6.6, that when benign samples are combined with the malware from the web and the evolved mutants, there is a significant drop in the accuracy of the classifier. This might be as a result of the amount of noise in the data, given that different sources of data are considered. Hence, experiments are conducted to see if this can be improved upon by using multi-class classification which comprises data

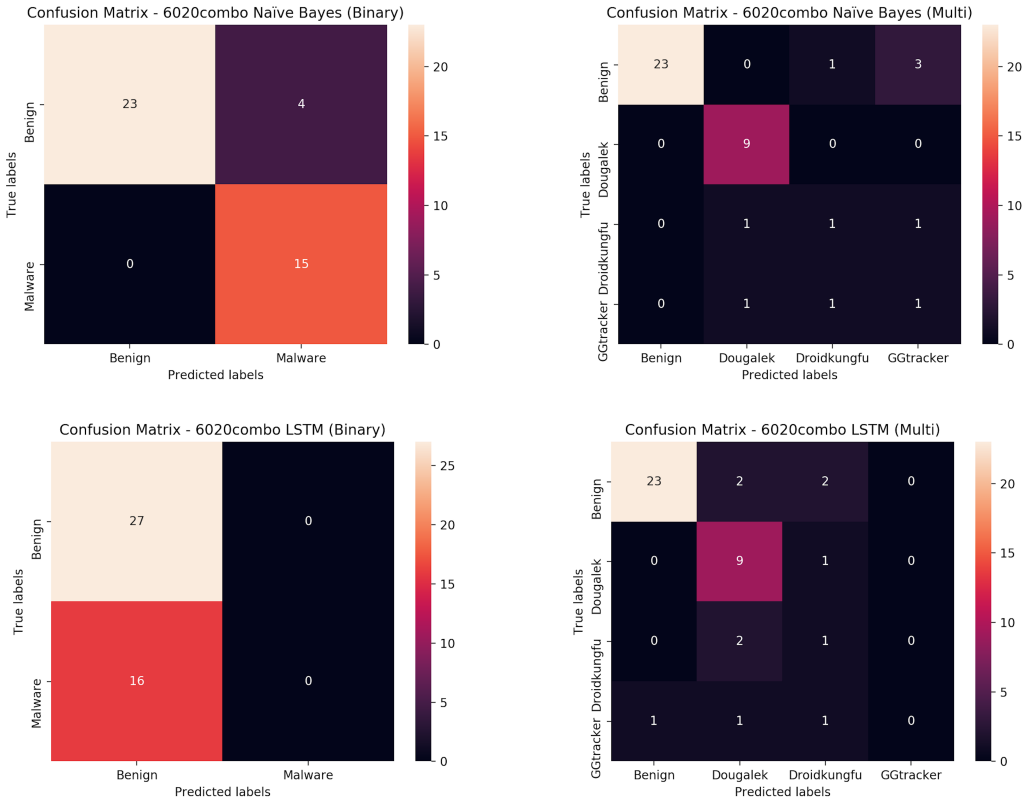


Fig. 6.4 Confusion matrices for the *6020combo* for both Naïve Bayes and LSTM models for both binary and multiclass classification

with multiple classes. In answering this research question, the *6020combo* and *6050combo* data sets described in Section 6.2.2, are used.

The results are given in Tables 6.9 and 6.10. It can be seen that for the *6020combo* and for the Naïve Bayes model, binary classification outperforms the multi-class classification. However, for the LSTM model, the multi-class model does better than the binary one. It can also be seen that the Naïve Bayes model does better than the LSTM model for both binary and multi-class classification.

From Table 6.10, it can be seen that for the *6050combo* and for the Naïve Bayes model, the binary classification outperforms the multi-class classification. However, for the LSTM model, again the multi-class model does better than the binary one. It can also be seen that the Naïve Bayes model does better than the LSTM model for both binary and multi-class classification. It is noteworthy that the LSTM model's performance is quite heavily dependent on the hyper-parameters and this is a common issue with deep learning models [126]. If the optimal set of hyper-parameters are not found, it impacts more on the performance of deep learning models like LSTM than for models such as Naïve Bayes which do not have hyper-parameters and so can be easily optimised.

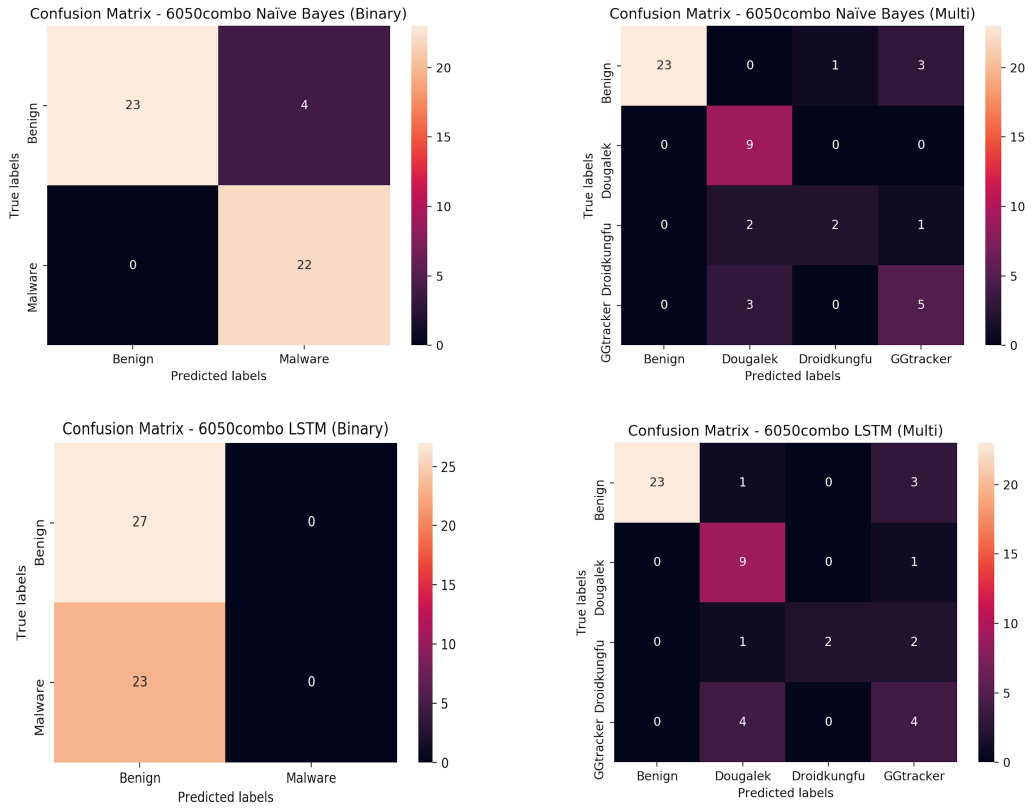


Fig. 6.5 Confusion matrices for the *6050combo* for both Naïve Bayes and LSTM models for both binary and multiclass classification

A look at the confusion matrices of both the *6020combo* and *6050combo* models give more insight to the classifiers. It can be seen from Fig. 6.4, the illustration for the *6020combo*, that the first set of confusion matrices are for the Naïve Bayes model, for the binary model it can be seen that it classifies 23 of the benign instances correctly and 4 of the benign instances incorrectly, However for the malicious instances, it classifies all of the instances correctly. For the multi-class classification however, 23 of the benign instances are classified correctly, 1 instance is misclassified as belonging to the Droidkungfu family while 3 instances are misclassified as belonging to the GGtracker family. It classifies all the Dougalek family correctly. However, it classifies only one of the Droidkungfu instances correctly and misclassifies two of the Droidkungfu instance as belonging to the Dougalek and GGtracker families. Similar observations are seen for the GGtracker family, it classifies only one of the GGtracker instances correctly and misclassifies two of the GGtracker instances as belonging to the Dougalek and Droidkungfu families.

For the LSTM model, for the binary classifier, it classifies all the 43 instances as benign instances (27 instances are classified correctly while 16 malware instances are misclassified as benign instances). For the multi-class classification, 23 of the benign instances are classified correctly, 2 instances are misclassified as belonging to the Dougalek family while 2 instances are misclassified as belonging to the Droidkungfu family. It classifies 9 of the Dougalek family correctly and misclassifies 1 instance as belonging to the Droidkungfu family. However, it classifies only one of the Droidkungfu instance correctly and misclassifies two of the Droidkungfu instance as belonging to the Dougalek family. For the GGtracker family, it classifies none of the GGtracker instances correctly and misclassifies 1 of the GGtracker instances as belonging to the benign family, 1 instance is misclassified as belonging to the Dougalek family while 1 instance is misclassified as belonging to the Droidkungfu family.

For the *6050combo*, it can be observed from Fig. 6.5, that like the confusion matrix for the *6020combo*, the first set of confusion matrices are for the Naïve Bayes model, for the binary model it can be seen that it classifies 23 of the benign instances correctly and 4 of the benign instances incorrectly, however for the malicious instances, it classifies all of the instances correctly. For the multi-class classification however, 23 of the benign instances are classified correctly, 1 instance is misclassified as belonging to the Droidkungfu family while 3 instances are misclassified as belonging to the GGtracker family. It classifies all the Dougalek family correctly. However, it classifies two of the Droidkungfu instances correctly and misclassifies three of the Droidkungfu instance as belonging to the Dougalek (2) and GGtracker (1) families. For the GGtracker family, it classifies five of the GGtracker instances correctly and misclassifies three of the GGtracker instances as belonging to the Dougalek family.

For the LSTM model, for the binary classifier, it classifies all the 50 instances as benign instances (27 instances are classified correctly while 23 malware instances are misclassified as benign instances). For the multi-class classification, 23 of the benign instances are classified correctly, 1 instance is misclassified as belonging to the Dougalek family while 3 instances are misclassified as belonging to the GGtracker family. It classifies 9 of the Dougalek family correctly and misclassifies only 1 instance as belonging to the GGtracker family. It classifies two of the Droidkungfu instances correctly and misclassifies three of the Droidkungfu instances as belonging to the Dougalek (1) and GGtracker (2) families. For the GGtracker family, it classifies four of the GGtracker instances correctly and misclassifies 4 of the GGtracker instances as belonging to the Dougalek family.

It can be observed from Figs. 6.4 and 6.5 that the models are generally good at classifying the benign samples. Also, for the binary classification, though the Naïve Bayes model is good at distinguishing between the benign and malicious instances, the LSTM models struggle to distinguish between the benign and malicious instances (they misclassify all the malicious instances as being benign for both the *6020* and *6050* data sets). For the multi-class classification, though the binary classification does better for the Naïve Bayes model for both the *6020* and *6050* dataset, the multi-class classifiers are still generally good at classifying the various classes for both data sets. For the LSTM models on the other hand, the use of multi-class classification improves the classification accuracy for both the *6020* and *6050* data sets and outperforms the binary classifiers. The multi-class classifiers are more capable of distinguishing between the various malicious groups and the benign instances. The misclassified instances for the various malicious groups by the multi-class classification, emanate from the fact that a number of these groups have overlapping functions (for instance the Dougalek and GGtracker families both steal personal information from mobile phone users) as such their system call vectors are similar making it more difficult for the classifier to distinguish between them.

It can be seen from the parallel coordinates plot (a visualisation tool for highly dimensional data which shows the relationship between features [75]) in Fig. 6.6, which is used to explain why the binary classifiers struggle to distinguish between the benign and malicious instances for both the *6020* and *6050* data sets for the LSTM model (that is, using the time ordered sequence of system calls), that although it can be seen that the benign samples are blue and the malicious samples are red for both data sets, there still seems to be an overlap in the plot between the benign and malicious groups which is possibly the reason why the LSTM model struggles to distinguish between the benign and malicious instances. Note that the x-axis of the parallel coordinates plot is the variable which in this instance is the system call ID. The y-axis of the parallel coordinates plot

Table 6.9 Comparison of accuracy obtained on the test set for *6020combo* using both Naïve Bayes and LSTM models for both binary and multi-class classification

Models	Binary	Multiclass
NB	0.91	0.81
LSTM	0.63	0.77

Table 6.10 Comparison of accuracy obtained on the test set for *6050combo* using both Naïve Bayes and LSTM models for both binary and multi-class classification

Models	Binary	Multiclass
NB	0.92	0.8
LSTM	0.54	0.76

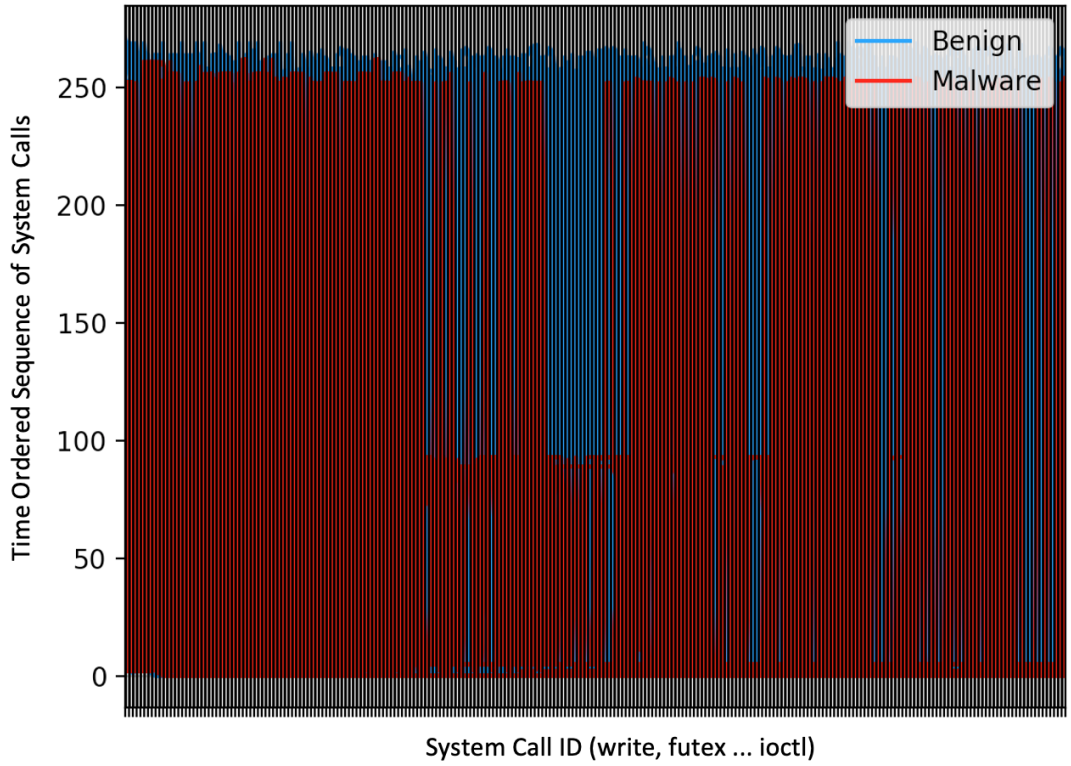
usually represents the coordinate value which in this case is the time ordered sequence of system calls.

6.3.6 Improving Classification Performance using a Transformer — BERT (pretrained on large NLP data sets) using the Evolved Mutants

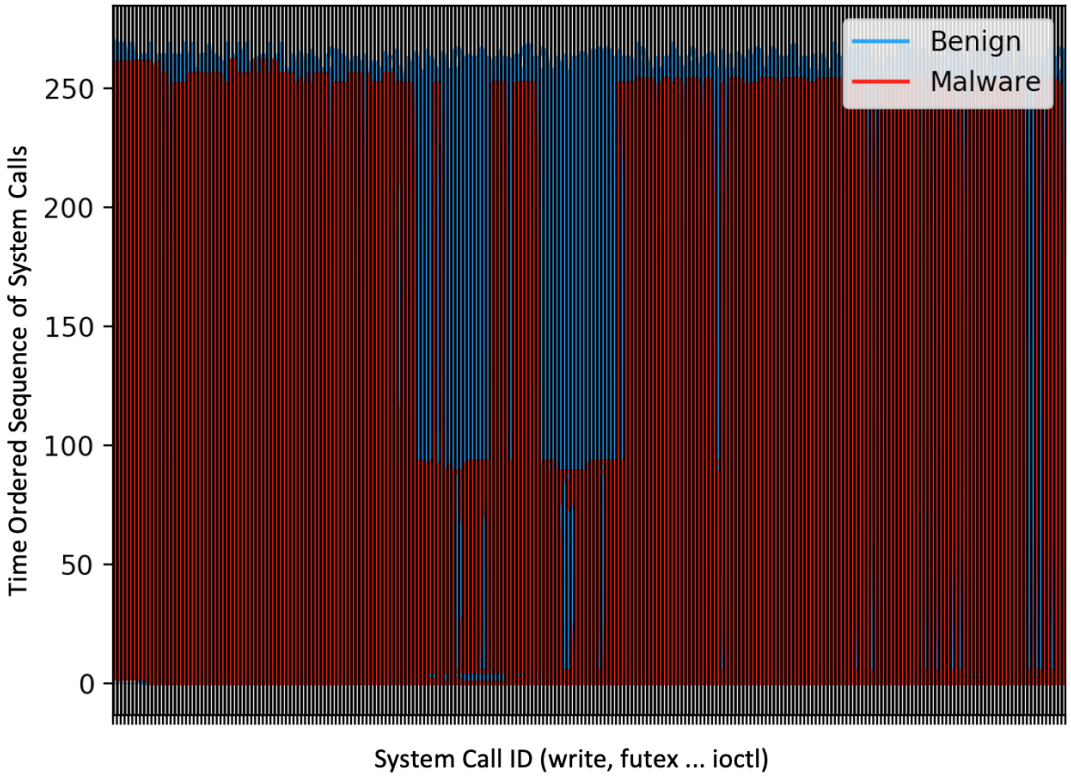
In this section of the chapter, the research question, "Can a transformer — BERT (that has been trained on large NLP datasets), be used in a transfer learning context to improve classification of metamorphic malware using the evolved samples?", is answered. The results are given in Tables 6.11 and 6.12. It is observed if the use of a transformer — BERT can improve the ML model's classification performance. The performance of the Naïve Bayes, LSTM and BERT models are compared using the *6020combo* and the *6050combo* data sets.

From the results given in Tables 6.11 and 6.12, it is seen that for the *6020combo*, BERT performs the best for the binary classification with an accuracy of 93%, an improvement to the Naïve Bayes model (91%) and the LSTM model (63%). For the multi-class classification however, the Naïve Bayes model performs the best with an accuracy of 81% followed closely by the BERT and LSTM model which have the same accuracy of 77%.

For the *6050combo*, the binary classification results show that both BERT model and the Naïve Bayes model perform really well with the Naïve Bayes model producing an accuracy of 92% which is higher than the BERT model's 90% accuracy. The LSTM model performs the least with a 54% accuracy. The results of the multi-class classification however, has BERT performing the least with a 50% accuracy as opposed to an 80% accuracy for the Naïve Bayes model and a 76% accuracy for the LSTM model.



(a) Parallel Coordinates for the *6020combo* for binary classification



(b) Parallel Coordinates for the *6050combo* for binary classification

Fig. 6.6 Parallel Coordinates for the *6020combo* and *6050combo* for binary classification

Table 6.11 Comparison of accuracy obtained on the test set for *6020combo* for the Naïve Bayes, LSTM and BERT models for both binary and multi-class classification

Models	Binary	Multiclass
NB	0.91	0.81
LSTM	0.63	0.77
BERT	0.93	0.77

Table 6.12 Comparison of accuracy obtained on the test set for *6050combo* for the Naïve Bayes, LSTM and BERT models for both binary and multi-class classification

Models	Binary	Multiclass
NB	0.92	0.8
LSTM	0.54	0.76
BERT	0.9	0.5

6.4 Summary and Conclusion

6.4.1 Summary

The following comments are provided to summarise all the results obtained in this chapter of the thesis:

- It was shown that models trained only on existing samples of malware are vulnerable to potential mutants.
- The results also provide evidence that show that ML models can be improved by augmenting training data with evolved mutants.
- It was observed that a sequential model (LSTM) outperforms the best classical model (Naïve Bayes) in terms of accuracy in some test scenarios.
- The results also show that models trained on evolved data representing future mutants retain their ability to recognise existing malware and do not over-fit to the new mutants.
- It was shown that multi-class models do not always improve the classification of metamorphic malware as compared to binary models.
- It was observed that the use of a pretrained NLP model — BERT in a transfer learning setting, leads to an improved classification of metamorphic malware in some test scenarios.

Answering the Research Questions

In answering the first research question of this chapter, "To what extent are models trained only on existing samples of known malware capable of detecting potential mutants?", experiments were carried out using existing malicious samples from the web. It was shown that models trained on existing malicious samples are susceptible to potential malware mutants. This was addressed by augmenting training data with a diverse set of evolved mutant samples with results showing improved classification performance. This also answered the second research question, "Can these models be improved by augmenting the training set with evolved samples, and if so, what is the most appropriate method of combining the evolved samples with existing samples?"

By carrying out additional experiments where the models trained using evolved samples were tested on a set of unseen malicious samples from the web, the third research question, "Do models trained on evolved data representing future mutants retain their ability to recognise existing known malware?", was answered. The results showed that the models do not over-fit to the novel mutants as they were still able to detect existing malware.

The fourth research question, "Do multi-class models improve the classification of metamorphic malware than binary models?" was answered by comparing the classification performance of the multi-class and binary models. It was shown that the binary models mostly outperform the multi-class models in classifying the evolved mutants. Finally, the results showed that the use of BERT, an NLP model pretrained on large NLP data sets, leads to an improved classification of metamorphic malware in some test scenarios. This answered the last research question, "Can a transformer — BERT (that has been trained on large Natural Language Processing (NLP) datasets), be used in a transfer learning context to improve classification of metamorphic malware using the evolved samples?"

6.4.2 Conclusion

In this chapter, an attempt was made to address the problem of metamorphic malware detection through a method that combines evolutionary computing and machine-learning. The former evolves a large set of malicious mutant variants of malware; the latter then uses this data to augment training sets, to develop models that are capable of recognising both existing malware and its future variants.

It was shown experimentally that machine learning models trained on existing malicious samples are vulnerable to potential mutants, and that training with evolved data addresses this. In addition to evaluating classical feature-based ML models, an LSTM using a time-ordered sequence of system calls as input, was applied and it was shown that this can outperform the classical models in some cases. Also, the results obtained

display that the models generalise over both existing malware and the evolved variants, and that binary models mostly outperform the multi-class models in classifying the evolved mutants. Finally, it was shown that the use of a pretrained NLP model — BERT in a transfer learning setting results in improved classification performance of the ML model in some test scenarios.

Chapter 7

Conclusion and Future Work

7.1 Introduction

The work presented in this thesis provides significant contribution within the field of cyber-security and provides evidence that suggest EAs are effective at searching for malicious variants of code, outperforming random search. These contributions are summarised in this chapter. Some of the limitations encountered while carrying out this study are also highlighted. Finally, prospective areas of future research are suggested.

7.2 Review of Research Questions

In this section, the answers to the research questions presented in Chapter 1 of the thesis are provided.

RQ 1: *To what extent can evasive and diverse variants of malware be generated, using an EA whose fitness function evolves for specific characteristics of solutions by running the EA multiple times to get a repertoire of solutions?*

It has been established throughout this thesis that metamorphic malware are a dangerous class of malware that obfuscates their code stochastically. Hence, they evade detection by a number of machine learning detectors. The major problem is caused by the inability of these detectors to determine where this malware will morph to due to insufficient training data. In order to address this problem as well as answer the first research question, an EA (MAL_EA) guided by three different fitness measures: the evasiveness of the variants (DR(x)), and their behavioural (BS(x)) and structural (SS(x)) similarity to the original malware, is employed. MAL_EA had to be run multiple times to get a repertoire of solutions.

An analysis of the influence of fitness function on evasiveness of the evolved mutants, was carried out. It was observed that for each of the fitness measures, the resulting variants were more evasive than the original malware. This was observed for all three malware classes (Dougalek, Droidkungfu and GGtracker).

For the Dougalek family for instance, 40.3% of the detection engines failed to detect the original malware while for the best evolved variants of Dougalek, 72%, 66.7% and 67.3% of engines failed to detect the new variants created using functions DR(x), BS(x) and SS(x) respectively. It can also be seen that for the Droidkungfu family, 65% of the detection engines failed to detect the original malware while for the best evolved variants, 94%, 82.1% and 83% of engines failed to detect the new variants created using functions DR(x), BS(x) and SS(x) respectively. For the GGtracker family, it can be observed that only 38.3% of the detection engines failed to detect the original malware while for the best evolved variants, 73.3%, 62.1% and 62.1% of engines failed to detect the new variants created using functions DR(x), BS(x) and SS(x) respectively.

It can also be noted that the fitness function that resulted in the most evasive variants for the best evolved variants for all malware families considered is the function DR(x). This is obviously because, this fitness function is optimised to produce evasive variants. The function SS(x) is the second best at producing evasive variants for the Dougalek and Droidkungfu families. However for the GGtracker family, function SS(x) ties with function BS(x) as the second best at producing evasive variants. The function BS(x) however, is the least of the three functions for producing evasive variants for all malware families considered.

It is also interesting to note that even when not evolving directly for evasive variants (i.e. using fitness function DR(x)), evasive variants can still be created using fitness functions BS(x) and SS(x).

Also, after carrying out an analysis of evasion characteristics for the three fitness functions for the three classes of malware studied it was discovered that for all malware families, there were a number of engines which were fooled by the mutants which were not fooled by the original malware with some engines like AhnLab-v3 which was neither fooled by the mutants nor the original malware for the three malware families considered. For Dougalek family for instance, for fitness function DR(x), 14 engines were not fooled by any of the mutants, some of such engines include AVG and Tencent. 17 engines on the other hand, were fooled by all the mutants with examples like AVware and McAfee. The number of engines not fooled by any variant when fitness function BS(x) is used, is 19. Examples include AVG and Fortinet. 11 engines however, were fooled by all the mutants for function BS(x)

with examples like GData and McAfee. Also, 16 engines were not fooled by any variants for the function SS(x). Examples include Symantec Mobile and Fortinet. 17 engines however were fooled 100% of the time for function SS(x) - examples include McAfee-GW and BitDefender. For Droidkungfu family, 3 engines were not fooled by any of the mutants for function DR(x). Examples include Avast Mobile and NANO Antivirus. 12 engines however, were fooled by all the mutants 100% of the time for function DR(x) - examples include Fortinet and Kaspersky. For fitness function BS(x), 6 engines were not fooled by any of the mutants, some of such engines include Avast Mobile and NANO Antivirus. 7 engines on the other hand, were fooled by all the mutants with examples like Symantec Mobile and Tencent. Furthermore, 7 engines were not fooled by any mutant for function SS(x) with examples such as AVG and Cyren. On the other hand, 11 engines were fooled by all the mutants for function SS(x). Examples include Kaspersky and ZoneAlarm. Furthermore for the GGtracker family, the number of engines not fooled by any variant when fitness function DR(x) was used, is 9. Examples include CAT-QuickHeal and DrWeb. 13 engines however, were fooled by all the mutants for function DR(x) with examples like Arcabit and BitDefender. Also, 15 engines were not fooled by any variants for function BS(x). Examples include K7GW and Kaspersky. 16 engines however were fooled all the time for function BS(x) - examples include AVware and TrendMicro. In addition, 16 engines were not fooled by any mutant for function SS(x) with examples such as Avast and AVG. On the other hand, 18 engines were fooled by all the mutants for function SS(x). Examples include Cyren and McAfee.

In order to measure the diversity of the variants produced using the three fitness functions for the three classes of malware, recall that diversity in terms of detection signature, behavioural signature and structural similarity, was compared. It is evident that in terms of uniqueness of the detection signatures, the highest percentage of unique variants were found in the Droidkungfu family when evolving for evasive (with 90% uniqueness) and behaviourally dissimilar (with 89% uniqueness) variants. However, when evolving for structurally dissimilar variants, the most unique variants were found in the Dougalek family (with a 50% uniqueness score). In terms of the behavioural signature, Dougalek was the most unique family with 100% uniqueness when evolving for evasive and behaviourally dissimilar variants. GGtracker on the other hand, was the most unique family in terms of the evolution for structurally dissimilar variants.

When comparing the diversity obtained in terms of behavioural and detection signatures of the evolved mutants, it was observed that there was more diversity in

terms of the behavioural signatures than detection signatures, but that diversity is apparent with respect to both signatures.

For structural similarity, it can be noted that the fitness function that produced the most structurally diverse variants was function SS(x) for GGtracker family, function BS(x) for Droidkungfu family and function SS(x) for Dougalek family. It is interesting to note that the least structurally diverse variants were produced by the Dougalek family and the most structurally diverse variants were produced by the Droidkungfu family.

RQ2: *How can a Quality Diversity (QD) algorithm (MAP-Elites) be used to generate variants of malware which are both diverse, with respect to multiple characteristics and evasive in a single run?*

As mentioned while answering the first research question, MAL_EA used in RQ1, had to be run multiple times to get a repertoire of solutions. To address the second research question, an EA - QD algorithm (MAP-Elites) was employed to generate solutions which were both diverse with respect to multiple characteristics and evasive in a single run. This was found to be the first use of a QD algorithm in the cyber security domain. The solutions were diverse with respect to their behavioural and structural similarity to the original malware.

A qualitative and quantitative comparison of MAL_EA described in RQ1 and MAP-Elites, was carried out. It was shown experimentally that MAP-Elites produces a large archive of solutions that are evenly distributed along the range of each feature (that is, behavioural and structural similarity to the original malware). Also, MAP-Elites does better than MAL_EA in terms of its coverage of the feature space and the reliability of the algorithm in terms of its ability to consistently locate solutions found in the performance map and finds equally good solutions for the performance metrics, and the solutions derived are diverse while being evasive.

RQ 3: *How does augmenting training data with the novel variants influence classic classification algorithms in Machine Learning? Which Machine Learning approach benefits most from augmenting with data from the novel variants?*

Machine learning models have been shown to perform poorly when classifying metamorphic malware due to insufficient training data. While answering the first two research questions two EAs (MAL_EA and MAP-Elites) were used to create mutant variants of malware that can serve as training data to improve the classification of metamorphic malware. In answering the third research question, the generated mutant variants of malware were used to augment existing data-sets so as to see if this improves the classification of metamorphic malware.

The experimental results show that models trained only on existing samples of malware are vulnerable to potential mutants. The results also provide evidence that show that using our generated mutant variants of malware improves the classification performance of ML models. It was also discovered that in some instances sequential model (LSTM) is better in terms of accuracy than the best classical model (Naïve Bayes) studied. It is also important to note that models trained on evolved data representing future mutants retained their ability to recognise existing malware and do not over-fit to the new mutants. Finally, in the problem instance, multi-class models did not always improve the classification of metamorphic malware as compared to binary models.

RQ 4: *How can one benefit from the use of a transformer — BERT (that has been trained on large Natural Language Processing (NLP) datasets), in a transfer learning setting to improve classification of metamorphic malware using the novel variants?*

It has been established in the previous chapters of the thesis, that one challenge with detecting metamorphic malware is the fact that there is insufficient training data and as such, ML models do not generalise over this class of malware as they can not predict where these malware will morph to.

To address this, the generation of mutant samples of malware to serve as training data for ML models was explored in Chapters 4 and 5, which research questions RQ1 and RQ2 sought to answer. The evolved samples were seen to improve classification of metamorphic malware as demonstrated in Chapter 6, which answered the third research question (RQ3).

However in answering this research question (RQ4) and to also address the problem of limited training data for ML models to detect metamorphic malware, the use of a model — BERT, that requires limited training samples as it has been pretrained on large NLP data sets, was explored, with this knowledge transferred to observe if this results in an improved classification of metamorphic malware using the evolved samples as part of the training data. The results showed that the use of BERT led to an improved classification of metamorphic malware in some test scenarios particularly performing well for binary classification.

7.3 Generalisability of the Research Work

The research work provides a framework and methodology for the generation of evasive and diverse mutants of malware which serve as rich sources of training data for ML models. Recall from Chapter 3 of thesis that the framework is comprised of five functional

modules, namely, a data source (i.e., a mobile malware dump), a disassembly tool (i.e., apktool), a mutation engine, a data store for APK variants, and a malware detector. The framework can be generalised and its various components can be adapted for various applications. The data source for instance comprised of a mobile malware dump i.e. Android malware source. This could be generalised to other kinds of malware depending on the platform (such as desktop based malware) or operating system (such as iOS malware) in which the malware are executed. This also applies to the disassembly tool i.e. apktool. Apktool was used as the disassembly tool because Android samples i.e. apk files were used in this research. Other disassembly tools could be used depending on the platform the malware is run such as the use of Portable Executable (PE) Viewers (such as CFF Explorer[1]) or Network Analysers (e.g. Wireshark[2]) for desktop based malware. The mutation engine module uses an EA to generate novel malware mutants. This module treats the disassembled malware as a piece of code and creates/uses mutation operators to transform the code in order to create mutants. In other words, as far as the disassembled malware can be represented as a code it can be generalised but the EA operators need to be tailored/customised depending on the kind of code being studied. The transformation of the malicious binaries to create diverse mutants was done based on characteristics of malware such as its structure, behaviour and ability to evade existing detection engines. Other characteristics of malware could also be used to guide the EA towards finding malicious and diverse mutants. The malware detector module uses the mutants created by the mutation engine module to train ML models in order to provide improved protection against future mutants. This module does not develop new ML models but uses standard ML models whose complexities are known and referenced in Section 6.2.4 of the thesis. The detector module uses some feature based and sequence based ML models. Again, other ML models could be used by the detector module.

It is important to note that due to the complex fitness function used by the EAs to create the mutant variants of malware, this required a lot of computational time. For a complete experimental run of 100 iterations (with each iteration potentially producing one mutant depending on the executability of the mutant created), it took between 150 and 350 minutes depending of the fitness function used. For some other applications requiring the generation of more data, this will mean more computational time. This could be solved by parallelisation [79] wherein multiple applications are run on several cores to speed up computational performance. It is noteworthy that EAs are very amenable to parallelisation [38]. In addition, cloud computing solutions could be used to scale up and speed up the computational performance of applications requiring the generation of large dataset.

[1]CFF Explorer - https://ntcore.com/?page$_id = 388$

[2]Wireshark - https://www.wireshark.org/

7.4 Summary of Contribution and Limitation of Studies

The following are the main contributions of this thesis. In this thesis, two methods of generating mutant variants of malware that serve as training data to improve detection by ML models, using EAs, are proposed. In the first, the fitness mechanism is used to control diversity by evolving for specific characteristics of solutions (that is solutions that are as behaviourally and structurally dissimilar to the original malware as possible while being as evasive or more evasive than the original malware) and the first EA needed to be run multiple times to get a repertoire of solutions. In the second instance, the EA uses a QD algorithm to generate solutions which are both diverse with respect to multiple characteristics (structural and behaviour similarity to the original malware) and evasive in a single run. In both instances, solutions which were evasive and diverse were created. Previous approaches suffered from lack of diverse training data which ML models could learn from. The two EAs previously described were tested on three malware families, namely, Dougalek, Droidkungfu and GGtracker.

The second EA (that is the one that uses a QD algorithm) produces larger sets of data with more diversity for all three malware families as compared to the first EA while still producing comparable performance in terms of minimising detection rates. It can be noted that upon analysing the evasion characteristics for the three classes of malware, it can be observed that for the first EA for all malware families, there were a number of engines that were fooled by the mutants that were not fooled by the original malware. However, some engines like AhnLab-v3 were neither fooled by the mutants nor the original malware for the three malware families considered. For the second EA however, it can be noticed that for the three malware classes, the evolved mutants were also able to evade more engines than their parent malware. It can also be seen that engines such as McAfeeGW were more susceptible to the Dougalek and Droidkungfu families while engines like Avira was robust to all families of malware mutants in all of the analysis.

It was also shown that machine learning models trained on existing malicious samples are vulnerable to potential mutants, and that training with evolved data addresses this. Furthermore, it was shown that the models generalise over both existing malware and the evolved variants, and that binary models mostly outperform the multi-class models in classifying the evolved mutants. It was also noted that by using BERT model (an NLP language model pretrained on large NLP data sets) in a transfer learning context, the classification of metamorphic malware was improved in some instances.

In this work, the major drawback encountered was the computational time used in generating the mutant samples given the complex fitness function used. This was partly solved by using the BERT model (a model that is suited for problem instances with small or limited data-set) which meant that a smaller data-set could be used without having to

generate large dataset which would have been computationally demanding. A GPU or TPU could have been used to run our experiments to speed up this process, however this was not used because for most of the experiment the focus was on the solution quality and not computational cost.

7.5 Possible Future Work

There is still scope for improving the EAs used in this research work, for example, in developing further mutation operators or methods of crossover that will result in runnable APKs; further tuning of the algorithm parameters is also likely to yield improvements.

In this thesis, it was shown, using two types of EAs, how mutant variants of malware could be created. This will potentially yield more interesting results, particularly if they can be combined in a setting typical in Generative Adversarial Networks (GAN [62]) in which improvements in the generated samples drive improvements in the detection method and vice versa. This may further improve the classification performance of metamorphic malware.

Also, MAP-Elites algorithm, a QD algorithm, was used in this work to generate an archive of diverse and evasive mutant variants of malware. There remains room for exploring other QD algorithms such as Novelty Search with Local Competition (NSLC) as well as multi-objective MAP-Elites to see if they outperform the standard MAP-Elites algorithm.

7.6 General Conclusions

The areas presented in this thesis span across several areas in the application of EC in Metamorphic malware analysis and detection. The results presented with the use of EC methods in metamorphic malware analysis show the promise EC has in this field and the cybersecurity domain. Although the contributions of the research are more beneficial to the cybersecurity community given that the work uses “standard” EA methods to create a rich source of training data which was shown to improve the classification of metamorphic malware, nevertheless, it is also beneficial to the EC community in that it provides evidence that the operators in EC are useful in cybersecurity applications.

This work presents the first ever use of a QD algorithm in metamorphic malware analysis and detection as well as in the cyber security domain at large. It shows that EAs can be used in generating diverse yet evasive solutions to serve as training data to improve metamorphic malware analysis and detection.

Also, it was shown that current ML models are not capable of detecting novel malware mutants, and training these models with mutant variants of malware addresses this. It was recognised that there is still room for future work such as in the use of the two methods of generating mutant samples in combination with a Generative Adversarial Networks (GAN), and the exploration of other QD algorithms, genetic operators, among others. However, it is believed that creating novel mutant samples of malware and using them as training data both in typical ML models or in a transfer learning context, presents steps in the right direction towards addressing challenges currently faced in metamorphic malware analysis and detection.

References

[1] APKSIGNER, 2018. URL https://developer.android.com/studio/command-line/apksigner.

[2] APKTOOL, 2018. URL http://ibotpeaches.github.io/Apktool.

[3] Run apps on the Android Emulator, 2018. URL https://developer.android.com/studio/run/emulator.

[4] ANDROID STUDIO, 2018. URL https://developer.android.com/studio.

[5] ANDROID STUDIO FEATURES, 2018. URL https://developer.android.com/studio/features.

[6] ZIPALIGN, 2018. URL https://developer.android.com/studio/command-line/zipalign.

[7] Vidhya. K. A and G. Aghila. A survey of naïve bayes machine learning approach in text document classification, 2010.

[8] Shahid Alam, Issa Traore, and Ibrahim Sogukpinar. Current trends and the future of metamorphic malware detection. In *Proceedings of the 7th International Conference on Security of Information and Networks*, SIN '14, pages 411–416, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3033-6.

[9] A. Alvarez, J. M. M. Font Fernandez, S. Dahlskog, and J. Togelius. Interactive constrained map-elites: Analysis and evaluation of the expressiveness of the feature dimensions. *IEEE Transactions on Games*, pages 1–1, 2020. doi: 10.1109/TG.2020.3046133.

[10] P.R.L. Alves, L.G.S. Duarte, and L.A.C.P. da Mota. Detecting chaos and predicting in dow jones index. *Chaos, Solitons Fractals*, 110:232 – 238, 2018. ISSN 0960-0779. doi: https://doi.org/10.1016/j.chaos.2018.03.034. URL http://www.sciencedirect.com/science/article/pii/S0960077918301322.

[11] Hyrum S Anderson, Bobby Filar, and Phil Roth. Evading Machine Learning Malware Detection. *BlackHat DC*, 2017.

[12] Android-Studio. UI/Application Exerciser Monkey. URL https://developer.android.com/studio/test/monkey.

[13] Apkpure. Spyster, 2016. URL https://apkpure.com/spyster/net.wappie.spyster.

[14] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh. A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation. *IEEE Access*, 6:12382–12394, 2018. doi: 10.1109/ACCESS.2018.2808340.

[15] Emre Aydogan and Sevil Sen. Automatic generation of mobile malwares using genetic programming. In Antonio M. Mora and Giovanni Squillero, editors, *Applications of Evolutionary Computation*, pages 745–756, Cham, 2015. Springer International Publishing. ISBN 978-3-319-16549-3.

[16] Kehinde O. Babaagba, Zhiyuan Tan, and Emma Hart. Nowhere metamorphic malware can hide - a biological evolution inspired detection scheme. In Guojun Wang, Md Zakirul Alam Bhuiyan, Sabrina De Capitani di Vimercati, and Yizhi Ren, editors, *Dependability in Sensor, Cloud, and Big Data Systems and Applications*, pages 369–382, Singapore, 2019. Springer Singapore. ISBN 978-981-15-1304-6.

[17] Kehinde O. Babaagba, Zhiyuan Tan, and Emma Hart. Automatic generation of adversarial metamorphic malware using map-elites. In Pedro A. Castillo, Juan Luis Jiménez Laredo, and Francisco Fernández de Vega, editors, *Applications of Evolutionary Computation*, pages 117–132, Cham, 2020. Springer International Publishing. ISBN 978-3-030-43722-0.

[18] Kehinde O Babaagba, Zhiyuan Tan, and Emma Hart. Improving classification of metamorphic malware by augmenting training data with a diverse set of evolved mutant samples. In *IEEE WORLD CONGRESS ON COMPUTATIONAL INTELLIGENCE (WCCI)*, Glasgow, UK, 2020. IEEE.

[19] Kehinde O. Babaagba, Zhiyuan Tan, and Emma Hart. Improving classification of metamorphic malware. https://github.com/KehindeOloye/Improving-Classification-of-Metamorphic-Malware.git, 2020.

[20] Wolfgang Banzhaf. Genetic Programming for Pedestrians. *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, page 628, 1993. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/GenProg_forPed.ps.Z.

[21] Babak Bashari Rad, Maslin Masrom, Suhaimi Ibrahim, and Subariah Ibrahim. Morphed Virus Family Classification Based on Opcodes Statistical Feature Using Decision Tree. In Azizah Abd Manaf, Akram Zeki, Mazdak Zamani, Suriayati Chuprat, and Eyas El-Qawasmeh, editors, *Informatics Engineering and Information Science*, pages 123–131, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-25327-0.

[22] Donabelle Baysa, Richard M. Low, and Mark Stamp. Structural entropy and metamorphic malware. *Journal in Computer Virology*, 9(4):179–192, 2013. ISSN 17729890.

[23] Nitin Bhatia and Vandana. Survey of nearest neighbor techniques, 2010.

[24] Niket Bhodia, Pratikkumar Prajapati, Fabio Di Troia, and Mark Stamp. Transfer learning for image-based malware classification. *CoRR*, abs/1903.11551, 2019. URL http://arxiv.org/abs/1903.11551.

[25] Concha Bielza and Pedro Larrañaga. Discrete bayesian network classifiers: A survey. *ACM Comput. Surv.*, 47(1), July 2014. ISSN 0360-0300. doi: 10.1145/2576868. URL https://doi.org/10.1145/2576868.

[26] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317 – 331, 2018. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2018.07.023. URL http://www.sciencedirect.com/science/article/pii/S0031320318302565.

[27] Jorge Blasco, Julio Hernandez-Castro, Juan Tapiador, Arturo Ribagorda, and Miguel Orellana-Quiros. Steganalysis of hydan. volume 297, pages 132–142, 05 2009. doi: 10.1007/978-3-642-01244-0_12.

[28] Markus F. Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Springer Science & Business Media, 2007.

[29] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. The Wadsworth statistics/probability series. Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, CA, 1984. URL https://cds.cern.ch/record/2253780.

[30] Michael Brückner, Christian Kanzow, and Tobias Scheffer. Static prediction games for adversarial learning problems. *J. Mach. Learn. Res.*, 13(1):2617–2654, September 2012. ISSN 1532-4435.

[31] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Matthias Hölzl, Alberto Lluch Lafuente, Andrea Vandin, and Martin Wirsing. *Reconciling White-Box and Black-Box Perspectives on Behavioral Self-adaptation*, pages 163–184. Springer International Publishing, Cham, 2015. ISBN 978-3-319-16310-9. doi: 10.1007/978-3-319-16310-9_4. URL https://doi.org/10.1007/978-3-319-16310-9{_}4.

[32] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, 2007. ISSN 15407993.

[33] Mayank Arya Chandra and S S Bedi. Survey on SVM and their application in image classification. *International Journal of Information Technology*, 2018. ISSN 2511-2112. doi: 10.1007/s41870-017-0080-1. URL https://doi.org/10.1007/s41870-017-0080-1.

[34] Li Chen. Deep transfer learning for static malware classification. *CoRR*, abs/1812.07606, 2018. URL http://arxiv.org/abs/1812.07606.

[35] Kim-Kwang Raymond Choo. Zombies and botnets. *Trends & Issues in Crime & Criminal Justice*, (333), 2007.

[36] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 32–46, 2005. ISBN 0769523390. doi: 10.1109/SP.2005.20.

[37] Brendan Cody-Kenny, Edgar Galván-López, and Stephen Barrett. locogp: Improving performance by genetic programming java source code. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 811–818, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3488-4.

[38] Péter Cserti, Szabolcs Szondi, Balázs Gaál, and István Vassányi. Game: Gpu accelerated multipurpose evolutionary algorithm library. *Int. J. Innov. Comput. Appl.*, 5(3):163–172, August 2013. ISSN 1751-648X. doi: 10.1504/IJICA.2013.055936. URL https://doi.org/10.1504/IJICA.2013.055936.

[39] A. Cully and Y. Demiris. Quality and diversity optimization: A unifying modular framework. *IEEE Transactions on Evolutionary Computation*, 22(2):245–259, April 2018. doi: 10.1109/TEVC.2017.2704781.

[40] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426, May 2013. doi: 10.1109/ICASSP.2013.6638293.

[41] Christopher Day. Chapter 72 - intrusion prevention and detection systems. In John R. Vacca, editor, *Computer and Information Security Handbook (Third Edition)*, pages 1011 – 1025. Morgan Kaufmann, Boston, third edition edition, 2013. ISBN 978-0-12-803843-7. doi: https://doi.org/10.1016/B978-0-12-803843-7.00072-7. URL http://www.sciencedirect.com/science/article/pii/B9780128038437000727.

[42] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

[43] A. Dhakal, A. Poudel, S. Pandey, S. Gaire, and H. P. Baral. Exploring deep learning in semantic question matching. In *2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS)*, pages 86–91, Oct 2018. doi: 10.1109/CCCS.2018.8586832.

[44] S. Di, H. Zhang, C. Li, X. Mei, D. Prokhorov, and H. Ling. Cross-domain traffic scene understanding: A dense correspondence-based transfer learning approach. *IEEE Transactions on Intelligent Transportation Systems*, 19(3):745–757, 2018.

[45] John Dolak. The Code Red Worm. 2001.

[46] E-Zest. Mobile platforms, frameworks & environments, 2018. URL http://www.e-zest.com/mobile-operating-system/.

[47] Kenneth S. Edge, Gary B. Lamont, and Richard A. Raines. A retrovirus inspired algorithm for virus detection & optimization. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 103–110, New York, NY, USA, 2006. ACM. ISBN 1-59593-186-4.

[48] Agoston Endre Eiben and Jim E. Smith. What is an Evolutionary Algorithm? In *Introduction to Evolutionary Computing*, pages 15–35. Springer Publishing Company, Incorporated, 2003. ISBN 978-3-642-07285-7.

[49] Pedro G. Espejo, Sebastián Ventura, and Francisco Herrera. A Survey on the Application of Genetic Programming to Classifcation. *Ieee Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 40(2):121–144, 2010. ISSN 1094-6977. doi: 10.1109/TSMCC.2009.2033566.

[50] F-secure. 2014 Mobile Threat Report. page 9, 2014.

[51] F-Secure. Trojan:Android/GGTracker.A, 2019. URL https://www.f-secure.com/v-descs/trojan{_}android{_}ggtracker.shtml.

[52] F-Secure. Trojan:Android/DroidKungFu.C, 2019.

[53] Yongsheng Fang and Jun Li. A review of tournament selection in genetic programming. In Zhihua Cai, Chengyu Hu, Zhuo Kang, and Yong Liu, editors, *Advances in Computation and Intelligence*, pages 181–192, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16493-4.

[54] M Flageat and A Cully. Fast and stable map-elites in noisy domains using deep grids. pages 273–282. Massachusetts Institute of Technology, 2020. doi: 10.1162/isal_a_00357. URL http://dx.doi.org/10.1162/isal_a_00357.

[55] David B Fogel. Advances in genetic programming : Kenneth {E}. Kinnear, Jr., (ed.), {MIT} Press, Cambridge, {MA}, 1994, 518 pp., $45.00. *Biosystems*, 36(1):82–85, 1995. ISSN 0303-2647. doi: doi:10.1016/0303-2647(95)90007-1. URL http://www.sciencedirect.com/science/article/B6T2K-4CHS0P6-5/2/2474f3669e7a25204939e72cbb4d7253.

[56] Matthew C. Fontaine, Scott Lee, L. B. Soros, Fernando De Mesentier Silva, Julian Togelius, and Amy K. Hoover. Mapping hearthstone deck spaces through map-elites with sliding boundaries. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 161–169, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6111-8.

[57] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 947–954, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-325-9.

[58] Carlos García-Martínez and Manuel Lozano. Local search based on genetic algorithms. In Patrick Siarry and Zbigniew Michalewicz, editors, *Advances in Metaheuristics for Hard Optimization*, pages 199–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-72960-0.

[59] Marko Gargenta. *Learning Android*. 2011. ISBN 9781449390501.

[60] Jorge Gomes, Pedro Mariano, and Anders Lyhne Christensen. Novelty-driven cooperative coevolution. *Evol. Comput.*, 25(2):275–307, June 2017. ISSN 1063-6560. doi: 10.1162/EVCO_a_00173. URL https://doi.org/10.1162/EVCO_a_00173.

[61] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages

2672–2680. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

[62] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[63] Alex Graves. *Supervised Sequence Labelling*, pages 5–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-24797-2. doi: 10.1007/978-3-642-24797-2_2. URL https://doi.org/10.1007/978-3-642-24797-2_2.

[64] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh. Automatic Generation of String Signatures for Malware Detection. In Engin Kirda, Somesh Jha, and Davide Balzarotti, editors, *Recent Advances in Intrusion Detection*, pages 101–120, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04342-0.

[65] Alexander Hagg, Alexander Asteroth, and Thomas Bäck. Modeling user selection in quality diversity. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 116–124, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6111-8.

[66] D. Heres. Source Code Plagiarism Detection using Machine Learning. (August), 2017. ISSN 1932-6203. doi: 10.1371/journal.pone.0101673. URL https://dspace.library.uu.nl/handle/1874/355678.

[67] Wael H.Gomaa and Aly A. Fahmy. A Survey of Text Similarity Approaches. *International Journal of Computer Applications*, 2013. doi: 10.5120/11638-7118.

[68] Julien I.E. Hoffman. Chapter 33 - logistic regression. In Julien I.E. Hoffman, editor, *Basic Biostatistics for Medical and Biomedical Practitioners (Second Edition)*, pages 581 – 589. Academic Press, 2 edition, 2019. ISBN 978-0-12-817084-7. doi: https://doi.org/10.1016/B978-0-12-817084-7.00033-4. URL http://www.sciencedirect.com/science/article/pii/B9780128170847000334.

[69] J H Holland. Adaptation in Natural and Artificial Systems. *Ann Arbor MI University of Michigan Press*, Ann Arbor:183, 1975. ISSN 10834419. doi: 10.1137/1018105. URL http://www.citeulike.org/group/664/article/400721.

[70] John H Holland. Genetic Algorithms and Adaptation. In Oliver G Selfridge, Edwina L Rissland, and Michael A Arbib, editors, *Adaptive Control of Ill-Defined Systems*, pages 317–333. Springer US, Boston, MA, 1984. ISBN 978-1-4684-8941-5. doi: 10.1007/978-1-4684-8941-5_21. URL https://doi.org/10.1007/978-1-4684-8941-5{_}21.

[71] Holger H. Hoos and Thomas Stützle. 8 - travelling salesman problems. In Holger H. Hoos and Thomas Stützle, editors, *Stochastic Local Search*, The Morgan Kaufmann Series in Artificial Intelligence, pages 357 – 416. Morgan Kaufmann, San Francisco, 2005. ISBN 978-1-55860-872-6. doi: https://doi.org/10.1016/B978-155860872-6/50025-1. URL http://www.sciencedirect.com/science/article/pii/B9781558608726500251.

[72] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on GAN. *CoRR*, abs/1702.05983, 2017. URL http://arxiv.org/abs/1702.05983.

[73] TaeHyun Hwang and Rui Kuang. *A Heterogeneous Label Propagation Algorithm for Disease Gene Discovery*, pages 583–594. doi: 10.1137/1.9781611972801.51. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611972801.51.

[74] L. Jiang, Z. Cai, D. Wang, and S. Jiang. Survey of improving k-nearest-neighbor for classification. In *Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007)*, volume 1, pages 679–683, 2007.

[75] Jimmy Johansson and Camilla Forsell. Evaluation of parallel coordinates: Overview, categorization and guidelines for future research. *IEEE Transactions on Visualization and Computer Graphics*, 22:1–1, 11 2015. doi: 10.1109/TVCG.2015.2466992.

[76] Myles Jordan. Dealing with metamorphism. *Virus Bulletin*, pages 4–6, 2002.

[77] Niels Justesen, Sebastian Risi, and Jean-Baptiste Mouret. Map-elites for noisy domains by adaptive sampling. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '19, pages 121–122, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6748-6. doi: 10.1145/3319619.3321904. URL http://doi.acm.org/10.1145/3319619.3321904.

[78] Hilmi Güne\cs Kayacik, Malcolm Heywood, and Nur Zincir-Heywood. On Evolving Buffer Overflow Attacks Using Genetic Programming. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1667–1674, New York, NY, USA, 2006. ACM. ISBN 1-59593-186-4.

[79] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011. doi: 10.1109/MM.2011.89.

[80] V. Kecman. *Support Vector Machines – An Introduction*, pages 1–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32384-6. doi: 10.1007/10984697_1. URL https://doi.org/10.1007/10984697_1.

[81] Jin-Young Kim, Seok-Jun Bu, and Sung-Bae Cho. Malware detection using deep transferred generative adversarial networks. In Derong Liu, Shengli Xie, Yuanqing Li, Dongbin Zhao, and El-Sayed M. El-Alfy, editors, *Neural Information Processing*, pages 556–564, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70087-8.

[82] Jin-Young Kim, Seok-Jun Bu, and Sung-Bae Cho. Zero-day malware detection using transferred generative adversarial networks based on deep autoencoders. *Information Sciences*, 460-461:83 – 102, 2018. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2018.04.092. URL http://www.sciencedirect.com/science/article/pii/S0020025518303475.

[83] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[84] Carl Kingsford and Steven L Salzberg. What are decision trees? *Nature Biotechnology*, 26(9):1011–1013, 2008. ISSN 1546-1696. doi: 10.1038/nbt0908-1011. URL https://doi.org/10.1038/nbt0908-1011.

[85] Evgenios Konstantinou. Metamorphic Virus: Analysis and Detection. Technical Report January, 2008. URL http://digirep.rhul.ac.uk/items/bde3a9fe-51c0-a19a-e04d-b324c0926a4a/1/.

[86] Evgenios Konstantinou. Metamorphic Virus: Analysis and Detection. Technical report, Department of Mathematics Royal Holloway, University of London, 2008.

[87] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. 1992. ISBN 0262111705.

[88] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[89] Arun Lakhotia and Prabhat K. Singh. Challenges in getting 'formal' with viruses. *Virus Bulletin*, pages 15–19, sep 2003.

[90] William B. Langdon and Mark Harman. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, 2015. ISSN 1089778X.

[91] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi. Photo-realistic single image super-resolution using a generative adversarial network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 105–114, 2017.

[92] Jared Lee, Thomas H Austin, and Mark Stamp. Compression-based Analysis of Metamorphic Malware. *International Journal of Security and Networks*, 10(2): 124–136, jul 2015. ISSN 1747-8405.

[93] Joel Lehman and Kenneth O. Stanley. Novelty Search and the Problem with Objectives. 2011. doi: 10.1007/978-1-4614-1770-5_3.

[94] Joel Lehman and Kenneth O. Stanley. Evolvability is inevitable: Increasing evolvability without the pressure to adapt. *PLOS ONE*, 8(4):1–9, 04 2013. doi: 10.1371/journal.pone.0062186. URL https://doi.org/10.1371/journal.pone.0062186.

[95] Linux Man Pages Strace. strace(1) - Linux manual page, 2017.

[96] Stan Lipovetsky. Pareto 80/20 law: derivation via random partitioning. *International Journal of Mathematical Education in Science and Technology*, 40(2): 271–277, 2009. doi: 10.1080/00207390802213609. URL https://doi.org/10.1080/00207390802213609.

[97] Zachary Chase Lipton. A critical review of recurrent neural networks for sequence learning. *ArXiv*, abs/1506.00019, 2015.

[98] Raymond W. Lo, Karl N. Levitt, and Ronald A. Olsson. MCF: a malicious code filter. *Computers and Security*, 1995. ISSN 01674048.

[99] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2005. doi: 10.1145/1081870.1081950.

[100] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 641–647, New York, NY, USA, 2005. ACM. ISBN 1-59593-135-X.

[101] R. Lu. Malware detection with lstm using opcode language. *arXiv:1906.04593*, 2019.

[102] Andrew Mackie, Jensenne Roculan, Ryan Russell, and Mario Van Velzen. Nimda Worm Analysis. Technical report, SecurityFocus, California, USA, 2001.

[103] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers and Security*, 51(March 2014):16–31, 2015. ISSN 01674048. doi: 10.1016/j.cose.2015.02.007. URL http://dx.doi.org/10.1016/j.cose.2015.02.007.

[104] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, and S. P. Smolley. Least squares generative adversarial networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2813–2821, 2017.

[105] Muazzam Maqsood, Faria Nazir, Umair Khan, Farhan Aadil, Habibullah Jamal, Irfan Mehmood, and Oh-Young Song. Transfer Learning Assisted Classification and Detection of Alzheimer's Disease Stages Using 3D MRI Scans. *Sensors (Basel, Switzerland)*, 19(11):2645, jun 2019. ISSN 1424-8220. doi: 10.3390/s19112645. URL https://pubmed.ncbi.nlm.nih.gov/31212698https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6603745/.

[106] A Martin, H D Menéndez, and D Camacho. Genetic boosting classification for malware detection. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 1030–1037, 2016. ISBN VO -.

[107] Gary McGraw and Greg Morrisett. Attacking malicious code: A report to the Infosec Research Council. *IEEE Software*, 17(5):33–41, 2000. ISSN 07407459. doi: 10.1109/52.877857.

[108] Julian F. Miller and Peter Thomson. Cartesian genetic programming. *Genetic Programming, Proceedings of the Third European Conference on Genetic Programming (EuroGP2000).*, 1802:121–132, 2000. ISSN 1098-6596. doi: 10.1017/CBO9781107415324.004.

[109] Melanie Mitchell. *An introduction to genetic algorithms*. 1996. ISBN 0-262-13316-4. URL https://svn-d1.mpi-inf.mpg.de/AG1/MultiCoreLab/papers/ebook-fuzzy-mitchell-99.pdf.

[110] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security Privacy*, 1(4):33–39, July 2003. ISSN 1558-4046. doi: 10.1109/MSECP.2003.1219056.

[111] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites, 2015.

[112] Nadim Nachar. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology*, 4, 03 2008. doi: 10.20982/tqmp.04.1.p013.

[113] M Nehéz, M Lelovský, J Bendžala, and T Blaho. On the k-center Problem in Social Networks. *Researchgate.Net*, (September 2014), 2015. URL https://www.researchgate.net/profile/Martin_Nehez/publication/277652673_On_the_k-center_Problem_in_Social_Networks/links/556f122f08aec226830a4f18.pdf.

[114] Robert Nisbet, Gary Miner, and Ken Yale. Chapter 9 - classification. In Robert Nisbet, Gary Miner, and Ken Yale, editors, *Handbook of Statistical Analysis and Data Mining Applications (Second Edition)*, pages 169 – 186. Academic Press, Boston, 2 edition, 2018. ISBN 978-0-12-416632-5. doi: https://doi.org/10.1016/B978-0-12-416632-5.00009-8. URL http://www.sciencedirect.com/science/article/pii/B9780124166325000098.

[115] University of Windsor. General Information of Malware, 2015. URL http://www1.uwindsor.ca/its/virus/sites/uwindsor.ca.its.virus/files/MalwareInfo.pdf.

[116] Una-May O'Reilly. Genetic Programming II: Automatic Discovery of Reusable Programs. *Artificial Life*, 1(4):439–441, 1994. ISSN 1064-5462. doi: 10.1162/artl.1994.1.4.439. URL http://dx.doi.org/10.1162/artl.1994.1.4.439%5Cnhttp://www.mitpressjournals.org/doi/pdf/10.1162/artl.1994.1.4.439%5Cnhttp://www.mitpressjournals.org/toc/artl/1/4.

[117] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[118] PC-SPYWARE. KeySnatch - USB Keylogger, 2018.

[119] F. Periot. Defeating Polymorphism Through Code Optimization. *Virus Bulletin*, pages 142–159, 2003.

[120] Raphael Petegrosso, Sunho Park, Tae Hyun Hwang, and Rui Kuang. Transfer learning across ontologies for phenome–genome association prediction. *Bioinformatics*, 33(4):529–536, nov 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw649. URL https://doi.org/10.1093/bioinformatics/btw649.

[121] Riccardo Poli. Parallel Distributed Genetic Programming. (May), 1996.

[122] Justin Pugh, Lisa Soros, and Kenneth Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3, 07 2016. doi: 10.3389/frobt.2016.00040.

[123] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40, 2016.

[124] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2015.

[125] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. A comparison of code similarity analysers. *Empirical Software Engineering*, 2018. ISSN 15737616. doi: 10.1007/s10664-017-9564-7.

[126] Nils Reimers and Iryna Gurevych. Optimal hyperparameters for deep lstm-networks for sequence labeling tasks, 2017.

[127] E. Rezende, G. Ruppert, T. Carvalho, F. Ramos, and P. de Geus. Malicious software classification using transfer learning of resnet-50 deep neural network. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1011–1014, 2017.

[128] Sebastian Ruder, Matthew E. Peters, Swabha Swayamdipta, and Thomas Wolf. Transfer learning in natural language processing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pages 15–18, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-5004. URL https://www.aclweb.org/anthology/N19-5004.

[129] Justin Sahs and Latifur Khan. A Machine Learning Approach to Android Malware Detection. In *2012 European Intelligence and Security Informatics Conference*, 2012. ISBN 978-1-4673-2358-1.

[130] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013. ISSN 0020-0255.

[131] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20, Oct 2015. doi: 10.1109/MALWARE.2015.7413680.

[132] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, pages 38–49, 2001.

[133] Pritam Sil. Tournament Selection (GA), 2018.

[134] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala. Static Analyzer of Vicious Executables (SAVE). In *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 2004. ISBN 0769522521.

[135] Péter Ször and Peter Ferrie. Hunting for metamorphic. In *In Virus Bulletin Conference*, pages 123–144, 2001. doi: 10.1007/s11416-006-0028-7.

[136] S. Momina Tabish, M. Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics - CSI-KDD '09*, 2009. ISBN 9781605586694.

[137] Roelof Temmingh and Haroon Meer. Setiri: Advances in Trojan Technology. In *BlackHat Asia*, Singapore, 2002. BlackHat. URL https://www.blackhat.com/presentations/bh-asia-02/Sensepost/bh-asia-02-sensepost.pdf.

[138] The-Honeynet-Project. Droidbox, 2011. URL https://www.honeynet.org/taxonomy/term/191.

[139] Annie H Toderici and Mark Stamp. Chi-squared Distance and Metamorphic Virus Detection. *J. Comput. Virol.*, 9(1):1–14, feb 2013. ISSN 1772-9890.

[140] N P Tran and M Lee. High performance string matching for security applications. In *International Conference on ICT for Smart Society*, pages 1–5, jun 2013.

[141] TRENDMICRO. ANDROIDOS_DOUGALEK.A, 2012. URL https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/androidos{_}dougalek.a.

[142] TRENDMICRO. The Michelangelo Virus, 25 Years Later. *Cybercrime & Digital Threats*, mar 2017.

[143] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123 – 147, 2019. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2018.11.001. URL http://www.sciencedirect.com/science/article/pii/S0167404818303808.

[144] Laurens Van Der Maaten. Accelerating t-SNE using tree-based algorithms. *Journal of Machine Learning Research*, 15:3221–3245, 2015. ISSN 15337928.

[145] Amit Vasudevan and Ramesh Yerraballi. SPiKE: Engineering malware analysis tools using unobtrusive binary-instrumentation. In *Conferences in Research and Practice in Information Technology Series*, volume 48, pages 311–320, 2006. ISBN 1920682309.

[146] C Vatamanu, D Gavrilut, R Benchea, and H Luchian. Feature Extraction Using Genetic Programming with Applications in Malware Detection. In *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 224–231, 2015. ISBN VO -.

[147] Tim Vidas. Contagio Mobile – Mobile Malware Mini Dump, 2015. URL http://contagiominidump.blogspot.com/2015/01/android-hideicon-malware-samples.html.

[148] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 21–30, New York, NY, USA, 2004. ACM. ISBN 1-58113-961-6. doi: 10.1145/1030083.1030088. URL http://doi.acm.org/10.1145/1030083.1030088.

[149] Virustotal. Virustotal, 2004. URL https://developers.virustotal.com/reference#getting-started.

[150] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. Normalizing metamorphic malware using term rewriting. In *Proceedings - Sixth IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2006*, 2006. ISBN 0769523536.

[151] Andrew Walenstein, Rachit Mathur, Mohamed Chouchane, and Arun Lakhotia. The design space of metamorphic malware. In *ICIW 2007: 2nd International Conference on i-Warfare and Security*, pages 241–248, 2007.

[152] Geoffrey I. Webb. *Naïve Bayes*, pages 713–714. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_576. URL https://doi.org/10.1007/978-0-387-30164-8_576.

[153] David R. White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011. ISSN 1089778X.

[154] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2006. ISSN 17729890. doi: 10.1007/s11416-006-0028-7.

[155] Weilin Xu, Yanjun Qi, and David Evans. Automatically Evading Classifiers: A Case Study on PDF Malware Classifier. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA*. The Internet Society, 2016.

[156] Mohd Najwadi Yusoff and Aman Jantan. A framework for optimizing malware classification by using genetic algorithm. In Jasni Mohamad Zain, Wan Maseri bt Wan Mohd, and Eyas El-Qawasmeh, editors, *Software Engineering and Computer Systems*, pages 58–72, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22191-0.

[157] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings - IEEE Symposium on Security and Privacy*, 2012. ISBN 9780769546810. doi: 10.1109/SP.2012.16.

[158] Yajin Zhou and Jiang Xuxian. Android Malware Genome Project, 2012. URL http://www.malgenomeproject.org/.

[159] Zhi Hong Zuo, Qing Xin Zhu, and Ming Tian Zhou. On the time complexity of computer viruses. *IEEE Transactions on Information Theory*, 51(8):2962–2966, 2005. ISSN 00189448. doi: 10.1109/TIT.2005.851780.

Appendix A

Publications

A.1 Nowhere Metamorphic Malware Can Hide - A Biological Evolution Inspired Detection Scheme

K. O. Babaagba, Z. Tan, and E. Hart, "Nowhere metamorphic malware can hide - a biological evolution inspired detection scheme," in Dependability in Sensor, Cloud, and Big Data Systems and Applications, G. Wang, M. Z. A. Bhuiyan, S. De Capitani di Vimercati, and Y. Ren, Eds. Singapore: Springer Singapore, 2019, pp. 369–382.

A.2 Automatic Generation of Adversarial Metamorphic Malware Using MAP-Elites

K. O. Babaagba, Z. Tan, and E. Hart, "Automatic Generation of Adversarial Metamorphic Malware Using MAP-Elites," in 23rd European Conference on the Applications of Evolutionary and bio-inspired Computation, P.A. Castillo et al, Ed. Seville: Springer-Verlag New York, Inc., 2020, pp. 117-132.

A.3 Improving Classification of Metamorphic Malware by Augmenting Training Data with a Diverse Set of Evolved Mutant Samples

K. O. Babaagba, Z. Tan, and E. Hart, "Improving Classification of Metamorphic Malware by Augmenting Training Data with a Diverse Set of Evolved Mutant Samples," in 2020 IEEE World Congress on Computational Intelligence, Glasgow, UK, 2020.