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

Kehinde O. Babaagba$^{[0000-0003-0786-2618]}$, Zhiyuan Tan$^{[0000-0001-5420-2554]}$, and Emma Hart$^{[0000-0002-5405-4413]}$

School of Computing, Edinburgh Napier University, Edinburgh EH10 5DT, United Kingdom {K.Babaagba,Z.Tan,E.Hart}@napier.ac.uk

Abstract. The ability to detect metamorphic malware has generated significant research interest over recent years, particularly given its proliferation on mobile devices. Such malware is particularly hard to detect via signature-based intrusion detection systems due to its ability to change its code over time. This article describes a novel framework which generates sets of potential mutants and then uses them as training data to inform the development of improved detection methods (either in two separate phases or in an adversarial learning setting). We outline a method to implement the *mutant generation* step using an *evolutionary algorithm*, providing preliminary results that show that the concept is viable as the first steps towards instantiation of the full framework.

Keywords: Metamorphic Malware; Evolutionary Algorithm; Mutant Generation; Mobile Devices; Detection Methods; Adversarial Learning.

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 2019 Internet Security Threat Report by Symantec revealed that there has been an increase in malicious attacks in form of form-jacking attacks, with approximately 4,800 websites being victims monthly. Ransomware is now targeting enterprises with a 12% rise in the number of infections as compared to the previous year's attack incidence.

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, 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. [1] provide a detailed overview, including Opcode-Based Analysis (OBA), Control Flow Analysis (CFA) or Information Flow Analysis (IFA) which differ depending on the type of information being used in the analysis. We propose an alternative approach via a novel framework which contains two components: the first component generates a set of potential mutants of existing malware; the second method trains a detection system to recognise the new mutants. The two components maybe used sequentially or in an adversarial setting, that is, the mutant generator can create increasingly more complex mutants as the detection system gets better at recognising mutants.

In this paper we describe the generic concept of the framework then propose a method for instantiating the mutant-generating component using an *evolutionary algorithm* [8]. This population based search technique has been successfully used in code-modification scenarios, e.g. to fix bugs [10] or speed-up code [6], as well as some previous attempts to evolve mutants of existing malware [2]. The main contributions of the paper are as follows:

- A review of current metamorphic malware detection techniques.
- A proposal for a novel framework for developing malware detectors capable of recognising future malware mutations.
- A proof-of-concept that an Evolutionary Algorithm as the mutation engine component of the framework can be used to generate new mutant samples.

The rest of the paper is structured as follows. Section 2 gives a brief history of metamorphic malware detection. Section 3 discusses the challenges to existing detection techniques. In Section 4, evolutionary based malware detection and its challenges is discussed. Section 5 describes the proposed detection framework. In Section 6, preliminary experimentation and discussion is presented. A conclusion is drawn in Section 7.

2 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.

2.1 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 [17] was one of the first signature-based malware detection scheme and has subsequently been referenced by other research works such as [24]. The authors of [17] 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 [12] used string signatures for metamorphic malware detection and they achieved a false alarm rate of less than 0.1%. The work, presented in [26], introduces Aho-Corasick (AC), which is a string matching algorithm for detecting metamorphic malware. [23] 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 [14], 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.

2.2 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) [3], Hidden Markov Models (HMM) [25], and Support Vector Machines (SVM) [20].

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 [22] which uses Naïve Bayes (NB) in automatically identifying malicious patterns in malware. Their NB approach, calculated from the programs 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 [21], 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 [25] in detecting metamorphic viruses. [3] also uses a statistical-based classifier that employs a DT in metamorphic malware detection. A single class SVM is used by [20] in Android based metamorphic malware classification. These works led to increased metamorphic malware detection rate.

2.3 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 [19] whose approach took advantage of various code optimisation strategies in enhancing the detection of malware. [29] 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 [5].

In addition, similarity-based approaches, for instance structural entropy and compression-based techniques, were applied in [4] and [16] 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 [4], 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 [16] 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.

3 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. 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.

4 Evolutionary based Malware Detection

We propose that the use of an Evolutionary Algorithm to generate new malware samples in order to create detectors that can recognise potential future variants of a class of malware will address some of the above issues.

The term EA refers to a class of problem-solving techniques inspired by Darwin's theory of evolution [8], 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 (crossover) and/or by a mutation process that makes small, random, changes to solutions. As new fitter solutions replaces 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 [27], classification problems [32] among others . [7] described 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;
- Provides operators that enable easy manipulation of code;
- Proven ability in transformation, optimisation and improvement of software code [6], [15] and [30]

The idea of using EA based techniques for malware analysis and detection is not a new concept. For example, [18] use EAs to improve classifier selection and performance for malware detection. In [13], the authors use a variant of an EA called Genetic Programming (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.

In [2], the authors also 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, [31] 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 [2] and operators like deletion, insertion and replacement for [31]. 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.

4.1 Challenges with Current EA-based Approaches

Although the approaches described above provide evidence that EAs are a useful methodology, there remains much scope for improvement. [13] focus on buffer-overflow attacks rather than metamorphic malware, while [31] focus on pdf malware. Although [2] focus on metamorphic malware, they only consider 8 antivirus 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.

Our proposed scheme addresses the above weaknesses as follows. Firstly, we focus on mobile platforms as they are currently targeted by recent malware attacks. We evaluate evasiveness using a large set of 63 AV engines to evaluate the variants created. Rather than only considering evasiveness as the fitness metric, we also measure the structural similarity and the behavioural similarity between the evolved mutants and the original malware and include this information in the fitness function. This makes it easier for the EA to traverse the fitness landscape as the fitness function is more fine-grained and hopefully discovers better solutions. It also increases the diversity of evolved variants. Finally, we also ensure that all variants retain their malicious nature once mutation has occurred.

5 Framework of Detection Scheme

This section proposes a framework of metamorphic malware detection for mobile computing platforms in order to address some of the challenges raised above. The framework comprises 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.1.

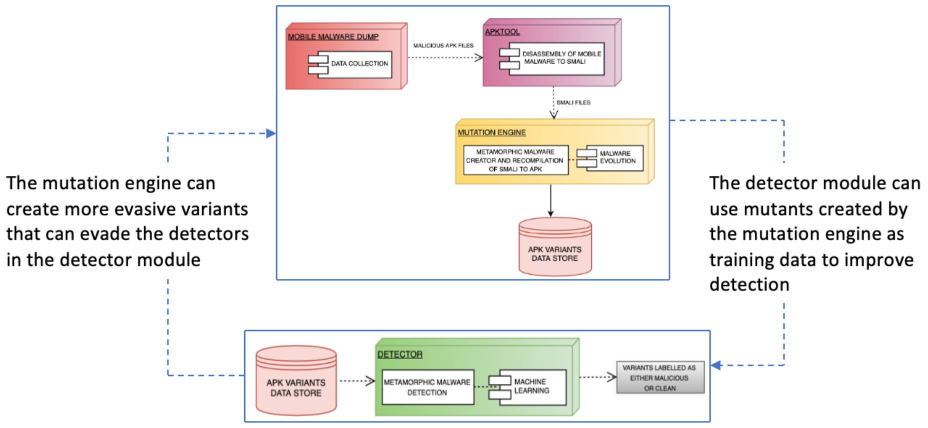

MOBILE MALWARE DUMP
DATA COLLECTION
MALICIOUS APK FILES
APKTOOL
DISASSEMBLY OF MOBILE MALWARE TO SMALI
SMALI FILES
MUTATION ENGINE
METAMORPHIC MALWARE CREATOR AND RECOMPILATION OF SMALI TO APK
MALWARE EVOLUTION
APK VARIANTS DATA STORE
The mutation engine can create more evasive variants that can evade the detectors in the detector module
The detector module can use mutants created by the mutation engine as training data to improve detection
APK VARIANTS DATA STORE
DETECTOR
METAMORPHIC MALWARE DETECTION
MACHINE LEARNING
VARIANTS LABELLED AS EITHER MALICIOUS OR CLEAN

Fig. 1. A conceptual overview of the proposed detection framework.

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 can be 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 drives the generation of more diverse mutants, hence driving further improvement in the detection system.

To implement the malware generation module, we propose the use of Evolutionary Algorithm which is explained in the next section. This technique is well-known for its ability to search vast spaces of potential solutions [11]; we use it 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 detection module can employ 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. We suggest that the probabilistic and evidence-based nature of Bayesian inference techniques makes it a suitable machine learning candidate. Metrics that might be used to evaluate the suitability of a machine-leaner are suggested below.

- Training time of the model – This refers to the time taken to train the new machine learning adaptation.
- Classification time - This refers to the time taken for the new machine learning adaptation to classify the sample as either malicious or benign.
- Model Accuracy - This refers to how accurate the model is in detecting the malicious binaries.

At the current stage of the research, we focus on the mutation engine in order to generate high quality and diverse samples to serve as rich a training set to the machine learning model. At a later stage of the research, we will then train a machine learning model on the generated diverse variants.

5.1 Malware Evolution

The mutation engine shown in Fig.1 is implemented using an EA. The goal of the EA is to generate a new set of malware variants that evade current detection engines, and are diverse with respect to their behavioural and structural similarity. The malware, used for demonstration in this paper, was collected from Contagio Mini Dump [28], a mobile malware dump. The code of this mobile malware is, first, reverse engineered from an Android Package (APK) to smali and then converted to a document vector that serves as an input to the evolutionary algorithm.

An EA given in Algorithm 1 is used to evolve the instances of given malicious code. It begins with an initial population of solutions created by applying a random mutation process to the original malware. Each solution is evaluated using a fitness function, defined in equation (5.1). The fitness function is minimised by the algorithm, i.e. the smaller the value, the more evasive the variant, and the more it is structurally and behaviourally dissimilar to the original malware. The main loop of the algorithm then selects a mutant as a *parent*, and mutates this to create a child mutant. The child replaces the solution in the population that has worst fitness if its own fitness is better than the worst.

Algorithm 1. Evolutionary Algorithm

```
1: Initialize pop of $m_i$ random mutants $i \in [0, m-1]$
2: Assign fitness to each mutant
3: while Maximum number of iterations not reached do
4:     Randomly select $k$ variants from pop, and set parent $p_{best}$ to fittest variant
5:     Generate a new mutant $m_{new}$ from $p_{best}$ by mutating $p_{best}$, selecting mutation
       operator with uniform probability
6:     Evaluate fitness $fit_{new}$ of new mutant
7:     if $fit_{new} > fit_{worst}$ then
8:         Replace the worst fit in pop with $m_{new}$
9:         Update $fit_{worst}$
10:    end if
11: end while
12: return The variants created
```

The fitness function referred to in algorithm 1 is given below and returns a value between 0 and 1:

$$f(x) = \begin{cases} 1 & \text{if variant not executable} \\ w_0 DR(x) + w_1 SS(x) + w_2 BS(x) & \text{otherwise} \end{cases}$$

$$\text{subject to} \begin{cases} \sum_{i=0}^{2} W_i = 1 \\ 0 \leq DR(x), SS(x), BS(x) \leq 1 \end{cases}$$

The defined fitness function takes into consideration of the code level similarities (denoted as $SS(x)$) between the original malicious file and its variants; the behavioural similarities (denoted as $BS(x)$) between the original malicious file and its variants and the detection rate (denoted as $DR(x)$) of its variants. The three functions ($DR(x)$, $SS(x)$ and $BS(x)$) can be weighted with values between 0 and 1 to favour one type of solution over another. These functions are described below:

The Code Level/Structural Similarity: $SS(x)$ measures the similarity between the original smali file and its mutants. Text similarity (cosine similarity, levenshtein and fuzzy string matching) and source code similarity (jplag and sherlock plagiarism detectors as well as normalised compression distance) metrics are employed. The structural similarity between the original APK file and its mutants is an average of all the similarity metrics employed where a value of 0 means the original APK file and its mutants are completely dissimilar and 1 means the original APK file and its mutants are identical.

The Behavioural Similarity: The behavioural analysis of the APK files mutants is measured using Strace and Monkey runner. Strace is used to monitor the system calls of the variants while monkey runner is used to simulate user action. A feature vector is constructed from the log of strace where each vector element represents the frequency of a system call. The behavioural similarity between the original APK file and its mutants measures the cosine similarity between the original APK file and its mutants' feature vector. The result of the function is a value between 0 and 1 where 0 means the original APK file and its mutants are completely dissimilar and 1 means the original APK file and its mutants are identical.

Evasiveness of Variants: The function $DR(x)$ assesses the ability of a mutated APK to evade detection by antivirus engines. It is measured with the analysis report from Virustotal to determine the APKs evasive ability. Virustotal comprises of 63 antivirus engines that represent most of the state-of-the-art antivirus engines. The function checks to see which of these antivirus engines flags a submitted file as malicious or benign. $DR(x)$ returns the percentage of engines that detects the variants where a lower percentage indicates a more evasive variant.

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.

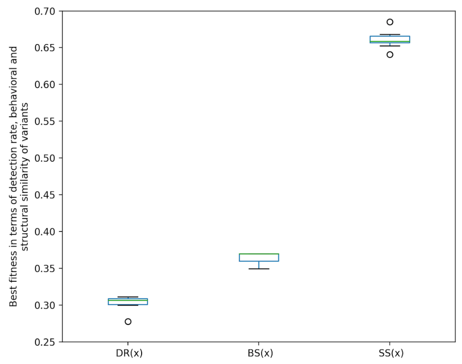
- To check for the compilation of an APK file: apktool, apksigner and zipalign.
- To check that the file runs: Android emulator.

We wrap the functions that check to see that the APK variants run and compile properly in a bash script. Finally, in order to ensure that the evolved variants retain their maliciousness we use Droidbox, a dynamic analysis tool, to check that the variants are still malicious. We only analyse the results for the variants that retain their maliciousness.

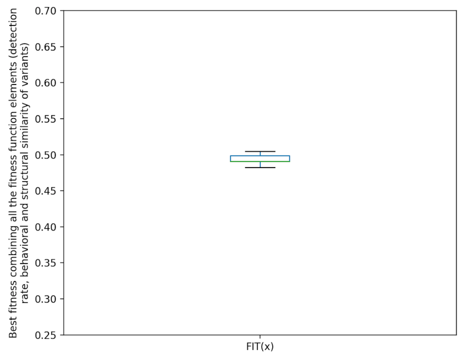
6 Experiments

In this preliminary study, the malware utilised in this work for demonstration is from the Contagio Minidump. The original parent malware was selected from the dump at random. The experiment was conducted in a VM Workstation running

Ubuntu operating system. The EA consists of a population size of 20. Parents are selected using tournament selection [9] with $k = 5$. The best of the k selected parents is mutated by adding either junk codes like line numbers, reordering its variables or distorting the program code's control flow through the insertion of a goto statement that jumps to a label that does nothing. The EA is then run 10 times for 100 iterations, and the best variant produced in each of the 10 runs is recorded.



(a) Boxplot of best fitness in terms of detection rate ($DR(x)$, behavioral similarity ($BS(x)$) and structural similarity ($SS(x)$) of the variants for the ten runs of the EA



(b) Boxplot of best fitness for 5 runs of the EA where the fitness is a weighted combination of $DR(x)$, $BS(x)$ and $SS(x)$ as given in eq. (1) in Section 5.1

Fig. 2. Boxplots of best fitness

First, we conduct three experiments in turn in which the fitness function only uses one of the three metrics DR, BS, SS to understand the influence of each individual component i.e. the weight of interest is set to 1 and the weights for the other two metrics set to 0. The initial experiments using each of the functions in the fitness function are illustrated in Fig. 2(a).

The original malware had a detection rate ($DR(x)$) of 0.597. We see from Fig.2(a) that we are able to create variants in which the best has a rate of 0.278 and all 10 values are less than 0.32. Also, we see that we are able to create variants that are only 35% behaviourally ($BS(x)$) similar to the original malware and 64% structurally ($SS(x)$) similar to the original malware, indicating behavioural and structural diversity.

When we combine all the functions in the fitness function as given in equation (5.1), we are able to generate a diverse set of mutants that are executable, evasive and behaviourally and structurally different compared to the original malware which would have a weighted fitness value of 0.8657 (with a value 1 for BS(x) and SS(x) and 0.597 for DR(x)), as seen in Fig.2(b).

We also analyse which engines are more likely to be fooled by the new evasive mutants as seen in Fig.3. The original malware was detected by 37 out of the

63 antivirus engines. We analyse how many of these engines that detected the original malware were able to be fooled by the best variant found in each of the 10 runs of the EA (Fig. 3).

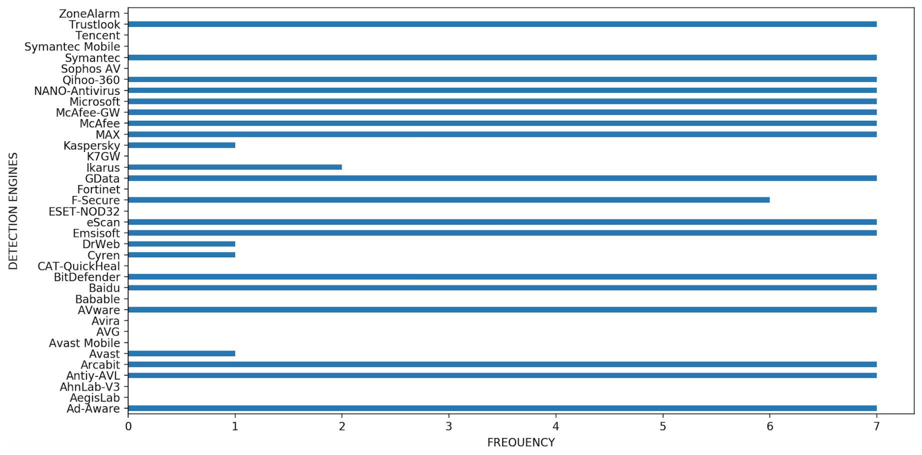


Fig. 3. Analysis of the detection engines

Some detection engines (e.g. Trustlook, McAfee and F-Secure) are fooled by all 10 new variants. On the other hand, Avast Mobile and Babable were not fooled in any of the runs (i.e. they were able to detect the new malware). Also, 14 of the engines were not fooled in any of the 10 runs while 17 of the engines were fooled in all the 10 runs of the EA.

7 Conclusion

In this paper we carried out a review of current methods to metamorphic malware analysis and detection. We also proposed a framework that could be used to tackle detection of malware by creating a mutation engine that provides new training examples representing potential states the malware can morph too, that can then be used to train better detectors.

Furthermore, we provide a proof of concept for the mutation engine that uses an EA showing it is capable of generating a diverse set of malicious mutants. This is advantageous in that it will help in determining the effectiveness of the existing IDS. To complete the framework, future work will conduct a more thorough experimental analysis of the EA, including EA parameters and investigating how the approach generalises to other classes of malware. Also, it will focus on designing new machine learning (ML) methods that can be trained using the new metamorphic malware created in order to be robust to future attacks.

References

1. Alam, S., Traore, I., Sogukpinar, I.: Current trends and the future of metamorphic malware detection. In: Proceedings of the 7th International Conference on Security of Information and Networks. pp. 411–416. SIN '14, ACM, New York, NY, USA (2014)
2. Aydogan, E., Sen, S.: Automatic generation of mobile malwares using genetic programming. In: Mora, A.M., Squillero, G. (eds.) Applications of Evolutionary Computation. pp. 745–756. Springer International Publishing, Cham (2015)
3. Bashari Rad, B., Masrom, M., Ibrahim, S., Ibrahim, S.: Morphed Virus Family Classification Based on Opcodes Statistical Feature Using Decision Tree. In: Abd Manaf, A., Zeki, A., Zamani, M., Chuprat, S., El-Qawasmeh, E. (eds.) Informatics Engineering and Information Science. pp. 123–131. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
4. Baysa, D., Low, R.M., Stamp, M.: Structural entropy and metamorphic malware. Journal in Computer Virology **9**(4), 179–192 (2013)
5. Bruschi, D., Martignoni, L., Monga, M.: Code normalization for self-mutating malware. IEEE Security and Privacy **5**(2), 46–54 (2007)
6. Cody-Kenny, B., Galván-López, E., Barrett, S.: 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. pp. 811–818. GECCO Companion '15, ACM, New York, NY, USA (2015)
7. Edge, K.S., Lamont, G.B., Raines, R.A.: A retrovirus inspired algorithm for virus detection & optimization. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation. pp. 103–110. GECCO '06, ACM, New York, NY, USA (2006)
8. Eiben, A.E., Smith, J.E.: What is an Evolutionary Algorithm? In: Introduction to Evolutionary Computing, pp. 15–35. Springer Publishing Company, Incorporated (2003)
9. Fang, Y., Li, J.: A review of tournament selection in genetic programming. In: Cai, Z., Hu, C., Kang, Z., Liu, Y. (eds.) Advances in Computation and Intelligence. pp. 181–192. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
10. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation. pp. 947–954. GECCO '09, ACM, New York, NY, USA (2009)
11. García-Martínez, C., Lozano, M.: Local search based on genetic algorithms. In: Siarry, P., Michalewicz, Z. (eds.) Advances in Metaheuristics for Hard Optimization, pp. 199–221. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
12. Griffin, K., Schneider, S., Hu, X., Chiueh, T.c.: Automatic Generation of String Signatures for Malware Detection. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) Recent Advances in Intrusion Detection. pp. 101–120. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
13. Kayacik, H.G., Heywood, M., Zincir-Heywood, N.: On Evolving Buffer Overflow Attacks Using Genetic Programming. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation. pp. 1667–1674. GECCO '06, ACM, New York, NY, USA (2006)
14. Konstantinou, E.: Metamorphic Virus: Analysis and Detection. Tech. rep., Department of Mathematics Royal Holloway, University of London (2008)

15. Langdon, W.B., Harman, M.: Optimizing existing software with genetic programming. IEEE Transactions on Evolutionary Computation **19**(1), 118–135 (2015)
16. Lee, J., Austin, T.H., Stamp, M.: Compression-based Analysis of Metamorphic Malware. International Journal of Security and Networks **10**(2), 124–136 (jul 2015)
17. Lo, R.W., Levitt, K.N., Olsson, R.A.: MCF: a malicious code filter. Computers and Security (1995)
18. Martin, A., Menéndez, H.D., Camacho, D.: Genetic boosting classification for malware detection. In: 2016 IEEE Congress on Evolutionary Computation (CEC). pp. 1030–1037 (2016)
19. Periot, F.: Defeating Polymorphism Through Code Optimization. Virus Bulletin pp. 142–159 (2003)
20. Sahs, J., Khan, L.: A Machine Learning Approach to Android Malware Detection. In: 2012 European Intelligence and Security Informatics Conference (2012)
21. Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G.: Opcode sequences as representation of executables for data-mining-based unknown malware detection. Information Sciences **231**, 64–82 (2013)
22. Schultz, M.G., Eskin, E., Zadok, F., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001. pp. 38–49 (2001)
23. Sung, A.H., Xu, J., Chavez, P., Mukkamala, S.: Static Analyzer of Vicious Executables (SAVE). In: Proceedings - Annual Computer Security Applications Conference, ACSAC (2004)
24. Tabish, S.M., Shafiq, M.Z., Farooq, M.: 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)
25. Toderici, A.H., Stamp, M.: Chi-squared Distance and Metamorphic Virus Detection. J. Comput. Virol. **9**(1), 1–14 (feb 2013)
26. Tran, N.P., Lee, M.: High performance string matching for security applications. In: International Conference on ICT for Smart Society. pp. 1–5 (jun 2013)
27. Vatamanu, C., Gavrilut, D., Benchea, R., Luchian, H.: Feature Extraction Using Genetic Programming with Applications in Malware Detection. In: 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). pp. 224–231 (2015)
28. Vidas, T.: Contagio Mobile Mobile Malware Mini Dump (2015), http://contagiominidump.blogspot.com/2015/01/android-hideicon-malware-samples.html
29. Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: Normalizing metamorphic malware using term rewriting. In: Proceedings - Sixth IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2006 (2006)
30. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation **15**(4), 515–538 (2011)
31. Xu, W., Qi, Y., Evans, D.: 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)
32. Yusoff, M.N., Jantan, A.: A framework for optimizing malware classification by using genetic algorithm. In: Zain, J.M., Wan Mohd, W.M.b., El-Qawasmeh, E. (eds.) Software Engineering and Computer Systems. pp. 58–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)