Automatic Generation of Adversarial Metamorphic Malware Using MAP-Elites

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. In the field of metamorphic malware detection, training a detection model with malware samples that reflect potential mutants of the malware is crucial in developing a model resistant to future attacks. In this paper, we use a Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) algorithm 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 two classes of malware as a test-bed, we show that the MAP-Elites algorithm produces a large and diverse set of mutants, that evade between 64% to 72% of the 63 detection engines tested. When compared to results obtained using repeated runs of an Evolutionary Algorithm that converges to a single solution result, 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.

Keywords: Metamorphic Malware, MAP-Elites, Machine-Learning

1 Introduction

The proliferation of malicious attacks on networked devices and internet infrastructures at large has become a source of concern for companies and cybersecurity researchers. These attacks often emanate from a vast range of malicious groups. One of such dangerous groups is metamorphic malware. These malware transform their code between generations using various obfuscation techniques thereby making detection difficult. These techniques include the insertion of junk code into the original program code i.e. garbage code insertion, renaming variables in the original program code a process calling variable renaming among others. [3] provides a comprehensive list of such techniques used by metamorphic malware.

One recent approach to improving detection of metamorphic malware is through the use of *adversarial learning* [11]: these approaches generate new malicious input data (attacks) that reveals vulnerabilities in the detection models,

then improve detection models as a result. Adversarial learning has received significant attention from cybersecurity experts recently for malware analysis and detection as it gives insights to researchers on the processes malware writers use in generating malware [7, 12, 17].

However, the search for adversarial samples which comprise of several 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 desirable 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 adversarial samples, for example in the domains of pdf-malware [19] and in android malware [1, 2]. 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 in addition, 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, we propose a solution that generates a *set* of variants that are *diverse* with respect to two features, the Structural Similarity (SS(x)) and Behavioral Similarity (BS(x)) of the variants with respect to the original malware. Specifically, we apply a quality-diversity algorithm — MAP-Elites [13] — 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. 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 [14] that aim to return an archive of diverse, high-quality behaviors in a single run. The algorithm has multiple documented successes in evolutionary robotics [13], but also in design applications, car wing-mirror design [8].

We address two questions in this paper:

1. How does diversity of samples produced by running MAP-Elites algorithm compare to repeated executions of the standard Evolutionary algorithm described in [2] and in Sect. 2?
2. How does the evasiveness of samples produced by MAP-Elites algorithm compare to repeated executions of a standard Evolutionary algorithm?

The contributions of the paper is three-fold. To the best of our knowledge, this is the first use of an illumination algorithm to generate a diverse set of adversarial samples of mutant 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, we provide a comparison to results obtained by running a single evolutionary algorithm multiple times in order to generate a set of variants [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, we provide novel insights into the factors that contribute to evasiveness, based on the results obtained from the illumination algorithm.

The rest of the paper is structured as follows. Section two presents a background of the work and reviews related works presenting distinguishing points between this work and the related works. In section three, we present our methodology. Our experimental design is explained in section four. We discuss and analyse our results in section five. Section six concludes the paper and provides areas of future research work.

2 Background

Previous research has been geared towards creating evasive malware that goes undetected by antivirus engines and other detectors. One of the pioneering systems used to assess the ability of antivirus engines in detecting evasive malware is ADAM [20]. This system automatically transforms an original malware sample to different variants via repackaging and obfuscation techniques in order to evaluate the robustness of different detection systems against malware mutation. Similar to the work of [20] is DroidChameleon [16] which extends [20] by considering more advanced forms of attacks including metamorphic and polymorphic attacks.

Recently, the use of evolutionary computing as a technique in the generation of evasive malware has been explored by a number of authors. Genetic Programming was used by [19] to create pdf malware that evades detection by pdf detectors while retaining their malicious functionality. [1] used Genetic Programming (GP) to create a single malware variant that maximised an evasiveness score when presented to 8 detection engines, however the characteristics of the evolved malware were not considered. In addition, it is likely that many distinct variants could map to the same fitness value, given that the fitness function takes one of only 9 distinct values.

Inspired by this work, we recently proposed a mutation only EA [2] for generating adversarial samples. Our work advanced that of [1] in (1) evaluating a set of new fitness functions that optimised for behavioral and structurally diverse variants as well as for evasiveness and (2) extended the evaluation to a much larger set of 63 detection engines. However in both this work and prior work of [1], it was necessary to run the evolutionary engine multiple times to generate a set of samples. While the work presented in [2] was shown in fact to lead to some diversity across multiple runs, this cannot be guaranteed.

On contrast, the MAP-Elites algorithm was explicitly designed with the goal of providing multiple high-performing solutions that are diverse with respect to a user-defined feature-space [13]. The seminal paper showed that the technique can illuminate the close links between performance and interesting features in the search space as well as creating diverse and high quality solutions. Following the initial work in the robotics domain, the algorithm has found application in

other domains such as video games, with the introduction of MAP-Elites with Sliding Boundaries (MESB)[5] showing MAP-Elites ability to discover varying and diverse styles of playing the game, and in the combinatorial optimisation domain in evolving delivery schedules in a feature-space that includes carbon-emissions and staff-costs [18].

Here, we use MAP-Elites algorithm in the malware analysis domain to generate new malicious variants that evade current detectors and are structurally and behaviourally dissimilar to their parent malware. As far as we are aware, this is the first time that this algorithm has been used in the exploration of the search space of malware.

3 Methodology

In this section, we describe 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.

3.1 MAP-Elites Algorithm

The algorithm is given in Alg. 1. 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 Sect. 3.2, 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 goto statement in the original program code that 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 the original malware is in the form of an executable *apk file*, to create mutants we first reverse engineer the apk by converting it to a *smali* format using apktool[1]. Thereafter, we execute the following steps:

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.

[1] APKTOOL - http://ibotpeaches.github.io/Apktool

3. Sign the recompiled apk using apksigner[2] and align using zipalign[3].
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 1. MAP-Elites algorithm for mutant generation, modified from [13]

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

[2] APKSIGNER - https://developer.android.com/studio/command-line/apksigner
[3] ZIPALIGN - https://developer.android.com/studio/command-line/zipalign

3.2 Feature Descriptor

The feature descriptor 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[4] using Monkeyrunner[5] 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 behavioral 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. The text based similarity measures the cosine similarity, fuzzy string match [4] and Levenshtein distance [10] between the original malware and the mutant. The code level similarity on the other hand, uses the jplag and sherlock plagiarism detectors [9] and normalised compression distance [15] in computing the similarity between the original malware and the mutant. 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 [15].

3.3 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[6] which comprises of 63 up-to-date antivirus engines and reports how many of its engines flags a sample as malicious. 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.

4 Experimental Design

The dataset utilised in this work is the Contagio Minidump[7] which comprises of mobile malware archived as APKs collected between the period of December 2011 and March 2013. We randomly select two samples belonging to two

[4] Strace - https://linux.die.net/man/1/strace
[5] Monkeyrunner - https://developer.android.com/studio/test/monkey
[6] Virustotal - https://developers.virustotal.com/reference#getting-started
[7] Contagio Minidump - http://contagiominidump.blogspot.com/2015/01/android-hideicon-malware-samples.html

families from this dump to serve as the parent malware. The parent malware chosen belong to Dougalek[8] and Droidkungfu[9] family. Dougalek family is known for stealing personal information from mobile phone users such as the user's account details or contacts. The Droidkungfu family on the other hand is known for privilege escalation and unauthorised remote control of mobile phones. Experiments are conducted separately on each family.

To assess the quality of the MAP-Elites based mutant generator, we compare its performance against that of an EA proposed in [2] for the same task. The EA in [2] is a classical EA which uses a single objective performance-based fitness function to drive evolution with no regard to features of the resulting solutions. The EA is referred to from here on as *MAL_EA*. 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 reader is referred to [2] for a detailed description of this algorithm and parameter settings. The parameters used are given in Table 1. The parameters used in MAL_EA were derived following empirical analysis. For MAP-Elites on the other hand, two of its parameters namely, selection and mutation rate are standard settings from previous literature. However, as a result of the computational cost involved in running the experiments, we limit the number of iterations to 120. The number of bootstrap iterations is then set proportionally.

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

Table 1. Evolutionary based Parameter Settings

To compare between the MAL_EA method proposed in [2] and the new MAP-Elites approach proposed here, we use four standard metrics for measuring algorithm performance which include global performance, coverage, reliability and precision, taken from [13].

Global performance (equation: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, we take the value for the best solution possible to be the single best solution obtained from any run of either algorithm. This is given

[8] Dougalek - https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/androidosdougalek.a

[9] Droidkungfu - https://www.f-secure.com/v-descs/trojan_android_droidkungfu_c.shtml

as:

$$Global\ Performance = \frac{s}{S} \tag{1}$$

Coverage (equation: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.

$$Coverage = \frac{n(f_c)}{n(F_c)} \tag{2}$$

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. 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

$$M_{x,y} = \max_{\{i \in [1,...,k]\}} m_i(x,y) \tag{3}$$

The reliability of a performance map m is given as:

$$Reliability = \frac{1}{n(M)} \sum_{x,y} \frac{m_{x,y}}{M_{x,y}} \tag{4}$$

where $x, y \in \{[x_{min}, ..., x_{max}; y_{min}, ..., y_{max}]\}$, 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, averages only the performance of only 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:

$$Precision = \frac{1}{n(M)} \sum_{x,y} \frac{m_{x,y}}{M_{x,y}} \tag{5}$$

for $x, y \in \{[x_{min}, ..., x_{max}; y_{min}, ..., y_{max}]\} | filled_m(x,y) = 1$, 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, we define performance as (1-*detection_rate*), using the definition of *detection_rate* given in section 3.3, i.e. a

solution that was not detected by any of the antivirus engines has a performance of 1.

To ensure a fair comparison, we run each algorithm for exactly same number of fitness evaluations, i.e. 120. In the case of MAP-Elites, this includes bootstrapping with $\mathcal{G} = 20$ iterations (step 4 of the Alg. 1) 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 [2], a population size of 20 is used. The EA was run 10 times with each of the three fitness functions defined in [2], the first optimising directly for evasiveness, the second optimising for behavioral 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.

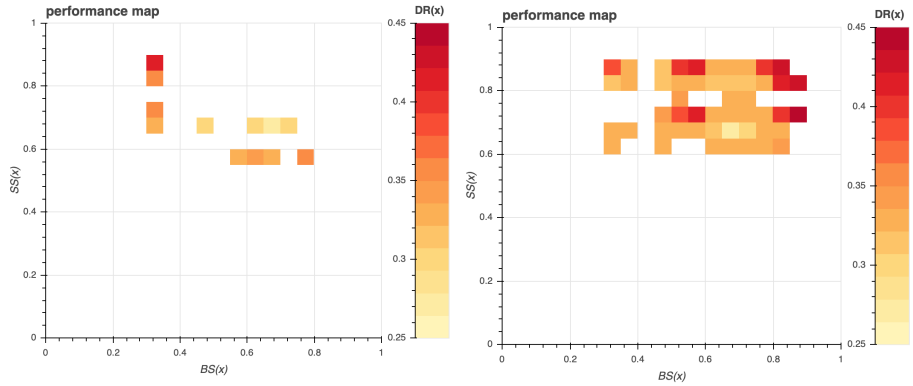
5 Results and Analysis

Here, we first provide a qualitative comparison between the MAP-Elites based mutant generator and the EA from [2]. Then, using the metrics described in Sect. 4, we carry out a quantitative comparison. We then provide additional analysis to gain insights into which of the anti-virus engines prove weakest in failing to detect the evolved variants.

5.1 Qualitative Comparison of MAP-Elites and MAL_EA

Figs. 1 and 2 show the maps obtained from merging the repeated runs of (a) the EA and (b) MAP-Elites for both Dougalek and Droidkungfu families. The x and y axes are defined by the selected feature descriptors, i.e. the behavioral (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.

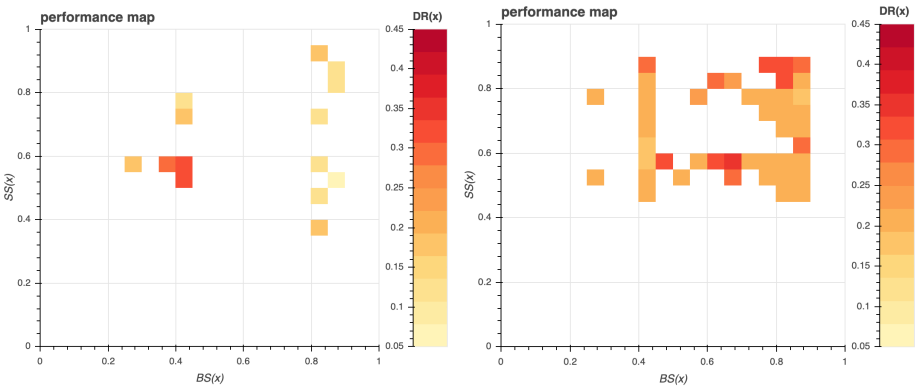
It is obvious from Fig. 1 that MAP-Elites generates a larger archive of solutions than MAL_EA for the Dougalek family. Although the solutions from both algorithms cover approximately the same *range* for each feature, the MAL_EA map is sparsely occupied with the 30 solutions obtained from the multiple 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. Similar observations apply to the Droidkungfu archives shown in Fig. 2. Although the range of the structural similarity feature extends more widely in the MAL_EA



(a) MAL_EA (b) MAP-Elites

Fig. 1. Performance map of MAL_EA and MAP-Elites for Dougalek family

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.



(a) MAL_EA (b) MAP-Elites

Fig. 2. Performance map of MAL_EA and MAP-Elites for Droidkungfu family

5.2 Quantitative Comparison of MAP-Elites and MAL_EA

In Figs. 3 and 4, we show the performance of both MAP-Elites and MAL_EA in terms of the global performance, coverage, reliability and precision metrics

		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**

Table 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).

for both Dougalek and Droidkungfu families. Mann-Whitney U tests with a 95% confidence interval are used to determine statistical significance. Table 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.

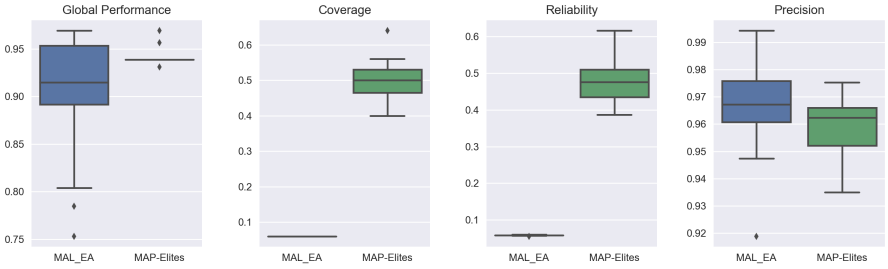
We see from Table 2 that for Dougalek, MAP-Elites does significantly better than MAL_EA in terms of global performance, reliability and coverage. For *precision* however, the significance tests fails to reject the null hypothesis that the distributions are different.

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.

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

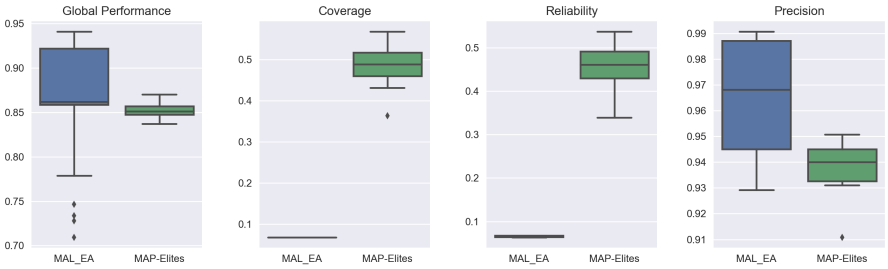
Table 3. Fitness of the best, median and worst variants produced by MAP-Elites and MAL_EA, where fitness is defined by the detection-rate, i.e. the percentage of detectors that recognise the variant as malicious. Hence, 0 represents a failure of all 63 detectors.

From Table 3, we see that both MAL_EA and MAP-Elites produce variants that are more evasive than the original malware for both Dougalek and Droidkungfu, i.e their detection rates are lower. For Dougalek, both methods produce a variant that is only detected by 28% of the detectors (compared to the original malware that was detected by 60%). 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%.



(a) Global Performance (b) Coverage (c) Reliability (d) Precision

Fig. 3. Boxplots of Global Performance, Coverage, Reliability and Precision for MAL_EA and MAP-Elites for Dougalek family



(a) Global Performance (b) Coverage (c) Reliability (d) Precision

Fig. 4. Boxplots of Global Performance, Coverage, Reliability and Precision for MAL_EA and MAP-Elites for Droidkungfu family

Although the median values are similar, the worst variant produced by MAP-Elites is more evasive than the worst variant from MAL_EA for both Dougalek and Droidkungfu.

In summary, MAP-Elites consistently outperforms the 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 family, 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.

5.3 Analysis of the antivirus engines

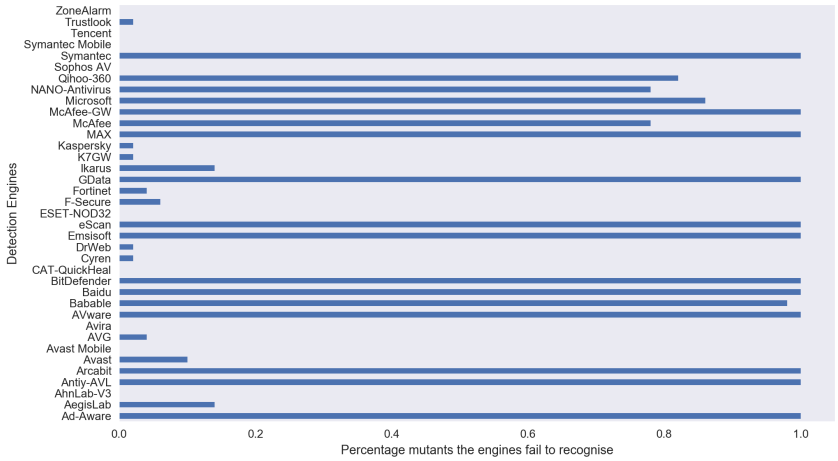
In order to gain more insight into which engines are most susceptible to potential mutated versions of the original malware, we determine the percentage of new variants evolved using MAP-Elites that a detector fails to recognise. We only consider the engines which recognised the original parent malware 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(a) and 5(b).

It can be seen from Fig. 5(a), 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. We note that the McAfeeGW engine appears very vulnerable to both families of metamorphic malware.

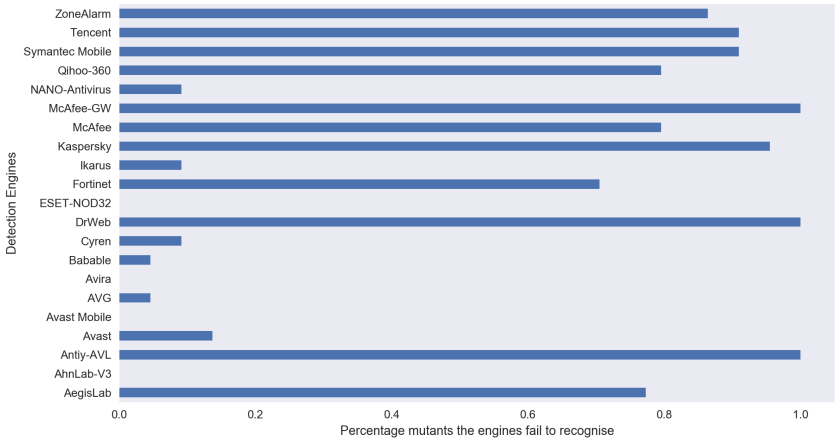
6 Conclusion

The ability of metamorphic malware to change its code over time poses significant challenges for detection models that are trained on static sets of data. One approach to this is to train models with datasets that include potential variants of the malware. It is therefore desirable to create new datasets 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.

Quality-Diversity (QD) algorithms that produce diverse archives of high-performing solutions offer an obvious solution to this problem. Although they



(a) Dougalek



(b) Droidkungfu

Fig. 5. Percentage of the mutants evolved from MAP-Elites that a specific detection engine failed to recognise for (a) Dougalek and (b) Droidkungfu

have proved effective in a range of domains in recent years, we believe this to be their first use in generating a diverse set of adversarial malware samples within the metamorphic malware detection domain. We have shown that MAP-Elites — an example of a QD algorithm — is capable of generating high-performing and diverse samples for two malware families. When compared to an EA (MAL_EA [2]), it produces larger sets of data with more diversity (with 50% (Dougalek) and 49% (Droidkungfu) coverage for MAP-Elites, as opposed to MAL_EA's coverage of 0.06% (Dougalek) and 0.07% (Droidkungfu)), while still producing comparable performance in terms of minimising detection rates. There remains significant scope to optimise the model suggested, particularly in terms of investigating an appropriate size for the archive and running the algorithm over longer periods to increase coverage.

We believe that quality-diversity algorithms are a ripe avenue for exploration in this field, particularly if they can be combined in a setting typical in generative adversarial networks (GAN [6]) in which improvements in the generated samples drive improvements in the detection method and vice versa. This will form the basis of future work.

References

1. 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)
2. Babaagba, K.O., Tan, Z., Hart, E.: Nowhere metamorphic malware can hide - a biological evolution inspired detection scheme. In: Wang, G., Bhuiyan, M.Z.A., De Capitani di Vimercati, S., Ren, Y. (eds.) Dependability in Sensor, Cloud, and Big Data Systems and Applications. pp. 369–382. Springer Singapore, Singapore (2019)
3. Bruschi, D., Martignoni, L., Monga, M.: Code normalization for self-mutating malware. IEEE Security and Privacy **5**(2), 46–54 (2007)
4. Dhakal, A., Poudel, A., Pandey, S., Gaire, S., Baral, H.P.: Exploring deep learning in semantic question matching. In: 2018 IEEE 3rd International Conference on Computing, Communication and Security. pp. 86–91. ICCCS '18 (2018)
5. Fontaine, M.C., Lee, S., Soros, L.B., De Mesentier Silva, F., Togelius, J., Hoover, A.K.: Mapping hearthstone deck spaces through map-elites with sliding boundaries. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 161–169. GECCO '19, ACM, New York, NY, USA (2019)
6. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: Advances in neural information processing systems. pp. 2672–2680 (2014)
7. Grosse, K., Papernot, N., Manoharan, P., Backes, M., McDaniel, P.: Adversarial examples for malware detection. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) Computer Security – ESORICS 2017. pp. 62–79. Springer International Publishing, Cham (2017)
8. Hagg, A., Asteroth, A., Bäck, T.: Modeling user selection in quality diversity. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 116–124. GECCO '19, ACM, New York, NY, USA (2019)

9. Heres, D.: Source Code Plagiarism Detection using Machine Learning. Ph.D. thesis, Utrecht University (2017)
10. H.Gomaa, W., A. Fahmy, A.: A Survey of Text Similarity Approaches. International Journal of Computer Applications (2013)
11. Lowd, D., Meek, C.: Adversarial learning. In: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining. pp. 641–647. KDD '05, ACM, New York, NY, USA (2005)
12. Maiorca, D., Biggio, B., Giacinto, G.: Towards adversarial malware detection: Lessons learned from pdf-based attacks. ACM Comput. Surv. **52**(4), 78:1–78:36 (2019)
13. Mouret, J.B., Clune, J.: Illuminating search spaces by mapping elites (2015)
14. Pugh, J.K., Soros, L.B., Stanley, K.O.: Quality diversity: A new frontier for evolutionary computation. Frontiers in Robotics and AI **3**, 40 (2016)
15. Ragkhitwetsagul, C., Krinke, J., Clark, D.: A comparison of code similarity analysers. Empirical Software Engineering **23**(4), 2464–2519 (2018)
16. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: Evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. pp. 329–334. ASIA CCS '13, ACM, New York, NY, USA (2013)
17. Suciu, O., Coull, S.E., Johns, J.: Exploring adversarial examples in malware detection. In: 2019 IEEE Security and Privacy Workshops. pp. 8–14. IEEE SPW '19 (2019)
18. Urquhart, N., Hart, E.: Optimisation and illumination of a real-world workforce scheduling and routing application (wsrp) via map-elites. In: Auger, A., Fonseca, C.M., Lourenço, N., Machado, P., Paquete, L., Whitley, D. (eds.) Parallel Problem Solving from Nature – PPSN XV. pp. 488–499. Springer International Publishing, Cham (2018)
19. 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)
20. Zheng, M., Lee, P.P.C., Lui, J.C.S.: Adam: An automatic and extensible platform to stress test android anti-virus systems. In: Flegel, U., Markatos, E., Robertson, W. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 82–101. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)