Adapting Novelty towards Generating Antigens for Antivirus systems

Ritwik Murali**
C Shunmuga Velayutham
m_ritwik@cb.amrita.edu
Dept. of Computer Science & Engg.
Amrita School of Engineering - Coimbatore,
Amrita Vishwa Vidyapeetham
Coimbatore, Tamil Nadu, India
cs_velayutham@cb.amrita.edu

ABSTRACT

It is well known that anti-malware scanners depend on malware signatures to identify malware. However, even minor modifications to malware code structure results in a change in the malware signature thus enabling the variant to evade detection by scanners. Therefore, there exists the need for a proactively generated malware variant dataset to aid detection of such diverse variants by automated antivirus scanners. This paper proposes and demonstrates a generic assembly source code based framework that facilitates any evolutionary algorithm to generate diverse and potential variants of an input malware, while retaining its maliciousness, yet capable of evading antivirus scanners. Generic code transformation functions and a novelty search supported quality metric have been proposed as components of the framework to be used respectively as variation operators and fitness function, for evolutionary algorithms. The results demonstrate the effectiveness of the framework in generating diverse variants and the generated variants have been shown to evade over 98% of popular antivirus scanners. The malware variants evolved by the framework can serve as antigens to assist malware analysis engines to improve their malware detection algorithms.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; **Intrusion/anomaly detection and malware mitigation**; • **Information systems → Expert systems**; **Decision support systems**; • **Applied computing → Personal computers and PC applications**; • **Computing methodologies → Bio-inspired approaches**; **Generative and developmental approaches**.

*Both authors contributed equally to this research.

GECCO '22, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9237-2/22/07…$15.00
https://doi.org/10.1145/3512290.3528693

KEYWORDS

Application of Evolutionary Algorithms, Evolutionary Algorithm, Malware, Malware Generation, Proactive Defence, Virus

ACM Reference Format:
Ritwik Murali and C Shunmuga Velayutham. 2022. Adapting Novelty towards Generating Antigens for Antivirus systems . In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3512290.3528693

1 INTRODUCTION

To the average user, a secure computer system is synonymous with the installation and usage of an automated antivirus software. These antivirus software are expected to detect and/or prevent malicious programs from affecting the end user's computing system. Most of the antivirus (AV) scanners identify malicious programs by examining new software/programs for predefined malicious patterns. These malicious patterns (called signatures) are identified previously by malware analysts and included in the antivirus scanner softwares database [28], thus enabling automated detection. While malicious software or malware have been prevalent since the early 1970s [1] [37] and the use of up-to-date antivirus software has simplified the process of securing a computing system, it is a fact that the AV scanners still struggle to identify variants of malicious programs. This is because even minor modifications in the malware code result in a change in the known malicious pattern, thus enabling the malware to evade detection by the AV scanners [22]. The identification / detection of malware variants usually requires manual intervention and it is quite impractical for an analyst to predict and identify signatures of unknown malware variants [29]. The significance of the problem is further emphasized by the 2021 SonicWall Cyber Threat Report, where 589,313 new malware variants were identified by the SonicWall team in 2020 [36]. To put in perspective, this is a 57.35% increase from 2019 where 153,909 malware variants were detected by the same company.

To counter this problem, anti-malware research has used two strategies with opposing perspectives. The first is a defensive strategy that applies Artificial Intelligence, Machine Learning [5, 15, 24, 33, 43, 44], Data Mining [8, 16, 42], Evolutionary Algorithms [23, 25, 34, 40], Optimization Strategies [21] and Knowledge Framework [12] based techniques to detect and predict malware variants

[1]Throughout this paper, the terms *malware* and *virus* are used interchangeably

by identifying specific characteristics or features of the malware executable in order to detect and/or classify its variants. However, the performance (in terms of accuracy and false positives) of these techniques are dependant on the underlying data set used for training and classification and are severely affected by the shortage of such publicly available labelled data sets [1]. The second is a more aggressive / proactive strategy that involves creating malware variants using generative techniques, such as Generative Adversarial Networks (GANs), or meta-heuristic algorithms, like Evolutionary Algorithms (EAs), to reduce the real-world impact of the variants. While GANs exploit the non-linear structure of neural networks to generate complex adversarial examples capable of evading the target model [2, 14, 19, 35, 38], EAs use biological evolution inspired strategies to generate malware capable of evading antivirus scanners [9, 30]. In most cases, the proactive malware generation approach acts to augment the effectiveness of the defensive strategy as the malware variants generated serve as a valid database upon which the former can be trained or tested.

The use of EAs for malware generation is mainly focused on two approaches. The first involved the identification of various malware features with the EA being used to search for new variants with novel combinations of the identified features. The features were at the application level and the EA ensured that the malware variants generated have similar malicious characteristics. This was demonstrated by Noreen et al.[31] who used a version of the email worm *Bagle* to illustrate the application of evolutionary algorithms in malware generation. The worm was represented as a genome, which is essentially a collection of all the attack features such as date, port number, domain, email body, email subject, etc. The evolution involved searching for different combinations of attack features' values facilitated by mutation. The work also served as a framework for the application level approach and also used genetic algorithms as an evolutionary tool while showcasing a comparative study of different parent selection and crossover techniques that could be used for the malware, more specifically, computer worm generation.

The second approach involves the use of the underlying assembly code of the malware as the platform for variation. Since all executables (irrespective of the source programming language) can be disassembled to their assembly code structure, the second approach is gaining popularity. The explored approaches include using byte level modifications of the Windows Portable Executable (PE) files [7], evolving entropy based polymorphic packers [26], identifying locations within valid executable to hide malicious code [6] and even a bottom up strategy of malware generation using Backus–Naur form (BNF) grammar to form the production rules to build code based on a designed grammar [32]. EAs have also been used to test the available AV solutions among mobile systems as well [3, 27, 41] in an effort to confuse the AV systems to misclassify malware and/or evolve variants to test the AV systems themselves.

While multiple algorithms are able to evolve an assortment of variants, the generation of a malware variant dataset, calls for creating a large number of malware variants possibly through code or feature modifications. In both cases, the modifications or transformations should also ensure and result in a diverse set of valid malware variants that serve as good representatives of the malware variants' space.The focus so far, has broadly been on evolving variants that evade the antivirus scanners. Additionally, these strategies employ diverse malware representation and evolutionary operators as directed by the representation. Therefore, it is also increasingly challenging for the practitioner to validate the quality and innovations (in terms of complex code mutations) of the generated malware variants.

Traditional evolutionary algorithms however are typically driven to converge to a fitness optimum. However, such a fitness strategy may not be apt when attempting to promote diversity and uniqueness in the resultant population. Novelty search is a divergent search algorithm that has the ability to promote evolvability [17]. Evolvability allows the evolutionary algorithm to generate variability [11]. This technique, inspired by natural evolution, rewards individuals that exhibit novel behaviours. The selection process for novel individuals therefore depends on the distance of the individual from its nearest neighbours in the behaviour space [18]. Novelty search also suggests several strategies to manage the archive of past solutions. These include collecting the individuals whose novelty was above a threshold when first evaluated, the most novel individuals at each generation or even randomly chosen individuals if not none at all [10].

In this work, we present an evolutionary algorithmic approach to evolve valid and potential (capable of evading effective anti-virus scanners) variants of a given malware. We consider the evolved (generated) malware variants as *antigens* which can be presented to malware analysis engines to train and improve their malware detection algorithms thereby providing active acquired immunity to the end system against the existing numerous malware variants. We hypothesize that a novelty search based approach is capable of generating malware variants of greater diversity than a simple similarity based approach and verify the hypothesis with our Malware Antigens Generating Evolutionary algorithm (MAGE). The algorithm would serve as a foundation for an assembly source code based framework towards generating diverse, valid and potential variants (capable of evading antivirus scanners). The modular design of the proposed framework facilitates extensibility in terms of malware representation, code transformation functions, quality indicator and the underlying evolutionary algorithm.

2 MALWARE ANTIGEN GENERATING EVOLUTIONARY ALGORITHM (MAGE) AS A FRAMEWORK

A framework that facilitates automated malware variant generation must handle two major challenges: (1) Ensure that the framework always results in a valid executable and the executable retains its maliciousness, and (2) The design of the framework should be flexible and modular such that the practitioner can induce application specific changes as per their requirement. In the context of an evolutionary algorithm (E), the automated malware variant evolution can be formally started as follows. To apply the evolutionary algorithm (E) on a given virus assembly code (ζ) to generate variants (ζ')

using assembly code transformation functions (T) for code mutations such that the maliciousness ($\Omega(\zeta')$) of the evolved variants is retained and the variant dataset generated by E is diverse in nature.

Therefore the proposed framework (F) includes the components E, ζ, ζ', Δ and ξ, where $T \in \{\psi_p, \tau_q, \sigma_r\}$ and R are variation operators (Δ), ξ refers to the quality indicator or fitness function and ψ, τ and σ are the control flow modifications, data transformations and code layout changes respectively. This implies that any E can operate on ζ and employ $\Delta = \{T, R\}$ as variation operators and ξ to evaluate the candidates to realize the automated generation of valid variants ζ' of ζ. In the perspective of an evolutionary algorithm, the framework models the different representations possible in an assembly language environment and discusses a few of the variation operators ($T, R \in \Delta$) and uses a modified version of novelty as the fitness function (ξ). The general structure of a typical population based meta-heuristic algorithm is given in Algorithm 1. Each module in the framework is flexible enough to be customized as per the practitioners requirement as can be observed during the description of the Malware Antigens Generating Evolutionary algorithm (MAGE) in the next section.

Algorithm 1 Structure of an evolutionary algorithm

```
1:  procedure EVOLUTIONARY ALGO()
2:      INITIALIZE a population using random individual solutions
3:      EVALUATE every individual in current population
4:      while NOT TERMINATION CONDITION do
5:          SELECT parents using any selection strategy
6:          RECOMBINE the parent pairs        ▷ i.e. perform crossover
7:          MUTATE the resultant child        ▷ i.e. the new individual
8:          EVALUATE the new individual
9:          SELECT individuals for the next generation
10:     end while
11: end procedure
```

2.1 Representation

From the perspective of the evolutionary algorithm, considering the case of the assembly language code structure, there are a number of ways the source code can be represented. Here each individual in the population is the entire assembly code (program) of the chosen malware (or even a variant). The representation is quite flexible and allows the individual to be modelled either in a linear fashion (figure 1) or as a graph (figure 2). Both the framework and the evolutionary algorithm proposed in this work adopt the linear representation of the malware code for the simulation experiments.

2.2 Quality indicator

The quality indicator ξ (fitness function in EA) of the framework quantifies the extent of the transformation introduced in the source assembly code. In the context of malware variant generation, this work attempts to explore a variation of novelty search as a strategy to yield divergent code structures. Considering each individual as a vector of assembly code statements (as the linear representation

Figure 1: Linear representation of virus code.

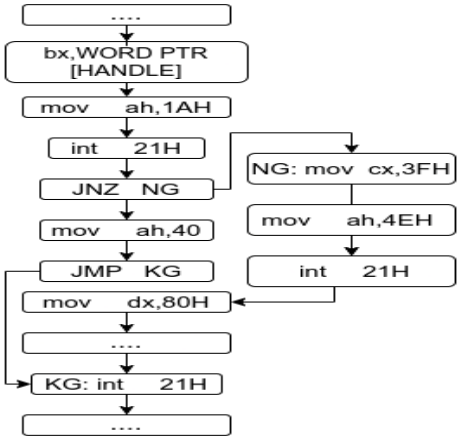

Figure 2: Graph representation of virus code.

dictates), the quality indicator (ξ) is calculated as the euclidean distance between each individual in the population and the mean vector ($\bar{S}$) of all intra-population similarity vectors in the population as shown in equation 1. Here P is the population size.

$$\xi = \sqrt{\sum_{i=1}^{P} (\bar{S}, \overrightarrow{SI_i})^2} \tag{1}$$

The similarity vector $\overrightarrow{S}I_i$ for the i^{th} individual within the population P is $\overrightarrow{SI_i} = J(p_1, p_i), J(p_2, p_i), ..., J(p_{NP}, p_i), J(\zeta, p_i)$ where $p_1, ..., p_{PG}$ is the rest of the population comprising all chromosomes except p_i , ζ is the source virus code and $J(p_j, p_i)$ is the Jaccard similarity index between a j^{th} and i^{th} chromosome as shown in equation 2.

$$J(h_j, h_i) = \frac{|h_j \cap h_i|}{|h_j \cup h_i|} = \frac{h_j \cap h_i}{|h_j| + |h_i| - h_j \cap h_i} \tag{2}$$

The Jaccard similarity index was chosen after preliminary experiments revealed that this was the most stringent metric for generating divergent malware. Nevertheless, it is worth reiterating that the choice(s) proposed for each key component of the framework

is merely suggestive and a practitioner, by virtue of the modular nature of the framework, can employ their choice(s) for each key component.

2.3 Variation Operations

Given the representation, the variation operations Δ are defined as mutation and crossover operations. For the chosen assembly language representation, the framework proposes code transformation operators as both mutation (T) and crossover (R) operations.

2.3.1 *Mutation:*

In the case of malware, the framework exploits the assembly level constructs commonly used by malware authors for evading antivirus scanners [13], to model the transformation operators (T). Therefore the transformation operators can be considered as a subset of the code evasion operators namely modifying control flow (ψ), Transforming Data (τ)) and Changing Code Layout (σ) (i.e. $T \subseteq \{\psi, \tau, \sigma\}$) and this includes the use of opaque predicates (T_{OP}), bogus insertions (T_{BI}), branching functions (T_{BF}), instruction transformation (T_{IT}), code block reordering (T_{RB}), variable substitution (T_{VS}) and code block substitution (T_{CBS}). Each of the transformation operators are limited by a set of constraints (C) that define the allowed structure resulting from transformations thus limiting the potentially infinite productions and ensuring valid code structures.

Based on generic transformation functions ($T_{OP}, T_{BI}, T_{BF}, T_{IT}$, T_{RB}, T_{VS} and T_{CBS}), five transformation function instances namely Fake Instruction (T_{FI}), Forced JMP (T_{FJ}), Untouchable Block (T_{UB}), Conditional Zero JMP (T_{CZJ}) and Conditional Non Zero JMP (T_{CNZJ}), have been defined and proposed. It is worth mentioning that every assembly language program code has a standard prologue (header), a program body and epilogue (footer). Therefore, all applications of code transformations are constrained by a common rule (C_{com}) which states that "Every code transformation must be applied in the program body of the input assembly code". It should be noted that only the common constraint c_{com} is applicable in the case of *Fake instructions* (T_{FI}), such as 'NOP', which can be inserted anywhere within the program body as it does not affect the code functionality in any manner. C_{com} here ensures that a valid executable is created post application of the transformation. However, while multiple *NOP* instructions can be inserted in any location within the program body, excessive insertion of this instruction would result in the code bloating thus increasing the chances of generating invalid executables.

"JMP" is an assembly language instruction that alters the control flow of the assembly code and can be inserted anywhere within a contiguous code block. This is a logical choice as insertion of a JMP statement beyond the boundaries of any function is meaningless since the command will never be called and would act as dead code. The JMP instruction is always paired with a *label* and this acts similar to a function call and in the framework F, must be defined beyond any continuous block of assembly code. This is to ensure that the label definition does not interrupt or affect any existing function blocks or contiguous code structures. The label definition should end with a reference/return to the line/command immediately after the JMP call so as to ensure continuity in the control flow of the code. This also ensures that nesting of such a transformation results in complex but linear code flow as shown in figure 3. Based on the above understanding of the JMP statement, the *Forced JMP* (T_{FJ}) transformation is a logical control flow modification choice for code mutations and it is defined as $T_{FJ} = (C(JMP, \$label, s_1, \$label :, s_1))$ where "label" refers to the logical position of the control flow transfer and s_i stands for each subsequent assembly language instruction. The flexibility of the JMP statement can also be exploited to create the *Untouchable Block* transformation, formally defined as $T_{UB} = (C(JMP, \$label, s_k^+, \$label :))$. Here, immediately after the JMP statement multiple assembly language statements (s_k^+) may be inserted. However, since none of these instructions will ever be executed, all of them act as dead code. The lines of code and structure of the code is changed here while retaining the control flow. *Conditional JMP* (T_{CZJ} and T_{CNZJ}) transformations check the status of the zero flag at that specific instance of code execution before executing a JMP instruction. This further increases the number of code level variations that can be evolved by the evolutionary algorithm.

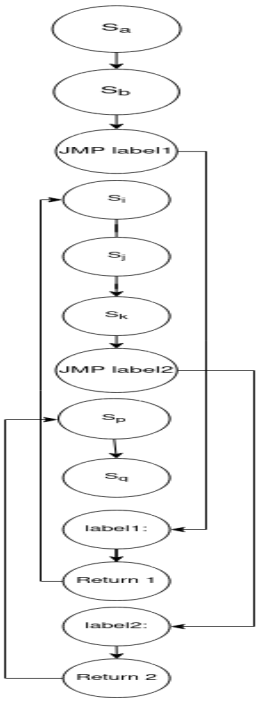

Figure 3: Forced JMP Transformation

There are also certain transformation functions that are classified under ψ, τ and σ, commonly used in higher level programming languages. These include operations such as loop unrolling, class transformations, array transformations, etc., which are not feasible to apply in the context of assembly language transformations because the format is not supported in assembly languages and/or they do not always result in a valid executable post application on code. Therefore such transformations have not been considered in the framework proposed. The chosen transformations ensure that

the transformed code is always a valid executable and thus form potential candidates for mutation operations $M \in \Delta$.

2.3.2 *Crossover:*

The crossover or recombination operation ($RE \in \Delta$) is another integral variation operation of an evolutionary algorithm. In the context of the code transformation functions described above, the recombination operation can be realized as *code block transformation* (T_{CBI}) and can be defined as follows. If P is the population of all individuals in the solution space of a single generation then p and $\hat{p}$ in the population represent two candidate parents for the crossover operation. In the context of the proposed assembly code based framework, $p = \{S^+, LOP^+, COP^+\}$ and $\hat{p} = \{\hat{S^+}, \hat{LOP}^+, \hat{COP}^+\}$. Then $\langle O, \hat{O} \rangle := RE(p, \hat{p}) := T_{CBI}\{(S^+, LOP^+, COP^+), (\hat{S^+}, \hat{LOP}^+, \hat{COP}^+), C_i\}$ where O, $\hat{O}$ are the offsprings resulting from recombination operation, LOP & COP represent the loop & conditional statements in the assembly language construct, and T_{CBI} represents code block interchange transformation function. Table 1 shows a summary of the instances of the variation operators.

The code block interchange transformation function T_{CBI} interchanges blocks of code that might contain statements as well as loop and conditional operands. In the absence of a constraint, T_{CBI} has the potential to yield invalid executables as offsprings and hence can be very disruptive. Consequently, the choice of code block location, (henceforth called pivot point) is very crucial to ensure that code block interchange does not interrupt the sequential execution of the resultant assembly code. The constraint then requires the interchange to be applied between code blocks above the pivot point and similarly swap code blocks below the pivot point as well. This constraint reduces the possibility of a label defined by any transformation function being lost during interchange operation else an undefined label would result in an invalid executable.

Table 1: Summary of variation operator instances

Notation	Description	Definition
T_{FI}	Fake Instruction	NOP
T_{FJ}	Forced JMP	$(C(JMP, \$label, s_1, \$label:, s_1))$
T_{UB}	Untouchable Block	$(C(JMP, \$label, s_k^+, \$label:))$
T_{CZJ}	Conditional Zero JMP	$(C(JZ, \$label, s_1, \$label:, s_1))$
T_{CNZJ}	Conditional Non Zero JMP	$(C(JNZ, \$label, s_1, \$label:, s_1))$
$RE(p, \hat{p})$	Crossover between 2 parents where T_{CBI} represents code block interchange	$T\_CBI\{(S^+, LOP^+, COP^+), (\hat{S^+}, \hat{LOP}^+, \hat{COP}^+), C_i\}$

As shown in Figure 4, the pivot point is randomly chosen to be any point beyond non-overlapping block of statements (s_i). By virtue of this pivot point, all transformation functions discussed so far (including T_{CBI}) can be applied both above and below the pivot point. Each shaded block in Figure 4 shows the space available for application of transformation operations as constrained by the pivot point. Since all the transformation functions described so far can be applied either within a block or between blocks, there remains ample opportunity for these transformation functions to introduce arbitrary complexity in the code yet retaining its capability for execution. Without loss of generality, the pivot point can

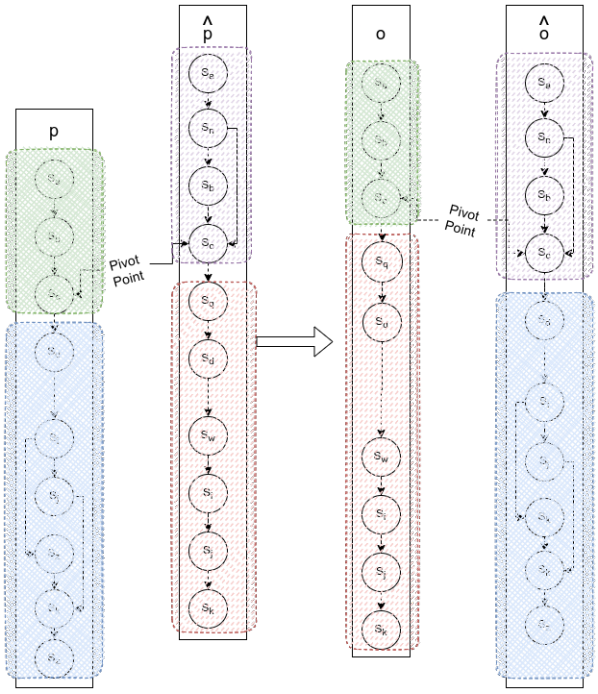

Figure 4: Code block interchange transformation

be considered to be in the middle of the source assembly code thus facilitating equal opportunity to the blocks above and below for arbitrary transformations. It is worth observing that T_{CBI} is a potential candidate transformation function for realizing single-point and multi-point crossover operations (with the latter requiring multiple pivot points).

2.4 Malware Antigen Generating Evolutionary algorithm (MAGE)

The Malware Antigen Generating Evolutionary algorithm (MAGE) has been designed based on the framework and the overview of MAGE is shown in algorithm 2. Therefore, MAGE also seamlessly merges into the form of a traditional EA as shown previously by algorithm 1. Each chromosome generated by MAGE, by virtue of the transformation functions, is a potential variant. This implies that MAGE is able to evolve $P \times G$ variants (P is the population and G is the generation) virus variants, making MAGE a generative algorithm. Additionally, following the novelty search ideals, MAGE also identifies the novel chromosomes in each generation and collates them in the form of a unique dataset. While every chromosome is a potential variant, this dataset of unique variants can serve as the antigens using which the antivirus systems can update their signature database. The intention behind the automated malware generation using MAGE is not just to generate virus variants evading AV scanners but to generate variants as diverse as possible in terms of the assembly code structure.

Algorithm 2 MAGE as a framework

procedure EVOLUTIONARY ALGO(ζ)
  Generate Initial Population using transformations $T_i \in \{\psi, \tau, \sigma\}$ with a probability ${p_m}^i$
  while (EXE Generated) || (Generations limit yet to be reached) **do**
    Compute fitness $\xi = \sqrt{\sum_{i=1}^{P} (\overrightarrow{SI_i}, J_i)^2}$ for every candidate i in current population ▷ EVALUATE
    while maximum population size limit not reached **do**
      `Select two parents using the tournament selection strategy` ▷ SELECTION
      `Perform crossover` ▷ RECOMBINE
      `Apply code transformation mutation functions (i.e.`$T_i \in \{\psi, \tau, \sigma\}$`) each with a probability` ${p_m}^i$ `on each of the resultant offspring` ▷ MUTATE
      `Add each evolved child to the next generation population`
      `Identify the novel individuals from current population and add them to a "Unique" variant dataset`
    end while
  end while
end procedure

3 EXPERIMENTS & DISCUSSION

In order to validate the capability of the framework in evolving divergent virus variants, MAGE was seeded with the assembly source of *Intruder* - a virus that can infect .EXE files and jump across directories and even drives [20]. The *Intruder* virus attaches itself to the end of any .EXE program and takes control of the program when it first starts. The virus first locates all files (including those from the sub directories) possible to infect and verifies if it can be infected before actually infecting the .EXE file. Since *Intruder* is an extremely infectious virus, the test-bed consisted of an isolated computer, inside which a 32bit Microsoft Windows 7 guest virtual machine was used. The host operating system was a 64bit Linux Mint system. This ensured that the experiments that were conducted did not escape the test environment and spread. The host machine had an Intel© Core™ i5-2400 CPU @ 3.10GHz with 4 CPU cores and 8 GB RAM with a 1TB Hard disk. The guest virtual machine used 3 CPU cores and 4GB of RAM with an execution cap of 80%. Regular snapshots of the virtual machine was taken to keep track of any changes the system might exhibit and thereby identify possible accidental infections. The guest windows OS also had a copy of the Microsoft Macro Assembler (MASM) which is required to make the virus an executable. Also, since the focus is on evolving divergent malware from a single source, no other obfuscation techniques or packers were employed during the test.

The purpose of the experiments was to verify if a novelty supported intra-population similarity based approach is capable of generating malware variants of greater diversity than a simple similarity-only based approach. Therefore, the simulation experiments involved evaluating the diversity, in terms of code structures, the result of applying MAGE with two different quality indicators

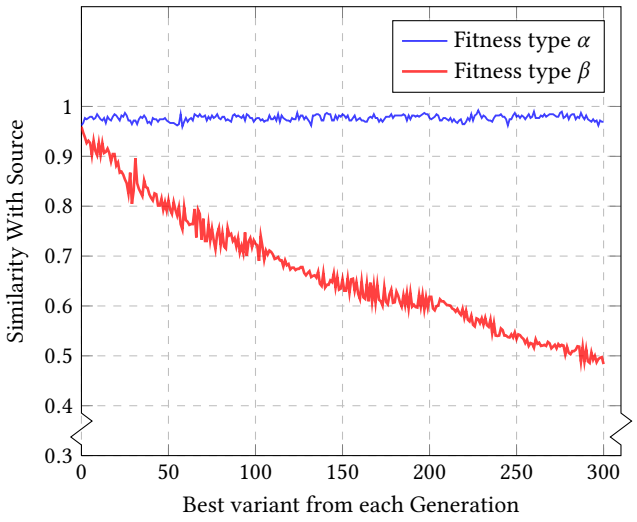

Figure 5: Similarity values of Initial and Final population using α and β

(fitness functions) α and β. The indicator α solely depended on Jaccard similarity as a metric for evaluation, while the fitness function β exploited the Jaccard similarity index based intra-population similarity. It is also worth mentioning that the focus of the work is on a framework for evolving divergent variants rather than benchmarking. In order to ensure fairness during comparison, the same random seed was set and used for the experiments and only the Jaccard similarity best fit individuals from each generation were compared with the source virus. The population size has been set at a random value of 20 and the EA has been run for 300 generations. The transformation functions T_{FI}, T_{FJ}, T_{UB}, T_{CZJ} and T_{CNZJ} are employed to mutate assembly code structure and T_{CBI} is employed for crossover operation. The results (figure 5) show that a novelty supported intra-population similarity based fitness function is able to induce more variations within the population.

A statistical analysis using the Mann-Whitney U test was also performed on the similarity value of the variants evolved in both the initial and final populations. The Mann–Whitney U test is a popular non-parametric hypothesis test that verifies a null hypothesis (H_0) and a research hypothesis (H_1). In the current analysis, the null hypothesis (H_0) states that the initial and final populations are the same. Conversely, in this context, the research hypothesis (H_1) states that the initial and final populations are different. The test calculates the value of U which ranges from 0 to $n_1 \times n_2$, where n_1 and n_2 are the sizes of each population. Based on the probability (p) that the results appear by chance, a high value of U accepts the null hypothesis (H_0) and rejects the research hypothesis (H_1). Conversely, a low value of U rejects the null hypothesis (H_0) and accepts the research hypothesis (H_1), also based on the probability (p) of the results appearing by chance. In the case of most biological analyses, it is usually admissible to use $p < 0.01$ as the threshold for acceptance for the Mann-Whitney U test.

Table 2: Similarity Values of Initial population vs Final population

	Similarity with Source (α)		**Novelty** (β)	
	Initial Population	*Final Population*	*Initial Population*	*Final Population*
1	0.9844155844	0.9973684211	0.981865285	0.4834183673
2	0.9717948718	0.9793281654	0.981865285	0.4922077922
3	0.9973684211	0.9921465969	0.9742930591	0.500660502
4	0.9844155844	0.9973684211	0.9973684211	0.4947780679
5	0.981865285	0.9844155844	0.961928934	0.4884020619
6	0.9717948718	0.9973684211	0.9793281654	0.4896640827
7	0.9947506562	0.9973684211	0.9947506562	0.5060080107
8	0.9844155844	0.9768041237	1	0.4834183673
9	0.961928934	0.9717948718	0.9869791667	0.5039893617
10	1	0.9973684211	0.9793281654	0.506684492
11	0.9844155844	0.9895561358	0.9768041237	0.4973753281
12	1	0.981865285	0.981865285	0.4980289093
13	0.981865285	0.981865285	0.9895561358	0.49672346
14	0.981865285	0.9844155844	0.981865285	0.5013227513
15	1	0.9947506562	0.981865285	0.4947780679
16	0.9895561358	0.9973684211	0.961928934	0.4947780679
17	0.9643765903	0.9768041237	0.9594936709	0.4922077922
18	0.9768041237	0.9793281654	0.9844155844	0.5039893617
19	0.9693094629	0.9947506562	0.9869791667	0.5039893617
20	0.9844155844	0.9895561358	0.9844155844	0.506684492

On performing the Mann-Whitney U Test on the initial and final populations (shown in Table 2) resulting from using α as the fitness metric in MAGE, the U, p and $zscore$ values were calculated. Both $U = 161$ and $zscore = -1.04143$ fall within the acceptable range (where 128.065 to 271.935 is the region of acceptance for U and between -1.96 and 1.96 is the region of acceptance for z score), and the value of p was 0.29834, which is not within the permissible threshold limit of 0.01. This implied that the hypothesis H_0 cannot be conclusively rejected and so the two populations are considered to be similar. On the other hand, when the Mann-Whitney U Test was performed on the initial and final populations resulting from using β as the fitness metric in MAGE, the difference between the randomly selected values of the initial and the final populations was large enough to be statistically significant. The value of U was 400 and $zscore$ was 5.4054 (both outside the region of acceptance, namely from 127.6622 to 272.3378 for U and from -1.96 to 1.96 for z score) and p was 6.467×10^{-8} which is definitely less than the threshold of 0.01. From this it is clear that the hypothesis H_0 can be conclusively rejected and the research hypothesis H_1 (which states that the two populations are different) can be accepted. To further illustrate that the evolution was indeed encouraged by the cascading effect of the fitness function, the Mann-Whitney U Test was also applied on the initial populations of both distributions. The results ($U = 223, zscore = 0.6133, p = 0.5397$) show that null hypothesis is true and both the populations are same. Since initial populations of both the distributions are level, it can be concluded that the diversity in the populations was more pronounced the during evolution, when using a novelty search based fitness function (β). This further supports the hypothesis that a novelty search based approach (β) is capable of generating malware variants of greater diversity than a simple similarity based (α) approach.

At this juncture, it is worth mentioning that, since the aim of the EA was to evolve divergent variants that are capable of evading antivirus scanners while not altering the maliciousness of the resultant variant, the evasion capability of the virus and its variants was

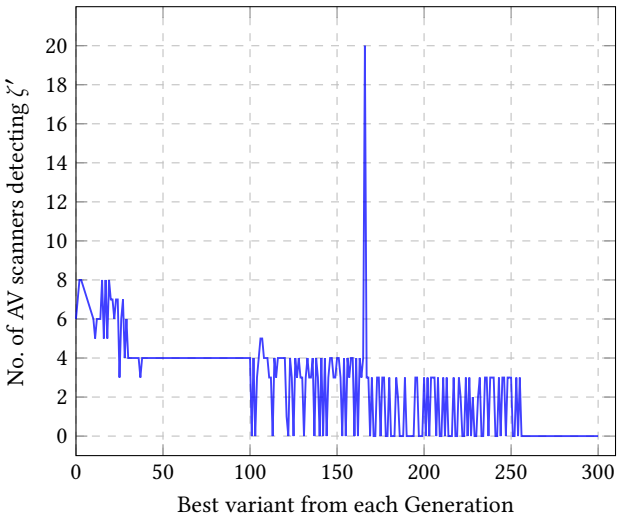

Figure 6: Evasion capability of MAGE: Number of antivirus scanners detecting the generated variants

also checked using VirusTotal [39] which used over 60 antivirus scanners to scan each executable. The VirusTotal results showed that 20 popular antivirus scanners were able to identify the source *Intruder* virus. However, it was observed that by the 250^{th} generation, almost all the antivirus scanners used by VirusTotal (over 98%) were evaded by variants evolved. By way of example, figure 6 displays evasion capability of variants evolved by MAGE when using β as the quality indicator. This observation was expected as it is a well known fact that even minor modifications in code is sufficient to evade most antivirus scanners. It is also worth noting that, by virtue of the transformation functions (T) and well defined recombination operator (R), MAGE is capable of evolving valid virus executables for over 600 generations i.e. $600 \times 20 = 12,000$ divergent virus variables successfully. This is because "bloat" is a common factor in EAs [4] and the Microsoft Macro Assembler (MASM) required that the files remained under the 64KB limit.

The above simulation experiments have demonstrated that a novelty supported intra-population similarity based approach is capable of generating diverse variants of a given malware. Subsequent Mann-Whitney U test based statistical analysis on the similarity value of the variants evolved revealed that the diversity in the populations was more pronounced during the evolution, when using a novelty search based fitness function (β). The validity of the framework was also demonstrated by the proposed Malware Antigen Generating Evolutionary algorithm (MAGE) which evolved diverse virus variants that evaded detection by over 98% of all antivirus scanners in VirusTotal.

4 CONCLUSION

This work discussed a generic assembly source code based framework that facilitates an evolutionary algorithm to generate diverse

and potential variants of an input malware, while retaining its maliciousness, yet capable of evading antivirus scanners. The generic code transformation functions $T \subseteq \{\psi, \tau, \sigma\}$ based on which five transformation function instances namely Fake Instruction (T_{FI}), Forced JMP (T_{FJ}), Untouchable Block (T_{UB}), Conditional JMP (T_{CZJ}) and Conditional JMP (T_{CNZJ}), were also proposed and defined as mutation operators. The code block interchange transformation function (T_{CBI}) was utilized in the design of the crossover operator to enable seamless recombination resulting in a valid executable.

The validity of the framework was also demonstrated by the proposed Malware Antigen Generating Evolutionary algorithm (MAGE) which utilized a novelty search supported intra-population based fitness function to evolve diverse variants of a source malware. The simulation experiments were performed using *Intruder* - a virus that attaches itself to the end of any .EXE program. The versatility, efficacy and flexibility of the MAGE was also demonstrated by utilizing two different quality indicators (fitness functions) α and β, to evolve valid variants of the *Intruder* virus. VirusTotal was utilized to observe the evasion capability of the evolved *Intruder* virus variants. Results show that almost all the antivirus scanners were evaded by the evolved variants after 250 generations of evolution. Statistical analysis on the initial and final populations further revealed that the novelty supported intra-population based fitness function (β) was able to direct MAGE to evolve divergent malware variants of the given source malware.

Since every candidate evolved by MAGE is a potential variant, the entire collection of variants generated could serve as the malware variant dataset. This dataset of antigens could then be presented to the malware analysis engines to improve their malware detection algorithms. Thus, in conclusion, the framework and MAGE could serve as a flexible platform for researchers and practitioners to develop novel applications under the aegis of automated malware generation for proactive defence.

REFERENCES

[1] Giovanni Apruzzese, Michele Colajanni, Luca Ferretti, Alessandro Guido, and Mirco Marchetti. 2018. On the effectiveness of machine and deep learning for cyber security. In *2018 10th International Conference on Cyber Conflict (CyCon)*. IEEE, IEEE, Tallinn, 371–390.

[2] Kshitiz Aryal, Maanak Gupta, and Mahmoud Abdelsalam. 2021. A Survey on Adversarial Attacks for Malware Analysis. *CoRR* abs/2111.08223 (2021), arXiv–2111. arXiv:2111.08223 https://arxiv.org/abs/2111.08223

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

[4] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. 1998. *Genetic programming: an introduction*. Vol. 1. Morgan Kaufmann Publishers San Francisco, California.

[5] Shamik Bose, Timothy Barao, and Xiuwen Liu. 2020. Explaining ai for malware detection: Analysis of mechanisms of malconv. In *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, IEEE, Glasgow, 1–8.

[6] Andrea Cani, Marco Gaudesi, Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. 2014. Towards automated malware creation: code generation and code integration. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, Gyeongju, Republic of Korea, 157–160.

[7] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo. 2019. AIMED: Evolving Malware with Genetic Programming to Evade Detection. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, Rotorua, 240–247.

[8] Rory Coulter, Qing-Long Han, Lei Pan, Jun Zhang, and Yang Xiang. 2020. Code analysis for intelligent cyber systems: A data-driven approach. *Information sciences* 524 (2020), 46–58.

[9] T Divya and Kandasamy Muniasamy. 2015. Real-time intrusion prediction using hidden Markov model with genetic algorithm. In *Artificial intelligence and evolutionary algorithms in engineering systems*. Springer, New Delhi, 731–736.

[10] Stephane Doncieux, Alban Laflaquière, and Alexandre Coninx. 2019. Novelty search: a theoretical perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Prague, 99–106. https://doi.org/10.1145/3321707.3321752

[11] Stephane Doncieux, Giuseppe Paolo, Alban Laflaquière, and Alexandre Coninx. 2020. Novelty search makes evolvability inevitable. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. ACM, Lille, 85–93.

[12] Weijie Han, Jingfeng Xue, Yong Wang, Fuquan Zhang, and Xianwei Gao. 2021. APTMalInsight: Identify and cognize APT malware based on system call information and ontology knowledge framework. *Information Sciences* 546 (2021), 633–664.

[13] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. 2018. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology* 104 (2018), 72–93.

[14] Weiwei Hu and Ying Tan. 2017. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. https://doi.org/10.48550/ARXIV.1702.05983

[15] Seungho Jeon and Jongsub Moon. 2020. Malware-detection method with a convolutional recurrent neural network using opcode sequences. *Information Sciences* 535 (2020), 1–15.

[16] Alireza Khalilian, Amir Nourazar, Mojtaba Vahidi-Asl, and Hassan Haghighi. 2018. G3MD: Mining frequent opcode sub-graphs for metamorphic malware detection of existing families. *Expert Systems with Applications* 112 (2018), 15–33.

[17] Joel Lehman and Kenneth O Stanley. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation* 19, 2 (2011), 189–223.

[18] Joel Lehman and Kenneth O Stanley. 2011. Improving evolvability through novelty search and self-adaptation. In *2011 IEEE congress of evolutionary computation (CEC)*. IEEE, New Orleans, 2693–2700.

[19] Yuanzhang Li, Yaxiao Wang, Ye Wang, Lishan Ke, and Yu-an Tan. 2020. A feature-vector generative adversarial network for evading PDF malware classifiers. *Information Sciences* 523 (2020), 38–48.

[20] Mark A. Ludwig. 1991. *The Little Black Book of Computer Viruses*. Amer Eagle Pubns Inc, Arizona.

[21] Noah MacAskill, Zachary Wilkins, and Nur Zincir-Heywood. 2021. Scaling Multi-Objective Optimization for Clustering Malware. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, Orlando, 1–8.

[22] Alexey V Malanov and Vitaliy A Kamlyuk. 2012. Rapid heuristic method and system for recognition of similarity between malware variants. US Patent 8,250,655.

[23] Farnoush Manavi and Ali Hamzeh. 2019. A new approach for malware detection based on evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, Prague, 1619–1624.

[24] KannanMani S ManiArasuSekar, Paveethran Swaminathan, Ritwik Murali, Govind K Ratan, and Surya V Siva. 2020. Optimal feature selection for non-network malware classification. In *2020 International Conference on Inventive Computation Technologies (ICICT)*. IEEE, IEEE, Coimbatore, 82–87.

[25] Syed Bilal Mehdi, Ajay Kumar Tanwani, and Muddassar Farooq. 2009. Imad: in-execution malware analysis and detection. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, Montréal, 1553–1560.

[26] Héctor D Menéndez, David Clark, and Earl T Barr. 2021. Getting ahead of the Arms Race: Hothousing the Coevolution of VirusTotal with a Packer. *Entropy* 23, 4 (2021), 395.

[27] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. 2016. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*. ACM, Xi'an, 365–376.

[28] Peter Morley. 2001. Processing virus collections. *VIRUS* 129 (2001), 129–134.

[29] Ritwik Murali, Akash Ravi, and Harshit Agarwal. 2020. A Malware Variant Resistant To Traditional Analysis Techniques. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*. IEEE, Chennai, 1–7.

[30] Ritwik Murali and C Shunmuga Velayutham. 2020. A preliminary investigation into automatically evolving computer viruses using evolutionary algorithms. *Journal of Intelligent & Fuzzy Systems* 38, 5 (2020), 6517–6526.

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

[32] Sadia Noreen, Shafaq Murtaza, M Zubair Shafiq, and Muddassar Farooq. 2009. Using Formal Grammar and Genetic Operators to Evolve Malware. In *Recent Advances in Intrusion Detection (RAID)*. LNCS, Springer, France, 374–375.

[33] TG Gregory Paul and T Gireesh Kumar. 2017. A framework for dynamic malware analysis based on behavior artifacts. In *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications*. Springer,

Springer, Bhubaneswar, 551–559.

[34] M Zubair Rafique, Ping Chen, Christophe Huygens, and Wouter Joosen. 2014. Evolutionary algorithms for classification of malware families through different network behaviors. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, Vancouver, 1167–1174.

[35] Abhishek Singh, Debojyoti Dutta, and Amit Saha. 2019. MIGAN: malware image synthesis using GANs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. Association for the Advancement of Artificial Intelligence, Honolulu, 10033–10034.

[36] SONICWALL. 2022. *2021 SonicWall Cyber Threat Report*. SONICWALL. Retrieved January 28, 2022 from https://www.sonicwall.com/medialibrary/en/white-paper/2021-cyber-threat-report.pdf

[37] Peter Szor. 2005. *The Art of Computer Virus Research and Defense: ART COMP VIRUS RES DEFENSE _p1*. Addison Wesley Professional, USA.

[38] Wee Ling Tan and Tram Truong-Huu. 2020. Enhancing Robustness of Malware Detection using Synthetically-adversarial Samples. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE, Taipei, 1–6.

[39] VirusTotal. 2021. Getting started with VirusTotal. https://developers.virustotal.com/reference Last accessed August 2021.

[40] Zachary Wilkins and Nur Zincir-Heywood. 2020. COUGAR: clustering of unknown malware using genetic algorithm routines. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. ACM, Cancún, 1195–1203.

[41] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang. 2017. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security* 12, 7 (2017), 1529–1544.

[42] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. 2017. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)* 50, 3 (2017), 1–40.

[43] Suyeon Yoo, Sungjin Kim, Seungjae Kim, and Brent Byunghoon Kang. 2021. AI-HydRa: Advanced hybrid approach using random forest and deep learning for malware classification. *Information Sciences* 546 (2021), 420–435.

[44] Nur Zincir-Heywood, Marco Mellia, and Yixin Diao. 2021. *Overview of Artificial Intelligence and Machine Learning*. Wiley Online Library, New Jersey. 19–32 pages.