Expert Systems With Applications 226 (2023) 120092

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Evolving malware variants as antigens for antivirus systems

Ritwik Murali [a,*], Palanisamy Thangavel [b], C. Shunmuga Velayutham [a]

[a] Department of Computer Science & Engineering, Amrita School of Computing- Coimbatore, Amrita Vishwa Vidyapeetham, Tamil Nadu, India
[b] Department of Mathematics, Amrita School of Physical Sciences- Coimbatore, Amrita Vishwa Vidyapeetham, Tamil Nadu, India

ARTICLE INFO

Keywords:
Antigens
Evolutionary algorithms
Malware
Malware evolution
Virus variants

ABSTRACT

This paper proposes MAGE — A Malware Antigen Generating Evolutionary algorithm that is capable of generating unseen variants of a given source malware. MAGE evolves malware variants by employing code transformation functions as mutation operators and intra-population Jaccard similarity metric as fitness function. By virtue of these design choices, MAGE is capable of generating active malware variants with diverse code structure variations while retaining the maliciousness of the source malware. These malware variants (similar to biological antigens) generated throughout the run of MAGE forms a potential dataset of malware variants. The dataset can be used to train an adaptive Antivirus engine to learn the code structure variations that make up the space of malware variants. This could augment the engines ability to detect unseen malware variants, thus preventing attacks from the same. The efficacy of MAGE has been demonstrated with two malware viz. *Timid* , a COM infector and *Intruder*, an EXE infector. The simulation experiments demonstrate the potential and versatility of MAGE towards generating diverse malware variants.

1. Introduction

Malicious programs and software have been prevalent since the early 1970s with the *Creeper* virus first appearing on the ARPANET. This, followed by the *Elk Cloner* and the *Brain* in the 1980s with latter as the first PC virus in the wild, heralded the age where even machines are prone to diseases (Dwan, 2000). Naturally, the genesis of anti-virus programs too date back to the early 1970s starting with *Reaper* (though a benevolent virus itself but developed to get rid of *Creeper*). Since then a number of virus and anti-virus programs have appeared with the latter combating the former and vice versa. With the advent of the digital age, the security of end users' computers largely rests with effective Anti-Virus (AV) scanners making anti-virus programs indispensable (Mawgoud et al., 2021).

However, despite their widespread use, it is also well known that these AV scanners struggle to identify the maliciousness of a program/file/software as even minor modifications to code structure result in the malware (used interchangeably with virus henceforth in this paper) variants evading detection (Malanov & Kamlyuk, 2012). Most AV scanners find it difficult to identify malware variants as they are minor modifications in the malware code and thus do not match with the pre-identified malware signature. This poses a major challenge for anti-malware researchers developing malware scanning engines as they are unable to predict and identify all the possible variants without manual intervention. To combat this, anti-malware companies are now embracing Artificial Intelligence & Machine Learning (Bose et al., 2020; Chen et al., 2018; Khalilian et al., 2018; Liu et al., 2020; Paul & Kumar, 2017; Santacroce et al., 2020; Stiborek et al., 2018; Wadkar et al., 2020), Deep Learning (Yazdinejad et al., 2020; Zhong & Gu, 2019) and Evolutionary Algorithm (Afaneh et al., 2013; Divya & Muniasamy, 2015; Wu & Banzhaf, 2010) based strategies to detect malware and its variants. However, the performance (in terms of accuracy and false positives) of these learning algorithms are dependent on the underlying data set used for training and classification and are severely affected by the lack of availability of such publicly available labelled data sets (Apruzzese et al., 2018).

The development of such data sets demand creating a large number of malware variants possibly through code modifications. Most importantly, the code modifications should also ensure and result in a diverse set of malware variants that serve as good representatives of the malware variants' space. This malware variants' data set creation is a herculean task if attempted through manual approach. Consequently, we need to resort to automated methods to create malware variants. In this paper, we present an evolutionary algorithmic approach to evolve valid and potential (capable of evading effective anti-virus scanners) variants of a given malware. These evolved malware variants (similar to biological *antigens*) can then be presented to malware analysis engines to train and improve their malware detection algorithms thereby providing active acquired immunity to the end system

* Corresponding author.
E-mail addresses: m_ritwik@cb.amrita.edu (R. Murali), t_palanisamy@cb.amrita.edu (P. Thangavel), cs_velayutham@cb.amrita.edu (C. Shunmuga Velayutham).

https://doi.org/10.1016/j.eswa.2023.120092
Received 2 June 2021; Received in revised form 14 March 2023; Accepted 8 April 2023
Available online 16 April 2023

against the existing numerous malware variants. The Evolutionary Algorithm (EA) presented in this paper, called *MAGE - M*alware *A*ntigens *G*enerating *E*volutionary algorithm, employs well-known code transformation functions to mutate malware code structure to evolve potential *active* malware variants. MAGE also employs an intra-population Jaccard similarity fitness function to ensure that the mutated variants are diverse in terms of code structure variations.

Using the well known COM infector virus "*Timid*" as source, MAGE was able to mutate the assembly level code structure using transformation functions to evolve a number of effective and unique *Timid* virus variants. The evolved variants were able to successfully evade over 60 well known antivirus scanners with as high as 90% evasion rate. The choice of transformation functions as mutations and crossovers, ensure that every chromosome generated during the course of MAGE's run to be an active *Timid* virus variant thus making MAGE a potential automated approach to assist anti-malware researchers towards generating a data set of virus variants. The generic nature of MAGE towards generating virus variants has been demonstrated on another EXE infector virus *Intruder*.

Specifically, our contributions in this paper are:

- An evolutionary algorithm (MAGE) to evolve potential virus variants from a single source virus towards creating a virus database.
- The choice of transformation functions and a Jaccard similarity based intra-population fitness function which serves as generic primitives for evolving assembly code structural modifications.
- Validation on two viruses (COM infector and EXE infector)
- MAGE as an Evolutionary Engine for antivirus researchers to generate valid & diverse variants for any given malware.

It is worth pointing out that the focus of this proposed work is towards demonstrating the potential of MAGE in generating unique virus variants and so creating a virus variants' data set than on the evasion efficacy of the dataset itself. This dataset could serve as antigens and be used to augment an antivirus scanners' ability to detect unseen malware variants thus preventing infection.

The rest of the paper is organized as follows: Section 2 reviews the related works in the literature aligning with the proposed work; Section 3 presents the design of MAGE for evolving diverse malware variants; Section 4 details simulation design, studies and analyses; finally Section 5 concludes the paper.

2. Background & related work

A computer virus is a program that can infect other programs by modifying them to include a possibly evolved copy of itself (Cohen, 1987) causing widespread destruction while remaining undetected. Consequently, antivirus (AV) scanners detect and prevent viruses from causing mayhem. The scanners use binary analysis techniques (roughly classified into static, dynamic and symbolic or concolic analysis) to analyse potentially malicious program executables to build its human readable representation for automated detection (Murali et al., 2020). Key features of the virus are identified/extracted to form a signature. The antivirus scanner scans all new binaries in the computing environment for these signature patterns which identify the malicious entity.

Alternatively, Biological Immune Systems (BIS) are also used as models which attempt to detect and purge the virus from the computing environment (Lamont et al., 1999). The BIS are comparable to AV systems where both have a "detector" to differentiate between host and non-host elements; a "classifier" to isolate non-host elements and group them based on their characteristics; a "cleansing agent" to eliminate the non-host element and a "memory" to identify such elements and the corresponding action taken for future responses. However BIS are usually used as Intrusion Detection Systems (IDS) meant to identify potential attacks in a network while AV systems focus on malware detection on the host device and evolutionary algorithms have been used successfully to evolve exploits (attacks) to automate the testing of IDS (Dasgupta, 2012; Kayacık et al., 2011). These strategies utilized system processes, API calls and other anomaly detection techniques to evolve exploits. However, this paper focuses on generating malware (antigens) variants for AV systems rather than BIS and uses the underlying assembly code structure of the malicious executables to generate diverse variants. The aim is not restricted to generating variants that evade the antivirus scanner, rather, it is centred around the possibility of generating diverse variants of a given malicious code (without affecting the functionality of the malware) so as to build a reliable malware variant dataset.

Most of the recent work available concentrates on generating signatures for malware analysis, detection and classification. Noreen et al. (2009a) proposed the first formal literature available that discussed the possibilities of joining the two domains of malware and evolutionary algorithms. The authors use a version of the email worm *Bagle* to validate the application of evolutionary algorithms in malware. The worm genome was represented as a collection of all the attack features such as date, port number, domain, email body, email subject, etc. The mutation was also on the application level by changing the content in the features. Shuffling between @hotmail.com, @msn.com, etc. for the domain feature could be considered as an example of the mutation in the domain feature. However, the genome representation, mutation, etc, remains in the application domain and consequently implies that the solution is not scalable towards other applications and is useful only if the application domain remains constant. In the case of computer viruses, such constant factors are extremely rare. Subsequently, Noreen et al. (2009b) proposed using formal grammar and genetic operators to evolve malware. The authors formally designed the *mini44* malware using Backus–Naur form or Backus Normal Form (BNF) grammar to form the production rules and leverage evolutionary algorithms to build code based on the grammar designed. Their work is an interesting direction towards malware evolution, however, creating grammars for each malware in order to evolve variants is in itself a tedious task.

Cani et al. (2014) proposed two ways to make use of evolutionary algorithms for proactive defence, namely "Code Generation" and "Code Integration". The focus of their work is in the *code integration* aspect which attempts to identify locations within valid executable which could hide malicious code. The *code generation* aspect is more related to this proposed work. The authors make use of a general purpose EA toolkit μGP (Squillero, 2005) to generate the virus, however, the authors are unable to emphatically state that the evolved program still retains the malicious character of the original. Castro et al. (2019) used genetic programming to demonstrate that malware can be evolved. It is interesting to note that their work implements byte level mutations that modify the Windows Portable Executable (PE) files (as detailed by Anderson et al. (2018) and includes manipulating debug info, packing or unpacking the file, etc.) by injecting modifications into the Windows PE files. However, as the authors point out, 53% of the malware variants generated were corrupt. Our proposed work aims to show that an alternate approach using vanilla virus source with an intra-population similarity based fitness function at its core is able to generate a far more divergent collection of valid virus variants which would be more useful to improve the AV scanners. However, it is worth mentioning that the GP based approach in Castro et al. (2019) can also be adapted for diversity. More recently, Menéndez et al. (2021) made use of evolutionary computation to evolve an entropy based polymorphic packer (El Empaquetador Evolutivo (EEE)) that entered into a co-evolutionary arms race with Virus Total. Their work also studied Virus Totals learning rate and detection rate by analysing the EEE generated variants which were detected by Virus Total (which took 3 days to learn the patterns and two days to forget them). In fact, in a prior work, the author (Menéndez et al., 2019) stated that entropy played a significant role in detecting patterns altered by concealment strategies. The summary of similar work is given in Table 1. At this juncture, it is also worth mentioning that there are techniques that

Table 1
Summary of similar work.

Authors	Variation type	Adaptable?	Comment
Noreen et al. (2009a)	Application level	N	Feature based variations are not adaptable to code level changes
Noreen et al. (2009b)	Grammatical	N	Defining grammar for malware is a challenge
Cani et al. (2014)	Code level	N	Focus of work is on identifying locations within an EXE to hide malicious code
Castro et al. (2019)	Code level	Y	Computes similarity of generated variant with source as a measure of diversity
Menéndez et al. (2019)	Code level	N	Focus is on packers as a wrapper over the malicious code and not vanilla malware

```
mov  bx,ax
push bx
mov  cx,5
..
  ..
mov ah,3FH
int  21H
pop  bx
...
```

Fig. 1. Snippet of chromosome (malware) representation in MAGE. Linear representation where a single malicious program is the chromosome and each line of code is a gene in the chromosome.

enlist the byte code to randomly modify the malware structure (Lin & Stamp, 2011; Madenur Sridhara & Stamp, 2013; Tamboli et al., 2014; Venkatachalam & Stamp, 2011) and direct it towards evading detection by antivirus scanners. While this proposed work does discuss evasion, it is focused on generating divergent variants to assist AV scanners (and does not focus on concealment/obfuscation strategies). The entropy of each of the evolved variants have been calculated to observe the level of diversity on our proposed evolutionary algorithm towards generating valid virus variants from vanilla assembly code.

EAs have also been used in the mobile malware domain to test the available AV solutions (Aydogan & Sen, 2015; Meng et al., 2016; Xue et al., 2017; Zheng et al., 2012) in an effort to obfuscate mobile malware, confuse the AV systems to misclassify the malware and/or evolve variants to test the AV systems themselves as well. In fact, Babaagba et al. (2020a) were successfully able to evolve diverse malware variants in the android environment using the MAP-Elites Algorithm and subsequently were able to validate that the evolved variants were useful in training various machine learning algorithms as well (Babaagba et al., 2020b). The feature descriptor used by Babaagba et al. (2020a) is of particular interest as the structural similarity metrics used a combination of cosine similarity, fuzzy string match, levenshtein distance, normalized compression distance and results from jplag and sherlock plagiarism detectors. All these values were then averaged to between 0 and 1. However, since this work focuses on generating diverse yet active virus variants capable of affecting the traditional Microsoft Windows operating system, the underlying language for mutations is the assembly language instructions. Consequently, many of the standard string matching approaches such as fuzzy hashing are quite poor when working on the assembly code (Haq & Caballero, 2021). This is further complicated by the fact that the disassembled assembly code would consist of a very large number of repeated strings (as the instruction set is usually limited), thus requiring extremely large computation resources (and resulting in hardware crashes), to use any string matching/ n-gram approaches to identifying text similarity. Therefore, only the metrics that can be effectively used without large computation resources requirements are used for generating diverse malware variants. A this juncture, it is worth mentioning that genetic algorithms are frequently used in malware detection and classification problems for feature selection (Al-Sahaf & Welch, 2019; Harahsheh et al., 2022) as well as to generate detection rules (Jerbi et al., 2021). The algorithms are used to augment the capabilities of existing machine learning and deep learning models using dynamic analysis features alongside prediction accuracy, precision, recall, etc.

The proposed work is significantly different from existing strategies as it does not attempt to generate signatures, rather, given a single malware assembly code structure, evolutionary algorithms are used to transform the underlying assembly code structure so as to generate a diverse set of variants of the input malware. Existing reverse engineering tools such as the Interactive Disassembler (IDA), Radare2, etc., can be used to extract the assembly code structure from a malware executable and MAGE is applied on the reversed assembly code. The generic code transformation functions employed by MAGE ensure that the malware thus generated retain its functionality while resulting in a valid executable.

3. Automated generation of malware variants using MAGE

The standard strategy to evade AV systems is to transform a virus code into a modified code structure, yet retaining its original behaviour, thereby masquerading the same virus as an acceptable computer program. The typical virus code transformation methods (called transformation functions henceforth in this paper) include modifying control flow, transforming data and changing code layout (Hosseinzadeh et al., 2018). This paper intends to employ EAs to automatically transform the code structure of a given virus code using the above mentioned transformation functions as code mutations in order to generate potential virus variants. At this juncture, we reiterate that the focus and potential of the work is on generating/evolving diverse virus variants rather than attempting to evade AV scanners. Formally, this EA based automated generation of malware variants can be stated as follows.

Given a virus code C *(and the possibility of a large number of potential variants* C_i', $\{\forall i \rightarrow 1, 2, 3, \ldots, n; n \rightarrow \infty\}$, *EA based malware variant generation problem is to apply an Evolutionary Algorithm* E *on* C *to generate potential variants* C_j' $\{j \rightarrow 1, \ldots, NP\}$; NP *is the population size (with* $C_j' \subset C_i'$ *possibly good representation of variant space), using transformation functions* F_j *for code mutation i.e.* $C_j' = F(C)$, *where* $F_j \subset \{\psi_p, \tau_q, \sigma_r\} \forall j\, p, q, r \rightarrow 1, 2, 3, \ldots, n$ *(comprising chosen code transformation function instances* (p, q, r) *respectively from control flow modification* (ψ), *data transformation* (τ) *and code layout change* (σ) *and* F *is the combination of transformation functions* $F_j s$*) such that the maliciousness of each of the generated variants is retained, the resultant variant dataset is diverse in nature and the generated variants evade good number of if not all of the well-known industry standard AV scanners.*

Timid, a well-known COM infector virus targeting Microsoft Windows OS has been chosen as the candidate virus for all simulation experiments and *The Little Black Book of Computer Viruses* (Ludwig, 1991) provides the source for the virus. The infection process of the virus involves searching the current directory for uninfected files and attaching itself to the end of these uninfected files. It does not traverse

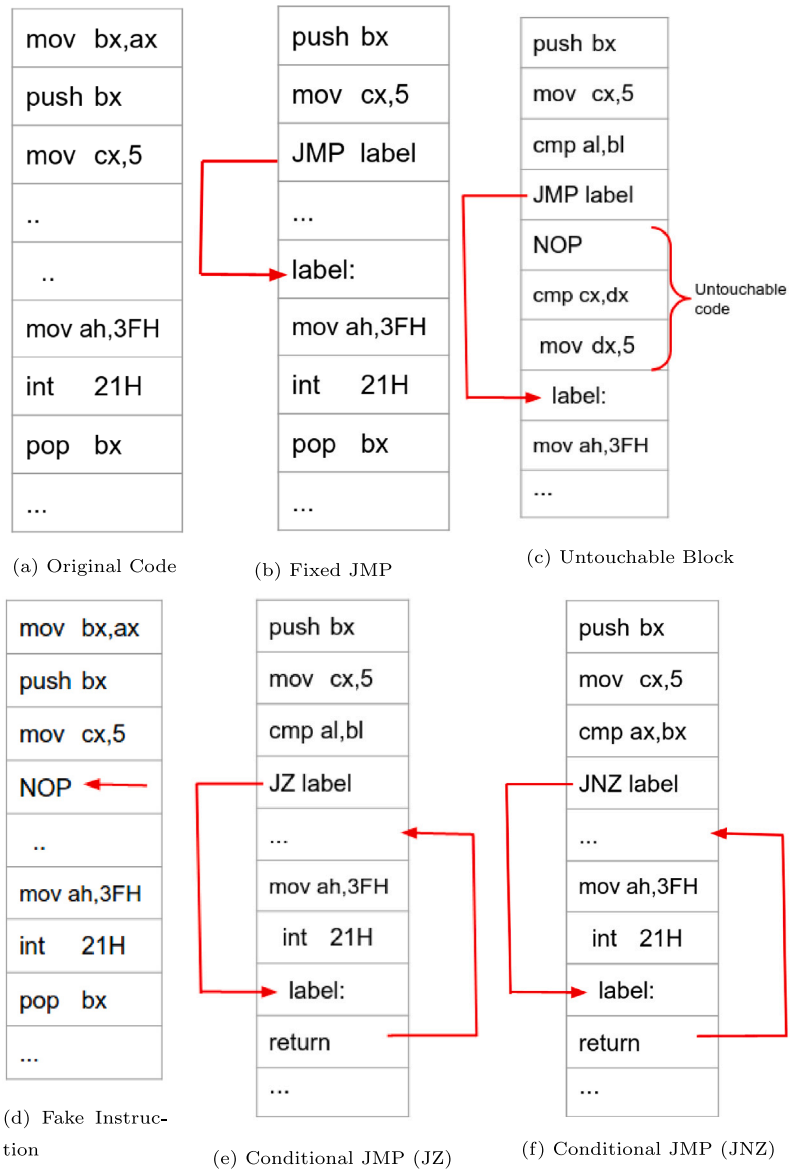


Fig. 2. Code transformation functions employed as mutation operators in MAGE.

multiple directories or files and so can be easily contained. It also does not perform any destructive actions such as file deletion and the like. It merely copies itself (infects) onto any COM files in current directory. Since this is also well known, it can easily be detected by antivirus scanners. The virus is also very small, consisting of only 264 bytes of machine language instructions, is reasonably safe (i.e. controllable) and the compilable assembly code of the virus is also available in public domain.

Factors such as code availability in public domain, the ability to control the virus (i.e. prevent accidental spread) and easy detection by most of the existing AV scanners (thus helping with checking the extent of virulence as a result of evolution), as well as related literature (Cani et al., 2014), led to this choice of the candidate virus for the simulation experiments. As the static malware analysis operations are predominantly on the assembly code of malware to develop the signatures required for AV systems, this work employs MAGE on the assembly code of *Timid* virus. Consequently each chromosome in the population is the entire assembly code (program) of *Timid* (or possibly its variants) with each line of code (i.e., each instruction) representing a gene. The genotype is the complete genetic makeup of the malware and a snippet of the genotype is shown in Fig. 1. Except for the header and footer portions of the malware code file (that stores information on the size of the file, meta-data about the code part, and details regarding the data part), the genotype is the same as the phenotype. The Fig. 1 clearly shows the instruction sequence that will be followed by the malware upon being executed. While the individual files are linear, the diversity is between the files are brought in through the evolution of the virus code. This linear representation of code using the genotype, when combined with careful choice of variation operators (mutation and crossover), that act upon the instruction (gene level), will ensure that every chromosome generated, as the evolution progresses, to be an active virus entity.

Often computer virus variants are created manually by using a combination of code transformation functions to modify the virus code structure. This in turn modifies the virus signature and helps it to evade AV scanners. Transformation functions thus serve as potential tools to masquerade computer viruses. A careful choice of transformation function instances, provided as mutation operators, will facilitate MAGE to compose and try out different combinations of those operations on different code locations of the input computer virus. The standard transformation functions include control flow modification, data transformation and code lay-out change. There are a number of code

transformation instances under each of the above said three functions. By way of an example, modifying code control flow includes strategies such as code reordering, instruction transformation, polymorphism, adding fake instructions etc. (Hosseinzadeh et al., 2018).

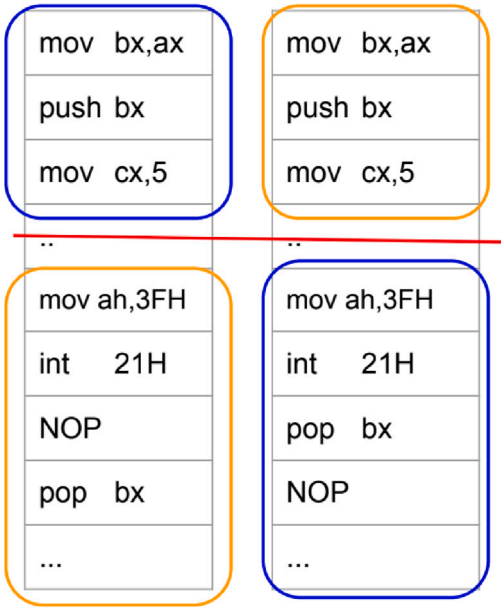
As the proposed work employs MAGE to mutate assembly level code structure of a given virus, the code transformation function instances have to be appropriately selected. Operations such as *adding fake instructions, control flow reordering* involving *fixed JMP, JNZ, JZ and untouchable block insertion* were chosen as the mutation operations. Fig. 2 shows a pictorial description of the chosen code transformation function instances. A snippet of the original *Timid* source code is shown in Fig. 2(a) and acts as the reference to demonstrate the other mutation operators. The JMP operator serves in a similar manner to a function call in usual programming environments. This changes the overall flow of execution by inserting another branch into the code flow structure as shown in Fig. 2(b). With respect to the instructions JZ and JNZ, they are usually preceded by a CMP instruction which verifies if the two values in the given register location are equal. If they are, then the zero flag is set. Figs. 2(e) and 2(f) show that the jump or call is itself conditional based on the current status of the zero flag.

Fake or bogus insertions are when instructions that have no effect on the assembly code are injected into the code itself. One of the most popular such instructions is NOP (Fig. 2(d)) which performs no action, but makes it harder to identify code memory locations for static analysis. Similarly, the untouchable block, as shown in Fig. 2(c) also inserts multiple lines of junk code into the assembly code structure, but this is preceded by a JMP command which, during execution, skips these inserted lines thereby ensuring code flow continuity while circumventing static pattern based analyses. It is worth pointing out that the chosen strategies do not change the nature of the virus but always result in active executable yet retaining the maliciousness. In addition, they have the potential to evade AV scanners; they are generic enough to be primitives for any virus assembly code mutations and they facilitate simple to complex code structure modifications.

A single-point crossover operator is also employed in MAGE to assist the mutation operators towards exploring the space of virus variants. As the crossover operator can be very disruptive more often yielding invalid executables as offsprings, the choice of crossover point (which we call *pivot point*) is very crucial. The pivot point is selected in such a manner that it does not interrupt any sequential execution of the assembly code. However identifying the similar pivot points across chromosomes is no straightforward task. We instruct MAGE to make this pivot point absolute by choosing it on the input computer virus itself and apply the mutation operations on code structures either above or below the pivot point. These code structures above and below the pivot point will be subjected to a variety of mutation combinations in the population across generations. Consequently, despite the absolute pivot point, swapping blocks above and below the pivot point results in sufficient code structure variety thus assisting in exploring the virus variant space. Fig. 3 depicts the crossover operation. Along with the code transformation mutation functions, the pivot point based crossover operator ensures that every chromosome evolved throughout the course of MAGEs' run is a valid active variant of the source malware.

The intention behind the automated malware generation using MAGE is to generate variants as diverse as possible in terms of the assembly code structure and not focused on generating virus variants evading AV scanners. Such a diverse set of variants form a potential data set to augment AV scanners and to enable them to detect unseen variants of the given computer virus. Besides in preliminary experiments it has been observed that even the most rudimentary mutations (code transformations) of a given computer virus enabled it to evade a number of AV scanners. A fitness function, based on the evasion of AV scanners as a matter of fact, misleads MAGE to resort to the most rudimentary and simplest of code transformations to achieve the goal of evasion (Murali & Shunmuga Velayutham, 2020). Instead, a

(a) Parent Pair



(b) After Crossover

Fig. 3. Single point crossover on malware code (pivot point shown as line).

fitness function involving code similarity has the potential to drive evolution to apply variation operators in such a way that more and more dissimilar code structures get generated resulting in diverse virus variants.

Given a population of chromosomes, Jaccard similarity index between each chromosome and the rest of the population (i.e. $NP-1$ chromosomes) as well as the original source virus is calculated. Jaccard similarity index measures similarity between two assembly code sets S_1 and S_2:

$$J(S_1,S_2)=\frac{S_1\cap S_2}{S_1\cup S_2}=\frac{S_1\cap S_2}{|S_1|+|S_2|-S_1\cap S_2} \tag{1}$$

where $0\le J(S_1,S_2)\le 1$. This intra-population Jaccard similarity index calculation results in a NP-dimensional similarity vector for each of the chromosome in the population where NP is the population size. The fitness value for each chromosome is the Euclidean distance between its intra-population similarity vector and the mean vector computed from the intra-population similarity vectors of all the chromosomes in the

population. The fitness function, thus, can be defined as

$$F = \|\bar{S} - S_i\| \tag{2}$$

with

$$S_i = (J(c_1, c_i), J(c_2, c_i), \ldots, J(c_{NP}, c_i), J(C, c_i)) \tag{3}$$

where $\bar{S}$ is the mean vector of all intra-population similarity vectors in the population, S_i is the intra-population similarity vector of the ith chromosome, $c_1, \ldots, c_{NP}$ is the rest of the population comprising all chromosomes except c_i, C is the source virus code and $J(c_j, c_i)$ is the Jaccard similarity index between a jth and ith chromosome. As can be seen from Eq. (2), the larger the distance between the intra-population similarity vector of a chromosome from the mean similarity vector, the more diverse the chromosome in terms of code structure from the rest of the chromosomes in the population. This intra-population similarity based fitness function drives MAGE to employ code transformation mutation functions in such a way that diverse code structure variations are effected on malware code.

Algorithm 1: Overview of MAGE

Input: A single known virus
Output: Pool of Distinct Virus Variants
Generate Initial Population using code transformation mutation functions each with a probability p_m;
while *Maximum number of generations yet to be reached* **do**
  for *every chromosome i in current population* **do**
    Generate a similarity vector S_i (Eq. (3));
  Generate mean vector $\bar{S}$ for current population;
  for *every chromosome i in current population* **do**
    Compute fitness $F = ||\bar{S} - S_i||$;
    if *unable to create EXE* **then**
      End Run ;
    else
      Continue ;
  while *population size limit not reached* **do**
    Select two parent chromosomes through tournament selection;
    Perform single-point crossover with a probability p_c;
    Apply code transformation mutation functions each with a probability p_m on each of the resultant offspring ;
    Add each evolved child to the next generation population;

Fig. 4 illustrates the antigen evolution process using MAGE as the generation engine and algorithm 1 gives the overview of MAGE. As each chromosome generated throughout the course of MAGE is a potential variant, the emphasis has been on generating more diverse malware variants (i.e. N× G variants where N is the population size and G is the total number of generations) by making MAGE a generational evolutionary algorithm. Experiments regarding the choice of the fitness functions and variation operators that resulted in MAGE are discussed in Section 4. We performed the experiments on the understanding that 32-bit executables work not just in a 32-bit environment, but also a 64-bit environment. MAGE has the flexibility to accept the updated transformation functions applicable to the underlying architecture and also has the potential to work equally effectively on a 64-bit architecture with only the instruction set being changed. By way of example, Fig. 5 shows the disassembled code of "Gen_Heur_PonyStealer_4" - A malware first seen in 2018 that usually steals passwords. The disassembly was performed using the free version of IDA pro - an extremely popular reverse engineering tool. IDA pro supports only a "generic assembler for 80 × 86 processors' (hex-rays, 2022) and this assembly code cannot be directly reassembled or recompiled (Wang et al., 2015)". Therefore, while this reverse engineered assembly code cannot be tested to generate a valid executable, MAGE can be applied by slightly updating the constraints of the transformation functions to suit the 64-bit instruction set during implementation. Since the code transformations employed by MAGE are logical transformations, the effect of the functions would remain the same irrespective of the "modernness" of the virus. The only additional challenge would be identification of the constraints such as the code prologue/epilogue markers during implementation of the algorithm and there is no change in the algorithm itself. Thus MAGE works even with modern malware. It is also worth reiterating that MAGE is also a generic evolutionary engine that can be applied to multiple malware corpora.

Table 2
Parameters used by MAGE to identify apt similarity function.

Parameter	Value
Population size (NP)	20
Number of runs	5
P_m	0.8
P_c	0.2
Maximum generation (G)	1000
Input malware (ζ)	*Timid*
Representation	Linear
Fitness function	Jaccard and Cosine

4. Simulation design, results and analysis

The simulation studies, as it involves computer viruses, demand a secure test environment. We used an Intel© Core™ i5-2400 CPU @ 3.10 GHz with 4 CPU cores, 8 GB RAM and a 1 TB hard disk running a 64-bit Linux Mint system having a guest virtual machine running a 32 bit Windows 7 OS, as the test environment. This ensured that the virus variants generated during the experiments conducted, did not escape the test environment and spread. The guest virtual machine used 3 CPU cores and 4 GB 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. The guest Windows OS also had a copy of the Microsoft Macro Assembler (MASM) which is required to make the virus an executable capable of infecting the Windows OS. It is worth mentioning that the 32-bit operating system was chosen as it is possible for programs designed for a 32-bit OS to run on a 64-bit system, but not vice versa — i.e., it is backward compatible. This is because the assembly instruction set (X86 instruction set) that serves as the underlying assembly code language for the 32-bit operating system continues to be valid for the 64-bit operating system (using X64 instruction set). Even the addressing modes of the X64 architecture is similar to the X86 architecture and instructions, such as JMP, CALL, etc., that implicitly refer to the instruction pointer and the stack pointer treat them as 64 bits registers on x64 (Marshall & Martis, 2023). Therefore, this ensures that the proposed algorithm MAGE works as designed, irrespective of the underlying assembly code architecture.

Both the choice of code transformation functions and the Jaccard similarity index based intra-population fitness function have been based on preliminary experiments with MAGE. The initial experiments focused on identifying the apt transformation functions from the literature keeping in mind the constraints posed by the assembly code. This involved simulating and rejecting several transformation functions (like code substitution, identifier renaming etc.) as they were found unsuitable for the assembly code mutation. Since the work used the Microsoft Macro Assembler (MASM), it is also a requirement that the resultant transformations ensured that the file size remains under the 64 KB limit as code bloat is a common factor in EAs (Banzhaf et al., 1998). During the course of these experiments, it was also noted that MASM converts all conditional jump statements into short jumps. Consequently, the choice of code transformation functions also had to ensure that the original *Timid* virus code itself did not break post the application of such transformations. Also, since the focus is on evolving malware variants from the source, no other obfuscation techniques or packers were employed during the experiment.

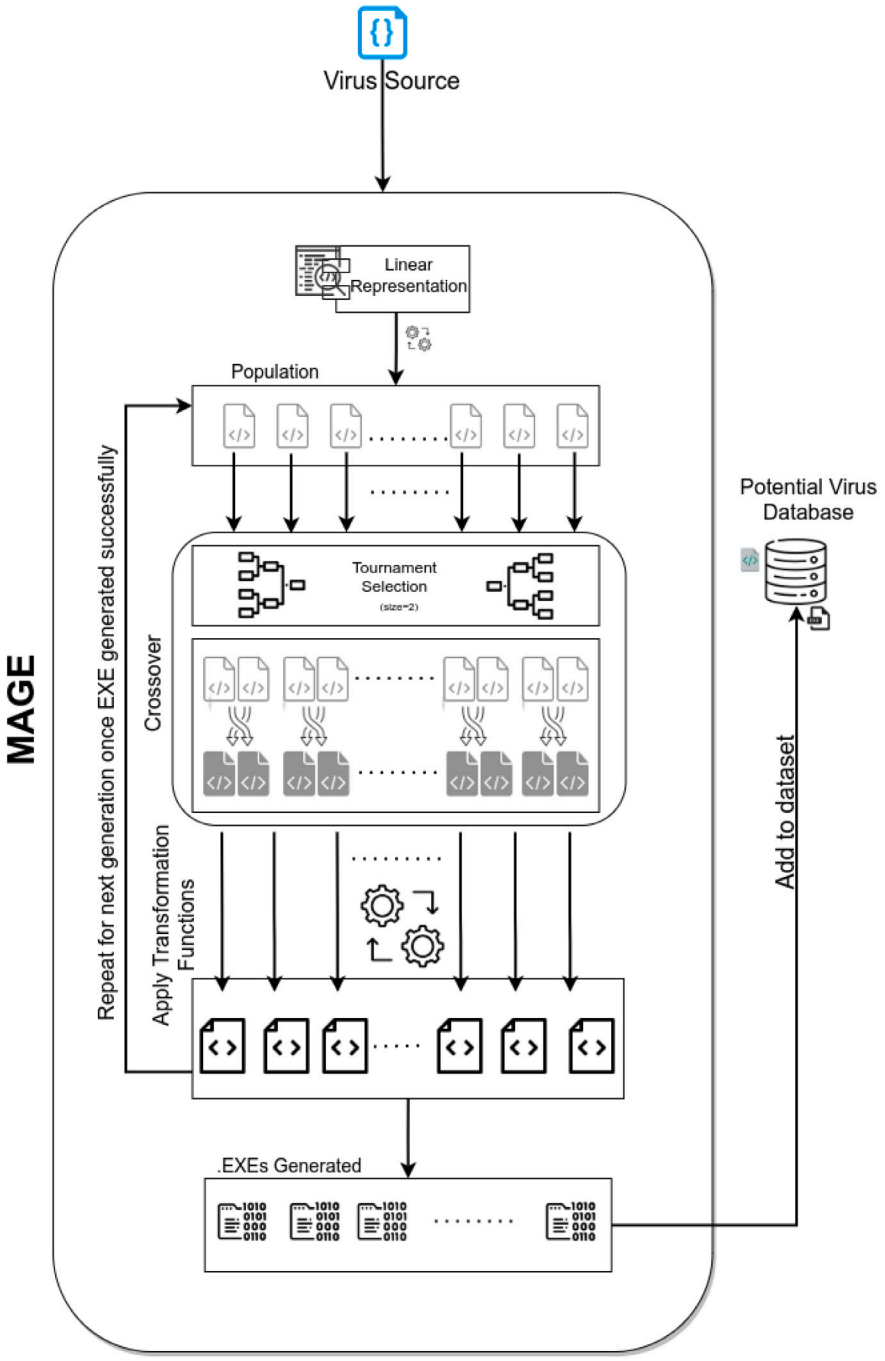


Fig. 4. Malware variant generation process using MAGE.

The choice of similarity as a metric of fitness was influenced by Castro et al. (2019) where the authors had calculated the differences between a given malware variant and the original malware with respect to the PE executables. Consequently, the proposed intra population similarity is explored as a fitness metric to evolve divergent virus variants a from vanilla virus source. This work also explores the use of a number of similarity metrics to find such a metric that would guide MAGE to evolve malware variants with diverse code structures. By way of an example, Fig. 6 shows a comparative performance between Cosine and Jaccard similarity metrics used in an experiment and the EA parameters for the same are depicted in Table 2. The mutations were found to drive the divergence and so were given a higher probability of occurrence over the crossover operation. The intention was to have a moderate population size while simulating MAGE for as high as 1000 generations. Fig. 6 plots the fitness values of 20,020 malware variants (20 variants in initial population and the 20 000 malware variant code structures generated during the course of 1000 generations). It is worth pointing out that 5 such simulation runs were carried out and Fig. 6 is the resultant fitness values distribution of one such run. In fact all the runs displayed similar characteristics. As Fig. 6 shows, Jaccard similarity based fitness function displays lesser variation i.e as MAGE evolves the malware variants tend to be more and more dissimilar amongst themselves compared to the Cosine similarity based fitness function with latter resulting in variants with both similar and dissimilar code structure characteristics. The latter, by virtue of measuring the angle between 2 vectors, identifies even minor variations in the code level thereby displaying higher variations (Xia et al., 2015). The intended objective was to generate more and more diverse variants in terms of

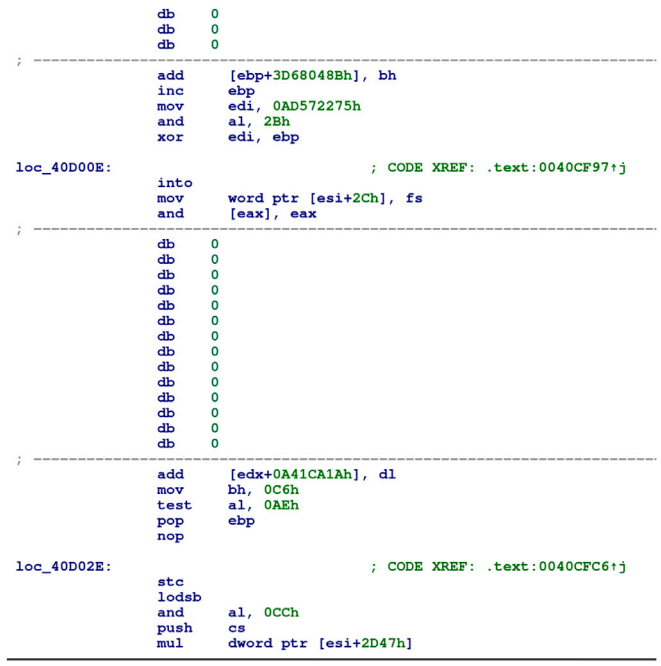


Fig. 5. Code of *Gen_Heur_PonyStealer_4* disassembled using IDA pro.

code structures within the population in each generation led to the choice of Jaccard as the effective similarity metric for MAGE. It is worth pointing out that all the 20 000 malware variants in fact are potential enough to be considered a malware variant data set for the chosen malware (i.e *Timid* virus in this case).

Experiments were conducted, with similar setup as above for 500 generations with population size as 20, to study the effectiveness of three different fitness function candidates. Fitness function α (Castro et al., 2019) explores the impact of the local minima in promoting diversity. This function computes Jaccard similarity index between each of candidate variants in the population against the source *Timid* malware alone (i.e. $J(C, c_i)$) and identifies the best candidate (where the best is the candidate with the least similarity with the source malware). The fitness function β is an intra-population Jaccard similarity based fitness function. This function calculates the distance between each candidate in the population and the mean vector of the Jaccard similarity of candidates within the population (as shown in Eqs. (2) and (3)). The candidate farthest from the mean vector are marked as the best fit candidate. The fitness function γ explores that possibility of evading AV scanners to also have a role to play in generating diverse variants. This fitness function utilized AV evasion ratio as the fitness function (no. of AV engines detected/no. of total AV engines), occasionally interspersed with the similarity based fitness function to observe the performance efficacy. We utilized VirusTotal (2021)[1] which allowed over 60 industry standard Anti-virus engines to evaluate the AV evasion ratio of all the chromosomes at the end of every 25 generations. The evasion ratio of each and every chromosome will be assigned as their respective fitness value (just for that one generation after every 25 generations) which will further be used in the tournament for parent selection. In fact such an interspersed manner of using fitness functions, has precedence in the literature (Shahrzad et al., 2020) and therefore the effect of γ (as a fitness function utilizing two different metrics in an interspersed manner) in encouraging diversity was also explored. The EA parameters for these experiments are shown in Table 3. The aim of the experiments were to identify the apt fitness metric that could be used by MAGE to evolve diverse malware variants.

[1] https://www.virustotal.com/.

Table 3
Parameters used by MAGE to explore different fitness functions.

Parameter	Value
Population size (NP)	20
Number of runs	5
P_m	0.8
P_c	0.2
Maximum generation (G)	500
Input malware (ζ)	*Timid*
Representation	Linear
Fitness function	α, β and γ

To verify that the variants generated by MAGE using the fitness metrics α, β and γ are indeed divergent, the Shannon entropy of the evolved variants are also calculated. In order to calculate the Shannon entropy, each virus executable was first read as a bytestream. This bytestream was then split into chunks of 256 bytes as suggested by Menéndez et al. (2019). Then the Shannon entropy for each chosen chunk(i.e. $H(U_i)$) is calculated by (Eq. (4)):

$$H(U_i) = -\sum_{b \in U_i} p(b) log_2 p(b) \tag{4}$$

with $p(b)$ being the probability of a byte$b$ within the jth chunk U_j of a variant c_j. In order to establish a baseline, the original timid.exe file entropy was calculated and found to be 0.073566 (rounded to 6 decimal places). The Mann–Whitney U test (a popular non-parametric hypothesis test) was used to compare the resultant population post evolution by MAGE using each fitness metric candidate. The Mann–Whitney U test is usually used to test the hypotheses H_0 which states that two populations (arising from the pairwise fitness metric experiments) are equal and the research hypothesis H_1 stating that the two populations are different. For any Mann–Whitney U test, the theoretical range of U is from 0 to $n_1 \times n_2$ where n_1 and n_2 are the corresponding sizes of the respective populations. The lower value of U implies that the hypothesis H_0 is false and supports the research hypothesis H_1. The p value of the Mann–Whitney U test calculates the probability of the sample results occurring by chance. According to MacFarland and Yates (2016), $p \leq 0.05$ is used by most explorative analyses. However, it is also not uncommon to see exploratory biological analyses use a more stringent level of $p \leq 0.01$. While a low value of p is sufficient to reject the hypothesis H_0, a large p value does not conclude that H_0 is true, it just implies that there is no compelling evidence to reject the hypothesis. Therefore for the simulation experiments, only when U is low and $p \leq 0.01$ it implies that the two distributions are different (i.e. the research hypothesis H_1 is correct).

Fig. 7 compares the results of MAGE execution using fitness metric α against the candidate fitness metric β and shows that the intra-population similarity based fitness function (β) evolved variants which are progressively becoming dissimilar to each other as reflected in decreasing similarity values as the generation proceeds. It should be noted that all comparative experiments used the same random seed to ensure a fair comparison for evolution. After 500 generations, with α as the fitness metric, the median entropy was 0.077785 ± 0.003035. This confirms that there is only a 5.74% increase in entropy which validated the inference from Fig. 7 that the variants being generated are only marginally different from the source *timid* virus. However, when the same analysis was conducted with β as the fitness metric, the median entropy was found to be 0.319802 ± 0.104216. This is a 334.71% increase in entropy and shows that increasingly divergent variants can be generated from a single vanilla source. Further, the Mann–Whitney U Test results on the similarity of variants generated using α and β show that the distributions are indeed different with $U = 1$ and $p = 1.62 \times 10^{-165}$. The low p value here indicates that the data did not occur by chance and therefore the two groups are indeed different. A comparison of the entropy levels of α, β and γ are depicted in Fig. 10 with the entropy results of β and γ analysed later in

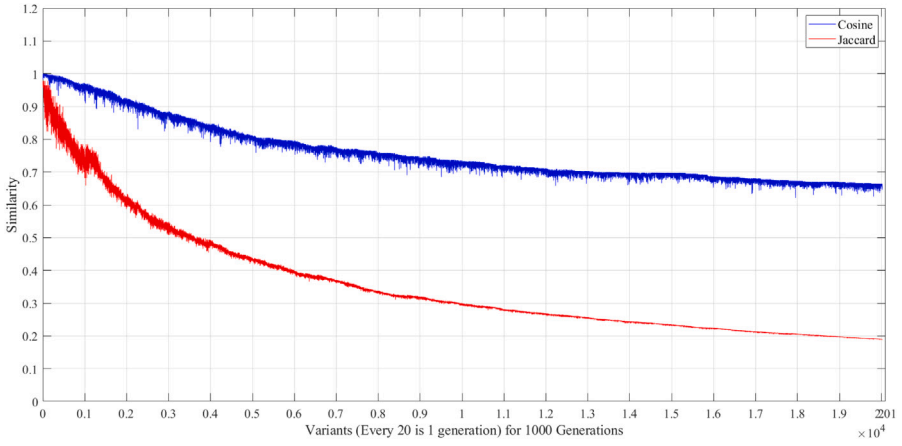


Fig. 6. Comparison of Jaccard and Cosine metrics.

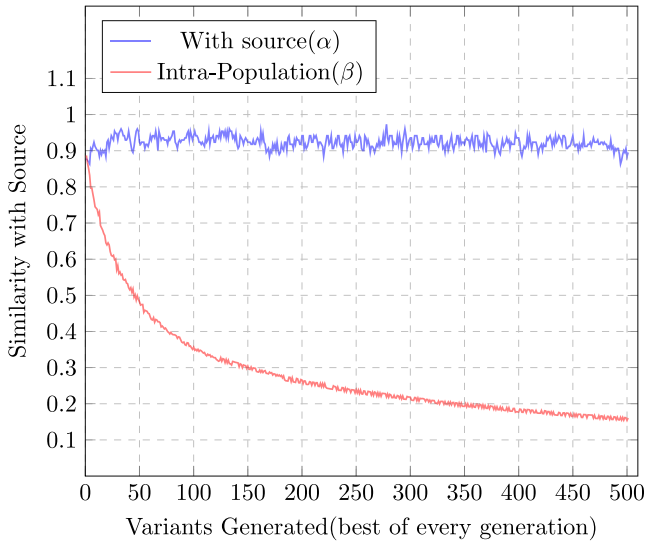


Fig. 7. Similarity of generated *Timid* virus variants using metrics α and β.

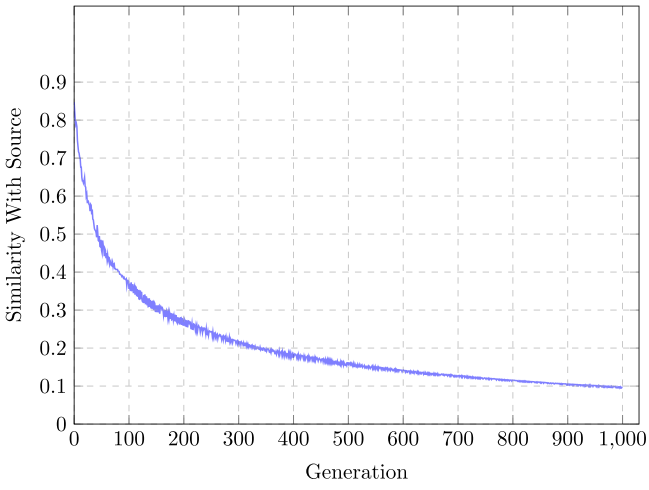


Fig. 8. Similarity values of best chromosomes against source malware obtained by MAGE with N = 100 and G = 1000.

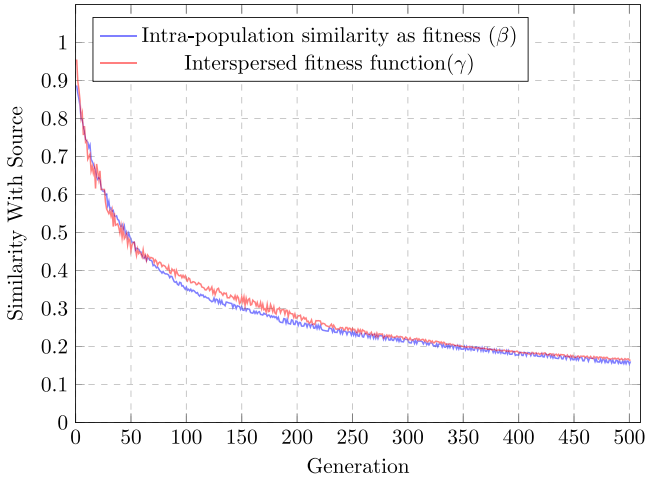


Fig. 9. Similarity values of best chromosomes calculated against source using intra-population similarity method (β) and interspersed fitness method (γ).

the paper. Fitness metric β subjects each variant to compete against a number of varied code structures thus promoting population diversity. In addition, the evolved variants increasingly become dissimilar to the source *Timid* virus thus becoming potential candidates for a malware variants data set. Fig. 8 plots the similarity between the most dissimilar (best) variant from each generation and the source *Timid* virus when MAGE was run with a population size NP of 100 for a maximum of 1000 generations (G). As the figure shows and reiterates, MAGE generates *Timid* variants which are increasingly becoming dissimilar to the source *Timid* virus. When considered in its entirety, the evolution of MAGE generates *Timid* variants of varied level of dissimilarity with source virus thus effectively representing the space of *Timid*'s variants.

Fig. 9 shows the performance of the fitness metric β and the interspersed fitness function (γ) in terms of the most dissimilar *Timid* variants evolved in each generation. Interestingly, as the figure shows, β is capable of evolving dissimilar variants in spite of not having a direct feedback about the evasion ratio of evolved structures albeit in every 25 generations. It is worth noting that a fitness function based only on AV evasion ratio will not guide the evolution towards dissimilar, unique and diverse code structures (unlike β and γ) towards realizing a potential data set of effective malware antigens. From Fig. 9 it can also be observed that the level of divergence of both β and γ are quite similar. To verify the same, the Shannon entropy of the variants evolved by each fitness function candidate was also calculated (Fig. 10).

It is observed that variants evolved by β show marginally higher levels of entropy. After 500 generations, with β as the fitness metric, the median entropy was observed to be 0.319802 $\pm$ 0.104216 while the median entropy with γ as the candidate fitness function was 0.298603

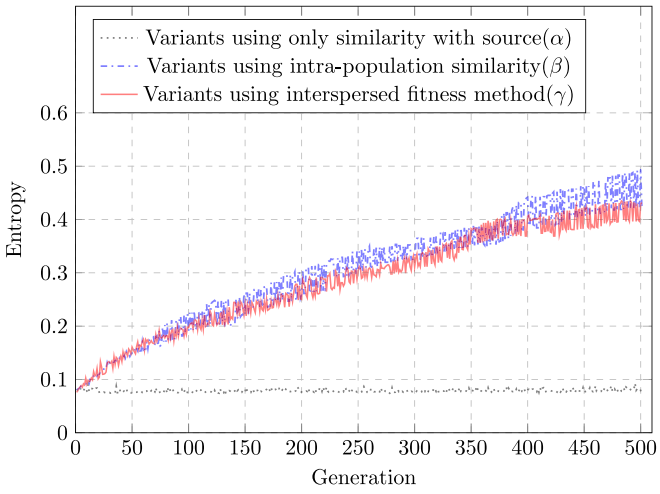


Fig. 10. Entropy values of best chromosomes calculated between variants generated using α, β and γ fitness metrics.

Table 4
Comparison of fitness measures.

Fitness measure	Mean similarity	Mean entropy	Mann Whitney U test
α[a]	0.923645	0.077785	Same
β	**0.271541**	**0.319802**	Diverse
γ	0.280414	0.298603	Diverse

[a]Adapted from Castro et al. (2019).

$\pm$ 0.094504. Interestingly, the Mann–Whitney U Test results resulted in $U = 1$ which is very low and indicative that the distributions are different. However, the p value obtained was 0.0886 which is greater than our set limit of 0.01. Considering this stringent value of p, H_0 cannot be conclusively rejected and so it can be concluded that the populations are *not* very different.

Based on the results (best value is marked as bold and summarized in Table 4), it is evident that MAGE is able to evolve the most divergent variants with β as the fitness function. From Fig. 10 it is quite clear that the fitness metric α (used by Castro et al., 2019) is unable to introduce significant changes in the vanilla executables of the generated virus variants while the proposed metrics β and γ are able to not just induce significant changes in the generated executables, but also ensure that the resultant variant population is diverse. While the variants evolved by both β and γ are quite similar, β does show a minor increase with respect to the entropy of the generated virus variants. Coupled with the fact that the time taken to evolve the variants is very low when compared to MAGE with γ as the fitness function, the intra-population based similarity metric (β) appears to be the better fit out of the three candidates to drive the evolution in MAGE. Therefore subsequent experiments primarily utilize β as the fitness function in MAGE.

MAGE, with this intra-population similarity based fitness function (β) combined with the choice of code transformation mutation functions, will not only evolve variants that are dissimilar from the source malware but also will ensure diversity amongst the evolved variants in terms of the code structure variations. By way of an example, Fig. 11, shows snippets of the same assembly code function in the original *Timid* (refer Fig. 11(a)), after the 1st generation (refer Fig. 11(b)) and after 100 generations (refer Fig. 11(c)) with the latter two sampled from random chromosomes. Fig. 11(c) shows the complex code control flow evolved by MAGE by virtue of the variation operators as well as the fitness function where the convolutions of the control flow can be clearly observed even in such a small code snippet. It is worth mentioning that despite such complex control flows, the code transformation functions do not break the original malware assembly code structure and ensure that all the evolved variants to be active malware.

The evolved diverse code structures possibly represent the variants' space of the employed source malware as MAGE evolved those variants using code transformation functions with emphasis on the unique code structure variations. Also, by virtue of the code structure variations, every chromosome that is evolved during the course of MAGE's run is a potential malware variant. It is then worth observing the anti-malware evasion ability of chromosomes that are evolved throughout the run of MAGE. Using VirusTotal, it was found that 29 anti-virus engines that were able to detect the source malware i.e. *Timid*. However in the simulation experiments reported, the evolved variants continued to be scanned with the full set of over 60 anti-virus engines (from VirusTotal) to verify whether any of the evolved variants are detected by other anti-virus engines not part of the above 29 anti-virus engines.

As these 29 anti-virus engines are capable of detecting the *Timid* virus, the anti-virus evasion ability of each *Timid* variant can be quantified by the number of anti-virus engines those variants are capable of evading from detection. It is definitely worth mentioning that if all the MAGE evolved variants are detected by even any one of the above listed anti-virus engines, it is a testimony to the fact that all the chromosomes evolved by MAGE are active malware. In fact it indeed has been verified that all variants of *Timid* evolved by MAGE has been detected as a virus by at least 1 or 2 AV engines. Fig. 12 shows the number of Anti-Virus engines that detected the most dissimilar *Timid* variant in each generation for both the intra-population similarity based fitness function and the interspersed fitness function cases (with population size as 20 for 100 generations). Though experiments were conducted for over 100 generations, the limit of 100 was selected as there were minimal changes in the detection ratio after 100 generations and so this served as a viable limit.

Interestingly, the intra-population similarity based fitness function, i.e. β (with no knowledge about the evasion ability of evolved structures) displayed a very competitive performance with that of the interspersed fitness function (γ) case. Additionally, in the case of MAGE evolved variants with the intra-population similarity based fitness function, there were a few scanners from the remaining AV engines (other than the 29 that initially detected the source virus *Timid*), that were able to identify the virus variants evolved by MAGE as malware. Table 5 shows a list of AV scanners that detected the original *Timid* virus. It can be noticed that while the virus is known to be *Timid*, AV scanners identify it under different names. For example, it can be noticed that Scanner 6 detects the original *Timid* virus as "Timid-305", Scanner 16 detects it as "Malware.Timid#4" and Scanner 15 detects the same virus as "A Variant of Acceptance". The table also showcases an interesting observation where one of the candidates generated during the initial generation (Table 1 - Evolved Timid) was detected by 29 AV scanners. Antivirus Scanner 6, which classified the original timid virus as "Timid-305", now classified the evolved candidate as "DOSMalware-gen [Trj]". Also, Scanner 11 which did not detect the original version of *Timid* detected the MAGE evolved variant as "$Malware@\#342r81r5umnd8$" while Scanners 23 & 24 consistently detected both the original and evolved *Timid* as "$Univ/j.dr$". This further emphasizes the fact that MAGE is capable of evolving potential malware variants without changing or affecting their functionality.

The simulation results above demonstrate that MAGE, by virtue of the code transformation mutation functions and intra-population similarity based fitness function (β), is capable of evolving diverse non-trivial assembly code structure variations for a given source *Timid* malware without changing or affecting the original malicious behaviour. These code structure variations have displayed good AV evasion performance as well. Consequently, the *Timid* variants evolved throughout the run of MAGE forms a potential data set which can be employed to train an adaptive AV engine to possibly augment its ability to detect the unseen *Timid* variants. In fact it can be argued that MAGE can use the assembly code of any source malware to evolve a number of potential

```
56 FIND_FILE:
57         mov     dx,[VIR_START]
58         mov     cx,3FH
59         mov     ah,4EH
60         int     21H
61
62
63 FF_LOOP:
```

(a) Original Function

```
67 FIND_FILE:
68         mov     dx,[VIR_START]
69         mov     cx,3FH
70         mov     ah,4EH
71 KFIUA:
72         call     OGEXG
73         int     21H
74
75
76 FF_LOOP:
```

(b) After 1 generation

```
514 FIND_FILE:
515         mov     dx,[VIR_START]
516         JMP     _PHPJ
517 NXVUI:
518         mov     cx,3FH
519 JZ TXCOB
520 JMP PVAVG
521 TXCOB:
522         mov     ah,4EH
523         JMP     _JTSW
524 TDCHC:
525         cmp     bx,cx
526 JNZ IOXQN
527 JMP ETAKS
528 IOXQN:
529         int     21H
530 FGZZY:
531         call     TPTID
532 PVAVG:
533         call     TXCOB
534 _PHPJ:
535         call     NXVUI
536 ETAKS:
537         call     IOXQN
538
539
540 FF_LOOP:
```

(c) same Function after 100 generations

Fig. 11. Effect of transformation functions in *Timid* assembly code - A snippet.

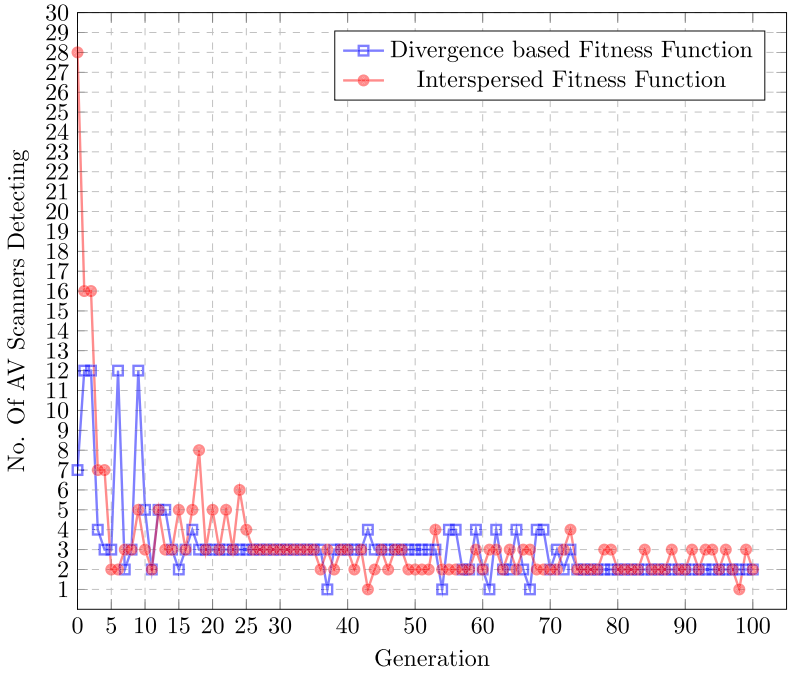


Fig. 12. Count of AV scanners detecting unique candidates of each generation.

Table 5
List of AV scanners that detected the original *Timid* virus and the name(s) under which the virus has been detected. The AV Scanner names have been changed to ensure anonymity.

AV name	Original timid	Evolved timid
*Scanner*1	*Gen* : *Trojan.FileInfector.aaW@aaaaa*	*GenTrojan.FileInfector.aaW@aaaaa*
*Scanner*2	*Virus.DOS.Timid.n!c*	*Virus.DOS.Timid.n!c*
*Scanner*3	*Virus* : *DOS/Timid.fddb*9066	*VirusDOS/Timid*.1708550*a*
*Scanner*4	*Gen* : *Trojan.FileInfector.aaW@aaaaa*	*GenTrojan.FileInfector.aaW@aaaaa*
*Scanner*5	*Trojan.FileInfector.ED*11*F*2	*Trojan.FileInfector.ED*11*F*2
Scanner6	**Timid-305**	**DOS Malware-gen [Trj]**
*Scanner*7	*Timid* − 305	*DOSMalware* − *gen*[*Trj*]
*Scanner*8	*Timid*#4	*DOS/Timid.yyrqa*
*Scanner*9	*Gen* : *Trojan.FileInfector.aaW@aaaaa*	*GenTrojan.FileInfector.aaW@aaaaa*
*Scanner*10	*Win.Trojan.Timid* − 4	*Win.Trojan.V*306 − 1
Scanner11	**FALSE(No detection)**	**Malware@#342r81r5umnd8**
*Scanner*12	*Malicious*(*score* : 85)	*malware*(*aiscore* = 85)
*Scanner*13	*Gen* : *Trojan.FileInfector.aaW@aaaaa*(*B*)	*GenTrojan.FileInfector.aaW@aaaaa*(*B*)
*Scanner*14	*Gen* : *Trojan.FileInfector.aaW@aaaaa*	*GenTrojan.FileInfector.aaW@aaaaa*
*Scanner*15	*AVariantOfAceptance*.311	*FALSE*(*Nodetection*)
Scanner16	**Malware.Timid #4**	**Malware.DOS/Timid.yyrqa**
*Scanner*17	*Gen* : *Trojan.FileInfector.aaW@aaaaa*	*GenTrojan.FileInfector.aaW@aaaaa*
*Scanner*18	*FileInfector.AAW*	*Timid.A!tr*
*Scanner*19	*Gen* : *Trojan.FileInfector.aaW@aaaaa*	*GenTrojan.FileInfector.aaW@aaaaa*
*Scanner*20	*Gen.DoS.FileInfector*	*DOS.Timid*
*Scanner*21	*Virus.DOS.Timid*.305.*a*	*Virus.DOS.Timid*.305.*a*
*Scanner*22	*malware*(*aiscore* = 83)	*malware*(*aiscore* = 83)
*Scanner*23	*Univ/j.dr*	*Univ/j.dr*
*Scanner*24	*Univ/j.dr*	*Univ/j.dr*
*Scanner*25	*Trojan* : *Win*32*/Occamy.CF*8	*TrojanScript/Wacatac.C!ml*
*Scanner*26	*Generic/Virus.DoS*.877	*Generic/Virus.DoS*.877
*Scanner*27	*Mal/Generic* − *S*	*Mal/Generic* − *S*
Scanner28	**Timid.305**	**FALSE(No detection)**
*Scanner*29	*Win*32*.Virus.Aceptance.Edyq*	*Dos.Virus.Timid.Htms*
*Scanner*30	*Virus.DOS.Timid*.305.*a*	*Virus.DOS.Timid*.305.*a*

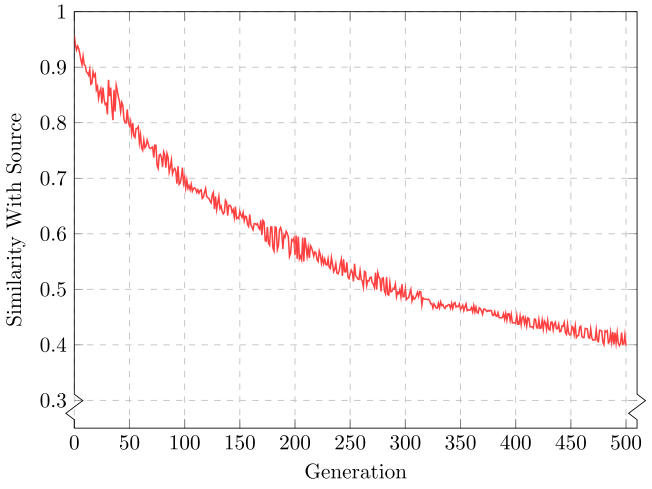


Fig. 13. Similarity values of best chromosomes against *Intruder* source obtained by MAGE with N = 20 and G = 500.

malicious variants. The validity of this argument is demonstrated by employing MAGE to evolve variants of another malware by way of an example.

Intruder (Ludwig, 1991), an EXE infector virus, was used as the source malware for MAGE to demonstrate its versatility. An EXE virus has the possibility to infect a greater number of files. The *Intruder* virus is a more sophisticated virus (when compared to *Timid*) that can infect EXE files and jump across directories and even across drives, thus making it much more complex and even more dangerous. The virus attaches itself to the end of an EXE program and gain control when the program first starts. A tree search routine is used to list the possible targets for infection. Conversely, if a suitable target is not found, the virus proceeds to search for sub-directories of the currently referenced directory for a suitable target. Fig. 13 shows the similarity

Table 6
Parameters used by MAGE for validation on *Intruder*.

Parameter	Value
Population size (NP)	20
Number of runs	5
P_m	0.8
P_c	0.2
Maximum generation (G)	500
Input malware (ζ)	*Intruder*
Representation	Linear
Fitness Function	β

of the most dissimilar variants in the population at each generation against the source *Intruder* virus. The best variants in each generation become increasingly more dissimilar from the source as was the case with the *Timid* experiments. Fig. 14 shows random *Intruder* assembly code structure variations evolved by MAGE both in the intermediate generation and in the final generation. The intricate and not so trivial code structure variations have been evolved by virtue of the chosen transformation functions. It is worth mentioning that the MAGE has been employed as is, with the transformation functions and intra-population similarity based fitness function intact (as well as other parameters used for *Timid* experiments) for evolving *Intruder* variants thereby demonstrating MAGEs' ability to generalize to multiple malware corpora. Table 6 presents the simulation parameters used by MAGE for validation on Intruder.

Fig. 15 shows the AV evasion ability of the best *Intruder* variants evolved by MAGE in every generation (for population size 20 and for 100 generations) in terms of the number of AV engines detecting them). It is worth mentioning that 20 popular AV engines detected the source *Intruder*. It is interesting to observe that the code structure variations effected by MAGE cause the *Intruder* variants to be detected by as high as 4 and as low as zero AV engines against the 20 engines that detected the source malware. As has been done in the case of *Timid*, all the best *Intruder* variants have been successfully tested for their maliciousness by allowing them to infect the virtual OS. It is worth noting that quite a

```
177 FINDEXE:
178         mov dx,OFFSET DTA1
179         mov ah,1AH
180         int 21H
181         mov di,OFFSET USEFILE
182         mov si,OFFSET EXEFILE
183         call CONCAT
184         push di
185         mov dx,OFFSET USEFILE
186         mov cx,3FH
187         mov ah,4EH
188         int 21H
```

(a) Original Function

```
296 FINDEXE:
297         mov dx,OFFSET DTA1
298         mov ah,1AH
299         JMP     CVRQV
300 JPRML:
301         int 21H
302         mov di,OFFSET USEFILE
303         mov si,OFFSET EXEFILE
304         call CONCAT
305         push di
306         mov dx,OFFSET USEFILE
307         mov cx,3FH
308         mov ah,4EH
309         cmp     cx,cx
310 JNZ LLJS_
311 JMP GGPNT
312 LLJS_:
313 CVRQV:
```

(b) Intermediate generation

```
435 RGMDF:
436         mov dx,OFFSET DTA1
437         mov ah,1AH
438         cmp     bx,dx
439 JNZ VFQGE
440 JMP RUFPL
441 VFQGE:
442         int 21H
443         mov di,OFFSET USEFILE
444         JMP     ETDIK
445 nop
446 nop
447 nop
448 nop
449 ETDIK:
450         mov si,OFFSET EXEFILE
451         call CONCAT
```

(c) same Function after 100 generations

Fig. 14. Effect of transformation functions on *Intruder* assembly code - A snippet.

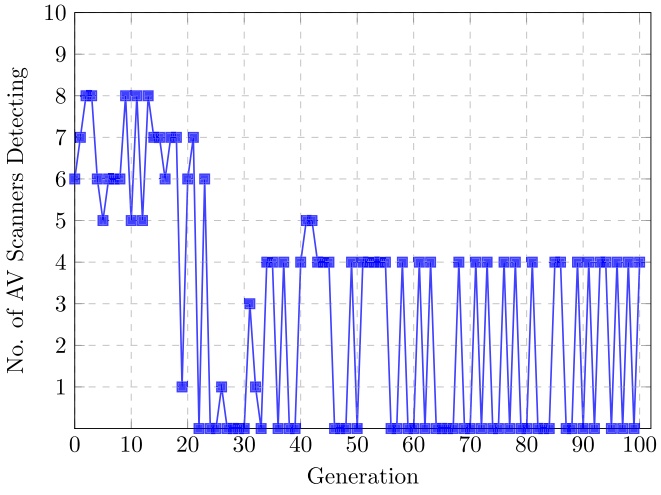


Fig. 15. *Intruder* virus - AV scanners evaded.

number of *Intruder* variants evolved by MAGE have successfully evaded all the AV engines from the AV engine pool.

It should be noted that with both *Timid* and *Intruder* viruses, MAGE was successfully able to generate malicious executables consistently for 600 generations i.e. $600 \times 20 = 12{,}000$ divergent virus variant executables from a single vanilla source. As mentioned previously, bloat is indeed a factor and beyond 600 generations there is the possibility of not being able to generate a valid executable. A failure is when if even a single candidate variant is unable to produce a successful executable. For example, during one of the runs, in generation 619, the population was able to generate only 19 successful virus variants and this is considered as a failure as 20 successful variants are expected (since population size was 20). It is worth pointing out that the failure in generating active malware variants is by virtue of the MASM limitations and not because of the variation operators (transformation functions in particular). Multiple such runs of MAGE revealed that with β as the fitness metric, MAGE was able to evolve variants successfully between 600–1700 generations. During our trials, the lowest number of generations where an executable failed to be generated was in generation 619 while one of the runs also evolved successful variants as executables for 1721 generations which resulted in $1721 \times 20 = 34\,420$ successful virus variants being generated. A similar experiment was attempted with γ as the fitness metric. However since γ as the fitness metric involved scanning the viruses at regular intervals (coupled with VirusTotals' scanning restrictions), even a single run of 500 generations took up to 72 h to complete. In comparison, with β as the fitness metric, 500 generations was successfully completed in under 30 min and so was used as the metric for testing the limits of the evolution with respect to number of generations.

At this juncture, it is worth mentioning that it is possible to use MAGE for malware generation in both 32-bit as well as 64-bit architectures. However, in case of non-availability of assembly source code, a malicious executable can be disassembled using popular disassembly tools and MAGE can be applied on the disassembled code. In this case, while MAGE still generates variants, it may not be possible to create executables using the same as disassembled codes may not be compilable. It is worth noting that MAGE's capability remains the same irrespective of the underlying assembly architecture. The only change would be in the test bed setup to include the 64-bit instruction

set of the reverse engineering tool used to generate the assembly language code. In addition, despite the fact that MAGE has been demonstrated to evolve diverse variants, for only two malware, viz Timid and Intruder, the proposed assembly code transformation functions are generic enough to transform any assembly source code. In fact, the purpose of the viruses was to assist in designing the transformation functions to be employed as variation operators as well as demonstrate the potential of MAGE to introduce diversity in assembly level codes. Consequently, experimenting MAGE with additional malware will be merely a reiteration of MAGE's potential that has been demonstrated so far. It is also worth mentioning that the idea of achieving arbitrary assembly code level diversity in an automated fashion by MAGE is applicable to any malware, irrespective of their period of appearance, and is relevant towards generating novel malware variants.

5. Conclusion and future directions

This paper proposed a malware antigens generating evolutionary algorithm (MAGE) to evolve potential malware variants of a given malware through assembly code structure transformations. Through a linear representation of the chromosome alongside carefully designed code transformations as variation operations, the simulation experiments revealed that MAGE was able to evolve valid executable variants of an input malware. A selective subset of code transformation functions as variation operations and an intra-population Jaccard similarity based fitness function augment MAGE to generate diverse malware variants in terms of non-trivial dissimilar code structure variations. Experimental evaluation using three different fitness metrics, namely α (adapted from literature), β (intra-population Jaccard similarity based) and γ (function incorporating AV results alongside similarity) revealed that the functions β and γ were able to evolve divergent variants of the input malware. Further statistical analysis using Mann Whitney U Test validated that the generated variants were indeed divergent. Virus Total (which was used to scan the evolved variants) scans revealed that MAGE was not only capable of evolving divergent variants, but the evolved variants were also misclassified by the AV scanners with a few of the scanners wrongly identifying the virus family itself. Thus MAGE was able to demonstrate the idea of achieving arbitrary code level diversity in an automated fashion which is relevant towards generating novel malware variants.

By proactively generating variants through code transformation that result in valid executables, the generated variants act as instances for a typical learning algorithm to create signatures. These signatures could be then updated in the antivirus database to trigger the immune response if any of these viruses are discovered in a client system. Thus, these malware variants (similar to biological antigens) have the potential to help improve adaptive AV engines to achieve active acquired immunity against unseen variants of a given malware.

Currently, algorithms do not evolve valid/diverse variants with high certainty. MAGE is able to achieve this with carefully designed but generic code transformation functions. The simulation experiments involving *Timid* and *Intruder* viruses demonstrate the versatility and efficacy of MAGE in evolving malware variants as executables with diverse code structures while retaining maliciousness and successfully evading detection by over 97% of over 60 AV scanners. As every candidate evolved throughout the run of MAGE is a potential variant, the complete set of malware variants evolved by MAGE can serve as a valid data set for a trainable AV engine to sample and learn the malware variants' space. This could augment the AV engines ability to detect unseen malware variants - a potential trait to detect zero day malware (variants') attack.

Our future work is a natural extension of the current work and would involve exploring the effect of MAGE across a variety of malware as well as investigating the potential of evolutionary packers alongside evolved antigens to train AV engines to proactively defend against unseen malware variants. Another interesting direction of our research is the potential co-evolution of malware generation and detection, where proactive strategies such as generative adversarial networks, meta-heuristic algorithms automatically generate both novel malware and malware variants.

CRediT authorship contribution statement

Ritwik Murali: Conceptualization, Methodology, Investigation, Software, Validation, Visualization, Writing – original draft. **Palanisamy Thangavel:** Formal analysis, Visualization. **C. Shunmuga Velayutham:** Methodology, Literature survey, Formal analysis, Writing – review & editing, Visualization, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgements

The authors would like to thank and acknowledge the inputs from the various anonymous reviewers for their thoughtful comments and efforts towards improving this work.

References

Afaneh, S., Zitar, R. A., & Al-Hamami, A. (2013). Virus detection using clonal selection algorithm with genetic algorithm (VDC algorithm). *Applied Soft Computing*, *13*(1), 239–246.

Al-Sahaf, H., & Welch, I. (2019). A genetic programming approach to feature selection and construction for ransomware, phishing and spam detection. In *Proceedings of the genetic and evolutionary computation conference companion* (pp. 332–333).

Anderson, H. S., Kharkar, A., Filar, B., Evans, D., & Roth, P. (2018). Learning to evade static PE machine learning malware models via reinforcement learning. arXiv preprint arXiv:1801.08917.

Apruzzese, G., Colajanni, M., Ferretti, L., Guido, A., & Marchetti, M. (2018). On the effectiveness of machine and deep learning for cyber security. In *2018 10th international conference on cyber conflict (CyCon)* (pp. 371–390). IEEE.

Aydogan, E., & Sen, S. (2015). Automatic generation of mobile malwares using genetic programming. In *European conference on the applications of evolutionary computation* (pp. 745–756). Springer.

Babaagba, K. O., Tan, Z., & Hart, E. (2020a). Automatic generation of adversarial metamorphic malware using map-elites. In *International conference on the applications of evolutionary computation (part of evostar)* (pp. 117–132). Springer.

Babaagba, K. O., Tan, Z., & Hart, E. (2020b). Improving classification of metamorphic malware by augmenting training data with a diverse set of evolved mutant samples. In *2020 IEEE congress on evolutionary computation* (pp. 1–7). IEEE.

Banzhaf, W., Nordin, P., Keller, R. E., & Francone, F. D. (1998). *Genetic Programming: an Introduction, Vol. 1*. Morgan Kaufmann Publishers San Francisco.

Bose, S., Barao, T., & Liu, X. (2020). Explaining AI for malware detection: Analysis of mechanisms of MalConv. In *2020 international joint conference on neural networks* (pp. 1–8). IEEE.

Cani, A., Gaudesi, M., Sanchez, E., Squillero, G., & Tonda, A. (2014). Towards automated malware creation: code generation and code integration. In *Proceedings of the 29th annual ACM symposium on applied computing* (pp. 157–160). ACM.

Castro, R. L., Schmitt, C., & Dreo, G. (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)* (pp. 240–247). IEEE.

Chen, Y., Jin, B., Yu, D., & Chen, J. (2018). Malware variants detection using behavior destructive features. In *2018 IEEE symposium on privacy-aware computing* (pp. 121–122). IEEE.

Cohen, F. (1987). Computer viruses: theory and experiments. *Computers & Security*, *6*(1), 22–35.

Dasgupta, D. (2012). *Artificial immune systems and their applications*. Springer Science & Business Media.

Divya, T., & Muniasamy, K. (2015). Real-time intrusion prediction using hidden Markov model with genetic algorithm. In *Artificial intelligence and evolutionary algorithms in engineering systems* (pp. 731–736). Springer.

Dwan, B. (2000). The computer virus—From there to here.: An historical perspective. *Computer Fraud & Security*, *2000*(12), 13–16.

Haq, I. U., & Caballero, J. (2021). A survey of binary code similarity. *ACM Computing Surveys*, *54*(3), 1–38.

Harahsheh, H., Alshraideh, M., Al-Sharaeh, S., & Al-Sayyed, R. (2022). Improving classification performance for malware detection using genetic programming feature selection techniques. *Journal of Applied Security Research*, 1–21.

hex-rays (2022). IDA help. URL: https://hex-rays.com/products/ida/support/idadoc/619.shtml. Last accessed July 2022.

Hosseinzadeh, S., Rauti, S., Laurén, S., Mäkelä, J.-M., Holvitie, J., Hyrynsalmi, S., & Leppänen, V. (2018). Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, *104*, 72–93.

Jerbi, M., Chelly Dagdia, Z., Bechikh, S., & Ben Said, L. (2021). Malware detection using rough set based evolutionary optimization. In *International conference on neural information processing* (pp. 634–641). Springer.

Kayacık, H. G., Zincir-Heywood, A. N., & Heywood, M. I. (2011). Can a good offense be a good defense? Vulnerability testing of anomaly detectors through an artificial arms race. *Applied Soft Computing*, *11*(7), 4366–4383.

Khalilian, A., Nourazar, A., Vahidi-Asl, M., & Haghighi, H. (2018). G3MD: Mining frequent opcode sub-graphs for metamorphic malware detection of existing families. *Expert Systems with Applications*, *112*, 15–33.

Lamont, G. B., Marmelstein, R. E., & Van Veldhuizen, D. A. (1999). A distributed architecture for a self-adaptive computer virus immune system. In *New ideas in optimization* (pp. 167–184). US: McGraw-Hill Inc..

Lin, D., & Stamp, M. (2011). Hunting for undetectable metamorphic viruses. *Journal in Computer Virology*, *7*(3), 201–214.

Liu, X., Lin, Y., Li, H., & Zhang, J. (2020). A novel method for malware detection on ML-based visualization technique. *Computers & Security*, *89*, Article 101682.

Ludwig, M. A. (1991). *The little black book of computer viruses*. Amer Eagle Pubns Inc.

MacFarland, T. W., & Yates, J. M. (2016). *Introduction to nonparametric statistics for the biological sciences using R*. Springer.

Madenur Sridhara, S., & Stamp, M. (2013). Metamorphic worm that carries its own morphing engine. *Journal of Computer Virology and Hacking Techniques*, *9*(2), 49–58.

Malanov, A. V., & Kamlyuk, V. A. (2012). Rapid heuristic method and system for recognition of similarity between malware variants. US Patent 8, 250, 655.

Marshall, D., & Martis, J. (2023). X64 architecture - windows drivers. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture.

Mawgoud, A. A., Rady, H. M., & Tawfik, B. S. (2021). A malware obfuscation AI technique to evade antivirus detection in counter forensic domain. In *Enabling AI applications in data science* (pp. 597–615). Springer.

Menéndez, H. D., Bhattacharya, S., Clark, D., & Barr, E. T. (2019). The arms race: Adversarial search defeats entropy used to detect malware. *Expert Systems with Applications*, *118*, 246–260.

Menéndez, H. D., Clark, D., & T. Barr, E. (2021). Getting ahead of the arms race: Hothousing the coevolution of VirusTotal with a packer. *Entropy*, (4), 395.

Meng, G., Xue, Y., Mahinthan, C., Narayanan, A., Liu, Y., Zhang, J., & Chen, T. (2016). Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on asia conference on computer and communications security* (pp. 365–376). ACM.

Murali, R., Ravi, A., & Agarwal, H. (2020). A malware variant resistant to traditional analysis techniques. In *2020 international conference on emerging trends in information technology and engineering (Ic-ETITE)* (pp. 1–7). IEEE.

Murali, R., & Shunmuga Velayutham, C. (2020). A preliminary investigation into automatically evolving computer viruses using evolutionary algorithms. *Journal of Intelligent & Fuzzy Systems*, *8*(5), 6517–6526.

Noreen, S., Murtaza, S., Shafiq, M. Z., & Farooq, M. (2009a). Evolvable malware. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation* (pp. 1569–1576). ACM.

Noreen, S., Murtaza, S., Shafiq, M. Z., & Farooq, M. (2009b). Using formal grammar and genetic operators to evolve malware. In *RAID* (pp. 374–375).

Paul, T. G., & Kumar, T. G. (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* (pp. 551–559). Springer.

Santacroce, M. L., Koranek, D., & Jha, R. (2020). Detecting malware code as video with compressed, time-distributed neural networks. *IEEE Access*, *8*, 132748–132760.

Shahrzad, H., Hodjat, B., Dollé, C., Denissov, A., Lau, S., Goodhew, D., Dyer, J., & Miikkulainen, R. (2020). Enhanced optimization with composite objectives and novelty pulsation. In *Genetic programming theory and practice XVII* (pp. 275–293). Springer.

Squillero, G. (2005). MicroGP—an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, *6*(3), 247–263.

Stiborek, J., Pevnỳ, T. A., & Rehák, M. (2018). Multiple instance learning for malware classification. *Expert Systems with Applications*, *93*, 346–357.

Tamboli, T., Austin, T. H., & Stamp, M. (2014). Metamorphic code generation from LLVM bytecode. *Journal of Computer Virology and Hacking Techniques*, *10*(3), 177–187.

Venkatachalam, S., & Stamp, M. (2011). Detecting undetectable metamorphic viruses. In *Proceedings of the international conference on security and management* (p. 1). Citeseer.

VirusTotal (2021). Getting started with VirusTotal. URL: https://developers.virustotal.com/reference. Last accessed August 2021.

Wadkar, M., Di Troia, F., & Stamp, M. (2020). Detecting malware evolution using support vector machines. *Expert Systems with Applications*, *143*, Article 113022.

Wang, S., Wang, P., & Wu, D. (2015). Reassembleable disassembling. In *24th USENIX security symposium (USENIX Security 15)* (pp. 627–642). Washington, D.C.: USENIX Association, URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-shuai.

Wu, S. X., & Banzhaf, W. (2010). The use of computational intelligence in intrusion detection systems: A review. *Applied Soft Computing*, *10*(1), 1–35.

Xia, P., Zhang, L., & Li, F. (2015). Learning similarity with cosine similarity ensemble. *Information Sciences*, *307*, 39–52.

Xue, Y., Meng, G., Liu, Y., Tan, T. H., Chen, H., Sun, J., & Zhang, J. (2017). Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security*, *12*(7), 1529–1544.

Yazdinejad, A., HaddadPajouh, H., Dehghantanha, A., Parizi, R. M., Srivastava, G., & Chen, M.-Y. (2020). Cryptocurrency malware hunting: A deep recurrent neural network approach. *Applied Soft Computing*, *96*, Article 106630.

Zheng, M., Lee, P. P., & Lui, J. C. (2012). ADAM: an automatic and extensible platform to stress test android anti-virus systems. In *International conference on detection of intrusions and malware, and vulnerability assessment* (pp. 82–101). Springer.

Zhong, W., & Gu, F. (2019). A multi-level deep learning system for malware detection. *Expert Systems with Applications*, *133*, 151–162.