AMVG: Adaptive Malware Variant Generation Framework Using Machine Learning

Jusop Choi[†], Dongsoon Shin[†], Hyoungshick Kim[†‡], Jason Seotis[*], Jin B. Hong[*]
[†]Sungkyunkwan University, South Korea
[*]University of Western Australia, Australia
[‡]Data61 CSIRO, Australia
[†]{cjs1992, ds.shin, hyoung}@skku.edu
[*]jseotis@gmail.com, jin.hong@uwa.edu.au

***Abstract*—There are advances in detecting malware using machine learning (ML), but it is still a challenging task to detect advanced malware variants (e.g., polymorphic and metamorphic variations). To detect such variants, we first need to understand the methods used to generate them to bypass the detection methods. In this paper, we introduce an adaptive malware variant generation (AMVG) framework to study bypassing malware detection methods efficiently. The AMVG framework uses ML (e.g., genetic algorithm (GA)) to generate malware variants that satisfy specific detection criteria. The use of GA automates the malware variant generations with appropriate modules to handle various input formats. For the experiment, we use malware samples retrieved from theZoo, a collection of malware samples[1]. The results show that we can automatically generate malware variants that satisfy varying detection criteria in a practical amount of time, as well as showing the capabilities to handle different input formats.**

***Index Terms*—Genetic Algorithm, Malware Detection, Malware Generation, Malware Variation, Source Code Similarity.**

I. Introduction

Malware generation techniques are continuously evolving in order to bypass new and emerging malware detection methods (e.g., polymorphic viruses). Consequently, existing malware detection methods are becoming less effective to detect new malware variants. To address this problem, the use of machine learning (ML) techniques are becoming popular, showing improved malware detection rate [1]–[3]. However, even though the overall detection rate is increasing, there is always a chance for attackers to generate malware variants to bypass detection methods as ML techniques inherently have errors, both in false positives and false negatives. Therefore, it is of paramount importance to understand what malware variants can bypass malware detectors and how such variants can be generated. Consequently, this understanding would be useful to develop more effective malware detection methods.

Many techniques are developed to bypass malware detectors [4], [5]. For example, ARMED [6] is a framework that can generate adversarial examples of malware automatically, which applies random perturbations to original malware samples until the variant version is wrongly classified by malware detectors. However, since the perturbations are randomly applied, the performance cannot scale to a large number of malware samples, as well as ensuring the operability of the variants (i.e., ensuring the original functionality of the malware after the perturbation has been applied). Hence, it is important that malware variants are carefully produced to bypass the existing detectors. We should also take into consideration that different malware detectors are implemented differently to detect malware. There are three main categories of malware detection techniques, which are: (1) signature-based, (2) behavior-based, and (3) heuristic-based [7]. The signature-based technique has been popularly used in practice due to its accuracy and high speed. Although they have limited capabilities to detect variants, the use of similarity measures have been adopted to provide the detection of malware variants to a certain degree [8], [9]. However, in academia, many papers using behavior-based and/or heuristic-based methods were also proposed. The behavior-based techniques have benefited the most from adopting the ML techniques [2], [3], but with advances in malware (e.g., unpredictable activity time periods), the speed of the detection is significantly slower than other detection techniques. The heuristic-based techniques typically use operational codes or file bytes to detect malware. Heuristic-based technique can also be significantly improved using ML techniques [5]. Hence, it is important that the malware variant generation framework has the flexibility to alter a malware detection method adopted for the malware variant generation process.

In this paper, we propose an adaptive malware variant generation (AMVG) framework taking into account different detection criteria used by multiple malware detectors. To efficiently generate malware variants, a genetic algorithm (GA) is incorporated into the framework to guide the variant generation processes. Other modules, such as *parser* and *scorer*, are used to guide GA to generate malware variants taking into account different input formats and malware detectors. To demonstrate, we utilize two malware scoring systems based on similarities, namely TLSH [8] and ssdeep [9] to show how the framework adapts to different malware detection systems. Using similarity measures for detecting malware variants are a growing field of study to detect them more efficiently and effectively [10]. Finally, we conduct an experimental analysis to evaluate the proposed framework. The contributions of this

[1]available at: https://github.com/ytisf/theZoo

paper are as follows.

- We propose an adaptive malware variant generation framework taking into account different input formats and malware detection criteria;
- We conduct two case studies using C and Python to evaluate the performance of malware variant generations with a real world malware corpus;
- We evaluate the effects of changing GA parameters, and conduct parameter optimization for the malware variant generation performance.

The rest of the paper is structured as follows. Section II presents related work on malware variant generations. Section III presents our framework for generating malware variants. Sections IV and V presents the specific Python and C-based implementations for the AMVG modules, respectively. Section VI presents the experimental analysis, evaluating the performance and effectiveness of the AMVG framework. Section VII discusses our findings and limitations, and finally, Section VIII concludes our paper.

II. Related Work

A. Malware detection techniques

The techniques for detecting malware can be categorized into three: signature-based, behavior-based and heuristic-based detection systems [7], [11]. Further, data mining techniques have been extensively studied due to the limitations of those malware detection systems [12], [13]. With advances in malware detection techniques using ML [14], higher detection rates are achieved with reduced false-positive and false-negative rates. For example, Rieck *et al*. [1] showed malware behavior analysis (e.g., usage of system calls) and applied ML algorithms to measure similarities between unknown applications detect malware variants. However, malware variants are also generated using adversarial learning [15]. As all malware detection techniques leverage a decision making module (i.e., signature-based techniques checking the signature similarities, and behavior-based techniques checking the behavioral score of the program under inspection), malware writers can develop methods to create variants that could bypass those decision making modules. For example, Shiel and O'Shaughnessy [16] presented fuzzy hashes for classifying malware, where the malware samples are scored based on their fuzzy hash values with the distance measured to various malware families. However, there are still gaps in existing malware detection systems that can be exploited by malware writers.

B. Bypassing malware detection systems

Various malware mutation techniques have been developed to modify the presentation of malware samples while preserving their program behaviors. A popular method is code obfuscation [17]–[19]. They also target various platforms (e.g., Android [20], [21]), where AI and ML techniques are used to enhance the efficiency of generating malware variants. Anderson *et al*. [4] proposed an attack using OpenAI Gym to bypass static malware detection techniques. The OpenAI Gym used reinforcement learning to guide its perturbations based on the detection outcomes. However, they did not consider checking the functionality of malware variants after injecting perturbations. Castro *et al*. [6] proposed ARMED, an automated malware variant generation framework by randomly injecting perturbations to malware samples. The framework checks whether the constructed malware variants are detected or not. However, they are known to have a performance issue as perturbations are randomly injected. Consequently, finding the narrow gap to bypass the detectors can be a significant challenge, especially by a random search for malware variants. Hu and Tan developed a tool called MalGAN [22], which generates adversarial examples by training a generative network. To run this tool efficiently, attackers must first generate malware variant samples by combining original malware with noise, and then input those variants with other benign applications to the black box classifiers. Then, the outputs from the classifier are used to label the inputs (both malware variants and benign applications). However, training this model could be a challenging task with many variables that can continuously be updated over time. Also, if the detector is known (i.e., a white box classifier), we can achieve better performance for generating malware variants by guiding the perturbation applications similarly as in [6].

C. Genetic algorithm for malware variant generations

GA has been used widely in various contexts of computer applications [23], including malware evolution. Noreen *et al*. [24] proposed an evolutionary framework to generate new malware families, and evaluated their work using a malware family called Bagle. However, this approach needs multiple samples from the same family. Cani *et al*. [25] proposed an automated malware creation method using GA to insert a specified set of instructions to variate the code representation. Given a malware sample, the proposed approach identifies code blocks to be modified in the sample while preserving its functional behaviors with inserted instructions. Generating malware variants using GA was also extended into mobile devices. Aydogan and Sen [26] presented malware variant generation in the Android platforms using GA. They retrieved smali codes from the target apk files and then analyzed their control flow graph where nodes are then mutated using GA. Their work showed that malware variant generations can work with various malware inputs including the codes. Meng *et al*. [27] proposed a framework named MYSTIQUE to automatically generate malware samples with four attack features and two evasion features. Their comprehensive evaluation results with 57 malware detectors showed that existing malware detectors may still be ineffective against malware variants generated using GA. Recently, Castro *et al*. [28] proposed AIMED, an extended work from the ARMED framework [6], which uses GA in order to improve the generation process of malware variants by up to 50% faster. However, their GA is guided by the functionality of mutated malware variants through random perturbations without using the properties of malware detectors (especially if they are white box classifiers),

so the initial setup could be time consuming, as well as requiring GA tuning as the malware detectors evolve.

To develop effective malware detection methods which are resilient against malware variants generated by several transformation techniques, we should first study how malware variants are generated. However, there are only a few studies dealing with automated malware generation in academia. In this paper, we propose a malware variant generation framework with a malware detection module to guide the generated variants to satisfy specified detection conditions (e.g., similarity, behavior, functionality, specifications, etc.).

III. Adaptive Malware Variant Generation Framework

The **A**daptive **M**alware **V**ariant **G**eneration (AMVG) framework is comprised of five components, as shown in Figure 1. Some components are highly coupled due to the dependency of program codes. For example, if *Parser* component is changed to support a new source code format, the other components should also be changed with the change of *Parser* component. The black arrows in Figure 1 represent input(s) and output(s) of those components (e.g., the outputs of the parser is the input of the modifier component). The red dotted arrows in the figure represents the dependency between the models (e.g., parser component depends on the sample format, verifier and validator implementations depend on the parser outputs, etc.). As seen in the figure, not all components have dependencies, so that different implementations can be made to those to change the behavior of the AMVG framework in an adaptable manner. The detailed explanation of components is as follows.

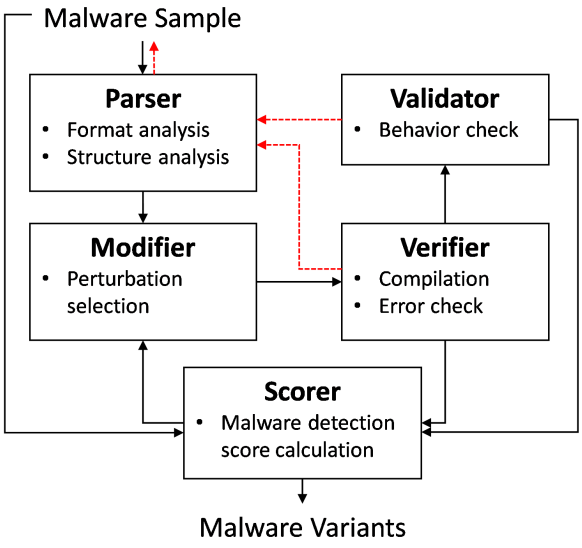


Fig. 1. AMVG framework

Parser processes a malware sample as input to generate its structured data for the modifier. In theory, the AMVG can be applied to any file format including binary executables and assembly programs [24]. However, the proprietary parser has to be implemented for each file format (e.g., C or Python source code) because there is no universal parser to process source codes written in various programming languages at the same time. The structured data from the parser is then passed to the *Modifier*.

Modifier applies modifications to the data structure to create variants. The variation can differ based on the inputs provided from the parser, but as long as applicable variations are specified for the given malware input, the operation of the modifier remains the same. For example, instruction sets can be used for assembly codes, while code formatting can be applied using source codes. As the modifier, different machine learning algorithms (e.g., genetic algorithm or deep neural network) can be chosen.

Verifier checks the syntactic correctness of the malware variant. The modifier could apply perturbations that cause the variant to not compile. Therefore, the verifier component is necessary to ensure that the generated variant works reliably without any functional issues. This is detected by checking any errors when the variants are compiled and executed.

Validator checks the logic of the malware. Even if the variant passes the verifier component, the behavior could be changed when perturbations are applied. Hence, the validator checks that the modifications by the modifier do not influence the operational flow of the original malware. In the same environment (i.e., with the same input), for example, if the original malware and the modified malware produce the same results, we may conclude that the modified malware has functionally equivalent behaviors to the original malware sample.

Scorer measures the degree of variation from the original malware sample to the generated malware variant. It takes into account the threshold values used in white box malware classifiers, which can be used to variate the score weights to different malware variants (i.e., closer to the threshold, higher weight value). If the malware classifier uses a black box model, then the framework behaves similar to the AIMED framework [28] (i.e., score value of 0 or 1 for detected and bypassed flags, respectively). The calculated score is then passed back to the modifier for future variant applications.

To demonstrate the effectiveness of our framework, we carry out two case studies: (1) Python and (2) C-based malware code samples in the following sections.

IV. Case Study 1: Python-based Malware

To demonstrate the effectiveness of the AMVG framework, we generate new malware samples written in Python. We use two representative malware samples, *Ares* and *Radium*, collected from theZoo repository [29].

The implementation of the AMVG framework components for the Python source is described. The *parser* processes the input code and identify variables, functions, and blocks. The *modifier* using GA applies a range of perturbations and apply them through a series of iterations. The *Verifier* checks for syntax and compile errors, and the *Validator* checks the behavior of the variant code. For the *scorer*, we selected to use ssdeep [9] and TLSH [8], fuzzy hashing techniques for

analyzing malware [16]. However, the *scorer* can be replaced with any other malware detection mechanisms.

A. Parser

For the Python-based malware codes, we implemented an abstract syntax tree (AST) constructor using the Python Standard Library. This process removes non-essential information such as comments, new lines, and excess white-space to sanitize the source code. Using the generated AST, we can quickly index variables, functions, and blocks of the malware source code. The AST also checks the syntax, which we use as a verifier for Python-based malware codes.

B. Modifier

The modifier component divides into two subcomponents: 1) *Applicable Perturbations* and 2) *Genetic Algorithm for Malware Variations*. We discuss each subcomponent as follows.

1) Applicable Perturbations: When applying perturbations, the outcome can either be functional (operational) or non-functional (broken). The difficulties of identifying the perturbation results are further discussed in Section IV-E. Further, possible perturbations can be classified onto (i) non-behavioral and (ii) behavioral changes. The non-behavioral changes modify the malware with perturbations that do not affect the logical operation of the malware (e.g., NOP operations), and the opposite for the behavioral changes. Because behavioral changes require extensive analysis in the validator component, we only consider the non-behavioral changes for the Python-based inputs (behavioral changes are considered later in Section V-B for C-based inputs).

The non-behavioral changes (NBC) are specified as follows:

(NBC1) Add No Operation: This mutation adds statements which have nothing affected to the results to a randomly selected function. For example, we insert a *pass* statement, which means that it does nothing. More complicatedly, multiple statements can also be inserted.

(NBC2) Add Print Function: This mutation adds a *print* function call to a randomly selected function. The *print* function affects to the standard output (e.g., occurring new line in the output text). However, the *print* function insertion does not affect the code behaviour because most malware does not use GUI. If malware is sensitive to the printed log, *AMVG* can disable this function.

(NBC3) Add Variable Declaration: This mutation creates a variable, assigns a value to it and randomly selects a function to add it to. For example, we declare a new *variable*, initiate it with meaningless value, and use it at a point that has no influence to the program logic.

(NBC4) Change Function/Variable Name: This mutation changes the name of an existing function/variable and updates all invocations with the generated name. The names can be changed to sequential numbering or random string. It is similar to one of the obfuscation technique which wipes the meaning of function/variable names.

(NBC5) Change Relational Operator: This mutation changes the relational operator within an if statement and updates the expression, so it is logically equivalent. For example, if a condition is that $var < 0$, it is converted to the condition that $!(var >= 0)$. These two conditions calculate the equivalent results.

(NBC6) AND and OR Reordering: This mutation shuffles the operands within an if statement. For example, if a condition is that var_1 *and* var_2 *or* var_3, it is converted to the condition that var_3 *or* var_1 *and* var_2.

(NBC7) Function Reordering: This mutation swaps two function blocks around. In other words, it changes the order of lines of functions. In some cases, the order between functions affects the operating program, because it is possible to invoke a function before it is declared due to interpreter characteristics of Python.

(NBC8) Function Keyword Arguments Reordering: This mutation shuffles the keyword arguments within a function. For example, if a function has an order like $func(arg_1, arg_2, arg_3)$, it is shuffled to $func(arg_3, arg_1, arg_2)$.

Because random perturbations are known to be slow [28], we guide our perturbations (i.e., NBCs) using malware detectors as our scorer, and feedback this information related to the malware variant generated to generate other variants. In particular a GA was chosen to generate malware variants.

2) Genetic Algorithm for Malware Variations: To use GA for generating variants, we first define various GA parameters. Then a set of operations are carried out, which are: *variant selection*, *mutation*, *cloning*, and *fitness calculations* are applied. The overall process of GA is as follows: (i) the population is initialized where each variant in the population is a copy of the original source, (ii) a subset of variants in the population is selected for mutation, and (iii) for each selected variant, a single NBC mutation is randomly applied. As only a single modification is applied to each selected variant per generation, the GA can guide variants towards the *false negative zone* using feedback and an iterative approach.

We configure 10 parameters to optimize GA.

Genetic Algorithm Mode specifies whether the GA should be run in *SINGLE* or *FIXED* mode. In *SINGLE* mode, the GA will terminate as soon as a variant that passes the scorer conditions. *FIXED* mode will continue running for the amount of iterations specified.

Max Generations define the maximum number the iterations the GA will execute, used for the *FIXED* mode.

Population Size specifies the number of variants in the population. As a general rule, larger populations have a higher chance of generating an optimal variant. However, larger populations consume more memory as more variants need to be stored in the population. Also, larger populations are computationally more expensive as more variants are likely to undergo mutation.

Number of Mutations defines the number of mutation options available for selection by the GA.

Mutation Probability specifies the probability that any variant in the population will be selected for mutation. For example, if the mutation probability is set to 0.5, there is a 50% chance that a variant in the population will undergo mutation.

Perform Cloning defines whether the GA will clone the fittest variants at the end of each iteration.
Number of Clones specifies the number of copies to make for each variant selected to be cloned. For example, if the number of clones is 2, then two copies are made resulting in three identical variants in the population.
Clone Rate defines the proportion of variants in the population to be cloned. For example, if the clone rate is 0.1, the fittest 10% of variants will be cloned.
Generated Identifier Length specifies the number of characters the generator will use to generate a random function or variable name.
Generated Max Number defines the maximum number the generator can assign a number variable. When the generator creates a number variable, the value assigned to it is randomly sampled between 0 and this parameter (has to be a numeric value, which can be negative).

Algorithm 1 describes the mutation process in detail. The mutation function has three parameters, which are the variant instance v, a set of NBC mutations m, and the mutation rate r_m. Once a random number r is less than r_m, then the mutation is applied for the particular v called. A random mutation m_r is selected from m and is applied to the copy of v (i.e., $v\prime$). This ensure the original variant is stored for later use if the new variant fails *validator* or *verifier* checks.

Algorithm 1 Malware variant mutation in GA

```
v: variant
m: set of NBC mutations
r_m: mutation rate
function APPLYMUTATION(v, m, r_m)
    r ← rand(0,1)                    ▷ random number [0,1) assigned
    if r_m > r then
        mod_applied ← False
        c ← copy(v)                  ▷ variant is copied
        while mod_applied = False do
            m_r ← random(m)          ▷ randomly select a mutation
            mutate(v', m_r)          ▷ apply selected mutation
        end while
    end if
    return v'
end function
```

Cloning variants creates larger population with higher scores, which leads to faster variant generations. After an iteration, the fittest variants are cloned to occupy a larger proportion of the population, while ones with low fitness values are removed. However, variant cloning can be turned off to use the original sets in the next iteration. *Number of Clones* is used to determine the number of replica variants.

C. Verifier

In AMVG experiments for the Python-based malware, Python compiler can be a *verifier* for the variants. In addition, AST can also verify the syntax error. The correctness of any perturbations is checked by AST prior to accepting the variant as functional, which is then passed as a new generation in the GA process.

D. Validator

Validator checks that the variants are the same as the original code. However, because non-behavioral changes are implemented, we can assume that the variants are the same as the original code without additional *validator* checks.

E. Scorer

The fitness value (between 0 and 1) of a variant is relative to the target score. First, the score value s of the variant v is calculated using the *scorer*. The misclassification range is not a single value, but given as a range with the midpoint value s_m. This is because instances of malware can become closer to other variants, and therefore the amount of changes should be moderated to specific needs (i.e., finding the gap). Also, setting a range for the framework allows control over how much change is applied by adapting the scorer. To smooth the fitness value, two smoothing variables O_l and O_f are introduced. O_l gives variant score s less than s_m a higher score, as the modifications are heading towards the target range. If the score is higher (i.e., $s > s_m$) then it is less likely to reduce with more changes so such variants do not get the advantage. O_f scopes the weight of the fitness value close to s_m If the O_f value is small, then only values very close to s_m will have significant fitness values, and vice versa. D is the maximum tolerable upper limit score which we will assign a fitness value. If the score value exceeds this value (i.e., $s > D$), then that variant received a fitness value zero. Finally, we assign highest fitness score (i.e., 1) if the variant score is the midpoint of the target range, and degrades as it gets further from the midpoint. This is captured using Equation (1). If no mutation happens, then the score will remain the same (i.e., 0), so the fitness value is also zero. If the score is exactly at the midpoint (i.e., $s = s_m$), then it is assigned the best fitness value. Otherwise linear weighting is used to assign the fitness value taking into account the difference of the variant score s and the midpoint score s_m.

$$Fitness(s) = \begin{cases} 0 & \text{if } s = 0 \\ 1 & \text{if } s = s_m \\ \frac{s-1}{s_m-1} \times (1-O_f) + O_f + O_l & \text{if } 0 < s < s_m \\ \frac{(2s_m-1)-s}{s_m-1} \times (1-O_f) + O_f & \text{if } s_m < s < 2s_m \\ \frac{D+(2s_m-1)-s}{D} \times O_f & \text{elsewhere} \end{cases} \quad (1)$$

V. Case study 2: C-based malware

The configuration of the AMVG framework components for the C-based inputs are as follows. The *parser* disassembles the variables, functions, and blocks of the code into a specific data structure we defined. The *modifier* using GA duplicates the input code, obtains the mutation probability, and then applies the mutation rules to the code. The *verifier* checks whether the mutated code is syntactically correct, and the *validator* checks

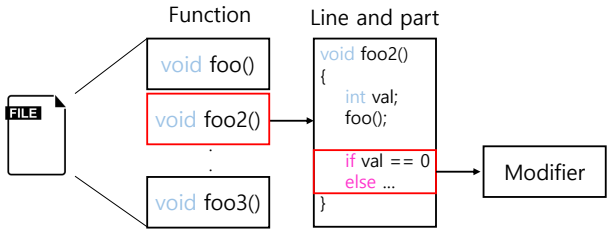


Fig. 2. Overview of C parser implementation.

whether the mutated code can generate the same results from the original code. For the *scorer*, we use ssdeep [9], which is a fuzzy hashing technique for analyzing malware [16]. Based on the ssdeep score, the *scorer* also selects the limited codes from the mutation source code and the before generation source code.

A. Parser

To convert the code into the structured data for the modifier, we defined the blocks at the *function* level, *line* level, and *part* level. For example, the *parser* disassembles the target code to the several blocks as shown in Figure 2.

Using the customized data structure, we can quickly adapt the mutation method to the variables, lines, and functions of the source code. Unlike AST we used in Section IV, the customized data structure has no ability of the syntax checks. Therefore, we need to check the correctness and validation of the variants additionally.

B. Modifier

We used GA as the *modifier* and applied two types of changes, such as non-behavioral changes (NBC) and behavioral changes (BC), which are mentioned in Section IV-B1. In the eight NBC, we chose two NBC (**Add Variable Declaration** and **Change Function/Variable name**) for modifying C-based source code. In addition, we applied another two NBC, and the specification of these NBC is as follows.

(NBC9) Extend operation: This mutation inserts a meaningless variable to the code line which has the arithmetic operations such as '+', '−', '×', and '÷'. For example, if a code is that $var_1 = var_2 + var_3$, it is converted to the several lines which are $var_1 = var_2 + var_{new}$, $var_1 = var_1 + var_3$, and $var_1 = var_1 - var_{new}$. The additional variable var_{new} is initiated to the random value which has no possibility of overflow or underflow errors. If we cannot expect the random range, we set the random range from the minimum value to the maximum value. The overflow and underflow errors can be detected at the *validator*.

(NBC10) Downsize conditional operator: This mutation decreases/increases the size of the conditional operator. Figure 3 shows how to change the original code using this mutation rule. With this mutation, the if statement on line 9 to 12 of Figure 3(a) is changed to the ternary operator on line 9 of Figure 3(b) and vice versa.

We also used two BC for modifying C-based source code.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int num1, num2;
7      printf("hello world");
8      gets(num1);
9      if (num1)
10         num2 = 100;
11     else
12         num2 = 200;
13 }
```

(a) Original source code.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6      printf("hello world");
7      int num1, num2;
8      gets(num1);
9      num2 = num1 ? 100 : 200;
10 }
11
12
13
```

(b) Variant source code.

Fig. 3. Example of a variant source code (b) that is semantically equivalent to the original source code (a).

(BC1) Add/Del Lines: This adds/deletes lines of code from the source code. When a line is added, the inserted line is not newly generated but brings from other source codes or variants. When a line is deleted, the deleted line is randomly chosen. Therefore, we need to use the *verifier* and the *validator* to check whether the deleted line is essential or not.

(BC2) Exchange Lines: This exchanges the position of lines of the code. Two random lines are selected and swapped. This mutation can be limited to lines within the same function or the file. Even though we can exchange the lines which are in other code, this is not considered as it produces the same result with the BC1 mutation.

C. Verifier

C compilers (e.g., GCC) can be a *verifier* for the variants. With the compile options of the source code, we check the functionality of the variants using GCC version 5.4.0. We considered that when the compiler can compile the variants, it passed through the *verifier*. The variants which can pass through the *verifier* are tested in the *validator*. Finally, the validity result is used in the *scorer*.

D. Validator

Validator checks that the variants are the same as the original code. However, it is hard to validate that the variants have the same operational flow to the original. We used the test case data for the variants and then compared the results from the answer of the test case data. If the variants output results which are the same to the answer, we assumed that the variants are functionally equivalent to the original code. Then the validation result is used in the *scorer*.

E. Scorer

ssdeep [9] was used as the *Scorer* for C-based inputs. The ssdeep outputs are the similarity score from 0 (different file) to 100 (same file). To apply to our experiment, we suggested the scoring equation, as shown in Equation 2 where v is a variant, o is the original code and $ssdeep$ is the ssdeep scoring function. Based on the score of the variants, we selected top-scored variants, and then if the variants exceeded over or did

TABLE I
PARAMETER CONFIGURATIONS FOR GA PARAMETER SETUP.

GC	Mode	Clone	Clone Rate	No. of Clones	Mutation Prob.	No. of Mutations	Gen String Length	Gen Max Number
1	SINGLE	FALSE	-	-	0.5	5	40	1,000,000
2	SINGLE	FALSE	-	-	0.5	9	40	1,000,000
3	SINGLE	TRUE	0.1	2	0.3	9	20	10,000
4	SINGLE	TRUE	0.1	2	0.3	5	40	1,000,000
5	SINGLE	TRUE	0.1	2	0.3	9	40	1,000,000
6	SINGLE	TRUE	0.15	3	0.5	5	40	1,000,000
7	SINGLE	TRUE	0.15	3	0.5	9	40	1,000,000
8	SINGLE	TRUE	0.25	2	0.5	5	40	1,000,000
9	SINGLE	TRUE	0.25	2	0.5	9	40	1,000,000
10	FIXED	TRUE	0.1	2	0.3	5	40	1,000,000
11	FIXED	TRUE	0.1	2	0.3	9	40	1,000,000

TABLE II
DESCRIPTION OF THE INPUT FILES.

File Name	Source	Type	LOC
Work Project	Python	Benign	300
Ares	Python	Malware	371
Radium	Python	Malware	563
You-Get	Python	Benign	940
Binary Tree	C	Benign	189

TABLE III
RUN CONFIGURATIONS FOR GA PARAMETER SETUP.

RC	Pop. Size	Max Generations	Distance Range
1	30	30	60-80
2	30	50	60-80
3	50	30	60-80
4	50	50	80-100
5	100	100	100-120

not reach the target score, they passed to the *modifier* and progressed GA again.

$$Score(v) = \{100 - ssdeep(v,o)\} * \begin{cases} 1 & \text{if } v \text{ passed the test} \\ -1 & \text{elsewhere} \end{cases} \quad (2)$$

VI. EXPERIMENTAL ANALYSIS

We investigate the use of our proposed AMVG framework in terms of applicability and performance, mainly focusing on the performance of malware variant generations under various conditions (e.g., file size, variability in operators etc.). Hence, a handful of input files were selected, representing various conditions for both benign and malware codes. First, we configure the GA parameter values to optimize the generation performance. Then, we evaluate the framework by creating malware variants. Finally, we experiment the performance of the framework to generate malware variants.

A. Experiment Setup

1) Test files: Four input files were used to evaluate Python-based codes, and one input file was used to evaluate C-based codes, which are summarized in Table II. We selected a benign C code, as we are also applying BCs that require validations. In addition to malware codes, benign codes were also used in the experiment to investigate the performance with respect to varying number of Lines of Codes (LOC). *Work Project* code is a simple data extraction program that aggregates different log files into a single one. *You-Get* is one of the program from an open-source utility project that handles media files. *Ares* is a cross platform remote access tool that acts as both a backdoor and botnet. *Radium* is a Windows-based keylogger that send recorded keystrokes via SMTP and screenshots uploaded via FTP. *Binary Tree* is a binary tree C code. All the input files perform different tasks with different structures. We demonstrate that our proposed framework can work with many different types of input files.

2) Configuring GA parameters: The first task is to configure the GA parameters to optimize the process of variant generations. Malware detectors using similarity measures set a threshold value against known malware samples. To bypass them, we set a similarity threshold values for TLSH and ssdeep score values, and test varying score ranges. In particular, three different distance ranges are tested, and the tested parameter combinations are shown in Tables I and III for the GA parameter configurations (GC) and run configurations (RC), respectively. Each GC is run against all five RCs for each of the four test files resulting, in 20 unique tests. As each test is run three times, statistics about each GC is averaged over 60 tests. Table IV shows the fitness values for each of the GCs, where GC_8 shows the highest fitness value and GC_3 the lowest in our setup. Hence, GC_8 will be used in our next experiment shown in Section VI-B. The GA parameters should be optimized for the best performance when using different input files.

3) Scoring: The scoring is done by the fitness calculation shown in Section IV-E. The parameters set for the scorer are as follows: (1) parameter O_l is assigned 0.001, (2) parameter O_f is assigned 0.25, and (3) the score upper limit is set to 1000. These parameter configurations smooth the score value assigned to the variant up to the upper limit in an equal manner.

B. Experimental Results

1) Python-based inputs: This experiment assesses the framework performance by testing the distance range beyond 100 (TLSH max score is 300) a similar analysis performed by Trend Micro [8]. Table V shows the RCs used in this

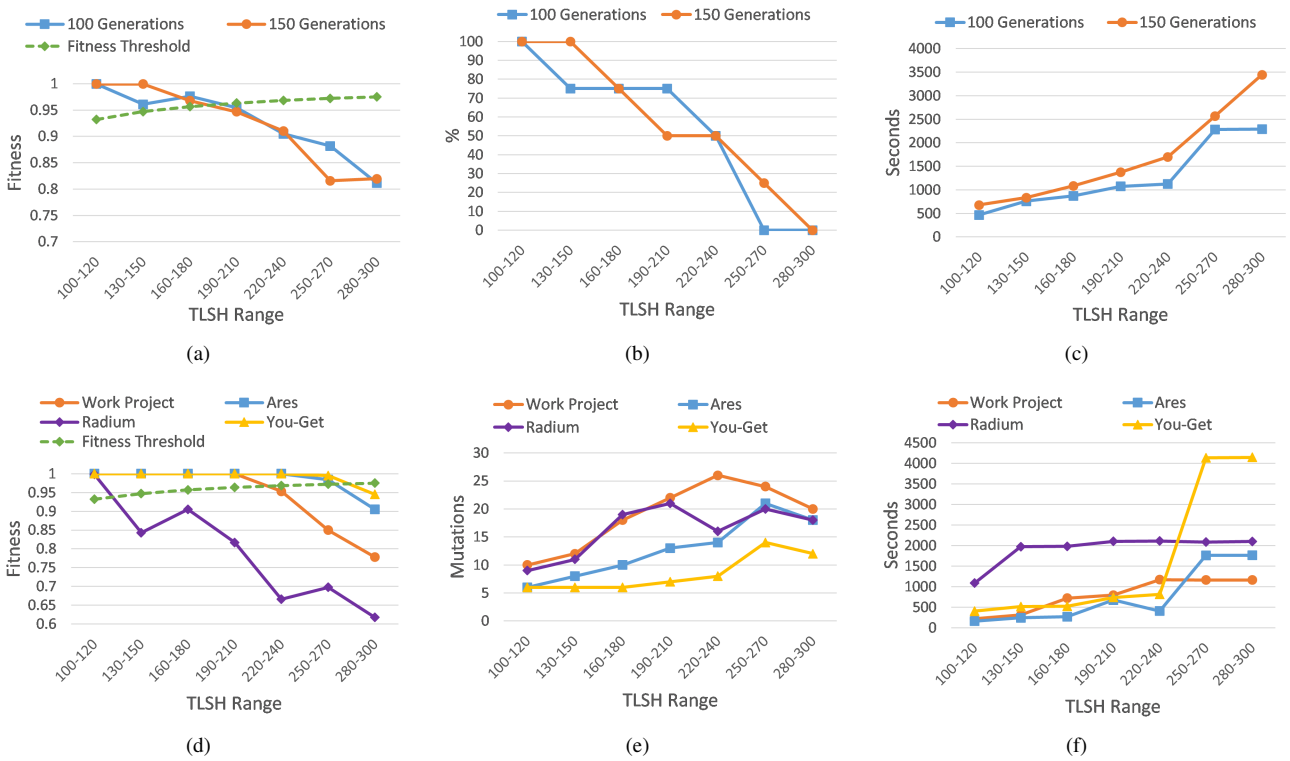


Fig. 4. Experimental results for Python-based sources based on TLSH scores.(a) compares the highest fitness score achieved when generating variants. (b) shows the optimal variant percentages out of the whole population generated. (c) shows the time taken to generate malware variants using different generation sizes. (d) shows the differences of the fitness values for selected sample sources. (e) shows the number of mutations required to achieve the optimal fitness score. (f) shows the time taken to generate the variant that satisfied the given TLSH score range.

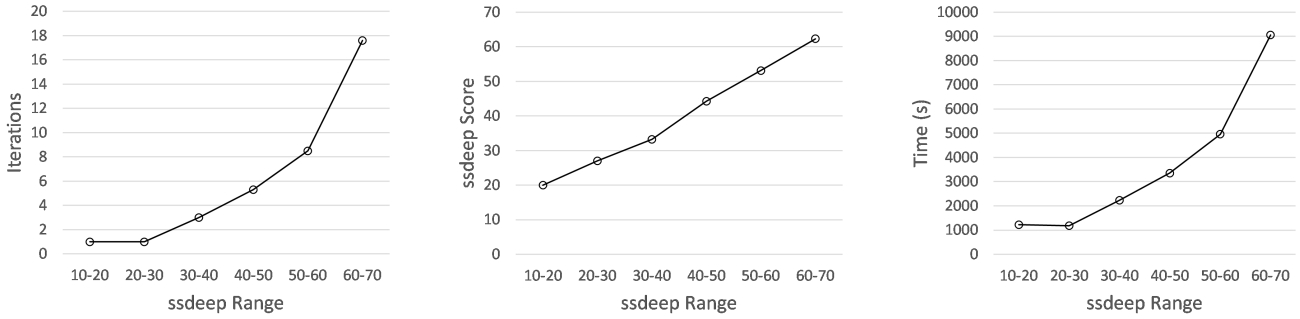
TABLE IV
FITTEST VARIANT FOR EACH GC.

GC	Fitness	GC	Fitness	GC	Fitness
1	0.984	5	0.955	9	0.984
2	0.974	6	0.981	10	0.967
3	0.932	**7**	**0.990**	11	0.966
4	0.979	8	0.944		

experiment, which span over seven distance ranges from 100-120 to 280-300. GC_8 (i.e., best configuration from experiment 1) was run against all 21 RCs listed in Table V for each of the four test files resulting in 81 unique tests. Figure 4 shows the experimental results.

First, we explore the effects of increasing the number of generations to find malware variants with respect to the performance. For this experiment, we compared the generation sizes of 100 and 150 (i.e., 50% difference in the number of generations). Figure 4(a) result shows that the two generation sizes had similar trend in achieving the fitness score for different TLSH score ranges. This result implies that there is no significant different in fitness level achieved for different population size used. The optimal variant percentage was analyzed as shown in Figure 4(b), which also indicate that there is no significant difference between increasing the generations and generating an optimal variant when looking at the percentage of optimal variants generated. Here, the percentage of optimal variants represents the proportion of variants in the population that satisfies the TLSH score range (e.g., for TLSH score range 220-240, about 50% of the generated variant population achieved the specified TLSH score). Finally, we compare the duration as shown in Figure 4(c). It also shows a negligent difference in terms of the performance time for the two different generation sizes. In conclusion, the difference in the generation size of less than 50% does not affect the performance of the AMVG framework significantly.

Next, we look at individual input files. Figure 4(d) shows a 100% optimal variant generation in three of the four files up to the TLSH score range 190-210. Also, variants for both *Ares* and *You-Get* have scores within the specified distance range up to 250-270. This show that regardless of the size of the input file, it is easier to achieve the desired fitness score with more diverse fields for perturbations. *Radium* has limited applicable mutations compared to other inputs, so it was more difficult to achieve the desired score as the target range increased. In contrast, all mutations outlined in Section IV-B1 were applicable to *You-Get*, and so it was still significantly easier to achieve the desired fitness score. Figure 4(e) also shows that *You-Get* required the least number of mutations across all distance ranges, although its size was the largest. Finally, Figure 4(f) shows the duration of generating the variants. The general trend shows a linearly increase. The big spike observed



(a) (b) (c)

Fig. 5. Experimental results for C-based source. (a) shows the number of iterations in GA to reach the target ssdeep score range. (b) shows the average ssdeep score achieved for the given target range. (c) shows the time taken to generate the variant to reach the target ssdeep score range.

TABLE V
RUN CONFIGURATIONS FOR EXPERIMENTS (C-BASED VARIANTS)

RC	Pop. Size	Max Generations	Distance Range
1	1000	200	10-20
2	1000	200	20-30
3	1000	200	30-40
4	1000	200	40-50
5	1000	200	50-60
6	1000	200	60-70
7	1000	200	70-80
8	1000	200	80-90

for *You-Get* from the range 250-270. It is possible that if the GA finds specific mutation to be effective, it may exhaust all options until it realizes that other mutations should be used. In conclusion, this experiment shows that the AMVG framework performance is more affected by the available modifiable fields in the input rather than the size of the input.

2) C-based inputs: Next, we investigate how the AMVG framework performs when the scorer (ssdeep) and input (C-based) are changed. The scoring of the ssdeep is done such that the score value of 0 means there is no similarity between the two compared files, up to 100 indicating that they are very similar or identical. Because we used TLSH scores between 0 and 300 for identical to not similar respectively, we apply $score_{ssdeep} = 100 - ssdeep(f_1, f_2)$, where $ssdeep(f_1, f_2)$ is the ssdeep score as described above (i.e., we reverse the range used for the ssdeep score in our experiment). Figure 5 shows the results for comparing the performance of finding the malware variant when a different scorer was used in the AMVG framework.

Figure 5(a) shows the changes in the number of iterations as the ssdeep range increases, which increases exponentially as the target range increases. However, ssdeep score is capped at 100, so it is possible to find such variant within a finite time. Figure 5(b) shows the average ssdeep score achieved for the target range. Figure 5(c) shows the average time taken to generate the variant that satisfied the score range. To verify the variant, we used the GCC compiler to ensure the variant is functional. Because the population size was reasonably large, compiling variants were the most time consuming task. As shown in Figure 5(a), the number of iterations increases exponentially, so the performance time also increased exponentially. In conclusion, the most time consuming part for C-based variants were the verifier checking each variant.

VII. DISCUSSION

Availability of malware samples: Malware samples are easily accessible nowadays online[2]. Although it enables researchers to access them freely, it also give malicious users more resources, such as malware source codes, assembly codes and so on. Hence, it is of paramount importance to understand the variability of malware families that could potentially be developed and used. Therefore, it is equally important to know various ways that the malware variants can be generated, in order to develop more effective defense methods.

In this paper, we used only a small selection of malware samples in our experiments. Although the selected samples represented various types of malwares (long, short, complex and simple), the differences between malware families are multi-dimensional (i.e., the variations are not easy to categorize). In our future work, more malware families will be examined, as well as different malware formats.

Protections against malware variants generated using ML: Malware variant generation frameworks (e.g., AMVG, ARMED and AIMED) exploit the limitations of the malware detectors (i.e., false-positive and false-negative). In order to prevent malware variants generated by ML or other similar techniques, the threshold value used for classifying malware and benign applications must also adaptively change to reduce FP and FN rates (e.g., for each classes of malware family). Such techniques will be further investigated in our future work.

Automated malware variation: We setup a list of NBCs and BCs in sections IV-B and V-B, but these are not exhaustive lists of changes. The challenge is that those variations are *input-dependent*. If we develop an adaptive variation plugin based on different input types, then the AMVG framework can generate variants more efficiently. However, this also requires

[2]e.g., at https://www.kernelmode.info/forum/viewtopic.php?f=16&t=308

a functional *validator* to examine the logical equivalence to the original input, and developing it is a challenging task (e.g., time consuming to observe the malware behaviors, such as executing malicious payload after certain time or after a trigger). More efficient means to validate variants will be explored in our future work.

VIII. Conclusion

To efficiently detect malware and its variants, we must understand how they are generated at various stages, starting from their creations to deployments. In this paper, the AMVG framework was proposed to generate malware variants for the chosen target system by adapting the framework components based on the target system details (e.g., setting the scorer using the malware detector on the target system). Using the AMVG framework, we can generate malware variants more efficiently by limiting the bypass conditions based on the target system configurations. Our experimental analysis showed that regardless of the type and size of the input malware source, the AMVG framework was able to generate multiple variants that satisfied the scorer conditions. We demonstrated this by using two different malware formats, Python and C code-based malware samples, with the same scorer to show that detection criteria can be bypassed in a practical time. With the capabilities easily extend the AMVG framework by adding new interfaces for the parser, verifier and validator to handle other malware source types, it enables us to understand various malware variant properties that could be helpful for developing more sophisticated malware detectors in the future.

Acknowledgment

This project is supported by the ICT R&D program (No.2017-0-00545, IITP-2019-2015-0-00403) and the Australian Government through the Australia-Korea Foundation (AKF) of the Department of Foreign Affairs and Trade (DFAT) project "Intelligent Cybersecurity: Jumpstarting Collaboration on Smarter Security for Dynamic Networks".

References

[1] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and Classification of Malware Behavior," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008, pp. 108–125.

[2] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant Permission Identification for Machine-Learning-Based Android Malware Detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[3] "Automated Malware Signature Generation," U.S. Patent US9 996 693B2, 2012. [Online]. Available: https://patents.google.com/patent/US9996693B2/en

[4] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, "Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning," 2018.

[5] S. Qiu, Q. Liu, S. Zhou, and C. Wu, "Review of Artificial Intelligence Adversarial Attack and Defense Technologies," *Applied Sciences*, vol. 9, no. 5, 2019.

[6] R. Castro, C. Schmitt, and G. Rodosek, "ARMED: How Automatic Malware Modifications Can Evade Static Detection?" in *Proc. of International Conference on Information Management (ICIM 2019)*, 2019.

[7] F. Manavi and A. Hamzeh, "A New Approach for Malware Detection Based on Evolutionary Algorithm," in *Proc. of the Genetic and Evolutionary Computation Conference Companion (GECCO 2019)*, 2019, pp. 1619–1624.

[8] J. Oliver, C. Cheng, and Y. Chen, "TLSH–A Locality Sensitive Hash," in *Proc. of the 4th Cybercrime and Trustworthy Computing Workshop (CTC 2013)*, 2013, pp. 7–13.

[9] J. Upchurch and X. Zhou, "Variant: A Malware Similarity Testing Framework," in *Proc. of the 10th International Conference on Malicious and Unwanted Software (MALWARE 2015)*, 2015, pp. 31–39.

[10] H. Naderi, P. Vinod, M. Conti, S. Parsa, and M. H. Alaeiyan, "Malware Signature Generation Using Locality Sensitive Hashing," in *Proc. of the 3rd ISEA International Conference on Security & Privacy (ISAP 2019)*. Springer Singapore, 2019, pp. 115–124.

[11] N. Idika and A. P. Mathur, "A Survey of Malware Detection Techniques," *Purdue University*, vol. 48, 2007.

[12] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A Survey on Malware Detection Using Data Mining Techniques," *ACM Computing Survey*, vol. 50, no. 3, pp. 41:1–41:40, 2017.

[13] A. Souri and R. Hosseini, "A State-of-the-art Survey of Malware Detection Approaches using Data Mining Techniques," *Human-centric Computing and Information Sciences*, vol. 8, no. 1, p. 3, 2018.

[14] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho, "Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection," in *Proc. of International Conference on Advances in Computing, Control, and Telecommunication Technologies (ACT 2010)*, 2010, pp. 201–203.

[15] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar, "Can Machine Learning Be Secure?" in *Proc. of the ACM Symposium on Information, Computer and Communications Security (ASIACCS 2006)*, 2006, pp. 16–25.

[16] I. Shiel and S. O'Shaughnessy, "Improving File-level Fuzzy Hashes for Malware Variant Classification," *Digital Investigation*, vol. 28, pp. S88–S94, 2019.

[17] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding Malware Analysis Using Conditional Code Obfuscation," in *Proc. of Annual Network & Distributed System Security Symposium (NDSS 2008)*, 2008.

[18] J.-M. Borello and L. Mé, "Code Obfuscation Techniques for Metamorphic Viruses," *Journal in Computer Virology*, vol. 4, no. 3, pp. 211–220, 2008.

[19] I. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," in *Proc. of the International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA 2010)*, 2010, pp. 297–300.

[20] G. Suarez-Tangil et al., "DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware," in *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY 2017)*, 2017, pp. 309–320.

[21] A. Cimitile, F. Martinelli, F. Mercaldo, V. Nardone, and A. Santone, "Formal Methods Meet Mobile Code Obfuscation Identification of Code Reordering Technique," in *Proc. of the IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2017)*, 2017, pp. 263–268.

[22] W. Hu and Y. Tan, "Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN," 2017. [Online]. Available: https://github.com/yanminglai/Malware-GAN

[23] M. Gen and L. Lin, *Genetic Algorithms*. American Cancer Society, 2008, pp. 1–15.

[24] S. Noreen, S. Murtaza, M. Z. Shafiq, and M. Farooq, "Evolvable Malware," in *Proc. of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO 2009)*, 2009, pp. 1569–1576.

[25] A. Cani, M. Gaudesi, E. Sanchez, G. Squillero, and A. Tonda, "Towards Automated Malware Creation: Code Generation and Code Integration," in *Proc. of the 29th Annual ACM Symposium on Applied Computing (SAC 2014)*, 2014, pp. 157–160.

[26] E. Aydogan and S. Sen, "Automatic Generation of Mobile Malwares Using Genetic Programming," in *Proc. of the European Conference on the Applications of Evolutionary Computation (EvoApplications 2015)*, 2015, pp. 745–756.

[27] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving Android Malware for Auditing Anti-Malware Tools," in *Proc. of ACM on Asia Conference on Computer and Communications Security (ASIACCS 2016)*, 2016, pp. 365–376.

[28] R. Castro, C. Schmitt, and G. Rodosek, "AIMED: Evolving Malware with Genetic Programming to Evade Detection," in *Proc. of the 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2019)*, 2019.

[29] Y. Nativ, "theZoo," 2019. [Online]. Available: https://github.com/ytisf/theZoo