FUMVar: A Practical Framework for Generating Fully-working and Unseen Malware Variants

Beomjin Jin
Sungkyunkwan University
Suwon, Republic of Korea
jinbumjin@skku.edu

Jusop Choi
Sungkyunkwan University
Suwon, Republic of Korea
cjs1992@skku.edu

Hyoungshick Kim
Sungkyunkwan University
Suwon, Republic of Korea
hyoung@skku.edu

Jin B. Hong
University of Western Australia
Perth, Australia
jin.hong@uwa.edu.au

ABSTRACT

It is crucial to understand how malware variants are generated to bypass malware detection systems and understand their characteristics to improve the detectors' performances. To achieve this goal, we propose an evolutionary-based framework named FUMVar to generate Fully-working and Unseen Malware Variants. In particular, we applied FUMVar on portable executable (PE) files that have been used extensively to infect Windows operating systems. Compared to the state-of-the-art approach named *AIMED*, our experimental results show that FUMVar generated 25% more evasive malware variants while reducing the time taken to generate them by 23%. Furthermore, FUMVar generated malware variants that bypassed commercial anti-malware engines, such as TrendMicro, with an alarming rate of up to 73% false-negative rate. To improve the detection techniques, we evaluate how different perturbations enhance the evasiveness and how different malware categories are affected by those perturbations. The results show that perturbations' effectiveness varies significantly by up to 6 times (e.g., section add v.s. unpack), and more suitable perturbations can be selected for different malware categories due to their varying applications. This information can then be used to develop more robust malware detection systems to detect unseen malware variants more effectively.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies → Genetic algorithms**;

KEYWORDS

Malware Generation, Malware Variation, Windows PE

ACM Reference Format:
Beomjin Jin, Jusop Choi, Hyoungshick Kim, and Jin B. Hong. 2021. FUMVar: A Practical Framework for Generating Fully-working and Unseen Malware Variants. In *Proceedings of ACM SAC Conference (SAC'21).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3412841.3442039

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC'21, March 22-March 26, 2021, Gwangju, South Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442039

1 INTRODUCTION

Various anti-malware solutions have been developed to prevent malware attacks [18, 22]. However, most of those solutions lack capabilities to detect unknown and unseen malware samples. To exploit this, malware writers use various techniques to generate new malware samples. One of the most popularly used technique is to generate variants using an existing malware sample through polymorphic and metamorphic mutations [14]. For example, a *Cerber* ransomware variant was generated every 15 seconds [12], creating a difficulty detecting its variants efficiently. Therefore, it is important to understand how evasive malware variants are generated and what makes them evasive first in order to design malware detectors that are robust to those malware evasion techniques.

Many researchers proposed a framework to generate malware variants [1, 3–6, 8]. For example, Anderson *et al.* [1] proposed a framework based on reinforcement learning to attack static portable executable (PE) anti-malware solutions, and Castro *et al.* [3] presented a framework called *AIMED* using a genetic algorithm to apply random perturbations to generate malware variants. However, most existing techniques simply checked whether the generated variants are a valid executable format only, without checking their behaviors (i.e., for their designated tasks) nor what makes them evasive and effective. Consequently, those variants cannot guarantee that the original malware samples' behaviors are maintained (i.e., fully-working), as well as being unclear about which perturbations might be useful to generate evasive malware variants against existing malware detection solutions. Subsequently, those studies do not provide fine-grained inputs for improving malware detection techniques. To overcome these limitations, we need to examine various perturbation techniques as well as ensuring that malware variants still maintain the same behaviors as their original samples.

In this paper, we implement a novel malware variant generation framework named FUMVar, which performs perturbations on malware samples in the Windows PE format. We use a genetic algorithm (GA) with a practically useful fitness score function based on a real-world malware detection system, VirusTotal (https://www.virustotal.com/). We also validate the functionality of generated malware variants by comparing their systems calls against the original malware sample using the Cuckoo Sandbox

(https://www.cuckoosandbox.org). The effectiveness of FUMVar is demonstrated via comparison to the state-of-the-art malware variant generator *AIMED* [3], with an improvement to the performance of 23% faster generation and 25% more evasiveness. Further, we evaluate the variants generated against commercial anti-malware solutions (e.g., Avast, AVG, BitDefender, and Kaspersky), achieving a false-negative rate of up to 73% against current commercial anti-malware solutions. This highlights some critical limitations of existing solutions to detect malware variants. To enhance the performance of existing malware detection techniques, we evaluate different perturbations and malware categories to identify the cause of evasiveness in malware variants. The main contributions of this paper can be summarized below:

- We propose an efficient framework named FUMVar that generates fully-working and unseen malware variants to evade malware detectors using GA[1];
- We demonstrate the performance improvement of FUMVar compared with the state-of-the-art malware variant generation framework *AIMED* [3] in terms of evasion rate and generation time;
- We evaluate the robustness of seven commercial anti-malware solutions against the malware variants generated by FUMVar, achieving up to 73% false-negative rate on detection rate;
- We analyze various perturbations and malware categories to identify characteristics of malware variants' evasiveness that can be used to enhance malware detectors.

2 BACKGROUND: WINDOWS PORTABLE EXECUTABLE (PE) FORMAT

The PE format is a data structure that contains the necessary information for Windows OS loader [13] to manage the executable code[2]. Fig. 1 shows the high-level PE format structure, which consists of the Disk Operating System (DOS) header, PE header (DOS stub, Common Object File Format (COFF) file header, optional header, and section table), and PE sections. The PE format is then used later to identify where perturbations are injected in PE files.

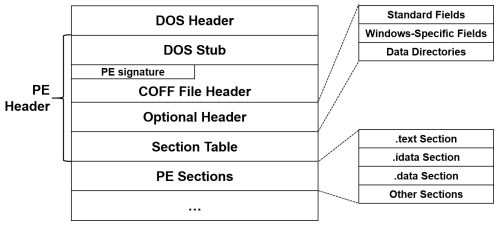


Figure 1: Overview of the PE format.

The *DOS header* and the *DOS Stub* are used for backward compatibility with DOS. The *COFF file header* contains essential information such as the type of machine for which the file is intended to execute, the type of the file (e.g., execution file, object file, or shared library), and the number of sections. However, it also includes some nonessential information such as timestamps and the number of symbols. Therefore, we can modify such fields that do not break the PE file format in the *COFF file header*.

[1]Our source code is available at https://github.com/FUMVar/FUMVar.
[2]The detailed specification is described on the official website (https://docs.microsoft.com/en-us/windows/win32/debug/pe-format).

The *optional header* contains data about how the executable must be loaded into memory. It consists of three major parts: *standard fields*, *Windows-specific fields*, and *data directories*. The *standard fields* are necessary for all COFF files. Therefore, we cannot perform perturbations in those fields. On the other hand, *Windows-specific fields* have 21 fields for the linker and loader information. We manually tried to perturb each field with various value ranges and checked whether the perturbations break the PE file format. Then, among 21 fields, we selected 15 of them (e.g., checksum and DLL characteristics) for perturbation. The *data directories* part has the address and size of the tables and strings that Windows uses. Similar to Windows-specific fields, we manually checked that all fields in the data directories part are modifiable.

The *section table* (also known as *section header*) contains the information of the section body. The *section table* consists of 10 fields directly related to the program activities. We choose two fields as examples of perturbation (i.e., virtual size and raw data size). The other fields, such as a virtual address, could be utilized by the program, so those fields cannot be modified. Besides, we also perturb the *rich header* which starts with a string "Rich." This field is not described in the official PE structure specification.

3 FUMVAR: AN OVERVIEW

In this section, we describe the overview of FUMVar, which comprises five modules, as shown in Fig. 2. The *Parser* module examines the structure of a given input PE file. The *Modifier* module applies mutations to the input malware sample using specified perturbations. The *Verifier* module confirms whether the modified malware sample is corrupted or not. The *Validator* module checks the modified malware sample's functionality to ensure that the original malware sample's behavior is maintained. Finally, the *Scorer* module computes the modified malware sample's score in terms of evasion and avoidance from malware detectors. Details of each module are described in the following subsections.

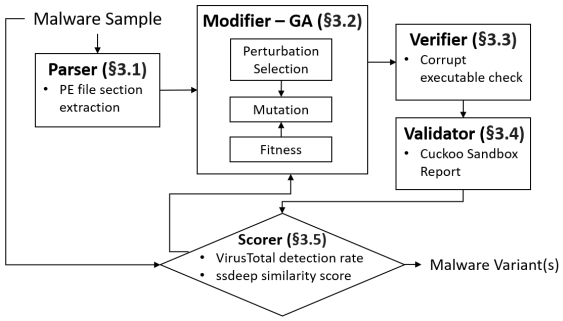


Figure 2: Overview of FUMVar.

3.1 Parser - extracting sections in PE files

A malware sample in raw PE format is given as the input to the *Parser* module. The *Parser* module examines the input PE file structure and identifies the sections and fields to be perturbed during the mutation process (as described in Section 3.2). After identifying those sections and tagged fields, the processed PE file is then passed to the *Modifier* module.

3.2 Modifier - applying mutations to PE files

The *Modifier* module iteratively modifies a given malware sample using perturbations, and perturbations are selectively chosen using a GA, but other selection process can replace GA as appropriate.

Perturbations on PE files: When applying perturbations to modify PE files, there are two main types of changes that can be made to malware samples: (a) behavioral changes (BC) and (b) non-behavioral changes (NBC). Table 1 shows the list of all perturbations used in FUMVar. The BCs can insert/modify/remove parts that affect how the malware functions when executed. Ideally, all critical functions are maintained for its malicious activities, but there is no guarantee that the original functionalities will be preserved. On the other hand, NBCs do not alter the existing functionalities, and instead add/modify/remove benign actions (e.g., NOP, variable name changes, etc.). We used LIEF [17][3] to implement perturbations.

Table 1: List of perturbations used in FUMVar.

Types	Name	Abbr	Description
NBC	Overlay append	OA	Adds the random length of zeroes to the end of the binary.
	DOS header	DH	Change the field values in DOS header.
	DOS stub	DS	Change DOS stub to random byte sequence.
	COFF header	CH	Change the field values in COFF header.
	Optional header	OH	Change the field values in Optional header.
	Data directory	DD	Change the field values in Data directory.
	Rich header	RH	Insert a new content info Rich header.
BC	Section rename	SRN	Change the name of a randomly selected section.
	Section add	SAD	Add a new section with random sequence of bytes.
	Section append	SAP	Append random bytes sequence to section content.
	Code cave inject	CI	Inject some random byte sequence to code cave.
	Pack	PA	Pack the file with UPX tool.
	Unpack	UP	Unpack the file with UPX tool.

Genetic algorithm: In FUMVar, we use a GA to "evolve" malware variants with effective perturbations. We use the *Scorer* module (see Section 3.5), providing the fitness value, such that the malware variant generated can be guided to reduce the detection rate and similarity to the original malware sample as the number of generations increases. In practice, multiple perturbations are applied to the malware sample through generations.

This process begins by initializing the population with a copy of the original malware sample. We then selected the top n fittest members from the population. All variants (i.e., the outputs from the GA) are then passed to the *Verifier* module, where functional ones are passed to the *Scorer* module. Once the scores are received from the *Scorer* module back to the GA, it populates the next generation.

The detailed procedure is presented in Algorithm 1. The algorithm first initializes the population in the first for-loop and continues until a specified number of generations is reached. Here, PL is the whole list of population as a set, and PL_s represents a subset that contains the top n malware variants with respect to the fitness scores, and PL_r represents a randomly selected subset of the population.

3.3 Verifier - checking corrupted variants

Applying perturbations may result in the malware being corrupted, which should be filtered in generations. To check the corruption state, we examine the debug information from the Cuckoo sandbox instead of manually executing the variants to reduce the computational overhead. If the PE file is corrupted, the Cuckoo sandbox displays the following error message: "`CuckooPackageError: Unable to execute the initial process, analysis aborted.`"

[3]LIEF is a popularly used tool that provides a cross-platform library that can parse and modify many executable formats, including PE format.

Algorithm 1 Malware variant generation with a GA

VT: VirusTotal detection rate
PS: Random perturbation selection function
FN: Functionality of the malware variant
b: Input malware bytes
p_{size}: Population size
g_{max}: Maximum generation
$PL \leftarrow []$: population list
$sol \leftarrow []$: solution list
function GA(b,p_{size},g_{max})
  for $i \leftarrow 1$ to p_{size} **do** ▷ Initialize population
    $perturbation \leftarrow PS(perturbation_list)$
    $PL.append(Mutate([b], [perturbation])$
  end for
  for $g \leftarrow 1$ to g_{max} **do** ▷ Main loop
    for $i \leftarrow 1$ to p_{size} **do**
      if $FN(PL[i]) = True$ and $VT(PL[i]) = 0$ **then**
        $sol.append(PL[i])$
      end if
    end for
    $PL_s \leftarrow Selection(Fitness(PL), n)$
    $PL_r \leftarrow Selection(RandomSelect(PL), PL - PL_s)$
    $PL \leftarrow PL_s + PL_r$
    $PL \leftarrow Mutate(PL, PS(perturbation_list))$
  end for
  return sol
end function

3.4 Validator - checking malware behaviors

It is not sufficient to verify only the correctness of a malware variant's PE file format. To address this, we perform validation by checking malware's behaviors, which is not done by AIMED. That is, we additionally check whether the modified variants' behaviors are the same as those of the original malware sample. To do this, we analyze the Cuckoo sandbox report about the malware variants against the original malware sample, which contains two attributes to represent malware samples' behaviors; API call sequence and signature fields. Here the signature fields are used to capture malware samples' suspicious behaviors (e.g., malicious network request, thread injection).

Since the API call sequences and signature fields in the Cuckoo sandbox reports are not always the same even when an identical malware sample is tested multiple times, we measure the similarity between the malware variant's API call sequence and the original malware sample's API call sequence rather than their exact matching sequence. We use ssdeep [10], a widely used fuzzy hash function for the API call sequences comparison. Also, we use Jaccard distance [9] to measure the dissimilarity between the sets of signature fields for the original malware sample and its variant, as we do not consider the sequence of signature fields.

To set the similarity threshold, we randomly selected ten malware samples manually and ran each malware sample ten times, respectively. Next, we calculated the pairwise similarity scores of all ten executions for each malware sample and selected the minimum similarity scores as thresholds. We believe that the selected thresholds are conservative values to be applied for all other variants.

To verify the feasibility of the chosen thresholds, we generate 30 variants from 10 malware samples by applying a few perturbations only to have the similarity scores over the thresholds. Then, we used Process Monitor [15] and tcpdump to manually examine their runtime behaviors (e.g., system call, file access, network requests) and confirmed that their behaviors are not changed. Therefore, we believe that those thresholds are sufficiently reasonable.

3.5 Scorer - measuring malware variants

In order to best reflect malware variants' evasiveness, we use malware detection tools directly in the *Scorer* module, which feedback the fitness value to the GA shown in Algorithm 1. Here, we use VirusTotal as the malware detection tool, which contains 74 anti-malware solutions such as Avast, AVG, BitDefender, and Kaspersky. That is, the VirusTotal detection rate is used as a fitness value. Besides, we also consider the code similarity score (computed by ssdeep) between the original malware sample and the generated variant because the code similarity can also be used to measure the dissimilarity between them. We note that the code similarity can often be used to detect malware samples [19]. A fitness value is calculated as the sum of them, as shown in Equation (1).

$$Fitness(var_{orig,i}) = w \times VT(var_{orig,i}) + (1-w) \times ssdeep(orig, var_{orig,i}) \quad (1)$$

The definitions are provided in Table 2 for Equation (1) and other equations presented later. VirusTotal detection rate is defined as the number of detected solutions divided by the total number of solutions in VirusTotal. Also, we use ssdeep to calculate the similarity score with byte codes of malware variants and original malware samples.

Table 2: Notations used in equations.

Notation	Description
$ORIG$	The set of original malware samples.
$orig$	The original malware sample $orig \in ORIG$.
VAR_{orig}	The set of variants generated from $orig$.
$var_{orig,i}$	The ith variant generated from $orig$.
$VT(x)$	A function that computes the VirusTotal detection rate for input malware x.
$ssdeep(x, y)$	A function that computes the ssdeep score for inputs x and y (e.g., malware binaries).
$VAL(x)$	A validation function that returns 1 if the input malware x is valid, else 0.
w	User-defined weight value that determines the proportional of the two scores the fitness score will rely on ($0 \leq w \leq 1$).

4 EXPERIMENTAL ANALYSIS

In this section, we carry out experimental analysis to evaluate the effectiveness of FUMVar. We collected 112 representative malware samples for the experiments to cover various malware (e.g., malware families, size, and API calls) from various malware repositories [11]. Using these samples, we first compare FUMVar against the state-of-the-art malware variant generator named *AIMED* [3] and compare their performances, as shown in Section 4.1. Next, we examine the detection ability of commercial anti-malware solutions[4]: Avast, AVG, BitDefender, Kaspersky, Malwarebytes, McAfee, and TrendMicro, against malware variants generated using FUMVar, as shown in Section 4.2. To identify malware variants' evasiveness, we then evaluate the effectiveness of different perturbations described in Table 1, as shown in Section 4.3. Moreover, we examine how different malware categories affect generating malware variants, as shown in Section 4.4.

To evaluate, we propose using the *valid variant generation success rate* (SR) and the *false-negative rate* (FN rate). The SR is defined as the number of functionally correct malware variants (i.e., passes both *Verifier* and *Validator*) divided by the total number of malware variants, as shown in Equation (2). The FN rate is defined as the difference between the original malware sample's detection rate to its functionally correct malware variants with chosen anti-malware solution(s)[5], as shown in Equation (3).

$$SR = \sum_{i \in |VAR_{orig}|} \frac{VAL(var_{orig,i})}{|VAR_{orig}|} \forall orig \in ORIG \quad (2)$$

$$FN\ rate = \frac{\sum_{orig \in ORIG} \max_{i \in |VAR_{orig}|} VT(orig) - VT(var_{orig,i})}{|ORIG|} \quad (3)$$

4.1 Baseline comparison: AIMED

In this experiment, 112 malware samples are used with $g_{max} = 100$ (i.e., max 100 generations) for both FUMVar and *AIMED*, with one perturbation applied at each generation. Further, we consider two perturbation selection strategies: (a) FUMVar-equal and (b) FUMVar-weighted. Using FUMVar-equal, the probability values are equally assigned to each perturbation used (the baseline strategy). Using FUMVar-weighted, each perturbation is assigned to a different probability value based on results obtained later in Section 4.3. That is, we assigned weight values for each perturbation by multiplying their associated SR and FN rate values as shown in Equation (4), where w_{pt_i} is the weight assigned to pt_i (i.e., the ith perturbation), and $SR(pt_i)$ and $FN\ rate(pt_i)$ are the SR and FN rate values for pt_i, respectively). The results are shown in Fig. 3, the SR and FN rate results for two strategies of FUMVar: FUMVar-equal (FUMVar-e), and FUMVar-weighted (FUMVar-w), compared with *AIMED*.

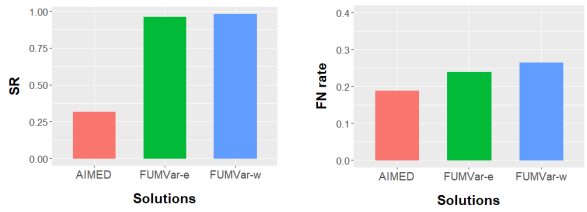
$$w_{pt_i} = \frac{SR(pt_i) \times FN\ rate(pt_i)}{|perturbation_list|} \quad (4)$$

Fig. 3a shows that both proposed strategies (i.e., FUMVar-equal and FUMVar-weighted) generated malware variants with over 90% SR, while AIMED only achieving around 30% SR. This result indicates that AIMED produces incorrectly working malware variants six times higher than the proposed strategies. Further, AIMED does

[4]The list can be found at www.trustedantiviruscompare.com/best-free-antivirus. The other three solutions in the list not analyzed are not used in VirusTotal.
[5]In the case of multiple anti-malware solutions being used, their average is calculated.

not check the validity of malware variants generated while FUMVar does it by checking the behavior of malware variants generated using the Cuckoo sandbox report. In addition, Fig. 3b shows that both proposed strategies achieved a higher FN rate compared with AIMED by 18% and 25%, respectively.

Next, we compare the time taken to generate malware variants using FUMVar and AIMED. Since the only difference between FUMVar-equal and FUMVar-weighted is the probability weights on perturbations, their execution time is the same. Therefore, we just use FUMVar in Fig. 4. To compare the time taken in a fair manner, we used the same parameters and environment setups with AIMED (e.g., number of perturbations, perturbation probabilities). We used Ubuntu 16.04 x64 with four cores and 16GB RAM to run both AIMED and FUMVar. According to the parameters used in the experiments for AIMED [3], we set the population size $p_{size} = 4$, generation max $g_{max} = 100$, and $n = 2$ (i.e., the top 2 are selected for the next generation round). Fig. 4 shows that both frameworks' time increased linearly for the number of generations, but FUMVar achieved, on average, 23% faster than AIMED for generating malware variants. Unlike AIMED, FUMVar was implemented to reduce the execution time by collecting both Cuckoo sandbox report and VirusTotal analysis results with two processes in a parallel manner.



(a) SR (b) FN rate

Figure 3: FUMVar and AIMED performance comparison.

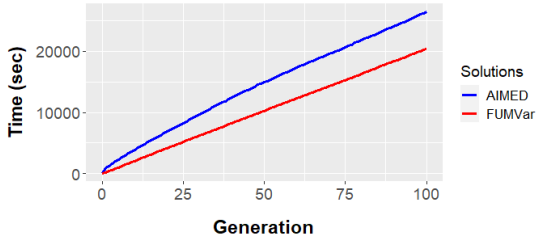


Figure 4: Avg. time taken to generate malware variants.

From the results above, we observe a critical limitation of the AIMED, which does not adopt malware solutions in their fitness function. Consequently, our experiment showed that only using similarity scores to generate evasive malware variants is significantly less effective (i.e., less evasive) than using malware solutions in the fitness function.

4.2 Evading anti-malware solutions

We examine commercial anti-malware solutions to detect malware variants, particularly the top 7 solutions used in VirusTotal: Avast, AVG, BitDefender, Kaspersky, Malwarebytes, McAfee, and TrendMicro. These solutions are used to evaluate the evasiveness of both proposed strategies[6]. Fig. 5 shows the results, where two observations are made: (Obs1) **all** solutions can be bypassed with a varying degree of detection rate, and (Obs2) FUMVar-weighted performed better than FUMVar-equal for almost all solutions tested.

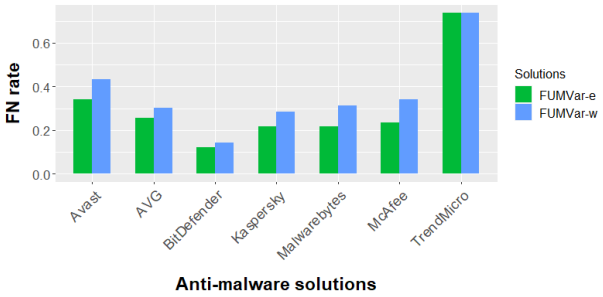


Figure 5: Bypassing anti-malware solutions using FUMVar.

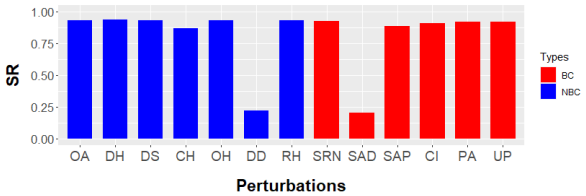
From (Obs1), no solutions were able to fully detect all malware variants generated by FUMVar. In fact, without BitDefender, the FN rate was at least 20% for all solutions using FUMVar-weighted (i.e., 1 in 5 malware variants evaded detection), and at least 10% FN rate is achieved for all solutions tested (i.e., 1 in 10 malware variant evaded detection). In our evaluation, the best performing solution was BitDefender with less than 15% FN rate, while the worst performing solution was TrendMicro, which had up to 73% FN rate (i.e., 2 in 3 malware variants evaded detection), closely followed by Avast with 45% FN rate. The significant gap between the FN rates indicates that not all anti-malware solutions are resistant against malware variants, and those can be exploited by carefully crafting malware variants. From (Obs2), allocating higher weight values to perturbations with a higher FN rate (i.e., FUMVar-weighted) improved the overall FN rate. In general, FUMVar-weighted outperformed FUMVar-equal for all solutions on average by 6%.

4.3 Perturbations and their effectiveness

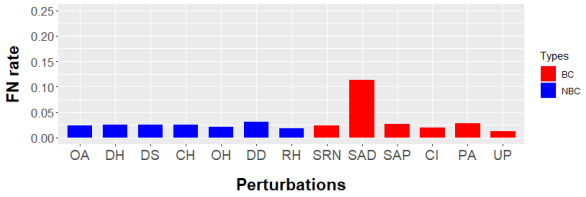
Section 4.2 shows that FUMVar is effective in bypassing commercial anti-malware solutions. To develop more robust anti-malware solutions against such variants, we need to understand what makes our variants more evasive. We applied a specific perturbation once on each of the 112 malware samples and calculated its SR and FN rate, respectively. For the FN rate, we used VirusTotal which consists of 74 anti-malware solutions. Fig. 6 shows the average of those rates across all malware samples. We only considered functionally working malware variants having the same behavior as the original malware sample after performing perturbations on them for the FN rate (i.e., passes both *Verifier* and *Validator* modules), as others cannot be used in practice.

Fig. 6a shows that most perturbation successfully generated the functional malware variants with over 90% except the addition of sections (*SAD*) and the changes in the field values in the data directory (*DD*). On the other hand, Fig. 6b shows that *SAD* perturbation is also the one that reduced the detection rate the most (i.e., 11%

[6] AIMED was omitted as it created many non-functional variants, as shown in Section 4.1, and performed consistently worse than FUMVar as shown in Section 4.4.



(a) SR



(b) FN rate

Figure 6: Effectiveness of individual perturbations.

after applying the perturbation once). However, as it has significantly less SR, it may not be an effective perturbation on its own. In contrast, other perturbations only reduced the detection rate by 3% or less. So although their SR is very high, their effectiveness in FN rate is not very significant. Therefore, we need to perform a sufficient number of perturbations to generate malware variants to bypass detectors with a high FN rate.

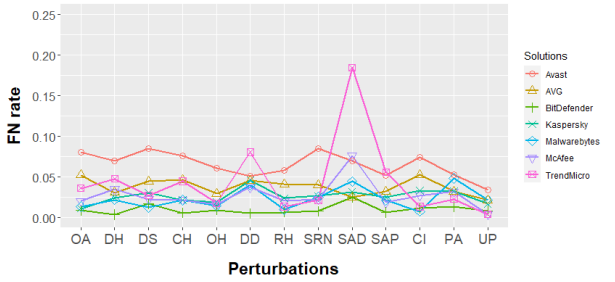


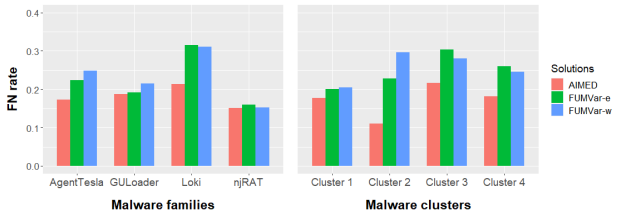
Figure 7: FN rates of the top 7 VirusTotal solutions with each perturbation.

We additionally measure the FN rate of an individual solution in VirusTotal for each perturbation. Fig. 7 shows the results of the top 7 most popular solutions presented in Section 4.2. The result shows that each malware detection solution is sensitive to different perturbations (i.e., better at detecting certain perturbations than others). For example, TrendMicro, BitDefender, McAfee, and Malwarebytes are more easily bypassed by adding a section (SAD); Avast is more easily bypassed by renaming the section name (SRN); and AVG is more easily bypassed by adding random byte sequence (CI) to code cave in a section. Hence, we can identify which types of perturbations are effective against different anti-malware solutions. These results can be used to enhance their detection rate.

4.4 Analysis based on malware categories

Here, we analyze how well FUMVar generates malware variants per malware family and category. We categorized the collected malware samples using the malware family information provided by the malware sharing site [11]. The collected samples are divided into four families: (M1) AgentTesla, (M2) GULoader, (M3) Loki, and (M4) njRAT. The FN rate results for malware families are shown in Fig. 8a. The result shows that FUMVar outperformed AIMED in all malware families in terms of FN rate, while the performance results between FUMVar strategies are mixed; FUMVar-weighted is more effective for AgentTesla and GULoader, but FUMVar-equal is more effective for Loki and njRAT. We surmise that this is because the weights used in FUMVar-weighted are not customized for each malware family. Therefore, it seems essential to optimize the weights of perturbations for FUMVar-weighted to improve the FN rate for a specific malware family (i.e., using adaptive weights). In addition, some malware families will be easier for malware writers to use and apply perturbations to bypass existing detectors. Hence, we can conduct further analysis for various malware families to identify and evaluate how and why some malware families can easily bypass existing anti-malware solutions using selected perturbations.

We also categorized malware samples based on the similarity of the API call sequences. For the clustering process, we extract the API call sequence from the Cuckoo sandbox report for each malware sample. Then, we measured the similarities between malware samples' API call sequences using ssdeep. Finally, we applied a hierarchical clustering algorithm *single linkage algorithm* [7] to cluster malware samples into categories until four[7] groups are left. The main characteristics of four malware clusters created are: (C1) Cluster 1 consists of malware samples with API calls related to processes, threads, and injections; (C2) Cluster 2 consists of malware samples with API, calls related to a registry key; (C3) Cluster 3 consists of malware samples with API calls related to network operations; and (C4) Cluster 4 consists of malware samples with API, calls related to searches and monitors.



(a) Malware families. (b) Malware clusters.

Figure 8: Comparing FN rate w.r.t. malware categories.

Fig. 8b shows that FUMVar outperformed AIMED in all clusters. For Cluster 1 and 2, FUMVar-weighted produced the best results, while FUMVar-equal produced the best results for Cluster 3 and 4. When FUMVar-equal was better, the performance gap with FUMVar-weighted was at most 5%, while FUMVar-weighted performed up to 13% better than FUMVar-equal. Therefore, our recommendation would be to use FUMVar-weighted. Another key observation is that FUMVar was more effective on malware in Cluster 3. In contrast, FUMVar was the least effect on malware in Cluster 1. These results show that FUMVar or similar perturbation techniques could more effectively generate (undetectable) malware variants with network operations.

[7]We matched the number of clusters with the number of families as before, but a different number of clusters can be used.

5 DISCUSSION

Bypassing commercial anti-malware solutions: Our experimental results (as shown in Fig. 5) demonstrate the effectiveness of FUMVar against commercial anti-malware solutions. Since most people only use a small number of anti-malware products in practice (mostly one), it would be easier to generate malware variants to bypass only those specific anti-malware products of the target. Given that many popular solutions are already included in our experiments (e.g., Avast, AVG, TrendMicro, BitDefender, McAfee, and Malwarebytes), we believe our experiments' coverage is reasonable with results reflecting practical impact. However, there are many experimental anti-malware solutions available today that are not included in VirusTotal, which we did not test FUMVar against. In future work, we will consider more anti-malware solutions to generalize our observations.

Validating malware variants and their behaviors: As described in Section 3.4, we used a sequence of API calls and signature fields to validate the behavior of malware variants generated. FUMVar uses the thresholds to check whether generated malware variants have the same behaviors as the original malware samples. Our analysis with 30 malware variants generated from 10 malware samples confirmed that no variant samples have different behaviors by manually analyzing their runtime behaviors. However, there may be other variants that do not meet this condition in real-world situations. To address this issue, we need to conduct a sensitivity analysis to determine the optimal thresholds in the *Validator* module, which we will consider in our future work to enhance FUMVar.

Another potential issue is that we may not capture malware samples' behavior with a snapshot of API call sequences and signature fields only. Although none of them were found in our manual inspection with 30 variants generated from 10 malware samples, there is a possibility to generate a malware variant with different behaviors, which are not detected in the current *Validator* module. To fully capture malware samples' behaviors, we believe that in-depth program behavior analyses would be required. However, such analyses place a significant overhead when applying perturbations. Therefore, more accurate and efficient program behavior methods should be investigated to enhance FUMVar.

Lastly, since we examined malware behaviors in the Cuckoo sandbox, some may not show their full malicious behavior by detecting its platform being a virtual machine [20]. Perhaps our current FUMVar implementation using the Cuckoo sandbox would fail to successfully generate such malware samples' variants. Therefore, we need to consider analyzing program behaviors in real computing machines instead of virtual machines.

Applying multiple perturbations: Most malware variants we generated have been applied with multiple perturbations that are randomly selected at each generation. However, there are some side effects when applying multiple perturbations in combinations. This case can happen due to the conflicting operations applied by multiple perturbations or previously applied operations. For example, it was impossible to pack or unpack the malware sample that was already packed or unpacked with the UPX tool. As a result, this particular perturbation did not apply any meaningful changes to the malware, which reduces the framework's effectiveness. For another case, the perturbation was not applied adequately because of the order of perturbation applied. It was impossible to apply UPX unpack after the other perturbations, such as section rename, were applied to a packed malware sample. Therefore, it is evident that perturbations and their dependencies should also be considered when applying them in combinations. However, our experimental results show that a malware variant generation framework, such as FUMVar, can produce a large number of variants efficiently even though applying multiple perturbations can sometimes lead to malware variants having incorrect behaviors.

Perturbation weight assignment: In this paper, we used Equation (4) to assign weights to different perturbations. To allocate the weight, we constructed FUMVar-weighted with different probabilities for perturbations based on our empirical studies found in Section 4.3. Although FUMVar-weighted performed the best overall as shown in the results, there were cases where the baseline approach, FUMVar-equal, performed better (e.g., as shown in Fig. 8 for malware family Loki and malware cluster 4). Hence, FUMVar-weighted can be improved further by optimally assigning the weights to perturbations. Also, according to the results in Section 4.4, we can dynamically adjust perturbation probabilities depending on malware categories (e.g., families, clusters, etc.). Those techniques will be investigated further in our future work.

6 RELATED WORK

Malware Detection: There are three representative approaches for detecting malware: signature-based, behavior-based, and heuristic-based [22]. Signature-based detection approaches extract some unique patterns such as string or hash values from previously collected malware samples and use them to detect malware. Behavior-based detection approaches analyze the differences in system behaviors between benign applications and malware. For example, the Cuckoo sandbox is an open-source tool to examine malware behaviors. However, checking the behavior may take a long time (e.g., up to a few minutes for each malware sample) to check various features such as signatures of files, API calls, network traffic, etc. Heuristic-based approaches detect malware by evaluating operational codes or file bytes. In addition to a signature check, it also inspects the sample on its structure and operational codes to detect and malicious behaviors [16]. To improve the heuristic-based approach, we can use data mining and machine learning (ML) techniques. For example, Yanfang *et al.* [23] proposed a PE-malware detection system using ML to conduct behavior analysis of Windows APIs call as a feature. Zhixing *et al.* [21] also proposed ML-based malware detection, analyzing patterns of virtual memory usage. However, malware writers are still developing malware variants by manipulating the values of features used for machine learning. Further, currently available solutions and tools take time for these new techniques to be adopted, leaving the gap for malware writers to develop new malware variants in advance.

ML for Malware Variant Generation: ML techniques have been a popular method to generate malware variants to automate and speed up the process. Anderson *et al.* [1] proposed a framework based on reinforcement learning (RL) to bypass static malware classifiers using the OpenAI gym model. However, they did not consider the functionality of malware variants after applying the

perturbations. Castro *et al.* [4] proposed the ARMED framework, which can automatically generate malware variants, applying random injection of perturbations per round. However, it has some performance issues for generating malware variants because it injects random perturbations. Furthermore, it just checks whether the PE file format is broken or not and does not check the functionality of malware variants. Hu *et al.* [8] introduced malGAN for generating the PE malware variants to bypass a target malware classifier using an adversarial model. However, there is a challenging issue for the training model, in which the variable can continuously be changed over time. Although many variant generation techniques are proposed, there are still gaps that can be exploited to generate evasive malware variants.

GA for Malware Variant Generation: GA is one of the metaheuristic search algorithms, which has been widely studied to generate malware variants. For example, Choi *et al.* [5] introduced the AVMG framework, which can generate malware variants from the malware code written in various languages. However, it only experimented with a small section, such as malware format or family, of malware samples. Cani *et al.* [2] inserted a set of instructions to modify the malware sample using GA while maintaining the original functionalities. Also, Castro *et al.* [3] proposed the AIMED framework, extended from ARMED, which applied GA for more effective malware variants generation process by up to 50% faster. They used several lists of perturbations, defined in [1], as the components of the population and implement the GA operation for more efficient malware variants generation. Also, Demetrio *et al.* [6] proposed an optimized malware manipulation method using the same perturbations with AIMED using GA. However, they did not check the full functionality (i.e., including behavior) of malware variants.

7 CONCLUSION

To improve the detection performance for new and unseen malware variants, we must first understand how those variants are generated against modern malware detectors. In this paper, we propose an evolutionary-based malware variant generation framework named `FUMVar`, which can generate evasive malware variants to bypass modern anti-malware solutions. Our experimental results showed that `FUMVar` compared with *AIMED*, a state-of-the-art malware variant generator using GA, achieved 23% faster generation time and 25% better evasiveness. Consequently, `FUMVar` generated malware variants that bypassed up to 73% on selected commercial anti-malware solutions. We then evaluated individual perturbations to better understand their evasiveness with respect to malware families and categories. The results showed that not all malware families and categories improved their evasiveness using the same perturbation, meaning their evasiveness can be customized against selected target malware detectors. We also identified most evasive perturbation (e.g., addition of sections), which can help enhance current commercial anti-malware solutions. Consequently, we believe that `FUMVar` can be used to understand the evasiveness of perturbations further to enhance existing and future anti-malware solutions.

As a part of future work, we plan to develop more effective perturbation techniques. For example, in the current `FUMVar` implementation, we just added sections consisting of random bytes. Instead, we will try to add sections that are actually used in benign programs so that `FUMVar` can generate malware variants that look like benign programs.

ACKNOWLEDGE

This work was supported by the MSIT, Korea (IITP-2020-2015-0-00403 and No.2018-0-00532), and the Australia-Korea Foundation of the Department of Foreign Affairs and Trade on the project "Intelligent Cybersecurity: Jumpstarting Collaboration on Smarter Security for Dynamic Networks." Note that Hyoungshick Kim and Jin B. Hong are the corresponding authors.

REFERENCES

[1] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. 2018. Learning to evade static PE machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917* (2018).
[2] Andrea Cani, Marco Gaudesi, Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. 2014. Towards Automated Malware Creation: Code Generation and Code Integration. In *Proc. of the 29th Annual ACM Symposium on Applied Computing (SAC 2014)*. 157–160.
[3] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo. 2019. AIMED: Evolving Malware with Genetic Programming to Evade Detection. In *Proc. of 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom 2019)*.
[4] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo Rodosek. 2019. ARMED: How Automatic Malware Modifications Can Evade Static Detection?. In *Proc. of 5th IEEE International Conference on Information Management (ICIM 2019)*.
[5] Jusop Choi, Dongsoon Shin, Hyoungshick Kim, Jason Seotis, and Jin B. Hong. 2019. AMVG: Adaptive Malware Variant Generation Framework Using Machine Learning. In *Proc. of 24th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2019)*.
[6] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. 2020. Efficient Black-box Optimization of Adversarial Windows Malware with Constrained Manipulations. *arXiv preprint arXiv:2003.13526* (2020).
[7] John C Gower and Gavin JS Ross. 1969. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 18, 1 (1969), 54–64.
[8] Weiwei Hu and Ying Tan. 2017. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983* (2017).
[9] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.
[10] Jesse Kornblum. 2006. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation* 3 (2006), 91–97.
[11] MalwareBazaar. [n. d.]. MalwareBazaar. https://bazaar.abuse.ch/ access date: 26 June 2020.
[12] Daniel Nieuwenhuizen. 2017. A behavioural-based approach to ransomware detection.
[13] Matt Pietrek. 1994. Peering inside the PE: a tour of the win32 (R) portable executable file format. *Microsoft Systems Journal-US Edition* 9, 3 (1994), 15–38.
[14] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. 2012. Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security* 12, 8 (2012), 74–83.
[15] Mark Russinovich. 2018. Microsoft sysinternals: Process monitor v3. 60.
[16] Ian Shiel and Stephen O'Shaughnessy. 2019. Improving file-level fuzzy hashes for malware variant classification. *Digital Investigation* 28 (2019), S88–S94.
[17] Romain Thomas. 2017. LIEF: Library to Instrument Executable Formats.
[18] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. 2019. Survey of machine learning techniques for malware analysis. *Computers & Security* 81 (2019).
[19] Jason Upchurch and Xiaobo Zhou. 2015. Variant: a malware similarity testing framework. In *Proc. of 10th IEEE International Conference on Malicious and Unwanted Software (MALWARE 2015)*.
[20] Timothy Vidas and Nicolas Christin. 2014. Evading Android Runtime Analysis via Sandbox Detection. In *Proc. of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2014)*.
[21] Zhixing Xu, Sayak Ray, Pramod Subramanyan, and Sharad Malik. 2017. Malware detection using machine learning based analysis of virtual memory access patterns. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE 2017)*.
[22] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. 2017. A Survey on Malware Detection Using Data Mining Techniques. *Comput. Surveys* 50, 3, Article 41 (2017).
[23] Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye, and Qingshan Jiang. 2008. An intelligent PE-malware detection system based on association mining. *Journal in computer virology* 4, 4 (2008), 323–334.