IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 14 February 2023, accepted 17 March 2023, date of publication 27 March 2023, date of current version 30 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3262265

RESEARCH ARTICLE

On the Effectiveness of Perturbations in Generating Evasive Malware Variants

BEOMJIN JIN[1], JUSOP CHOI[2], JIN B. HONG[3], (Member, IEEE), AND HYOUNGSHICK KIM[1]

[1]Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea
[2]Digital Transformation Response Support Department, Financial Security Institute, Yongin 16881, South Korea
[3]Department of Computer Science and Software Engineering, The University of Western Australia, Perth, WA 6009, Australia

Corresponding authors: Jin B. Hong (jin.hong@uwa.edu.au) and Hyoungshick Kim (hyoung@skku.edu)

This work was supported in part by the Korea Internet and Security Agency (KISA) under Grant 1781000003; and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) under Grant 2022-0-00495, and Grant 2019-0-01343. This work was also supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00532, Development of High-Assurance (>=EAL6) Secure Microkernel).

ABSTRACT Malware variants are generated using various evasion techniques to bypass malware detectors, so it is important to understand what properties make them evade malware detection techniques. To do this, a framework is proposed to effectively generate fully-working, unseen malware samples on Windows portable executable (PE) files with various perturbations such as code obfuscation and benign section addition. Using this framework, we were able to bypass various commercial anti-malware solutions (e.g., BitDefender, AVG, Kaspersky, and Avast) using the generated malware variants, with up to 86% more evasiveness than the original malware samples, and up to 28% more evasive compared with our previously proposed solution `FUMVar`. Our results are useful in terms of improving malware detection techniques, by analyzing different perturbations and their effectiveness, which leads to a better understanding of how malware variants could be generated that are more evasive and which malware categories they belong to. We found that the most effective perturbation is the code obfuscation using XOR – the malware variants generated by the code obfuscation can evade the detection of 28 anti-malware engines on average. Therefore, our experimental results and observations would be useful to develop anti-malware solutions that would be effective in detecting malware variants that have not been seen previously.

INDEX TERMS Malware detection, malware mitigation, malware analysis, malware generation, metamorphic malware, genetic algorithm.

I. INTRODUCTION

Various anti-malware techniques are available to prevent malware infections [1], [2]. However, the majority of these techniques lack the capabilities to detect new and unseen malware samples. Consequently, malware writers generate new malware variants from existing malware samples through polymorphic and metamorphic mutations for improved evasiveness [3]. For example, a *Cerber*'s variant was generated every 15 seconds [4], making it difficult to detect all of its variants in practice. Therefore, it is important to understand how evasive malware variants are generated and what makes them evasive first so that we can enhance malware detectors and their capabilities to detect malware variants generated using various malware evasion techniques.

Several studies have presented various techniques to generate malware variants [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. For example, reinforcement learning was used to generate malware samples against anti-malware products in [6], [13], and [14], as well as a genetic algorithm-based framework named *AIMED* [8] that randomly applies perturbations to generate malware samples. However, most existing techniques do not ensure that the generated variants' behaviors are identical to the original malware sample's behaviors. Jin et al. [11] proposed the framework called `FUMVar` using a genetic algorithm (GA) with a fitness score function that uses a publicly available malware detection

The associate editor coordinating the review of this manuscript and approving it for publication was Jiafeng Xie.

system named VirusTotal[1] to generate more evasive malware variants against commercial malware detectors. In contrast to previous works, the generated malware variants' functional behaviors are kept identical to the original malware's, achieved by analyzing them with the Cuckoo sandbox as ensured by FUMVar. However, the number of possible perturbations was limited, and the malware behavior validation step was inefficient, hindering its practical use.

In this paper, we extend and improve FUMVar [11] into a more effective and practical one dubbed FUMVar-Ex with (1) new perturbations reflecting the techniques used to generate real-world malware samples (2) a new malware behavior validation method using aggregated evaluation that improves the validation of generated malware variants' functionality and their equivalence to the original malware samples, and (3) more comprehensive experiments and improved results using FUMVar-Ex compared with FUMVar and other state-of-the-art techniques. For the first extension, we introduce new perturbation techniques modifying portable executable (PE) sections directly, whereas, in FUMVar, we only implemented the perturbation methods that only modified a few bytes of the header field, which did not reflect more complex malware perturbation methods used in practice.

For the second extension, we provide a more robust and generalized malware behavior validation method considering the *dynamically* changing runtime behaviors of programs (i.e., malware samples), which was not considered in FUMVar, and as a result, it did not fully guarantee the equivalent functionalities for dynamic malware samples. Finally, for the third extension, we show a significantly improved malware bypass rate against anti-malware solutions using FUMVar-Ex, by up to 28% more evasive against commercial anti-malware solutions compared with FUMVar. Hence, by considering the dynamically changing runtime behaviors, we are able to capture and validate dynamic malware samples more effectively, and also improve the evasiveness of malware when combined with more advanced perturbation techniques implemented in this paper.

The effectiveness of FUMVar-Ex has been compared to four state-of-the-art malware variant generation frameworks: FUMVar [11], AIMED [8], RL [6], and MAB-MALWARE [13]. The experimental results demonstrate that FUMVar-Ex outperforms the other frameworks, achieving a remarkable improvement in evasiveness of up to 19% compared to the second most effective framework, FUMVar [11]. Further, the detection rates against commercial anti-malware products (e.g., BitDefender, AVG, Kaspersky, and Avast) were evaluated using the malware variants generated by FUMVar-Ex, demonstrating that the detection rates for the variants significantly decreased by up to 86% compared with their original samples. These results highlight some key shortcomings of existing solutions in detecting malware variants in practice. Further, we evaluate different malware categories and perturbations to generate evasive malware variants. Those experimental results could be used to enhance the performance of malware detection techniques with better evasive malware samples. The key contributions of this paper can be summarized as follows.

- We present a framework dubbed FUMVar-Ex, a significant extension from the FUMVar framework [11], that efficiently generates fully working malware variants with improved evasion techniques to bypass malware detectors. The link to our source code can be found from https://github.com/FUMVar/FUMVar-Ex;
- We introduce new and complex perturbation methods, which mimic more practical perturbations used in real-world that improves the evasiveness of malware variants significantly;
- We propose an aggregated evaluation method considering the dynamically changing runtime behaviors to improve the validity of checking the equivalent behaviors between the generated malware variants and their original malware samples;
- We compare the performance of FUMVar-Ex against the state-of-the-art malware variant generation frameworks FUMVar, AIMED, RL, and MAB-MALWARE;
- We select seven commercial anti-malware solutions and compare their robustness against the malware variants generated by FUMVar-Ex, achieving an improved evasiveness of up to 86% than the original malware samples; and
- We analyze the effectiveness of various perturbation methods and malware types and examine evasive malware variants' characteristics, which can be used to enhance anti-malware solutions.

The rest of the paper is organized as follows. Section II provides the background information about the Windows PE format and the Cuckoo sandbox. Section III describes the overview of FUMVar-Ex, and the perturbation techniques are described in Section IV. In Section V, we present the experimental results. Section VI discusses our findings and limitations. Section VII presents related work on malware variant generation. Finally, Section VIII concludes this paper.

II. BACKGROUND

A. WINDOWS PORTABLE EXECUTION (PE)

The PE format contains the information for Windows OS loader [16] to handle the executable code. The official website describes the detailed specification (https://docs.microsoft.com/en-us/windows/win32/debug/pe-format). Fig. 1 illustrates the PE format structure at a high level, which contains the following sections: 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 *DOS header* and the *DOS stub* are needed to support backward compatibility with DOS. The *COFF file header* represents the type of machine for which the file is intended

[1] You can find more details about VirusTotal from (https://www.virustotal.com/)

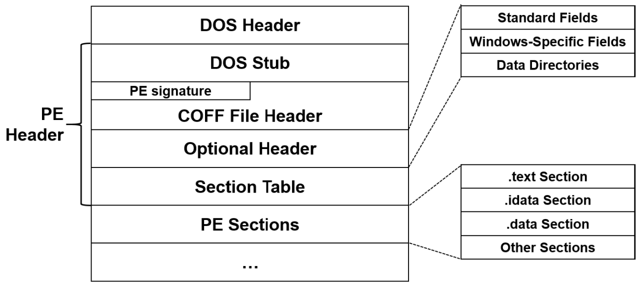


FIGURE 1. PE format structure.

to execute, the type of the file (e.g., execution or object files, or shared library), and the number of sections. The COFF file header additionally provides optional information such as timestamps and the number of symbols. As a result, these fields may be modified to maintain compatibility with the PE file format in the *COFF file header*.

The *optional header* defines instructional data to load executable to the memory, which has three parts: *standard fields*, *Windows-specific fields*, and *data directories*. The *standard fields* cannot be edited as it is required for all COFF files. However, *Windows-specific fields* have 21 fields for the linker and loader information. Out of the 21, we found that 15 of them can be modified without violating the validity of the PE file format. Similarly, the *data directories* part represents the address and size of the tables and strings that Windows OS uses, which were all modifiable without breaking the PE file.

The *section table*, also often called a *section header*, specifies the section body information. The *section table* has 10 fields that specify program activities. Here, we select two fields to apply perturbations as examples, which are virtual and raw data sizes. The program uses other fields, such as a virtual address, which cannot be modified. There are still more fields that can be modified, such as the *rich header* field [17], [18], [19].

B. CUCKOO SANDBOX

Cuckoo sandbox [20] is an open-source malware analysis tool that provides a detailed behavior report of an execution file, including Windows PE format. It automatically executes a suspicious execution file inside an isolated environment and monitors its behaviors. It can trace API calls and dump all traffic using a virtual switch, which forwards the traffic of the isolated environment to the host. The Cuckoo sandbox uses signatures to extract suspicious behaviors from low-level information, such as API calls and captured traffic. A signature is a piece of information that corresponds to low-level API calls or traffic to high-level information. It includes a high-level description, severity score, and patterns of low-level information. For example, it identifies suspicious behavior, such as unidentified thread injection, if the API calls include `CreateRemoteThread` or `CreateRemoteThreadEx`. To identify suspicious network traffic, it uses a blocklist, including malicious IP addresses and URLs.

TABLE 1. Field information of Cuckoo sandbox report.

Types	Field name	Information
Static	Target	File hash values, file size
	Static	File import table, sections, entropy
	Strings	File strings
Dynamic	Behavior	API calls, process commands
	Dropped	Files dropped
	Network	TLS, TCP, UDP traffic
	Signatures	Suspicious behaviors

Cuckoo sandbox provides a detailed behavior report of a suspicious execution file. Cuckoo sandbox report has various fields to express the analysis of a suspicious file, and they can be divided into two main types of fields: static and dynamic. Table 1 shows the list of fields in a Cuckoo sandbox report and the information they contain. Static type fields include file hash, strings, and PE file format. Dynamic type fields include the behavior of a file, such as API calls, dropped files, and network traffic during the file's runtime. *Signatures* is one of the dynamic type fields that includes the list of suspicious behaviors identified by signatures. We used the *signatures* field to validate malware variants' behaviors because they behave the same as the original. Each suspicious behavior represented in the *signatures* field consists of various information, such as timestamp, name, and description. We excluded some irrelevant information, such as a timestamp or process ID, that does not contribute to the behavior of the malware sample.

III. FUMVar-Ex: AN OVERVIEW

This section presents a high-level overview of FUMVar-Ex consisting of five modules (see Fig. 2). The *Parser* module ensures the validity of the PE file given to the framework. The *Modifier* module will select perturbation(s) and apply them to mutate the input PE file. The *Verifier* module checks the validity/operational status of the output PE file from the Modifier module (which can be corrupted after being modified). The *Validator* module checks the behavior of the modified malware sample against the original one to ensure their identical behaviors. Lastly, the *Scorer* module computes the evasiveness score of the generated variants against malware detectors. We explain each module's functionalities in detail in the following subsections.

A. PARSER - EXTRACTING SECTIONS IN PE FILES

A malware sample in PE format is inputted to the *Parser* module. The input sample is then examined to identify sections and fields (as described in section II) to be perturbed by the Modifier module (as described in Section III-B). So the key tasks in this module are identifying those mutable sections and tagging the fields.

B. MODIFIER - APPLYING MUTATIONS TO PE FILES

The *Modifier* module applies perturbations in an iterative manner to modify a given malware sample with the perturbations selected using a GA.

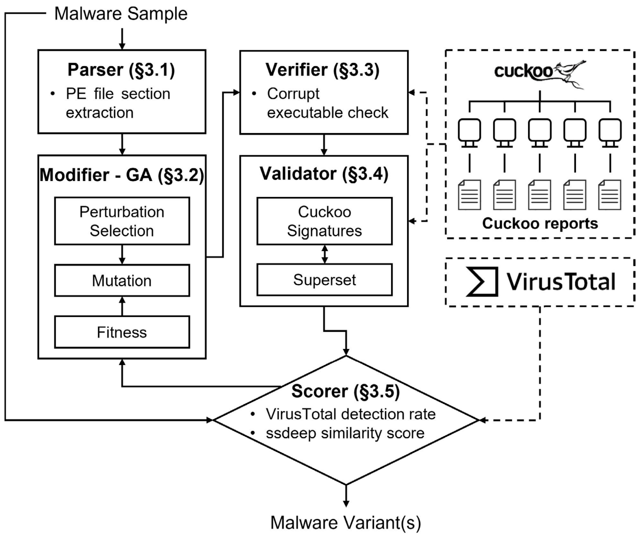


FIGURE 2. Overview of `FUMVar-Ex`**.**

1) PERTURBATIONS ON PE FILES

`FUMVar-Ex` uses 14 perturbation techniques (see Section IV for details). The *Modifier* module selects one or more perturbation techniques based on their effectiveness (as described in Section V-C) and applies them to the sample during the GA process. We used LIEF [21], a popularly used tool to parse and modify several program execution files (including the PE format), to implement perturbations on PE files.

2) GENETIC ALGORITHM

`FUMVar-Ex` uses a GA to "evolve" malware variants with a randomly selected perturbation at every mutation. The GA receives the feedback from the *Scorer* module (see Section III-E), which is used as the fitness value in order to guide the malware variant generations to decrease the detection rate, as well as to minimize the similarity to the original malware sample. As the malware goes through the GA multiple times, many random perturbations are applied, leading to enhanced evasiveness while reducing the similarity with the original input.

In the first step, the original input (i.e., the malware sample) is copied to initialize the population. After an iteration, the top n fittest members are selected from the population. All the malware variants generated from the GA are sent to the *Verifier* module to filter out malfunctioning ones prior to being sent to the *Scorer* module. The next generation is selected based on the scores received from the *Scorer* module.

Algorithm 1 shows the process in detail. The population is initialized in the outer for-loop. Note that the loop is terminated with a predetermined generation number. Here, PL represents the whole population as a set; PL_s is a subset populated with top n_s malware variants based on the fitness scores measured; PL_r is a subset populated with n_t randomly selected malware variants. Algorithm 1 uses the Selection and Mutate functions. The definitions of those functions are as follows:

Algorithm 1 Generating Malware Variants Using 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
        $perturbation \leftarrow PS(perturbations)$
        $PL.append(Mutate(b, perturbation))$
    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
        $PL_s \leftarrow Selection(PL, Fitness(PL), n_s)$
        $PL_r \leftarrow Selection(PL - PL_s, Random(PL - PL_s), n_t)$
        for $pl$ in $PL_s + PL_r$ do
            $PL.append(Mutate(pl, PS(perturbations)))$
        end for
        $PL \leftarrow Selection(PL, Fitness(PL), p_{size})$
    end for
    return $sol$
end function
```

- Selection(p, c, n): This function selects n elements from population p based on the criteria c.
- Mutate(b, p): This function applies the perturbation p to the byte code b.

C. VERIFIER - CHECKING CORRUPTED VARIANTS

Applying perturbations can result in the malware being corrupted, which should be avoided for generations. To check the corruption of the generated malware via perturbations, we inspect the debug information produced by the Cuckoo sandbox. If the PE file is corrupted, the Cuckoo sandbox shows the following error message: "`CuckooPackageError: Unable to execute the initial process, analysis aborted.`" Therefore, we run the Cuckoo sandbox with a generated malware instance and check whether the above error message was displayed.

D. VALIDATOR - CHECKING MALWARE BEHAVIORS

To generate effective malware variants, we must ensure that the generated malware variants' behaviors are identical to the original malware's. We use the Cuckoo sandbox to examine malware samples' functional behaviors. The

Cuckoo sandbox produces an analysis report including variants' behavior information, such as API calls and network activities (Section II-B). Among them, the *signatures* fields can be used to validate malware variants' behaviors.

One of the main problems when using the Cuckoo sandbox report is that the report outputs are inconsistent even for the same program file executed due to its dynamic properties. Consequently, the Cuckoo sandbox reports would contain different signatures whenever a program file is executed. To overcome this issue, we propose a method to obtain a superset of signatures through t multiple independent runs of a malware sample in the Cuckoo sandbox. That is, we repeatedly run the Cuckoo sandbox t times with the same malware sample and aggregate the report outputs until no new signature is produced from the Cuckoo sandbox report. Of course, some signatures may only appear under certain settings/environments. However, we empirically found that five reports (i.e., $t = 5$) are sufficient for most malware samples to reach an experimental saturation for signatures on the Cuckoo sandbox reports. In other words, the supersets created using $t = 5$ and $t = 5 + \alpha$ for any positive integer α resulted in over 99% of malware samples having the same superset.

Algorithm 2 Malware Signature Generation

```
$M$: Input malware
$R$: Cuckoo sandbox report
$t$: the number of iterations to generate a superset
$S \leftarrow []$: signatures
function Generate_Signatures($M$)
    for $i \leftarrow 1$ to $t$ do
        $R \leftarrow Cuckoo(M)$
        $CS \leftarrow Cuckoo\_signatures(R)$
        $S \leftarrow S \cup CS$
    end for
    return $S$
end function
```

Algorithm 2 shows how to obtain aggregated signatures for a given target malware. We first initialize S, which is the superset of signatures, and analyze a malware sample, using the Cuckoo sandbox, to extract the *signatures* for each iteration. Next, we determined the validity of a malware variant if the aggregated signatures from five Cuckoo sandbox reports were identical to the generated malware signature. We empirically found that five cuckoo sandbox reports sufficiently cover most malware variants' signatures.

E. SCORER - MEASURING MALWARE VARIANTS

The scorer measures the evasiveness of malware variants generated. We can use VirusTotal https://www.virustotal.com/), which contains 71 anti-malware solutions. The output from VirusTotal is then used as the evasiveness score (i.e., the number of anti-malware solutions bypassed). In addition, we use the code similarity score between the original malware sample and the generated variant (computed using *ssdeep* [22]) as part of the score. In practice, code similarity is often used (e.g., for signature-based solutions) to detect malware samples [23], so it is useful to measure the similarities between the original and the variant malware samples as well. Finally, the fitness value is the sum of the evasiveness score from VirusTotal and the similarity score from ssdeep, as shown in Equation (1).

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 proportion of the two scores the fitness score will rely on ($0 \leq w \leq 1$).

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

Table 2 provides the definitions of the notations used for Equation (1), as well as for notations used in other equations. VirusTotal detection rate ($VT(x)$) represents the number of malware detection products that detected the generated malware variant divided by the total number of malware detection products in VirusTotal. Finally, the similarity score is calculated using ssdeep ($ssdeep(x, y)$) using the byte code of a generated variant and its original samples because it has been popularly used for malware detection [24].

IV. PERTURBATIONS

FUMVar-Ex uses 14 perturbations that can be categorized into two types to generate malware variants: (i) behavioral changes (BC) and (ii) non-behavioral changes (NBC). The BC-type perturbations can insert/modify/remove some parts that can affect the program's functionalities. Inserting some random byte code into a section containing program operation code directly related to the program's behavior is an example of BC-type perturbation. On the other hand, NBC-type perturbations do not change the existing functionalities because those perturbations modify parts that are not related to the program's behaviors (i.e., logic is maintained). Most NBC-type perturbations change values in header fields unrelated to the program's behavior. Changing *stub value* in DOS stub and *timestamp* field in COFF header are examples of NBC-type perturbations, shown in Fig. 3. Table 3 shows the list of all perturbations used in FUMVar-Ex, and the detailed information for each type of perturbations will be explained in Section IV-A and IV-B.

A. NBC-TYPE PERTURBATIONS

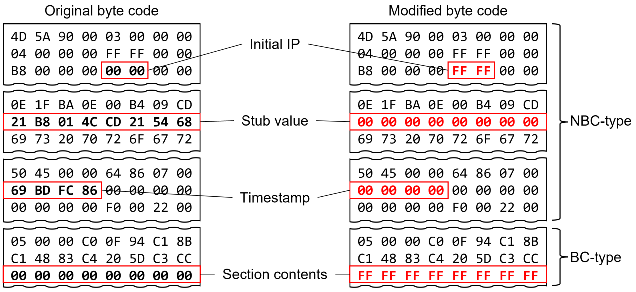
The NBC-type perturbations are specified as follows:

1) OVERLAY APPEND (OA)

This perturbation appends a random length of zeros to the end of the PE file, which does not break the PE file format

TABLE 3. List of perturbations used in `FUMVar-Ex`.

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



```
4D 5A 90 00 03 00 00 00        4D 5A 90 00 03 00 00 00
04 00 00 00 FF FF 00 00        04 00 00 00 FF FF 00 00
B8 00 00 00 00 00 00 00        B8 00 00 00 FF FF 00 00

0E 1F BA 0E 00 B4 09 CD        0E 1F BA 0E 00 B4 09 CD
21 B8 01 4C CD 21 54 68        00 00 00 00 00 00 00 00
69 73 20 70 72 6F 67 72        69 73 20 70 72 6F 67 72

50 45 00 00 64 86 07 00        50 45 00 00 64 86 07 00
69 BD FC 86 00 00 00 00        00 00 00 00 00 00 00 00
00 00 00 00 F0 00 22 00        00 00 00 00 F0 00 22 00

05 00 00 C0 0F 94 C1 8B        05 00 00 C0 0F 94 C1 8B
C1 48 83 C4 20 5D C3 CC        C1 48 83 C4 20 5D C3 CC
00 00 00 00 00 00 00 00        FF FF FF FF FF FF FF FF
```

FIGURE 3. Examples of perturbations.

because the program pointer never points after the end of the file.

2) DOS HEADER (DH)

This perturbation changes field values in the DOS header. For example, it changes a field named initial IP to random bytes within its length, as shown in Fig. 3.

3) DOS STUB (DS)

This perturbation selects a random byte sequence within the DOS stub and replaces it with a randomly generated byte sequence, as illustrated in Fig. 3. This process does not violate the PE file format, as the DOS stub is solely used to display an error message indicating that the executable is incompatible with DOS. Furthermore, it is important to note that this message can also be modified.

4) COFF HEADER (CH)

This perturbation changes field values in the COFF header, such as *timestamp* field shown Fig. 3.

5) OPTIONAL HEADER (OH)

This perturbation changes field values in the Optional header, which does not affect the PE file format, including the fields named *checksum*, *linker version*, and *operating system version*.

6) DATA DIRECTORY (DD)

This perturbation randomly chooses a data in the data directory list and changes the values in the data, such as *relative virtual address (rva)* and *size*.

7) RICH HEADER (RH)

This perturbation inserts new content information to the Rich header. The content information includes *id*, *build id*, and *count*.

B. BC-TYPE PERTURBATIONS

The BC-type perturbations are specified as follows:

1) SECTION RENAME (SRN)

This perturbation changes the name of a randomly selected section in PE sections to a random string. For example, it changes the section named *.data* to *random*.

2) SECTION ADD (SAD)

This is one of our new perturbation techniques to extract some sections from benign programs, which are then added to a malware variant sample in order to camouflage a target malware sample's (statistical) characteristics with added binary code (i.e., making it indistinguishable from benign programs). To add sections from benign programs, we collected 45,328 sections from 8,851 benign samples in the Windows System32 folder.

3) SECTION APPEND (SAP)

This perturbation appends some random byte sequence to the end of a section for the case the length of the content is smaller than the allocated size. The BC-type perturbation in Fig. 3 shows an example of appending a random byte sequence to the section.

4) CODE CAVE INJECT (CI)

This perturbation injects some random byte sequence into the code cave, which is unused byte sequences in a PE file. It usually exists between each section and header.

5) PACK (PA)

This perturbation packs the PE file with the UPX tool. We randomly selected the options for executing the UPX tool, such as *compression level*.

6) UNPACK (UP)

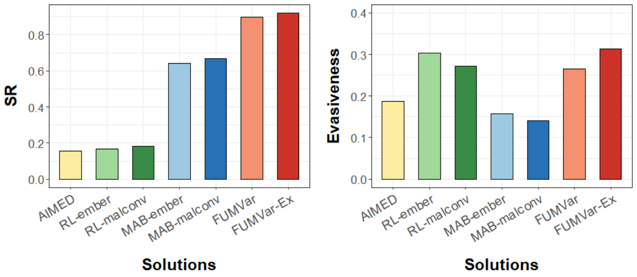
This perturbation unpacks the PE file with the UPX tool.

7) XOR OBFUSCATION (XO)

This perturbation is a popular technique for generating real-world malware samples. It simply encrypts some binary code using XOR operation with random key bytes, and the encrypted file then decrypts itself and executes during the runtime. The XOR obfuscation technique is simple yet effective in thwarting the analysis process. Algorithm 3 shows the process of XOR obfuscation. First, we encrypt malware bytes with a randomly generated key k and then decrypt the encrypted bytes using the same key. Next, we write the decrypted malware bytes to file f and execute the file.

Algorithm 3 XOR Obfuscation

```
$b$: Input malware bytes
$f$: Name of file written
$XOR_{enc}$: Encrypt bytes with a key
$XOR_{dec}$: Decrypt bytes with a key
function XOR_Obfuscation($b$)
    $k \leftarrow RandomKey()$                         ▷ $0 < k < 256$
    $b_{enc} \leftarrow XOR_{enc}(b,k)$
    $b_{dec} \leftarrow XOR_{dec}(b_{enc},k)$
    $WriteFile(f, b_{dec})$
    $Execute(f)$
end function
```



(a) SR (b) Evasiveness

FIGURE 4. SR and evasiveness results.

V. EXPERIMENTS

The aim of our experiments is to evaluate the effectiveness of `FUMVar-Ex`. In total, we used 112 representative malware samples from various malware repositories [25] that broadly cover various malware types, such as malware families, size, and API calls.

Using these samples, we first evaluate the performance of `FUMVar-Ex` against the state-of-the-art malware variant generators: `FUMVar` [11], `AIMED` [8], `RL` [6], and `MAB-MALWARE` [13] (see Section V-A). Next, we examine the detection performance of commercial anti-malware solutions, in particular: Avast, AVG, BitDefender, Kaspersky, Malwarebytes, McAfee, and TrendMicro (see Section V-B). We then look at perturbations listed in Section IV and evaluate their effectiveness in order to identify malware variants' evasiveness. Finally, we look at how different malware categories could affect generating malware variants (see Section V-D).

To evaluate, two metrics are used: (1) *valid variant generation success rate* (SR) and (2) *evasiveness*. SR represents the number of functionally correct malware variants (i.e., ones that were successful in checks from the *Verifier* and *Validator* modules) in proportion to the total number of malware variants, as shown in Equation (2). The evasiveness is defined as the proportion of anti-malware solutions bypassed by the generated variants to the original malware samples. The average of chosen anti-malware solutions is calculated in the case of multiple anti-malware solutions being used. The resulting equation is shown in Equation (3). The notations used in the equations are described in Table 2.

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

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

A. BASELINE COMPARISON

We used 112 malware samples with $g_{max} = 100$ (i.e., max 100 generations) for generating malware variants against the baseline of `FUMVar` and `AIMED`, which is based on GA. We applied one perturbation technique for each generation on `FUMVar-Ex`– different probability values are assigned to each perturbation based on the results obtained as shown in Section V-C. This is done by using the weight values of each perturbation and then by multiplying their associated SR and evasiveness 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 $Evasiveness(pt_i)$ are the SR and evasiveness values for pt_i, respectively). We also compared the performance against `RL` and `MAB-MALWARE`, which generate malware variants based on reinforcement learning using the malware samples. We used default settings (e.g., maximum iteration, reward, etc.) defined in their GitHub repositories for each framework. Since both frameworks are designed to evade machine learning detection models (Ember [26] and Malconv [27]), we tested them against each model (i.e., `RL-ember`, `RL-malconv`, `MAB-ember`, and `MAB-malconv`).

$$w_{pt_i} = \frac{SR(pt_i) \times Evasiveness(pt_i)}{|perturbation\_list|} \tag{4}$$

Fig. 4a shows that `FUMVar-Ex` achieved the best SR result, and both `FUMVar` and `FUMVar-Ex` generated malware variants with nearly 90% SR, whereas `AIMED`, *RL-ember*, and `RL-malconv` only achieved 20% or less SR. That is, `AIMED`, `RL-ember`, and `RL-malconv` produced incorrectly working malware variants eight times higher than `FUMVar` and `FUMVar-Ex`. In addition, `MAB-ember` and `MAB-malconv` achieved around 70% SR, which means they produced incorrectly working malware variants three times higher than `FUMVar` and `FUMVar-Ex`. These SR results demonstrate that it is integral for malware generation frameworks to check the validity of malware variants generated.

In addition, Fig. 4b shows that `FUMVar-Ex` achieved a better evasiveness rate of 32% compared with other solutions, demonstrating the superiority of `FUMVar-Ex`, which adopts the results of malware solutions in their fitness function and using more effective perturbation methods. `RL-ember` also achieved 30% evasiveness. However, the SR result is significantly lower than `FUMVar-Ex`. On the other hand, `AIMED` produced the worst results (i.e., easier to be detected), implying that using only the similarity scores to generate evasive malware variants would significantly be less evasive

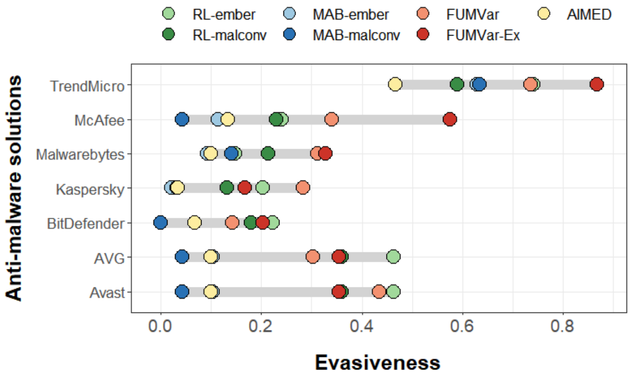


FIGURE 5. Bypassing anti-malware solutions using each framework.

than using anti-malware solutions in the fitness function. The performance gap will become larger also, as the similarity scores are static (i.e., these scores are not affected by the actual malware detection rates), whereas the anti-malware solutions are dynamic and tend to get better (i.e., the detection rate improves, also affecting the fitness function).

B. EVADING ANTI-MALWARE SOLUTIONS

To evaluate the evasiveness of generated malware variants, we choose seven commercial anti-malware solutions: Avast, AVG, BitDefender, Kaspersky, Malwarebytes, McAfee, and TrendMicro. We compare the evasiveness of FUMVar-Ex against FUMVar and AIMED.

The two main observations made from Fig. 5 are: (Obs. 1) regardless of the detection rate, all solutions can be bypassed using FUMVar-Ex, and (Obs. 2) FUMVar-Ex outperformed AIMED, MAB-ember, and MAB-malconv for all solutions tested and performed better than FUMVar for the five solutions (TrendMicro, BitDefender, AVG, Malwarebytes, and McAfee).

From (Obs. 1), both FUMVar and FUMVar-Ex were able to bypass the malware detection of all anti-malware products. If it was not for Kaspersky, the evasiveness achieved would be over 20% against all solutions using FUMVar-Ex (i.e., FUMVar-Ex can generate malware variants at a rate that 1 in 5 would bypass the detection solutions), and over 10% against for all anti-malware solutions tested. In contrast, almost all evasiveness results of AIMED, MAB-ember, and MAB-malconv were less than 15%. Unlike FUMVar, the best performing solution in detecting malware variants generated by FUMVar-Ex was Kaspersky in our evaluation with less than 20% evasiveness. However, the worst-performing solution against FUMVar-Ex was also TrendMicro, where we observed up to 86% evasiveness (i.e., almost 9 in 10 generated malware variants were able to bypass the detection by TrendMicro), followed by McAfee with 58% evasiveness. This also means that we can use FUMVar-Ex to carefully craft malware variants against targeted anti-malware solution(s).

From (Obs. 2), some new effective perturbations, such as XOR obfuscation, would be helpful to improve the overall evasiveness. In general, FUMVar-Ex outperformed FUMVar by up to 20% for McAfee, with an average of 4% improved evasiveness against all commercial anti-malware solutions.

There were cases that FUMVar-Ex performed worse, such as RL-malconv performing better than FUMVar-Ex against the four solutions (i.e., Kaspersky, BitDefender, AVG, and Avast). However, all solutions had significantly worse SR (i.e., the SR value of RL-malconv was only 19% compared to FUMVar-Ex with 90%), hindering their practicality over FUMVar-Ex.

C. PERTURBATIONS AND THEIR EFFECTIVENESS

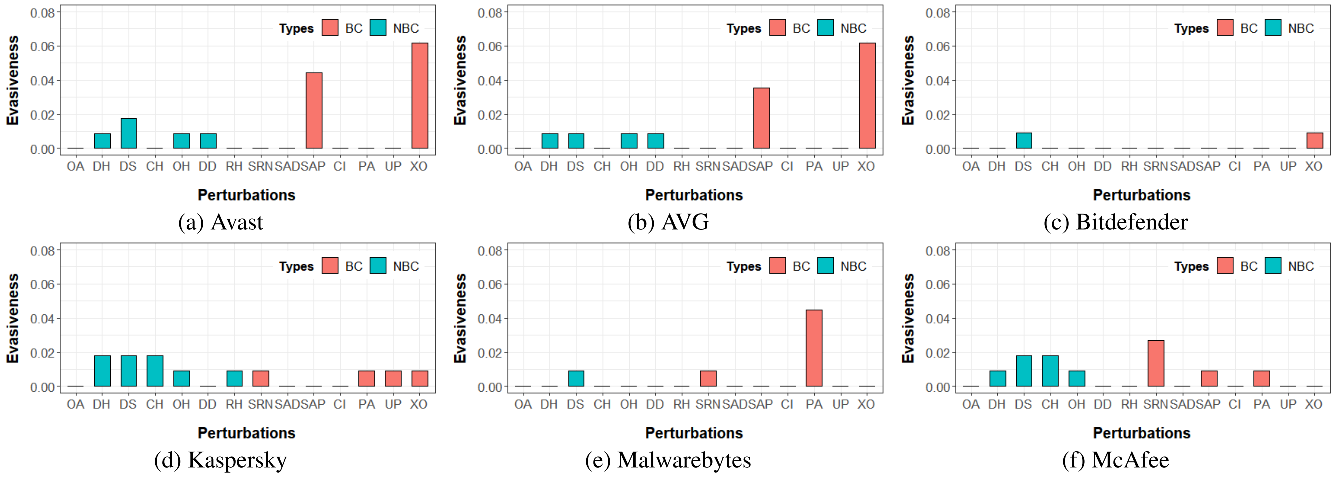
Section V-B showed our proposed framework FUMVar-Ex was effective against commercial anti-malware solutions. In order to develop more robust anti-malware solutions to combat such malware variant generation frameworks, we need to better understand what makes our variants more evasive. To examine this, we individually tested our used perturbations on all of our malware samples and calculated the variations in SR and evasiveness. The results are shown in Fig. 7, where the average rates across all malware samples are displayed. Please note that in this experiment, we discarded all non-functional malware variants and variants without the same behavior as the original malware sample, as those cannot be used in practice.

Fig. 7 shows the effectiveness of individual perturbations in SR and evasiveness metrics. Fig. 7 shows that most perturbation techniques would be effective in SR; SRs are over 75% except *SAD*, *DD*, and *XO*. In contrast, Fig. 7a shows that *XO* is the most effective perturbation technique in evasiveness; evasiveness is about 40% with just one perturbation. The second effective perturbation is *SAD*; evasiveness is about 25% with just one perturbation. Based on these findings, *XO* would be a better perturbation technique than *SAD*.

We measure an individual solution's evasiveness in VirusTotal for each perturbation. The results of the 6 most popular anti-malware solutions used (see Section V-B) is shown in Fig. 6. The result shows that the sensitivity of detection varies on the anti-malware solutions depending on the perturbations used (i.e., some perturbations are better detected than others). For example, Avast is easier to bypass by *SAP*; AVG is easier to bypass by packing a sample with *XO*, and Malwarebytes is easier to bypass by packing a sample with a UPX packer. Interestingly, the *XO* reduced the detection rate the most; however, it was not the most effective perturbation for bypassing the top 6 methods. Hence, the effective perturbations can be identified from our results when used against different anti-malware solutions we evaluated, as well as the ability to check with more perturbations and anti-malware solutions. A better understanding of the effectiveness of perturbations to create evasive malware variants can be used to enhance the detection rate of anti-malware solutions.

D. ANALYSIS BASED ON MALWARE CATEGORIES

We analyze from the perspectives of malware family and category to see how well FUMVar-Ex generates malware



(a) Avast

(b) AVG

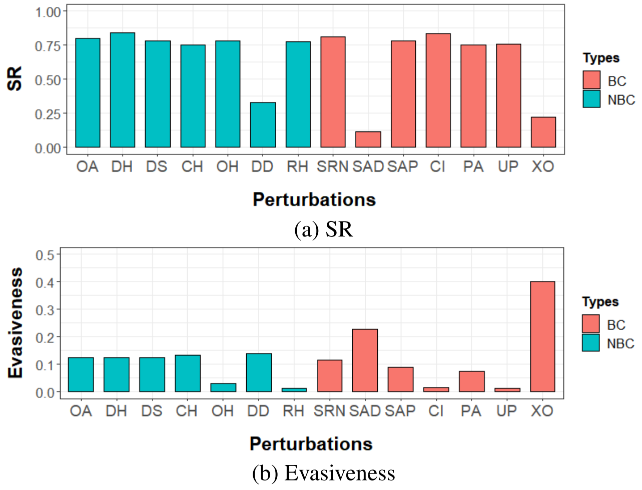
(c) Bitdefender

(d) Kaspersky

(e) Malwarebytes

(f) McAfee

FIGURE 6. Effectiveness of individual perturbations on evasiveness from the top 6 VirusTotal solutions.



(a) SR

(b) Evasiveness

FIGURE 7. Effectiveness of individual perturbations.

variants. We categorized the collected malware samples using the malware family information provided by the malware sharing site [25]. Four malware families are used to classify the collected malware samples: (M1) AgentTesla, (M2) GULoader, (M3) Loki, and (M4) njRAT. Fig. 8 shows the evasiveness of different malware families using `FUMVar-Ex`. The result shows that `FUMVar-Ex` outperformed `AIMED` and `FUMVar` for almost all malware families in evasiveness. In particular, `FUMVar-Ex` achieved better performance than `FUMVar` with significant gaps for Loki and njRAT, 11% and 15%, respectively, which means `FUMVar-Ex` bypassed over seven more solutions. However, `RL-ember` was more effective than `FUMVar-Ex` for the GULoader family, and `RL-malconv` and `FUMVar` for the AgentTesla family. This indicates that the global optimization performed for `FUMVar-Ex` is less effective on GULoader and AgentTesla malware families, and it can further be improved (e.g., optimized for specific malware families for enhanced evasiveness). From this result, we can still conclude that the newly defined perturbations (i.e., *XO* and *SAD*) may be adequate to generate evasive malware variants for all those malware families.

We also used the similarity of the API call sequences to categorize malware samples. First, we used the Cuckoo sandbox report and extracted each malware sample's API call sequence. The similarities between the samples' API call sequences are then measured. Lastly, we used a hierarchical clustering algorithm *single linkage algorithm* [28] to cluster malware samples into categories until four[2] groups are left. These malware clusters have the following characteristics: (C1) Cluster 1's API calls were related to processes, threads, and injections; (C2) Cluster 2's API calls were related to a registry key; (C3) Cluster 3's API calls were related to network operations; and finally (C4) Cluster 4's API calls were related to searches and monitors.

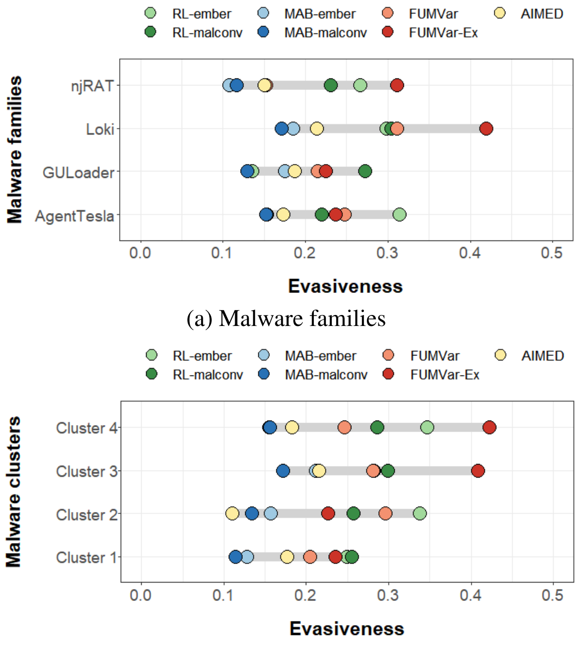
Fig. 8b shows that `FUMVar-Ex` outperformed `AIMED`, `MAB-ember`, and `MAB-malconv` in all clusters. For Cluster 3 and 4, `FUMVar-Ex` produced the best results, while `RL-malconv` and `RL-ember` produced the best results for Cluster 1 and 2, respectively. In addition, `FUMVar-Ex` performed better compared to `FUMVar`; the performance gap with `FUMVar` was 7% on average and at most 17%. Therefore, our recommendation would be to use `FUMVar-Ex`.

Another critical observation is that `FUMVar-Ex` was more effective on malware in Cluster 3 and 4 with a considerable gap compared with other frameworks. This result shows that `FUMVar-Ex` (and other similar perturbation techniques) could generate (undetectable) malware variants more effectively with network operations and API calls related to searches and monitors.

E. GENERATION EFFICIENCY

We compared the efficiency of malware variant generation using various state-of-the-art malware variant generators,

[2] For simplicity, we used the same number of clusters as the number of malware families used, but other cluster sizes can be used.



(a) Malware families

(b) Malware clusters

FIGURE 8. Comparing the evasiveness w.r.t. malware categories.

as shown in Fig. 4. To measure efficiency, we evaluated the *evasion time*, *CPU usage*, and *memory usage* for each generator. To compare them fairly, we used the same parameters and environment setups for GA-based malware variant generators, such as the number of perturbations and perturbation probabilities, and the default parameters, such as the reward and maximum iteration, defined in the open-source code for RL-based models. We used a machine with Ubuntu 18.04 × 64, four cores, and 16GB RAM to run the malware generators.

We first measured *evasion time*, representing the average time required to evade a target anti-malware solution. Since several malware variant generators, such as FUMVar and FUMVar-Ex, are intended to evade multiple anti-malware solutions, we measured the average time taken to evade a randomly selected anti-malware solution for them. The results of the *evasion time* measurements are presented in Table 4. The table shows that FUMVar-Ex achieved the best performance (1358.13 seconds) compared to the GA-based generators (i.e., AIMED and FUMVar). However, FUMVar-Ex required a longer time to evade anti-malware solutions compared to RL-based generators (i.e., RL and MAB-MALWARE). We found that the main reason for this difference was the behavior validation using the Cuckoo sandbox, which FUMVar-Ex implemented, but RL-based generators did not. Therefore, FUMVar-Ex could achieve better or similar performance if RL-based generators implement behavior validation.

Next, we analyzed *CPU usage* and *memory usage*, which denote the percentage of CPU usage and the amount of memory allocation, respectively. To compute CPU and memory usage, we measured the CPU utilization rate and memory usage per second using psutil [29] while generating malware variants. The CPU and memory usage measurement results are also presented in Table 4. The table shows that FUMVar-Ex exhibited acceptable performance in terms of both CPU and memory usage.

TABLE 4. Generation efficiency results.

Approach	Evasion time (s)	CPU (%)		Memory (GB)	
		Avg.	Std.	Avg.	Std.
RL	5.10	55.34	36.77	1.41	0.02
MAB-MALWARE	6.93	26.03	2.99	1.44	0.02
AIMED	7295.45	0.53	15.34	0.49	0.04
FUMVar	4892.10	0.47	14.11	0.51	0.03
FUMVar-Ex	1358.13	1.05	18.62	0.73	0.33

VI. DISCUSSION

A. BYPASSING COMMERCIAL ANTI-MALWARE SOLUTIONS

Our experimental results (as shown in Fig. 5) demonstrate the effectiveness of FUMVar-Ex against commercial anti-malware solutions. In practice, most users use a single anti-malware product, and a few users have more than one. In such a scenario, if a target user is found, it would be significantly easier to bypass the target anti-malware product. As we have already included many of the popular solutions in our experiments, our coverage seems reasonable to show the practical impact of the effectiveness of FUMVar-Ex against commercial anti-malware solutions and the urgent need to improve our anti-malware solutions by better understanding how evasive malware variants can be generated.

B. VALIDATING MALWARE VARIANTS' BEHAVIORS

As presented in Section III-D, we used signature fields to validate the behavior of malware variants generated. We generated a superset of signature fields using multiple Cuckoo sandbox analysis reports for each malware sample. Our empirical analysis suggests that five independent reports would sufficiently build a reasonable set of signature fields to capture malware samples' behaviors. However, real-world malware variants may have dynamically changing behaviors when some specific conditions are met, which may not be captured in how we identify their behaviors. For example, some may not show all their malicious behaviors by detecting their executing platform being a virtual machine [30]. Therefore, in such cases, our current FUMVar-Ex implementation using the Cuckoo sandbox would fail to obtain malware samples' signature fields successfully. To address this issue, we should consider in-depth program behavior analyses to capture malware samples' behaviors comprehensively.

C. DEVELOPING EVASIVE MALWARE VARIANTS

As shown in Fig. 7, there is a trade-off between SR and evasiveness. For example, *XO* and *SAD* would be effective in evasiveness while having lower SRs than other techniques,

TABLE 5. Comparisons between FUMVar-Ex and other state-of-the-art solutions.

Approach	Base tech.	Target tech.	Perturbation	# Perturbations	# Baseline approaches	Behavior check	Category Analysis
Anderson et al. [6]	RL	ML	BC, NBC	11	(-)	No	No
Song et al. [13]	RL	ML, Commercial	BC, NBC	13	3	No	No
Zhong etal. [14]	RL	ML	BC	3	3	No	No
Qiao et al. [12]	Prototype	ML, Commercial	BC	7	3	No	No
Castro et al. [8]	GA	ML	BC, NBC	13	1	No	No
Jin et al. [11]	GA	Commercial	BC, NBC	13	1	Yes	Yes
Our approach	GA	**Commercial**	**BC, NBC**	**14**	**6**	**Yes**	**Yes**

indicating that a direct modification of PE sections would increase the likelihood of evading detection, but it is hard to keep the PE file format compatible. Our findings suggest that no perturbation technique is useful in both SR and evasiveness. Therefore, it seems better to apply a combination of multiple perturbation techniques and perform a sufficient number of perturbations to generate fully-working malware variants to bypass detectors with high evasiveness.

VII. RELATED WORK

A. MALWARE DETECTION

There are two representative approaches for detecting malware: static and dynamic analysis approaches. The static analysis is to identify some specific code patterns and the hash value of files collected from known malware samples. Static analysis is widely used to determine whether a given file is malicious or not because it is simple and can easily be deployed with signatures or predefined rules. However, various techniques (e.g., code obfuscation, dynamic code loading, encryption, and packing) can be used by malware writers to evade static analysis [31]. Dynamic analysis would be more robust to such techniques because dynamic analysis tools have mainly been designed to capture malware samples' malicious behaviors. Recently, Machine learning (ML) techniques have become an integral part of malware detectors to improve their detection accuracy. Ye et al. [32] presented a malware detection system using ML with the features of Windows API call behaviors. Xu et al. [33] proposed a malware detection method using ML with the features of virtual memory usage. Ahmed et al. [34] suggested using an NLP-based deep learning model to identify malware's behaviors by leveraging Bidirectional Encoder Representations from Transformers (BERT) pre-trained model. However, malware writers are still trying to develop more evasive malware variants by manipulating the features used for ML models.

B. ML FOR MALWARE VARIANT GENERATION

ML techniques are popularly used to automate and speed up malware generation. For example, Anderson et al. [6] proposed a framework based on reinforcement learning (RL) to evade the detection of static malware classifiers. Zhong et al. [14] proposed a framework named MalInfo to evade an ML malware classifier based on RL with several new obfuscation techniques. Song et al. [13] proposed an efficient RL-based malware variant generator, MAB-MALWARE, applying the multi-armed bandit (MAB) problem to the model. However, these RL-based frameworks do not consider the generated malware variants' behaviors. Castro et al. [9] suggested the framework dubbed ARMED to generate malware variants with randomly selected perturbations. At first glance, this approach seems similar to FUMVar-Ex. Unlike FUMVar-Ex, however, ARMED cannot ensure that generated malware variants are correctly working. Hu and Tan [5] suggested using adversarial ML for generating evasive malware variants against a specific target malware classifier. Similarly, Qiao et al. [12] proposed a malware generation framework using adversarial examples. Their proposed method extracts some bytes of a representative benign sample and adds them into a malware sample to fool a deep learning-based detection model. However, even though generated malware samples can evade a targeted deep learning-based detection model, they achieved a lower evasiveness rate (7.1%) against VirusTotal engines than our proposed framework, FUMVar-Ex.

C. GA FOR MALWARE VARIANT GENERATION

GA is a computational model widely studied to generate malware samples. For example, Choi et al. [7] proposed the framework dubbed AVMG, which can generate malware variants at the source code level. Cani et al. [35] suggested a set of instructions to modify malware samples using GA while preserving their original behaviors. Expanding on the same concept, AIMED framework was proposed by Castro et al. [8], which achieved up to 50% faster speed using several perturbations defined in [6]. Jin et al. [11] also proposed a framework called FUMVar using GA to generate malware variants. In this paper, we extend FUMVar into a more effective and generalized one by introducing new perturbation techniques such as *XO* and *SAD* that can directly modify PE sections and a more robust method to validate malware samples' behaviors. This paper extends FUMVar into a more effective and practical framework dubbed FUMVar-Ex with 14 perturbations and a new malware functionality validation method. The experimental results show that FUMVar-Ex is superior to FUMVar in the evasiveness against commercial anti-malware products. Table 5 presents a comprehensive overview of the comparison results between FUMVar-Ex and other state-of-the-art methods, including FUMVar. The table covers several details, including the base technique employed, the target of the method, the types and number of supported perturbations, the number of approaches analyzed for baseline comparison, and the functionalities provided by each method.

VIII. CONCLUSION

We must understand how evasive malware variants could be generated to enhance anti-malware solutions. This paper proposes a GA-based malware variant generation framework dubbed `FUMVar-Ex` to automatically generate malware variants to evade the detection of anti-malware solutions. The evaluation results demonstrated that `FUMVar-Ex` achieved 28% and 78% better evasiveness than `FUMVar` and `AIMED`, the state-of-the-art malware variant generators using GA, respectively. We also showed that `FUMVar-Ex` outperformed the state-of-the-art malware variant generators using RL, namely `RL` and `MAB-MALWARE` with 70% and 25% better SR with notably improved evasiveness, respectively. Further, we showed that `FUMVar-Ex` can be used to generate malware variants that bypassed up to 86% on selected commercial anti-malware products. To evaluate the effectiveness of `FUMVar-Ex` in more details, we analyzed the effectiveness with respect to different malware families and categories, where we experimented using individual perturbations and observed varying SR and evasiveness. We found that the XOR obfuscation and adding benign section techniques, the two new perturbation techniques introduced in this paper, are specifically useful for generating evasive malware variants from antivirus engines from VirusTotal. Furthermore, the results showed that individual perturbations' effectiveness is greatly varied under different malware families and categories. This finding implies that the evasiveness techniques could be selected carefully and customized against targeted malware detectors, which can be used to improve our anti-malware solutions based on our findings. Therefore, we believe that `FUMVar-Ex` would be a useful tool to enhance the effectiveness of anti-malware solutions against new and unforeseen malware variants.

REFERENCES

[1] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 1–40, May 2018.

[2] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Comput. Secur.*, vol. 81, pp. 123–147, Mar. 2019.

[3] B. B. Rad, M. Masrom, and S. Ibrahim, "Camouflage in malware: From encryption to metamorphism," *Int. J. Comput. Sci. Netw. Secur.*, vol. 12, no. 8, pp. 74–83, 2012.

[4] D. Nieuwenhuizen, "A behavioural-based approach to ransomware detection," MWR Labs, Basingstoke, U.K., Tech. Rep. 2017-04-5, 2017.

[5] W. Hu and Y. Tan, "Generating adversarial malware examples for black-box attacks based on GAN," 2017, *arXiv:1702.05983*.

[6] 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, *arXiv:1801.08917*.

[7] J. Choi, D. Shin, H. Kim, J. Seotis, and J. B. Hong, "AMVG: Adaptive malware variant generation framework using machine learning," in *Proc. IEEE 24th Pacific Rim Int. Symp. Dependable Comput. (PRDC)*, Dec. 2019, p. 24609.

[8] R. L. Castro, C. Schmitt, and G. Dreo, "AIMED: Evolving malware with genetic programming to evade detection," in *Proc. 18th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun./13th IEEE Int. Conf. Big Data Sci. Eng. (TrustCom/BigDataSE)*, Aug. 2019, pp. 240–247.

[9] R. L. Castro, C. Schmitt, and G. D. Rodosek, "ARMED: How automatic malware modifications can evade static detection?" in *Proc. 5th Int. Conf. Inf. Manage. (ICIM)*, Mar. 2019, pp. 20–27.

[10] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Functionality-preserving black-box optimization of adversarial Windows malware," 2020, *arXiv:2003.13526*.

[11] B. Jin, J. Choi, H. Kim, and J. B. Hong, "FUMVar: A practical framework for generating fully-working and u nseen m alware var iants," in *Proc. 36th Annu. ACM Symp. Appl. Comput.*, Mar. 2021, pp. 1656–1663.

[12] Y. Qiao, W. Zhang, Z. Tian, L. T. Yang, Y. Liu, and M. Alazab, "Adversarial malware sample generation method based on the prototype of deep learning detector," *Comput. Secur.*, vol. 119, Aug. 2022, Art. no. 102762.

[13] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin, "MAB-malware: A reinforcement learning framework for blackbox generation of adversarial malware," in *Proc. 17th ACM Asia Conf. Comput. Commun. Secur. (ASIACCS)*, 2022, pp. 990–1003.

[14] F. Zhong, P. Hu, G. Zhang, H. Li, and X. Cheng, "Reinforcement learning based adversarial malware example generation against black-box detectors," *Comput. Secur.*, vol. 121, Oct. 2022, Art. no. 102869.

[15] Z. Fang, J. Wang, B. Li, S. Wu, Y. Zhou, and H. Huang, "Evading anti-malware engines with deep reinforcement learning," *IEEE Access*, vol. 7, pp. 48867–48879, 2019, doi: 10.1109/ACCESS.2019.2908033.

[16] M. Pietrek, "Peering inside the PE: A tour of the win32 (R) portable executable file format," *Microsoft Syst. J.*, vol. 9, no. 3, pp. 15–38, 1994.

[17] G. D. Webster, B. Kolosnjaji, C. von Pentz, J. Kirsch, Z. D. Hanif, A. Zarras, and C. Eckert, "Finding the needle: A study of the *PE32* rich header and respective malware triage," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2017, pp. 119–138.

[18] R. Keldorph. (2019). *E Format*. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

[19] M. Poslušný and P. Kálnai, "Rich headers: Leveraging this mysterious artifact of the PE format," in *Proc. Annu. Virus Bull. Int. Conf.*, 2019, pp. 1–24.

[20] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser. (2012). *The Cuckoo Sandbox*. [Online]. Available: https://www.cuckoosandbox.org

[21] R. Thomas, "LIEF: Library to instrument executable formats," Quarkslab, Paris, France, Tech. Rep. 17-07-RMLL-LIEF, 2017.

[22] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digit. Invest.*, vol. 3, pp. 91–97, Sep. 2006.

[23] J. Upchurch and X. Zhou, "Variant: A malware similarity testing framework," in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2015, pp. 31–39.

[24] M. Botacin, V. H. G. Moia, F. Ceschin, M. A. A. Henriques, and A. Grégio, "Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios," *Forensic Sci. Int., Digit. Invest.*, vol. 38, Sep. 2021, Art. no. 301220.

[25] *MalwareBazaar*. Accessed: Jun. 26, 2020. [Online]. Available: https://bazaar.abuse.ch/

[26] H. S. Anderson and P. Roth, "EMBER: An open dataset for training static PE malware machine learning models," 2018, *arXiv:1804.04637*.

[27] M. Al-Fawa'reh, A. Saif, M. T. Jafar, and A. Elhassan, "Malware detection by eating a whole APK," in *Proc. 15th Int. Conf. Internet Technol. Secured Trans. (ICITST)*, Dec. 2020, pp. 1–7.

[28] J. C. Gower and G. J. S. Ross, "Minimum spanning trees and single linkage cluster analysis," *J. Roy. Stat. Soc. C, Appl. Statist.*, vol. 18, no. 1, pp. 54–64, 1969.

[29] G. Rodola. (2009). *Psutil*. Accessed: Jan. 10, 2023. [Online]. Available: https://github.com/giampaolo/psutil

[30] T. Vidas and N. Christin, "Evading Android runtime analysis via sandbox detection," in *Proc. 9th ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS)*, Jun. 2014, pp. 447–458.

[31] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—A state of the art survey," *ACM Comput. Surv.*, vol. 52, no. 5, pp. 1–48, Sep. 2020.

[32] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang, "An intelligent PE-malware detection system based on association mining," *J. Comput. Virol.*, vol. 4, no. 4, pp. 323–334, Nov. 2008.

[33] Z. Xu, S. Ray, P. Subramanyan, and S. Malik, "Malware detection using machine learning based analysis of virtual memory access patterns," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 169–174.

[34] M. E. Ahmed, H. Kim, S. Camtepe, and S. Nepal, "Peeler: Profiling kernel-level events to detect ransomware," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2021, pp. 240–260.

[35] A. Cani, M. Gaudesi, E. Sanchez, G. Squillero, and A. Tonda, "Towards automated malware creation: Code generation and code integration," in *Proc. 29th Annu. ACM Symp. Appl. Comput. (SAC)*, Mar. 2014, pp. 157–160.

BEOMJIN JIN received the B.S. degree from the Department of Software, Sungkyunkwan University, Suwon, Republic of Korea, in 2019. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, Sungkyunkwan University. His research interests include malware analysis and cyber threat intelligence.

JUSOP CHOI received the B.S. degree from the Department of Information Engineering, Sungkyunkwan University, Suwon, Republic of Korea, and the Ph.D. degree from the Department of Electrical and Computer Engineering, Sungkyunkwan University. He is currently the Chief of the Digital Transformation Security Team, Digital Transformation Response Support Department, Financial Security Institute. His current research interests include compliance, cloud security, and financial security.

JIN B. HONG (Member, IEEE) received the Ph.D. degree in computer science from the University of Canterbury, New Zealand. He is currently a Senior Lecturer with the Department of Computer Science and Software Engineering, The University of Western Australia, Australia. His research interests include security modeling and analysis of computer and networks (including cloud computing, SDN, and the IoT), moving target defense, humans, and cyber.

HYOUNGSHICK KIM received the B.S. degree from the Department of Information Engineering, Sungkyunkwan University, in 1999, the M.S. degree from the Department of Computer Science, KAIST, in 2001, and the Ph.D. degree from the Computer Laboratory, University of Cambridge, in 2012. After completing his Ph.D., he was a Postdoctoral Fellow with the Department of Electrical and Computer Engineering, The University of British Columbia. Previously, he was with Samsung Electronics as a Senior Engineer, from 2004 to 2008. He was also a Distinguished Visiting Researcher with CSIRO, Data61, from 2019 to 2020. He is currently an Associate Professor with the Department of Computer Science and Engineering, College of Software, Sungkyunkwan University. His current research interests include usable security, blockchain, security vulnerability analysis, and data-driven security.