MultiEvasion: Evasion Attacks Against Multiple Malware Detectors

Hao Liu
University of Cincinnati
liu3ho@mail.uc.edu

Wenhai Sun
Purdue University
whsun@purdue.edu

Nan Niu
University of Cincinnati
nan.niu@uc.edu

Boyang Wang
University of Cincinnati
boyang.wang@uc.edu

***Abstract*—End-to-end malware detection analyzes raw bytes of programs with deep neural networks. It is considered as a new promising approach to simplify feature selection in static analysis but still provide accurate detection. Unfortunately, recent studies show that evasion attacks can modify raw bytes of malware and force a well-trained detector to predict the crafted malware as benign. In this paper, we propose a new evasion attack and validate the vulnerability of end-to-end malware detection in the context of multiple detectors, where our evasion attack MultiEvasion can defeat two (or even three) classifiers simultaneously without affecting functionalities of malware. This raises emerging concerns to end-to-end malware detection as running multiple classifiers was considered as one of the major countermeasures against evasion attacks. Specifically, our experimental results over real-world datasets show that our proposed attack can achieve 99.5% evasion rate against two classifiers and 18.3% evasion rate against three classifiers. Our findings suggest that the security of end-to-end malware detection need to be carefully examined before being applied in the real world.**

I. Introduction

End-to-end malware detection [1]–[6], i.e., *detecting malware by analyzing raw bytes of programs using deep neural networks*, is considered as a new promising approach to simplify feature selection in static analysis while still provide accurate detection. However, recent research [7]–[12] has shown that end-to-end malware detection is vulnerable under evasion attacks, where an adversary could modify raw bytes of a malware without affecting its functionality but force a neural network classifier predicting it as benign. As existing evasion attacks generate crafted malware based on single classifiers, one of the suggested countermeasures against evasion attacks is to utilize multiple classifiers to examine raw bytes of a program, where the classifiers jointly predict the result of malware detection [13], [14].

In this paper, we propose a new evasion attack, referred to as *MultiEvasion*, which can defeat multiple classifiers simultaneously. Specifically, our attack can modify a malware to a crafted program, which still has the same functionalities but can force two (or even three) well-trained neural networks to predict it as benign. *Our main idea* is to first obtain intermediate crafted perturbations from each neural network. Next, our method generates final perturbations based on intermediate perturbations such that the final perturbations can defeat multiple neural networks. Our findings indicate that, although end-to-end malware detection has many advantages compared to existing methods, *we should be cautious about its potential vulnerabilities and more analyses on the security of end-to-end malware detection should be investigated before we apply it in the real world*. Our main contributions and results are summarized below:

- We propose a new evasion attack which can defeat multiple neural network classifiers simultaneously without affecting the functionalities of malware.
- We evaluate our method against multiple malware detectors over two Windows PE program datasets. We show that existing evasion attacks based on single classifiers are not able to defeat multiple classifiers at the same time. On the other hand, our method can achieve an evasion rate of 99.5% against two classifiers and an evasion rate of 18.3% against three classifiers.
- We conduct comprehensive evaluations by leveraging different crafted byte ratio, functionality-preserving functions, gradient-based algorithms, and different formats (including vectors and images) of raw bytes of malware. For instance, even two classifiers use two different formats of raw bytes, our method can still achieve an evasion rate of 96.4%.

Reproducibiltiy. The code and datasets can be found at https://github.com/UCdasec/MultiEvasion.

II. Background

A. System and Threat Model

System Model. The *system model* is described in Fig. 1. A malware submitted to the malware detection service, and the detection service performs malware analysis, decides whether the program is a malicious program or a benign program. The detection service makes the decision by running multiple pre-trained *classifiers* (or also referred to as *malware detectors*) over the raw bytes of a program. Each classifier makes its own prediction. The final prediction is the ensemble of the predictions across all the classifiers, where we assume each classifier has an equal weight. The malware detection, in essence, is a binary classification.

Threat Model. We assume that an attacker can modify the malware before sending it to the malware detection service. The goal of this attacker is to modify the raw bytes of a malware and force all the classifiers on the server side to predict it as benign. In addition, the modifications on the raw bytes would

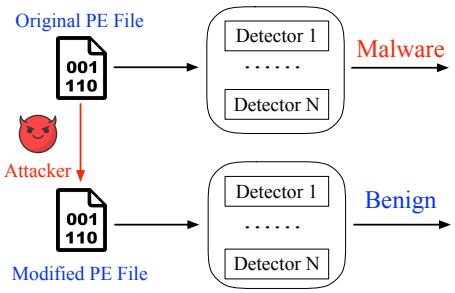

Fig. 1. Evasion Attacks against Multiple Malware Detectors.

not affect the functionality of the malware, i.e., the malware is still functional. This attack is commonly known as an *evasion attack*.

We assume that each classifier is a neural network. Every classifier is pre-trained and does not change during the attack. We assume that the attacker has the *white-box access*, where the information of each target classifier, including architecture and parameters, are public and known to the attacker.

Scope. In this study, we consider programs that are in Portable Executable format in Windows, and we consider static analysis over raw bytes only. In addition, we assume that all the programs are unpacked as some previous studies [15]–[17].

Data Representation. We assume that the raw bytes of a program is represented as a one-dimensional vector of integers starting from the first byte of a program. Each integer represents one byte and it is a decimal number within $[0, 255]$. The order of the integers in a vector follows the same order of their associated bytes in a program. This vector is utilized as an input to a neural network, where the vector length is pre-defined and fixed across programs. For instance. some studies [1], [2] suggest to use 102,400 or even 2,000,000 as the vector length to achieve promising performance in malware detection. If the actual size of a program is greater than the vector length, the bytes after the vector length will be ignored. If the actual size of a program is less than the vector length, padding 0s will be applied. Leveraging raw bytes of programs as inputs to a classifier offers several advantages in malware detection [1]–[3]. For instance, it can skip manual feature selections, which can be time-consuming.

B. Portable Executable Format

The Portable Executable (PE) format is a format for executable programs in Windows. A program in PE format often consists of several parts, including DOS Header and Stub, PE Header, Optional Header, Section Table, and Sections. A high-level description of PE format is illustrated in Fig. 2.

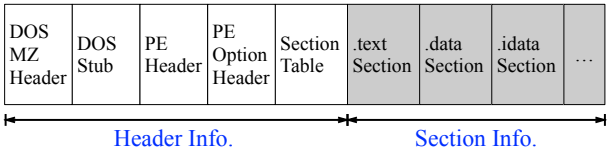

Fig. 2. The Structure of PE file format.

DOS Header and Stub. The main purpose of DOS Header and Stub is to ensure programs are backward compatible with MS-DOS [1]. As a result, when running a program on modern Windows systems, bytes (except the ones from two subsections) within DOS Header and Stub are redundant and can be modified without affecting the functionality of a program. One subsection that cannot be altered is the first two bytes `0x4D5A`, which are also known as a magic number "MZ". The other subsection that cannot be altered is the 4-byte signature at offset `0x3C`, where the signature indicates the entry point of PE header.

PE Header. PE header contains information, including head signature (e.g., a magic number "PE", which is `0x50450000`), header size, and file attributes, that are utilized by OS loader.

Optional Header. It is optional for object files and starts with a 2-byte magic number indicating the architecture (e.g., `0x010B` for PE32, `0x200B` for PE64, and `0x0107` for ROM) [2]. It also includes the size and virtual base of the code and data, entry point, and the number of directories. In addition, it specifies a value named *file alignment*, which suggests the size of each section of a program must be a multiple of this value.

Section Table. It contains information, such as the number of sections, the name of each section, the size of each section, and the address of each section.

Sections. Common sections in a program include executable code section (*.text*), data section (e.g., *.data*, *.rdata*, etc), resources section (*.rsrc*), export data section (*.edata*), import data section (*.idata*), debug information section (*.debug*).

C. Functionality-Preserving Modifications

We present existing functionality-preserving modifications [8], [10], [11], [18] on malware, where these modifications can change or add bytes in a program without affecting its functionality. We will leverage those methods or the combination of those to locate bytes that can be modified in a PE program in our study. In other words, these existing methods can answer the question — *which bytes in a malware can be modified*.

Padding (P). Padding [11], [18] appends arbitrary bytes to the end of a program. It does not affect the functionality of a program as the OS will not load those padded bytes.

Slack (S). Slack space [11] refers to the `0x00` bytes towards the end of a section. These zero bytes are not code or data of a program but introduced by the compiler as the size of each section needs to be a multiple of *file alignment* value defined in Optional Header. These bytes can be modified without affecting the functionalities of a program.

Partial DOS (PD) and Full DOS (FD). Partial DOS [10] modifies all the 58 bytes after the 2-byte magic number "MZ" but before the 4-byte signature of PE header at offset `0x3C` in DOS Header. Full DOS [8] modifies all the bytes in DOS Header and Sub except the 2-byte magic number "MZ" and 4-byte signature of PE header at offset `0x3C`.

[1]https://docs.microsoft.com/en-us/windows/win32/debug/pe-format
[2]https://wiki.osdev.org/PE

Extending (E). This method [8] extends the size of DOS Header and Stub such that a greater number of bytes can be modified compared to Full DOS. There are several steps that are needed to ensure the functionality of a program is not affect. First, the extended size should be a multiple of the *file alignment* value specified in Optional Header. Second, the size of headers and the bytes of the PE header signature at offset `0x3C` will need to be updated accordingly. Next, the PE header needs to be shifted according to the new PE header signature. Last, the offset of entry point of each section will need to be shifted accordingly based on the the number of extended bytes in DOS Header and Stub.

Content Shift (CS). This method [8] can insert a new space with arbitrary bytes before the start of a section in a program. The size of this new space would be a multiple of the *file alignment* value. As a result, the offset of the entry point of each section defined in Section Table will need to be updated according to the shift size. While it is feasible to insert a new space before any section, we only focus on inserting a new space before the first section of a program in this study.

Validation of Functionality-Preserving Modifications. We validate the feasibility of all these methods in our experiments. Specifically, given an unpacked PE program (e.g., `putty.exe`, a popular program for SSH and telnet in Windows), we perform each modification method described above by using a hex editor named 010 Editor in Windows. We save the modified program generated by each method and run it. In addition, we utilize an open-source behavior analysis tool, Cuckoo Sandbox, to confirm the functionalities of the modified program remains the same as the ones in the original program.

thd

III. Proposed Evasion Attacks

In this section, we describe the details of our method and answer the following open research question — *how bytes in a malware can be modified to defeat multiple classifiers*.

A. Problem Formulation

Evasion against Single Classifiers. For existing evasion attacks targeting on single classifiers, the problem can be described as follows. Given a program z and a target classifier f, where the predication of f on program z is malicious, i.e., $y = f(z) = 1$, an attacker generates a crafted program z' based on program z by performing functionality-preserving modifications such that the predication of f on crafted program z' is benign, i.e., $y' = f(z') = 0$, but z' has the same functionalities as original program z. In addition, the modifications to program z are expected to be minimal such that the attack is as stealthy as possible. It can be mathematically formulated as an optimization problem below

$$\min_{z'} L(f(z'), 0) \quad \text{and} \quad z' = \tau(z, t), \quad \text{s.t.} \quad t \in T \quad (1)$$

where $z = (z[1], z[2], ..., z[m])$ is a vector of raw bytes obtained from program z, m is the vector length, $z[i] \in [0, 255]$, for $1 \leq i \leq m$, L is the loss function measuring the predication of $f(z')$ and target label $y' = 0$, $\tau(\cdot)$ is the transform function that changes program z to crafted program z' using functionality-preserving modification t, T is a set of functionality-preserving modification methods that can be applied.

According to recent studies, the above optimization problem can be solved by gradient-based algorithms that used to generate adversarial examples [7], [19], [20] in the other domains (e.g., image recognition) given the constraints of preserving functionalities on malware. In other words, compared to perturbations on an image, not all the bytes in a malware can be perturbed and the values of perturbed bytes should remain within $[0, 255]$. Gradient-based Algorithms, such as Fast Gradient Sign Method (FGSM) [7], can be utilized. *Put differently, a functionality-preserving modification decides which bytes can be perturbed and a gradient-based algorithm decides how these bytes are perturbed in a malware.*

Evaluation Metric. To measure the effectiveness of an evasion attack, *evasion rate* over a set of malicious programs can be computed. Specifically, given a number of n malicious programs, if an attacker can obtain perturbed programs forcing target classifier f to predict incorrectly over a number of s malicious programs, where $s \leq n$, evasion rate is computed as s/n. A higher evasion rate indicates that the attack is more effective against target classifier f.

If an attack aims to defeat multiple classifiers, evasion rate is redefined as the ratio of the number of s crafted programs defeating all the classifiers over a given number of n malicious programs.

B. Our Method: Evasion on Multiple Detectors

Unfortunately, existing studies focus on defeating single detectors. Specifically, if a crafted program z' is obtained based on classifier f_1, there is no guarantee that this crafted program can also defeat classifier f_2. This is because the classifier f_2 has different architectures and parameters compared to classifier f_1, which leads to different gradients and perturbations over raw bytes of a program.

Our Main Idea. To address this limitation, we propose a new method, referred to as *MultiEvasion*, which adjusts perturbations on a malware such that a crafted program could defeat multiple detectors simultaneously. Our main idea is to first obtain an intermediate perturbation based on each classifier. Next, our method adjusts the perturbations across multiple intermediate perturbations to offset perturbations caused by inconsistent gradients over multiple classifiers, e.g., the perturbation over one byte is positive based on classifier f_1 but the perturbation over the same byte is negative based on classifier f_2. When gradients are consistent over multiple classifiers, e.g., the perturbations on the same byte are both positive according to classifier f_1 and f_2, the maximum perturbations among intermediate perturbations are applied to ensure the perturbations are sufficient to defeat all the classifiers. For ease of presentation, we describe our method against two detectors

in the following. The description can be easily extended to cases with more than two detectors.

Specifically, given two classifiers f_1 and f_2, a malicious program $z = (z[1], ..., z[m])$, where m is the number of vector length, $y_1 = f_1(z) = 1$ and $y_2 = f_2(z) = 1$, our method first obtains intermediate perturbation $r'_1 = (r'_1[1], ..., r'_1[m])$ from classifier f_1 and intermediate perturbation $r'_2 = (r'_2[1], ..., r'_2[m])$ from classifier f_2 by running the following equations

$$\begin{cases} r'_1 \leftarrow \texttt{SingleEvasion}(z, f_1) \\ r'_2 \leftarrow \texttt{SingleEvasion}(z, f_2) \end{cases} \tag{2}$$

and also runs

$$b \leftarrow \texttt{FindPerturbedIndex}(t) \tag{3}$$

where t is one type or a combination of multiple types of functionality-preserving modification and $b = (b[1], ..., b[m])$ suggests which bytes can be modified in z. We name b as perturbation-index vector. If $b[i] = 1$, where $i \in [1, m]$, it indicates the i-th byte of z can be perturbed according to functionality-preserving modification t. If $b[i] = 0$, it indicates the i-th byte can not be perturbed. We use $\texttt{SingleEvasion}(\cdot)$ as a general function representing any existing evasion attack on single classifiers, and use $\texttt{FindPerturbedIndex}(\cdot)$ as a general function to generate a perturbation-index vector.

Next, our method obtains the adjusted perturbation $r' = (r'[1], ..., r'[m])$. Specifically, given intermediate perturbations $r'_1 = [r'_1[1], ..., r'_1[m]]$ and $r'_2 = (r'_2[1], ..., r'_2[m])$, and perturbation-index vector $b = (b[1], ..., b[m])$, our method runs the following

$$\begin{cases} r'[j] = max(r'_1[j], r'_2[j]) & \text{if } (r'_1[j] > 0) \wedge (r'_2[j] > 0) \wedge (b[j] = 1) \\ r'[j] = min(r'_1[j], r'_2[j]) & \text{if } (r'_1[j] < 0) \wedge (r'_2[j] < 0) \wedge (b[j] = 1) \\ r'[j] = 0 & \text{otherwise} \end{cases} \tag{4}$$

for $1 \leq j \leq m$.

In other words, for $r'[j]$, if the corresponding byte can be modified without affecting functionalities (i.e., $b[j] = 1$), the perturbation $r'[j]$ will be updated as the maximum of $r'_1[j]$ and $r'_2[j]$ if $r'_1[j]$ and $r'_2[j]$ are both positive or the minimum of $r'_1[j]$ and $r'_2[j]$ if if $r'_1[j]$ and $r'_2[j]$ are both negative. Otherwise, $r'[j]$ will be set to zero.

With adjusted perturbation $r' = (r'[1], ..., r'[m])$, we can obtain the final crafted program $z' = (z'[1], ..., z'[m])$, where $z'[j] = z[j] + r'[j]$ for $1 \leq j \leq m$, to defeat the two detectors.

IV. Evaluation

A. Datasets

We examine our method over two real-world datasets. The first dataset, referred to as *ME dataset*, is collected by us in 2021. It includes 1,000 benign PE programs across multiple Windows versions, including Windows XP, Vista, 8 and 10, and 1,000 malicious PE programs downloaded from VirusShare [3]. The second dataset is a public dataset referred to as *phd dataset* [4]. It contains 977 benign PE programs and 2,597 malicious PE programs collected from 2011 to 2016.

We further split each dataset for training (70%), validation (10%), and testing (20%) as presented in Table. I.

TABLE I
THE TWO DATASETS

Dataset		Train	Validate	Test
ME	Benign	700	100	200
	Malicious	700	100	200
phd	Benign	684	98	195
	Malcious	1,817	260	520

B. Evaluation Setting

We conduct all the experiments on a Linux machine running Ubuntu 18.04.5 with Intel i9-9900K (3.60GHz) CPU, 64 GB Memory, and a NVIDIA Titan RTX GPU. We examine three neural-network-based target classifiers, including MalConv, FireEyeNet, and AvastNet, over raw bytes in our experiments. All the three classifiers are recently proposed by cybersecurity companies.

MalConv. MalConv [1] includes one 8-dimensional embedding layer, two 1-dimensional gated convolutional layers, a temporal max pooling layer, a fully connected layer with softmax. The embedding layer transfers bytes into a high-dimensional space. The convolutional layers are utilized to capture important features, where a great filter width (e.g., 500) and stride (e.g., 500) is used to minimize memory cost. The maximum input size examined in [1] is 2 MBs and it can achieve 98.1% AUC (Area Under the Curve) and 94% accuracy based on a private large-scale dataset from industry. The reported training time of MalConv over the private dataset in [1] is around one month.

FireEyeNet. FireEyeNet [2] was proposed by researchers from FireEye. It consists of one 10-dimensional embedding layer, five stacked 1-dimensional convolutional and max pooling layers, followed by a fully connected layer with sigmoid function. The maximum input size of each program examined in [2] is 102,400 and it achieves 98% AUC and 96% accuracy over a private large-scale dataset.

AvastNet. AvastNet was proposed in [3] by Avast. It includes one 8-dimensional embedding layer, four convolutional layers with a max pooling layer, a global average pooling layer, and four fully connected layers. As reported in [3], this model achieves 70.4% restricted AUC (96.0% accuracy) compare to MalConv with 66.1% restricted AUC (94.6% accuracy) over a private large-scale dataset from Avast, where restricted AUC is referred to as the area under the Receiver Operator Curve restricted to the interval [0,0.001] of the false positive rate.

Implementation Details of Our Method. We implement the three models and our method with Python 3.8.8 and PyTorch 1.8.0. When new bytes are inserted to an original program with a functionality-preserving modification, the values of those new bytes are initialized with bytes from benign programs rather

[3]https://virusshare.com/

[4]https://github.com/tgrzinic/phd-dataset

than zero bytes. Specifically, we randomly select and utilize bytes from *.text* sections of benign PE programs in our dataset. This is because the *.text* section has been proved that it has a significant impact on the predictions of neural networks [10]. With this step, it can defeat target classifiers with a less number of iterations during the generation of perturbations.

In addition, as all the three target classifiers apply an embedding layer, which acts as a non-differentiable feature mapping function, to map bytes to a higher dimensional space before running convolutional layers, the perturbations generated by gradient-based algorithms can not be obtained directly over the raw bytes in the problem space as the embedding function is non-differentiable. To address this issue, we adopt an approach from existing studies [8], [10] to obtain intermediate perturbations for each classifier, where the perturbations are obtained in the embedded space first with a gradient-based algorithm. Next, a reconstruction function [21] is used to transform a perturbed vector in embedded space back to perturbed bytes of a crafted program in the problem space.

We train all the three classifiers over ME and phd datasets respectively. The key training parameters of each classifier are presented in Table. II.

TABLE II
MODEL TRAINING PARAMETERS

Parameter	MalConv	FireEyeNet	AvastNet
Epoch	50	50	50
Batch Size	32	32	32
Learning Rate	0.0001	0.0001	0.0001
Input Size	102,400	102,400	102,400
Training Time (min) on ME	126	14	14
Training Time (min) on phd	221	17	18

C. Experimental Results

Experiment 1: Performance of Malware Detectors. We first examine the detection performance of target classifiers without evasion attacks. Specifically, we examine MalConv and FireEyeNet with input size of 102,400 for all programs over ME dataset and phd dataset. We defer the discussions on the input size to a later experiment. We use accuracy and Area Under the Curve (AUC) as the metrics to evaluate the detection performance of malware detectors. We present the results in Table. III. Overall, both classifiers are effective in detecting malware.

TABLE III
PERFORMANCE OF MALWARE DETECTORS

Dataset	MalConv		FireEyeNet	
	Acc	AUC	Acc	AUC
ME	95.5%	98.4%	98.5%	99.5%
phd	91.3%	97.3%	92.2%	96.5%

Experiment 2: Existing Evasion Attacks against Two Malware Detectors. In this experiment, we examine the performance of existing evasion attacks based on single classifiers and apply the perturbed programs to two target classifiers at the same time. The main purpose of this experiment is to show that these evasion attacks based on single classifiers are not able to defeat two classifiers simultaneously.

Specifically, we examine 3 existing methods with 5 different functionality-preserving modifications over our ME dataset. We generate crafted programs using MalConv as a target classifier for each method, and then examine the evasion rate of these crafted programs against two classifiers, including MalConv and FireEyeNet, simultaneously. The three existing methods include Slack FGM [11], Header Attack [10], and RAMEN attack [8]. We use 102,400 as the input size for each program. A total number of 200 malicious programs are examined in each attack. We present the evasion rate of these methods with different crafted byte ratio in Table. IV. The *crafted byte ratio* is defined as the number of crafted bytes over the input size, which is 102,400 in this experiment. As we can observe, these existing methods work well against a single classifier but are ineffective against two classifiers. This is expected as crafted programs from these methods are generated based on a single classifier. We have similar observations when generating crafted programs based on FireEyeNet and then examining evasion rate over two classifiers simultaneously. We skip the details due to space limitation.

TABLE IV
EVASION RATE OF EXISTING EVASION ATTACKS ON SINGLE CLASSIFIER AND MULTIPLE CLASSIFIERS.

No. of Classifiers	Evasion Attack	Crafted Byte Ratio			
		4%	8%	16%	32%
One	Slack FGM [11]	7.1%	7.1%	7.1%	7.1%
	Header Attack [10]	9.6%	9.6%	9.6%	9.6%
	RAMEN (FD) [8]	39.4%	39.4%	39.4%	39.4%
	RAMEN (E) [8]	90.4%	99.0%	99.5%	99.9%
	RAMEN (CS) [8]	93.9%	96.5%	99.5%	99.6%
Two	Slack FGM [11]	0%	0%	0%	0%
	Header Attack [10]	0%	0%	0%	0%
	RAMEN (FD) [8]	0%	0%	0%	0%
	RAMEN (E) [8]	0%	0%	1.5%	5.6%
	RAMEN (CS) [8]	0%	0%	1.5%	4.5%

Experiment 3: Evasion Rate of Our Method against Two Malware Detectors. In this experiment, we answer the two following questions: 1) *Can our method evade two malware detectors simultaneously?* 2) *Which functionality-preserving modification is more effective when using our method?*

To answer the first question, we generate crafted malware using two detectors, MalConv and FireEyeNet, by using our method. In addition, we examine 5 functionality-preserving modifications, including Partial/Full DOS, Extending, Content Shift, and Slack, as well as their combinations. We use ME dataset with an input size of 102,400. We leverage FGSM with $\epsilon = 0.1$ to generate intermediate perturbations. Parameter ϵ is the adversary strength that controls the amount of perturbations in FGSM algorithm.

As shown in Table. V, our method can achieve an evasion rate as high as 94.4 % over MalConv and FireEyeNet when we use the combination of Extending, Content Shift, and Slack as the functionality-preserving modification and modify 32% of bytes in each input (i.e., 32,768 bytes). Moreover, we also observe that if the modification includes Content Shift, the

attack is often more effective. Specifically, the modification with the combination of Content Shift and Slack outperform other modifications in most of the cases. We will use the combination of Content Shift and Slack as the functionality-preserving modification in the rest of this paper unless specified.

TABLE V
EVASION RATE OF OUR METHOD OVER MALCONV AND FIREEYENET WITH DIFFERENT FUNCTIONALITY-PRESERVING MANIPULATIONS

Manipu -lation	Crafted Byte Ratio					
	1%	2%	4%	8%	16%	32%
S	1.5%	1.5%	1.5%	1.5%	1.5%	1.5%
PD	0%	0%	0%	0%	0%	0%
FD	0%	0%	0%	0%	0%	0%
E	4.1%	5.6%	9.6%	13.6%	33.8%	93.4%
CS	5.6%	9.1%	19.2%	24.7%	42.4%	92.4%
PD+S	1.0%	1.0%	1.0%	1.0%	1.0%	1.0%
FD+S	0%	0%	0%	0%	0%	0%
E+S	5.1%	7.6%	12.1%	17.7%	37.9%	87.4%
CS+S	**8.6%**	**14.2%**	**19.8%**	**30.5%**	**47.2%**	92.9%
PD+CS	3.5%	3.0%	7.0%	13.6%	36.0%	89.4%
FD+CS	2.5%	5.1%	7.1%	14.7%	36.0%	87.3%
E+CS	4.1%	4.6%	7.6%	17.3%	39.1%	92.9%
PD+CS+S	4.1%	4.6%	7.6%	20.8%	39.1%	90.4%
FD+CS+S	3.6%	6.1%	9.1%	18.8%	42.1%	90.9%
E+CS+S	4.1%	5.6%	9.6%	20.8%	44.7%	**94.4%**

Experiment 4: The Impact of Adversary Strength ϵ of FGSM. We examine the impact of adversary strength ϵ of FGSM on the performance of our method. Specifically, we utilize ME dataset with an input size 102,400. We run the evasion attacks against two target classifiers, MalConv and FireEyeNet, multiple times, where each time we use a different adversary strength ϵ in FGSM – the gradient-based algorithm. We examine $\epsilon = \{0.1, 0.2, ..., 0.7\}$ and report the results in Fig. 3. We notice that if $\epsilon = 0.4$, our method achieves a higher evasion rate than the ones obtained from $\epsilon < 0.4$. If ϵ is greater than 0.4, the performance remains similar as the one with $\epsilon = 0.4$. This is likely because the perturbations on each byte are bounded with $[0, 255]$. When ϵ is higher than 0.4, although a higher perturbation can be generated by FGSM, the actual perturbation that can be applied to a byte of a crafted program is similar as the one with $\epsilon = 0.4$.

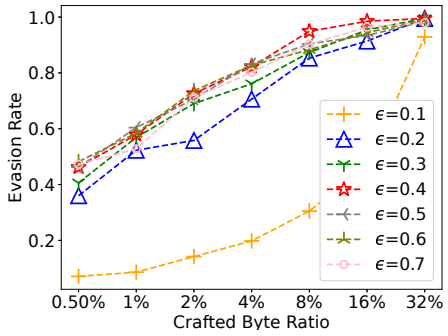

Fig. 3. Evasion rate of our method with different adversary strength ϵ.

Experiment 5: The Impact of Gradient-Based Algorithms. In this experiment, we examine the performance of our method when we use different gradient-based algorithms, including FGSM [7], PGD [20], FFGSM [22], to generate intermediate perturbations respectively. PGD is an iterative version of FGSM. FFGSM is an varient of FGSM, where FFGSM applies random noises to initialize perturbations, and it has been proved that it can achieve competitive performance as PGD. We examine ME dataset with an input size of 102,400. We use $\epsilon = 0.4$ (according to observation from the previous experiment) in FGSM and FFGSM. We set $\epsilon = 0.4$, $\alpha = 0.9$, and the number of iterations as 5 in PGD, where α controls the total amount of perturbations in each iteration of PGD.

As shown in Fig. 4, PGD is the most effective one compared to the other two, while FGSM obtains slightly lower evasion rate compare to PGD, but they are very close, especially when crafted byte ratio is greater than 2%. As for FFGSM, it achieves lowest evasion rate overall. On the other hand, as illustrated in Fig. 5, PGD takes a much longer time to generate a crafted program in our method. If an attacker would like to achieve a higher evasion rate with less generation time, FGSM is the best option among the three.

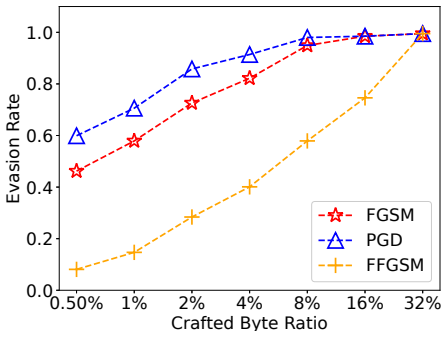

Fig. 4. Evasion rate of our method with different gradient-based algorithms.

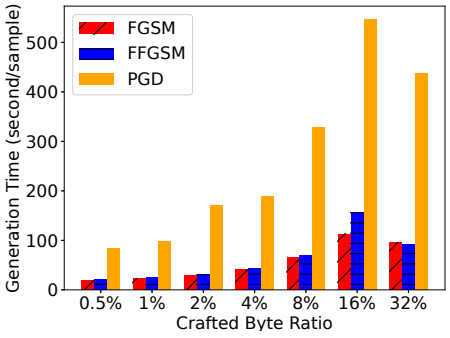

Fig. 5. Generation time of crafted programs in our method with different gradient-based algorithms.

Experiment 6: The Impact of Input Size. We investigate the impact of the input size on the evasion rate of our method. We still use ME dataset, and FGSM with $\epsilon = 0.4$ in this experiment. We attack two classifiers, MalConv and FireEyeNet, with our method over different input sizes, including 102,400, 204,800, and 408,600, of programs in ME dataset.

We first present the detection performance of two classifiers on ME dataset with different input sizes in Table. VI, and both classifiers are effective in detecting malware over different input sizes. For evasion performance, as we observe in Fig. 6, the evasion rate with input size 102,400 is higher than the ones with greater input sizes. In other words, our attack is less effective if a detector analyzes a greater number of bytes of a program. This is reasonable and expected as a detector is more difficult to defeat when it examines a greater number of bytes.

TABLE VI
DETECTION PERFORMANCE OF MALWARE DETECTORS WITH DIFFERENT INPUT SIZE

Input Size	MalConv		FireEyeNet	
	Acc	AUC	Acc	AUC
d=102400	95.5%	98.4%	98.5%	99.5%
d=204800	91.0%	97.8%	96.3%	98.1%
d=409600	96.3%	99.1%	97.0%	98.3%

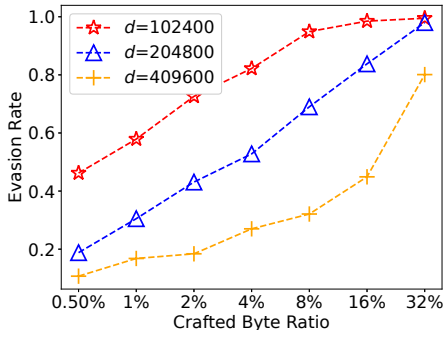

Fig. 6. Evasion rate of our method with different input size d.

Experiment 7: The Impact of Different Datasets. We investigate and compare the evasion rate of our method over two datasets, ME dataset and phd dataset, to validate whether the results are generic across different datasets.

We use an input size of 102,400, FGSM with $\epsilon = 0.4$ as the gradient-based algorithm in this experiment. We examine two detectors, MalConv and FireEyeNet, over each dataset. As shown in Fig. 7, our evasion attack has similar evasion rate over the two datasets given the same crafted byte ratio. Therefore, we believe that our proposed attack is robust across different datasets.

Experiment 8: The Impact of Different Payload Initialization. We investigate the impact of different payload initialization approaches of our method. As we mentioned in the beginning of this section, we initialize the values of crafted bytes with bytes from *.text* sections of benign programs. Besides this initialization approach, another approach is to initialize the bytes with random values between $[0, 255]$.

We still use ME dataset with an input size of 102,400, FGSM with $\epsilon = 0.4$ as the gradient-based algorithm in this experiment. We examine two detectors, MalConv and FireEyeNet. We compare the results between benign byte initialization and random byte initialization in Fig. 8. The results suggest that

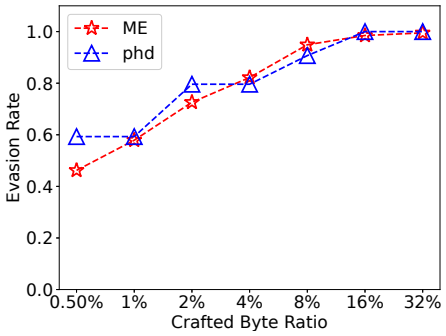

Fig. 7. Evasion rate of our method over different datasets.

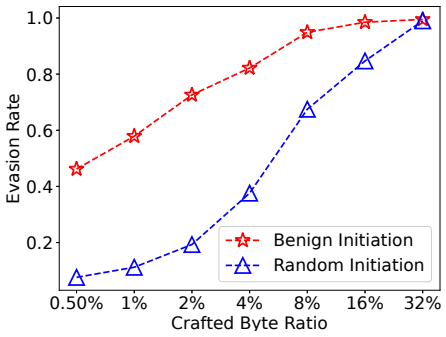

Fig. 8. Evasion rate of our method with different initialization approaches.

benign byte initialization is more effective and can further boost the evasion performance of our method.

Experiment 9: Evasion Rate of Our Method against Three Malware Detectors. We further investigate the evasion rate of our method against three malware detectors at the same time. Compare to previous experiments, we examine three target classifiers including MalConv, FireEyeNet, and AvastNet. Other settings remain the same. AvastNet achieves 99.0% accuracy and 99.7% AUC over ME dataset in malware detection in our experiments.

For evasion performance, as illustrated in Table. VII, our method can still defeat three classifiers with 18.3% evasion rate given the crafted byte ratio 32% by using FGSM. On the other hand, the evasion rate against 3 classifiers is much lower than the evasion rate (99.5%) against 2 classifiers. This is also expected as it is more difficult to defeat 3 classifiers at the same time. In addition, we also compare the generation time of our method (with FGSM) against different number of detectors in Fig. 9, where the generation time is longer while attacking a greater number of detectors.

TABLE VII
EVASION RATE OF OUR METHOD AGAINST THREE DETECTORS

Gradient-Based Algo.	Crafted Byte Ratio				
	2%	4%	8%	16%	32%
FGSM	0%	0.5%	1%	2%	**18.3%**
PGD	0%	0.5%	0.5%	1.5%	**24.4%**

Experiment 10: Evasion Rate of Our Method against Two Malware Detectors with Different Input Formats. In

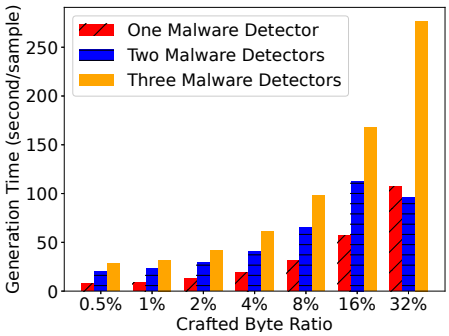

Fig. 9. Generation time of our method against different numbers of malware detectors.

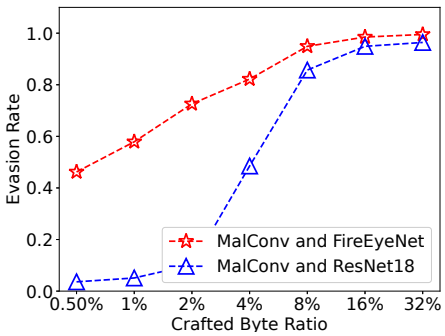

Fig. 10. Evasion rate of our method with different input formats.

this experiment, we investigate the evasion rate of our method against two malware detectors, where one detector represents raw bytes of PE programs as vectors for inputs and the other detector represents raw bytes of PE programs as greyscale images for inputs. Several recent studies [4]–[6], [18] show that the image format can also provide promising results in malware detection. For the detector using vectors, we use MalConv. For the detector using images, we utilize ResNet18 [23].

ResNet18 was proposed by He et al. [23]. It includes 18 convolutional layers and uses skip connection function to add output from one layer to the next layer, which can ease the training load of deeper networks and mitigate the gradient vanishing problem. We explore several models, including AlexNet [24], ResNet18, ResNet34, and Xception [25]. ResNet18 achieves the best performance in malware detection over our ME dataset.

When we train ResNet18 over ME dataset, we use 50 epochs, 128 batch size, 0.001 learning rate , and 320×320 as the input size for a number of 102,400 bytes of each program. ResNet18 achieves 98.0% accuracy and 97.5% AUC over ME dataset in malware detection as shown in Table. VIII.

TABLE VIII
DETECTION PERFORMANCE OF MALWARE DETECTORS

Dataset	ResNet18 (image format)		AvastNet (vector format)	
	Acc	AUC	Acc	AUC
ME	98.0%	97.5%	99.0%	99.7%

Next, we perform our evasion attack on MalConv and ResNet18. We still use FGSM with $\epsilon = 0.4$ as the gradient-based algorithm to generate perturbations. As shown in Fig. 10, our method can evade MalConv and ResNet18 at the same time, especially when the crafted byte ratio is greater than 2%. On the other hand, we also observe that attacking two detectors (i.e. MalConv and ResNet18) using two different formats is more difficult than attacking two detectors (i.e., MalConv and FireEyeNet) using the same format.

V. RELATED WORK

Malware Detection: There is a significant number of studies in the area of malware detection. Comprehensive surveys can be found in [26], [27]. Due to space limitation, we only briefly discuss the studies that are most related to this paper.

Many studies [4], [28]–[34] leverage static features, including signatures, API calls, n-grams features from disassembly or byte sequences, among others, to detect malware. For instance, Kolter et al. [28] extracted important byte sequences by adopting n-grams and leveraging machine learning classifiers (e.g., SVM, boosted models, decision trees) to detect malware. Anderson et al. [30] manually extracted features and utilized gradient boosted decision tree, which can even outperform MalConv. Daniel et al., [34] proposed a broad static analysis method to detect Android malware, named DREBIN. It is a lightweight and more explainable approach than deep learning based methods. However, feature engineering process is still needed and performed before training the models. Nataraj et al. [4] developed a malware detection by using malware images. Specifically, they convert a feature vector of a binary file to a grayscale image and then pass it into a SVM classifier.

Evasion Attacks: Several white-box evasion attacks [9]–[12], [21] have been proposed in the context of malware detection. For example, Kolosnjaji et al. [9] proposed an evasion attack against deep learning based malware detection (MalConv) by padding optimized values at the end of each input. Demetrio et al. [10] proposed a similar attack against MalConv by modifying DOS header. Kreuk et al. [21] proposed a gradient-based evasion attack that targets on either the slack space or the end-of-file space. It achieved around 30% evasion rate against MalConv. Suciu et al. [11] improved the evasion rate of [21] to 70% against MalConv. Sharif et al. [15] proposed an attack to defeat neural-network based malware detectors by transforming the instructions of the binaries (i.e., binary diversification) without breaking functionalities. Specifically, it applies in-place randomization to replace opcodes inside *.text* section with semantic equivalent opcodes or uses *jump* function to move opcodes into a different section without altering original functions. However, all these attacks focus on defeating single classifiers.

VI. DISCUSSIONS

In this section, we point out a couple of mitigations that could alleviate our evasion attacks on multiple detectors.

More Detectors with Different Feature Spaces. According to our results, the evasion attacks are less effective when there

is a greater number of detectors or the formats (or essentially the feature spaces) are different across detectors. Therefore, we believe that applying a greater number of detectors, where each detector utilizes a different feature space, could mitigate the evasion rate. In addition, the combination of detectors based on both static analysis and dynamic analysis would also improve the robustness of malware detection against evasion attacks. On the other hand, how to defeat multiple detectors, where some are based static analysis and the rest are based on dynamic analysis, would be an interesting and critical problem to investigate in future work from the perspective of an attacker.

Analyzing More Bytes. According to our results, if a detector examines a greater number of bytes, it is more challenging for an attacker to defeat a detector. Therefore, analyzing a greater number of bytes can help mitigate potential evasion attacks. On the other hand, increasing the number of bytes, or essentially the input size, requires much longer training time.

VII. Conclusion

We propose a new method which can evade multiple neural-network-based detectors in malware detection without affecting the functionalities of malware. Our results suggest that although detecting malware over raw bytes can significantly simply the process of malware detection, it is vulnerable against evasion attacks, even if two or three detectors are utilized simultaneously.

Acknowledgements

The authors would like to thank the anonymous reviewers for their comments and suggestions. The UC authors were partially supported by National Science Foundation (CNS-1947913).

References

[1] E. R., J. B., J. S., R. B., B. C., and C. N, "Malware detection by eating a whole exe," in *AAAI Workshops 2018*, 2018.

[2] S. E. C. and C. G, "Activation analysis of a byte-based deep neural network for malware classification," in *2019 IEEE Security and Privacy Workshops*, 2019.

[3] M. Krcál, O. Svec, M. Bálek, and O. Jasek, "Deep convolutional malware classifiers can learn from raw executables and labels only," in *ICLR*, 2018.

[4] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th International Symposium on Visualization for Cyber Security (VizSec '11)*, 2011.

[5] D. G. Llauradó, C. Mateu, J. Planes, and R. Vicens, "Using convolutional neural networks for classification of malware represented as images," *Journal of Computer Virology and Hacking Techniques*, vol. 15, pp. 15–28, 2018.

[6] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, "Lightweight classification of iot malware based on image recognition," *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 02, pp. 664–669, 2018.

[7] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Exmaples," in *Proc. of ICLR'15*, 2015.

[8] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, and F. Roli, "Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection," in *ACM Transactions on Privacy and Security*, vol. 24, no. 4, 2021, pp. 1–31.

[9] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," *2018 26th European Signal Processing Conference (EUSIPCO)*, pp. 533–537, 2018.

[10] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," in *CEUR Workshop Proceedings*, Pisa, Italy, 2019.

[11] O. Suciu, S. E. Coull, and J. Johns, "Exploring adversarial examples in malware detection," *2019 IEEE Security and Privacy Workshops (SPW)*, pp. 8–14, 2019.

[12] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Functionality-preserving black-box optimization of adversarial windows malware," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3469–3478, 2021.

[13] C. Jing, Y. Wu, and C. Cui, "Ensemble dynamic behavior detection method for adversarial malware," *Future Gener. Comput. Syst.*, vol. 130, pp. 193–206, 2022.

[14] C. Smutz and A. Stavrou, "When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors," in *NDSS*, 2016.

[15] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, "Malware makeover: Breaking ml-based static analysis by modifying executable bytes," in *ASIACCS*, May 2021, pp. 744–758.

[16] D. Park, H. Khan, and B. Yener, "Generation & evaluation of adversarial examples for malware obfuscation," *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pp. 1283–1290, 2019.

[17] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When malware is packin' heat; limits of machine learning classifiers based on static analysis features," in *NDSS*, 2020.

[18] A. Khormali, A. A. Abusnaina, S. Chen, D. Nyang, and A. Mohaisen, "Copycat: Practical adversarial attacks on visualization-based malware detection," *ArXiv*, vol. abs/1909.09735, 2019.

[19] H. Liu, J. Dani, H. Yu, W. Sun, and B. Wang, "Advtraffic: Obfuscating encrypted traffic with adversarial examples," *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, pp. 1–10, 2022.

[20] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks," in *Proc. of Workshop on Principled Approaches to Deep Learning*, 2017.

[21] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Deceiving end-to-end deep learning malware detectors using adversarial examples," in *Workshop on Security in Machine Learning (NeurIPS)*, 2018.

[22] E. Wong, L. Rice, and J. Z. Kolter, "Fast is better than free: Revisiting adversarial training," in *ICLR*, 2020.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proc. of Advances in Neural Information Processing Systems (NIPS)*, 2012.

[25] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1800–1807, 2017.

[26] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Human-centric Computing and Information Sciences*, vol. 8, pp. 1–22, 2018.

[27] Y. Ye, T. Li, D. A. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Computing Surveys (CSUR)*, vol. 50, pp. 1 – 40, 2017.

[28] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.

[29] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "Pe-miner: Mining structural information to detect malicious executables in realtime," in *RAID*, 2009.

[30] H. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading machine learning malware detection," in *Black Hat USA 2017, July 22-27, 2017, Las Vegas, NV, USA*, 2017.

[31] D. Vasan, M. Alazab, S. Wassan, B. Safaei, and Q. Zheng, "Image-based malware classification using ensemble of cnn architectures (imcec)," *Comput. Secur.*, vol. 92, p. 101748, 2020.

[32] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang, "An intelligent pe-malware detection system based on association mining," *Journal in Computer Virology*, vol. 4, pp. 323–334, 2008.

[33] A. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static analyzer of vicious executables (save)," in *20th Annual Computer Security Applications Conference*, 2004.
[34] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *NDSS*, 2014.