MalGuise: Generating Practical Adversarial Malware against Learning-based Windows Malware Detection

Anonym

ABSTRACT

Given the widespread popularity and impressive performance of learning-based malware detection in both academia and industry, in this paper, we present a novel attack framework of `MalGuise` for effectively and efficiently evaluating the security vulnerabilities of existing learning-based Windows malware detection systems. Based on the newly proposed semantics-preserving transformation of `call`-based `redividing` that concurrently manipulates both nodes and edges of the control-flow graph (CFG) representation, `MalGuise` employs a Monte Carlo tree search (MCTS) based optimization to search for an optimized sequence of `call`-based `redividing` transformations that would be performed in the input Windows malware and then reconstructs a successful adversarial malware file under the constraints of Windows executable format, thereby preserving the same semantics as the original inputted malware. We systematically evaluate our proposed attack of `MalGuise` against three state-of-the-art learning-based Windows malware detection systems under the strict black-box setting. Evaluation results demonstrate that `MalGuise` achieves a high attack success rate of mostly over 97% and more than 90% of them generate realistic adversarial malware that keeps the same semantics as the original one. Moreover, to understand the real threats of adversarial attacks against real-world anti-viruses, we additionally demonstrate the attack effectiveness of `MalGuise` on seven commonly used commercial anti-virus products and particularly observe that the highest attack success rate among them is up to 74.97%.

1 INTRODUCTION

With the sustainable development of information technology, especially computer systems, malware (*i.e.*, short for **mal**icious soft**ware**) has been constantly developed and evolved as one of the most security threats that could perform malicious activities on computer systems, including stealing sensitive information, demanding from a large ransom, or even disrupting national critical infrastructures, just to name a few. Meanwhile, as the most widely used operating system for both individuals and organizations in the world, the Windows family of operating systems (*i.e.*, Windows) has inevitably become the most preferred target operating system of malware for potential attackers, which is generally termed as Windows malware in this paper. According to the security statistics of AV-TEST [9], in the first three quarters of 2022, it is reported that approximately 59.58 million new Windows malware was discovered and these accounted for over 95% of all malware samples that were newly discovered during the time period [8].

To defend against the ever-increasing number of security threats caused by the ever-evolving Windows malware, considerable research efforts with the latest technological advances have been made to identify and mitigate Windows malware [16, 47, 48, 61, 76], namely Windows malware detection. Basically, Windows malware detection can trace its history back to signature-based malware detection in the 1990s, which mainly blacklists suspicious malware based on a frequently updated database of known malware signatures that are previously collected and analyzed. It is apparent that signature-based malware detection cannot detect new or previously unknown malware. In the past two decades, aiming at increasing the generalization in detecting newly emerging and previously unknown malware, a variety of conventional machine learning (ML) and deep learning (DL) models are continuously explored and employed for Windows malware detection, which is called the *learning-based Windows malware detection* in this paper. Due to the high learning capacities and capabilities of ML/DL models, learning-based Windows malware detection is demonstrated to generalize to detect other newly emerging and even zero-day malware, thereby becoming a critical building block of current mainstream anti-virus products in the competitive market [36, 53].

However, recent advanced studies have demonstrated that ML/DL models are highly and inherently vulnerable to adversarial attacks [46], by which the adversary maliciously creates adversarial examples as the input to trigger the target ML/DL model to malfunction, *e.g.*, make incorrect predictions or decisions. Since being initially proposed in 2015 [26], adversarial attacks have been successfully used to explore security threats in the domain of computer vision like autonomous driving [63], and further have been extended to many other domains, such as audio recognition, natural language processing, and graph analysis [12, 74]. Furthermore, considering the massive and advanced usage of learning-based Windows malware detection in both academia and industry [36, 53, 76], one natural research question that arises is, *is it feasible to generate practical adversarial malware against existing learning-based Windows malware detection for evaluating its security vulnerabilities?*

To answer this research question, in this paper, we attempt to explore an adversarial attack under the realistic black-box setting for effectively and efficiently generating practical adversarial malware. Towards this, we identify two key challenges (*i.e.*, **C#1** and **C#2**) that need to be addressed as follows.

- **C#1**: How to generate practical adversarial malware that can guarantee the same semantics as the original one as well as cannot be easily noticed by defenders? Existing adversarial attacks mainly consist of ① adding irrelevant API calls [4, 17, 29, 72], ② partially or globally manipulating raw bytes [6, 24, 39, 40, 50, 66], and ③ manipulating the control-flow graph (CFG) representation of malware by injecting semantic `nop`s [78]. We argue that the first two adversarial attacks (*i.e.*, ① and ②) are either impractical to generate adversarial features rather than practical adversarial malware, or strictly limited against a specific malware detection like MalConv. Although demonstrating better scalability against other malware detection based on CFG, the third adversarial attack (*i.e.*, ③) only considers a coarse-grained transformation

ACM CCS 2023, Nov. 26–30, 2023, Copenhagen, Denmark
2023. ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

that manipulates the nodes of CFG, which is quite noticeable to be mitigated by defenders. Therefore, to address **C#1**, our idea is to devise a more fine-grained and less noticeable transformation towards the CFG representation that not only manipulates the nodes (*i.e.*, instruction blocks) but also its edges, *i.e.*, the control-flow relationships between two instruction blocks.

- **C#2**: How to efficiently search in the large and discrete space of Windows malware under the strict black-box setting such that the optimized adversarial malware can bypass learning-based Windows malware detection? We investigate the state-of-the-art black-box adversarial attacks against Windows malware detection, typically including gradient estimations with surrogate models [4, 39, 72], evolutionary algorithms [17, 50], and reinforcement learning [6, 78]. Apparently, those adversarial attacks based on gradient estimations rely heavily on prior information (*e.g.*, training data, model architecture) about the target systems. On the other hand, adversarial attacks based on evolutionary algorithms and reinforcement learning are computationally expensive in the vast and discrete space of malware. Therefore, we address **C#2** by employing a Monte Carlo tree search (MCTS) based optimization to effectively and efficiently search the optimized adversarial malware that can successfully bypass the target malware detection.

Overall, in this paper, we propose a novel and practical framework of adversarial attacks against learning-based Windows malware detection under the black-box setting, namely `MalGuise`. As depicted in Fig. 2, `MalGuise` mainly consists of three backbone phases, *i.e.*, 1) adversarial transformation preparation, 2) MCTS guided searching, and 3) adversarial malware reconstruction. Specifically, `MalGuise` first represents the input Windows malware as CFG with the disassembled assembly files and presents a novel semantics-preserving transformation of `call`-based `redividing`, which lays the base for manipulating both nodes and edges of the CFG representation. Then, we employ an MCTS algorithm that can effectively and efficiently guide `MalGuise` to search for an optimized sequence of `call`-based `redividing` transformations among the vast and discrete space of malware. Finally, based on the optimized sequence of transformations, `MalGuise` reconstructs a successful adversarial malware under the constraints of Windows executable format, which guarantees that its format, executability, and maliciousness remain the same as the original Windows malware inputted.

In order to demonstrate the attack effectiveness of the proposed adversarial attack framework of `MalGuise`, we systematically evaluate the security vulnerabilities of three representative learning-based Windows malware detection systems (*i.e.*, MalGraph, Magic, and MalConv) compared with two state-of-the-art baseline adversarial attacks on the wild Windows dataset that contains hundreds of thousands of malware and goodware. The evaluation results show that `MalGuise` is agnostic to the target learning-based Windows malware detection systems and achieve a high attack success rate of mostly over 97%. Meanwhile, we automatically and empirically verify at scale that `MalGuise` could generate realistic adversarial malware files with a probability of more than 90%, whereas prior adversarial attacks can only generate with a probability of less than 50% or even fail to generate realistic adversarial malware files. Furthermore, to understand the real security threats of adversarial attacks against real-world anti-virus products, we additionally demonstrate the attack effectiveness of `MalGuise` on seven commonly used commercial anti-virus products and particularly observe that the attack success rate among them reaches a range of 5.64% to 74.97%. To summarize, we highlight our key contributions as follows.

- To understand and inspect the security vulnerabilities of existing learning-based Windows malware detection and advanced commercial anti-virus tools in real world, we propose a general and practical adversarial attack framework of `MalGuise` under the strict black-box settings.
- To the best of our knowledge, `MalGuise` is the first to apply a more fine-grained manipulation towards the CFG representation of Windows executable, which not only changes the nodes of CFG (*i.e.*, instruction blocks) but also the edges (*i.e.*, the control-flow relationship between two instruction blocks).
- Evaluations demonstrate that `MalGuise` could not only successfully bypass state-of-the-art learning-based Windows malware detection with an attack success rate of mostly over 97%, but also can bypass seven commercial anti-virus products with an attack success rate in a range of 5.64% to 74.97%.

2 PRELIMINARIES & THREAT MODEL

In this section, we introduce the necessary preliminaries on learning-based malware detection and our threat model.

2.1 Preliminaries on Learning-based Malware Detection Systems

The purpose of learning-based malware detection is to discriminate whether the suspicious executable is malicious or not. As shown in Fig. 1, we give an overview framework of learning-based malware detection systems.

First, as ML/DL models most only accept numeric vector data as inputs, the training samples and testing samples of executables are pre-processed by feature engineering before being inputted into ML/DL models. Formally, feature engineering can be formulated as $\phi : \mathcal{Z} \to \mathcal{X} \subseteq \mathbb{R}^n$, which actually produces a n-dimension feature vector $x \in \mathbb{R}^n$ in the *feature-space* $\mathcal{X}$ (*i.e.*, $x \in \mathcal{X}$) for a given executable z in the *problem-space* $\mathcal{Z}$, *i.e.*, $z \in \mathcal{Z}$. Then, based on all the training samples, the ML/DL model is employed to learn a classification boundary between malware and goodware via training a binary classifier as the learning-based malware detection system $f : \mathcal{Z} \to \mathcal{Y}$. That is, given an executable $z \in \mathcal{Z}$, f can predicts a corresponding label of y in the *label-space* $\mathcal{Y}$ (*i.e.*, $y \in \mathcal{Y}$), such that $y = f(z) \in \{0, 1\}$, in which $y = 0$ denotes goodware while $y = 1$ denotes malware. Additionally, for simplifying the notation in learning-based malware detection [43, 48], we denote the malware detection system that can return the malicious probability as $g : \mathcal{Z} \to \mathbb{R}$, in which $g(z)$ denotes the predicted malicious probability for the given z and the opposite benign probability is naturally inferred as $1 - g(z)$.

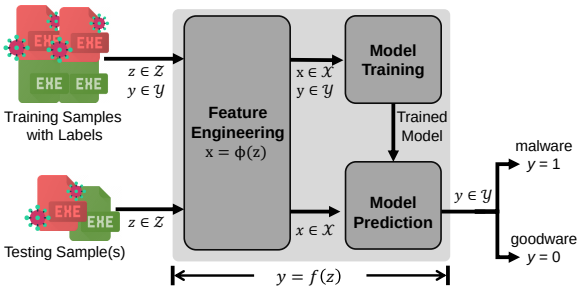


Figure 1: Overview of learning-based malware detection.

2.2 Threat Model

Following the most widely used framework of modeling threats in adversarial machine learning [12, 58], we report our threat model in terms of the adversary's goal, knowledge, and capability as follows.

Adversary's Goal. In this paper, we focus on *evasion attacks*, which can be generally categorized into *un-targeted attacks* and *targeted attacks* in multi-class settings like image recognition. In particular, targeted attacks mean to cause an incorrect classification towards a specified class, while un-targeted attacks mean to cause an incorrect classification without any specificity. However, in the context of our attack scenario that targets malware detection systems (*i.e.*, binary classification), the differences between *un-targeted* and *targeted attacks* are lost [64]. This is because, for the adversary that aims to compromise malware detection systems, it is hugely profitable to misclassify malware as goodware, but not vice versa. Therefore, the primary goal of the adversary is to generate a practical adversarial malware file z_{adv} from a legitimate malware file $z \in \mathcal{Z}$ (*i.e.*, $f(z) = 1$) with minimal efforts, such that $z_{adv} \in \mathcal{Z}$ can not only be misclassified by f (*i.e.*, $f(z_{adv}) = 0$) but also preserve the same semantics (including format, executability, and maliciousness) as z [48].

Adversary's Knowledge and Capability. We start with an adversary who intends to perform the classic *zero-knowledge black-box attack* [51, 58] against the target malware detection system. This indicates that the adversary has no prior information on the target system about the training data, the extracted feature set, the employed learning algorithm with parameters, and the model architecture along with corresponding weights. However, it should be clarified that the zero-knowledge black-box attack still has some minimal information on the target system, including what specific task is the target system employed to perform (*e.g.*, static or dynamic malware detection), which kind of feature type is employed to represent the input software file (*e.g.*, image, sequence or graph), and the querying feedback from the target system. To be specific, in our attack scenario against learning-based Windows malware detection systems, we restrict the adversary to only knowing that the feature type employed by our target system is mainly static control-flow information like CFG. Furthermore, the adversary can obtain the prediction label $f(z)$ along with or without the prediction probability $g(\phi(z))$ by inputting an arbitrary executable z into the target system.

Considering that the generated adversarial malware file must preserve the same semantics as the original malware, we thus resort to the problem-space attack [58] that attempts to apply *semantics-preserving transformations* over the original input malware so that the transformed malware is misclassified as goodware. That means the adversary has the capability of manipulating the input executable without destroying its semantics, including format, executability, and maliciousness.

3 DESIGN OF MALGUISE

3.1 Problem Formulation

In this paper, the primary goal of the adversary is to generate a practical adversarial malware file $z_{adv} \in \mathcal{Z}$ from a legitimate malware $z \in \mathcal{Z}$, such that the generated z_{adv} not only being misclassified as goodware but also preserves the same semantics as the original malware z. To preserve that z_{adv} has the same semantics as z, we devise semantics-preserving transformations in the problem space to transform the original malware step by step until a successful adversarial malware file is generated. Recall that we consider the classic *zero-knowledge black-box attack* against the learning-based malware detection system, in which the adversary can obtain the predicted label with or without the malicious probability. Thus, as formulated in Eq. (1), we start to define the first black-box attack scenario with the malicious probability $g(\cdot)$ by maximizing the difference of the predicted malicious probabilities between the original malware z and the generated adversarial malware z_{adv}, meaning to reduce the malicious probability that z_{adv} is predicted to be malicious as much as possible.

$$\arg\max_{\mathbf{T}} g(z) - g(z_{adv}) \quad \blacktriangleright \text{ when knowing } f(\cdot) \text{ with } g(\cdot) \tag{1}$$

$$\arg\min_{\mathbf{T}} f(z_{adv}) \quad \blacktriangleright \text{ or knowing } f(\cdot) \text{ without } g(\cdot) \tag{2}$$

$$\text{s. t.:} \quad f(z) = 1 \tag{3}$$

$$f(z_{adv}) = 0 \tag{4}$$

$$z_{adv} = \mathbf{T}(z) \in \mathcal{Z} \tag{5}$$

$$\mathbf{T} = T_1 \circ T_2 \circ \cdots \circ T_n \in \mathbb{T} \tag{6}$$

in which $T \in \mathbb{T}$ denotes one of atomic transformations that can transform one executable into another semantics-preserving executable; $\mathbf{T} = T_1 \circ T_2 \circ \cdots \circ T_n$ denotes a finite and ordered sequence of n transformations that z can be step-by-step transformed into an adversarial malware, *i.e.*, $z_{adv} = \mathbf{T}(z) \in \mathcal{Z}$.

Similarly, we further extend our attack scenario to a strict black-box setting where the adversary can only obtain the predicted label $f(\cdot)$ without the predicted probabilities, and further define it in Eq. (2) by simply minimizing the predicted label of z_{adv} to 0 since 0 means it is goodware for the target malware detection system f.

3.2 Overview of `MalGuise` Framework

We present the overview framework of `MalGuise` in Fig. 1, which mainly consists of three backbone phases, *i.e.*, adversarial transformation preparation, MCTS guided searching, and adversarial malware reconstruction. In general, building on the CFG representation of Windows malware, we first propose a novel and practical semantics-preserving transformation of `call`-based `redividing`, which prepares the groundwork for manipulate both nodes and edges of CFG in more fine-grained fashion. Then, we use the MCTS algorithm to search for an optimized sequence of transformations

that could be applied to transform the original CFG into the corresponding adversarial CFG representation under the black-box setting. Finally, abiding by the constraints of Windows executable specifications, we further reconstruct the adversarial CFG representation into a successful adversarial malware file. As below, we elaborate on each of three backbone phases in the following subsections of § 3.3, § 3.4, and § 3.5, respectively.

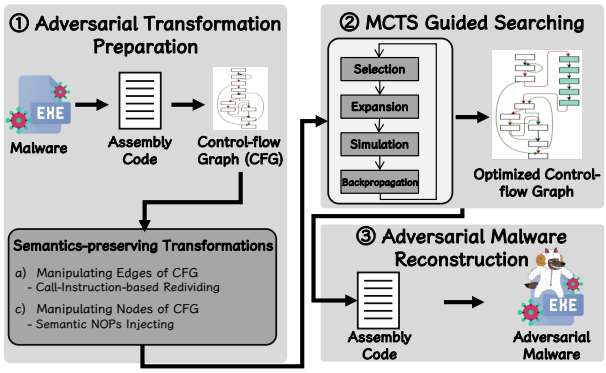


Figure 2: Overview of `MalGuise`, consisting of three backbone phases: adversarial transformation preparation, MCTS guided searching, and adversarial malware reconstruction.

3.3 Adversarial Transformation Preparation

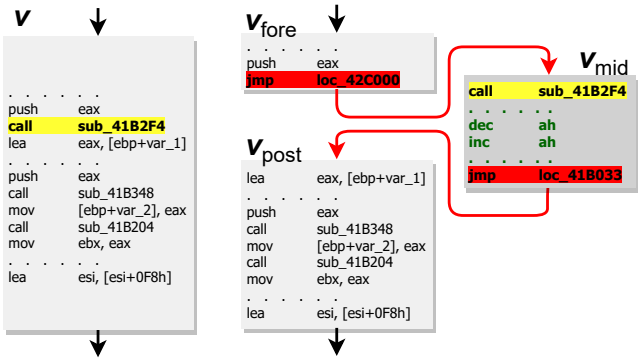
3.3.1 Representing the Windows malware as CFG. To prepare adversarial transformations that manipulate the given malware file while preserving its semantics, we start by first representing the malware file as CFG, in which each node denotes a basic block and each edge denotes a control-flow path between two basic blocks during the execution. Specifically, one basic block consists of a maximum sequence of assembly instructions without branching instructions (*e.g.*, "jmp␣[address]") except for the last instruction, and each assembly instruction has one opcode and a certain number of operands. Actually, the main reason for us to represent Windows malware as CFG is two folds. First, CFG encapsulates the intrinsic control flows during the execution of the given Windows malware, which actually contains rich semantic and structural information of assembly instructions. Therefore, if we can manipulate both nodes and edges of the CFG representation, we can change both semantics and structural information to further generate possible adversarial malware in a more fine-grained fashion, thereby making the generated adversarial malware less easy to be noticed. Second, the representation of CFG actually has been widely employed in a variety of applications of software analysis, and CFG-based malware detection is extensively proven to be technically advanced and highly effective in both industry and academia [47, 75]. Therefore, if we could successfully attack these advanced CFG-based malware detection systems as representative cases, it could clearly demonstrate the maximum effectiveness of our proposed `MalGuise`.

3.3.2 Semantic-preserving Transformations. However, if we transform one executable into another executable by directly manipulating its CFG, it is extremely easy to cause various unexpected software errors like addressing or processing errors, so that the transformed executable cannot be executed properly or even crashes immediately. Therefore, in order to guarantee the transformed executable has the same semantics as the original one, the proposed semantics-preserving transformation should satisfy two principles: 1) one the one hand, the proposed transformation should be able to manipulate not only the instruction information of the basic block itself (*i.e.*, nodes), but also its structural information of control-flow paths (*i.e.*, edges) in CFG; 2) on the other hand, the proposed transformation must not change the executing logic of the given executable, thereby ensuring the transformed executable can be executed properly with the same behaviors as the original executable.

To account for the two aforementioned principles, we propose a novel semantics-preserving transformation for the CFG representation of executables, namely `call-based redividing`, which mainly *redivides* one of the basic blocks that contain at least one assembly instruction with "call" (*e.g.*, "call␣subroutine") for concurrently manipulating both instruction information of the basic block itself and structural information of control-flow path in the corresponding CFG. To be specific, for one given executable, the proposed `call-based redividing` first annotates all available basic blocks with "call" instruction(s), which serve as the base for subsequent transformations. Then, supposing there is a basic block $\mathbf{v}$ with "call" instruction(s), `call-based redividing` considers the instruction where the "call" operator is located as the dividing line, and attempts to redivide the original basic block v into a combination of three new and consecutive basic blocks (*i.e.*, the fore-basic-block $\mathbf{v}_{fore}$, the post-basic-block $\mathbf{v}_{post}$, and the mid-basic-block $\mathbf{v}_{mid}$).

To be specific, by `call-based redividing`, $\mathbf{v}_{fore}$ mainly contains a sequence of consecutive instructions before the "call" instruction and is additionally appended with an added "jmp" instruction that can jump to the basic block of $\mathbf{v}_{mid}$. $\mathbf{v}_{post}$ contains a sequence of consecutive instructions after the "call" instruction without any changes. $\mathbf{v}_{mid}$ initially starts with the specified "call" instruction and ends with a newly added "jmp" instruction that jumps to the basic block of $\mathbf{v}_{post}$. Finally, to avoid the basic block of $\mathbf{v}_{mid}$ being easily noticed by defenders due to having only two instructions, `call-based redividing` further attempt to enrich the assembly instructions in $\mathbf{v}_{mid}$ by injecting semantic nops [19] between the "call" instruction and the "jmp" instruction. In particular, semantic nops denotes a sequence of consecutive instructions that do not have any effect on the memory and register during execution [19, 50]. Here, we specifically employ a context-free grammar developed in [50] to generate diverse semantic nops to be injected.

As illustrated in Fig. 3, we first show one basic block of the latest "LockBit 3.0" ransomware that has been the most active ransom gang during the third quarter of 2022 [49] as a representative example in Fig. 3(a), and further highlight one of the "call" instructions in **yellow**, which is considered as the dividing line by the `call-based redividing` transformation. After performing the transformation, Fig. 3(b) shows the transformed composite of three consecutive basic blocks, in which the ends of two basic blocks (*i.e.*, $\mathbf{v}_{fore}$ and $\mathbf{v}_{mid}$) are two newly added jump instructions with **red** highlighted, and those assembly instructions between the "call" instruction and the jump instruction in $\mathbf{v}_{mid}$ are newly added semantic nops in **green**.



(a) A basic block in CFG before transformations.

(b) A composite of three basic blocks in CFG after the call-based redividing transformation.

Figure 3: Illustration of call-based redividing transformation that semantics-preserving redivides one basic block in the "LockBit 3.0" ransomware (*i.e.*, Fig. 3(a)) into a composite of three consecutive basic blocks (*i.e.*, Fig. 3(b)) without interfering with all others.

3.4 MCTS Guided Searching

Recalling our adversarial attack formulated in § 3.1 and the transformation of call-based redividing that is defined upon the CFG representation (*i.e.*, $\phi(z)$) of a given Windows malware (*i.e.*, z) in § 3.3, we further decompose our adversarial attack into finding an optimized sequence of transformations (*i.e.*, $\mathbf{T} = T_1 \circ T_2 \circ \cdots \circ T_N \in \mathbb{T}$) that could first consecutively transform the original CFG representation $\phi(z)$ into a successful adversarial CFG representation $\mathbf{T}(\phi(z))$ under the black-box setting and then reconstruct the final adversarial malware file $z_{adv} = \phi^{-1}(\mathbf{T}(\phi(z)))$. Below, we focus on describing how to find an optimized sequence of transformations $\mathbf{T}$ under black-box settings in this subsection, and leave the introduction of the remaining adversarial malware reconstruction to the next subsection of § 3.5.

In essence, equipped with the proposed transformation of call-based redividing, the optimal solution we are solving here is an optimized sequence of transformations $\mathbf{T}$ of length n, and each transformation (*i.e.*, $T_i (1 \leq i \leq n) \in \mathbb{T}$) involves two decision-making processes: 1) the first is to select one of all available call instructions to be redivided, and it should be noted that every call instruction can be repeatedly and infinitely selected in a recursive manner; 2) the second is to determine proper semantic nops to be injected, and these semantic nops can be generated infinitely by the employed context-free grammar [50]. In short, the whole process of optimizing a sequence of transformation $\mathbf{T}$ in MalGuise requires exploring and optimizing in an infinite and discrete space under black-box settings.

Motivated by the great success of Monte Carlo Tree Search (MCTS) [23] in solving the long-standing challenging problem of computer Go [23, 25, 65] and other difficult problems of optimization and planning with little or no domain knowledge [14], we propose an MCTS guided searching algorithm to explore and optimize a sequence of semantics-preserving transformations $\mathbf{T}$ in the *infinite* and *discrete* space within limited computational budget under *black-box* settings. To be specific, MCTS is a heuristic searching method that incorporates the precision of tree search with the generality of Monte Carlo random sampling to find the optimal solution for large-scale optimization and planning problems [14, 23, 57]. MCTS progressively builds and searches an asymmetric game tree, in which each node in the game tree represents a state of CFG that stores statistics of a reward value and the number of visits, and each edge represents an action that transforms a parent state node into the child state node.

Algorithm 1: MCTS Guided Searching Algorithm

Input : a given malware z, malware detection system f with the CFG representation ϕ_{cfg}, max length N, simulation number S, computation budget C.

Output: the adversarial CFG representation x_{adv}.

```
1  begin
2      x ← φ_cfg(z) ▶ represent z as CFG;
3      I_call ← GetCallInsts(x) ▶ get all available call instructions;
4      v ← InitMCTSNode(x, I_call);
5      for i ← 1 to N do
6          for j ← 1 to C do
7              p ← random(0, 1);
8              if p < 0.5 then
9                  v_selected ← Selection(v);
10             else
11                 v_selected ← Expansion(v);
12             end
13             reward ← Simulation(v_selected, f, S);
14             BackPropagation(v_selected, reward);
15         end
16         v_node ← ChildWithHighestReward(v);
17         v ← v_node;
18         x_adv ← v.x;
19         if bypass(f, x_adv) == True then
20             return x_adv
21         end
22     end
23 end
```

As shown in Algorithm 1, we detail the main procedure of MCTS guided searching algorithm for finding an optimized sequence of transformations $\mathbf{T}$ that takes the graph representation $\phi(z)$ as the input and outputs an adversarial graph representation $x_{adv} = \mathbf{T}(\phi(z))$. In the beginning, we obtain both the CFG representation x of the given malware z and all corresponding available instructions $\mathbb{I}_{\text{call}}$ for initializing the root node of MCTS (*line 2–4*). Meanwhile, we limit the maximum length of the optimized transformation sequence to N (*line* 5–22) and limit the maximum number of iterations of MCTS to C, *i.e.*, computational budget (*line* 6–15). As for the MCTS optimization process, we follow the four typical steps (*i.e.*, Selection, Expansion, Simulation, and Backpropagation) in the MCTS-guided searching algorithm (*line 7–14*). It should be noted, as the transformation of call-based redividing can be performed unlimitedly, the game tree of MCTS can be unlimitedly expanded downwards, *i.e.*, Expansion. Therefore, we adaptively force to select the most "promising" child node (*i.e.*, Selection) in the established game tree via a simple random sampling. After C iterations of MCTS optimizations, we can thus obtain the child node v_{node} with the largest reward value, and further judge whether the

corresponding CFG representation x_{adv} of the child node could bypass the target malware detection system f (*line 16–21*). In particular, if x_{adv} cannot be bypassed, we continue to use v_{node} as the root node for the next round of MCTS until reaching the maximum computation budget C, otherwise return x_{adv} as the adversarial CFG representation. For the sake of simplicity, we leave further technical/implementation details of MCTS to Appendix A.

3.5 Malware Reconstruction

After determining an optimized sequence of `call`-based `redividing` transformations with length N (*i.e.*, $\mathbf{T} = T_1 \circ T_2 \circ \cdots \circ T_N$) and obtaining a corresponding adversarial CFG representation x_{adv} in § 3.4, we further reconstruct the original Windows malware z into the final adversarial Windows malware z_{adv} accordingly, namely malware reconstruction. Generally, in order to adhere to the specifications of Windows executables and avoid unexpected errors (*e.g.*, addressing errors) that may arise during malware reconstruction, we first add a new section to the original malware z (in § 3.5.1) and then apply the sequence of transformations $\mathbf{T}$ of length N to z with the newly added section (in § 3.5.2), resulting in the final adversarial Windows malware z_{adv}.

3.5.1 Adding a new section. Firstly, to ensure that there is sufficient and proper space for inserting all those instructions when applying the optimized sequence of `call`-based `redividing` transformations, we need to determine the size of the new section that will be added to the original malware z. Assuming that the total size of all those instructions in all $\mathbf{V}_{mid}$ from all their `call`-based `redividing` transformations is Δ^{new}, the size of the new section to be added can be computed as $\Delta^{new}_{align} = \text{RoundUp}(\Delta^{new}, page\_size)$, which guarantees that the size of the newly added section must be a multiple of the architecture's page size (*i.e.*, *page_size*) for facilitating the executable to be loaded into the memory [56]. In particular, the *page_size* for both Intel x86 and MIPS is 4096 bytes and for Itanium is 8192 bytes. Accordingly, as we append the new section right after the last section in the original executable, the starting address A^{new} of the new section is essentially the ending address of the original last section, which can be computed from the header information of the original executable. After both the size Δ^{align}_{new} and the address A^{new} of the newly added section are determined, we take further actions according to the specifications of Windows executables to achieve the purpose of adding a new section to the original malware. Examples of all such subsequent actions include: i) modifying both "number of sections" and "size of image" in PE Header; ii) adding a new entry in the Section Table with both size and address of the newly added section, *etc.* More details about how to add a new section to the original malware can be found in Appendix B.1 with Algorithm 4.

3.5.2 Applying transformations to the malware. Equipped with the newly added section in the original malware z, we further apply the optimized sequence of transformations of length N (*i.e.*, $\mathbf{T} = \{T_1, T_2, ..., T_N\}$) iteratively. To be specific, each transformation can be represented as $T_k = \{\mathbb{I}^{\text{call}}_k, A^{\text{call}}_k, \mathbb{I}^{semnops}_k\}$, $k = 1, 2, \cdots N$, in which $\mathbb{I}^{\text{call}}_k$ and A^{call}_k denotes the instruction and address of the selected "call" instruction by the k-th `call`-based `redividing` transformation, and $\mathbb{I}^{semnops}_k$ denotes the corresponding semantic nops to be injected. To avoid unexpected errors that might be caused by applying those transformations, for each transformation T_k, we first replace the selected call instruction of $\mathbb{I}^{call}_k$ with a new jmp instruction (*i.e.*, "jmp_$[A^{new}_k]$"), which transfers the control-flow to a new address for the k-th transformation in the new section, *i.e.*, A^{new}_k. Subsequently, in the address A^{new}_k of the newly added section, we deposit $\mathbb{I}^{call}_k$ and $\mathbb{I}^{semnops}_k$ in order, and inject a new jmp instruction (*i.e.*, "jmp_$[A^{\text{call}}_k + \text{GetSize}(\mathbb{I}^{\text{call}}_k)]$") in the end, which actually transfers the control-flow backwards to the next instruction of $\mathbb{I}^{call}_k$ in the original section. It is noted that, when $k = 1$, A^{new}_1 is equal to the starting address of the new section (*i.e.*, $A^{new}_1 = A^{new}$). Otherwise, each new starting address of A^{new}_k is arranged one by one. After all of N transformations are applied as above, we can reconstruct the final adversarial malware z_{adv} that preserves the same semantics as the original malware z. Details can be found in Appendix B.2 with Algorithm 5.

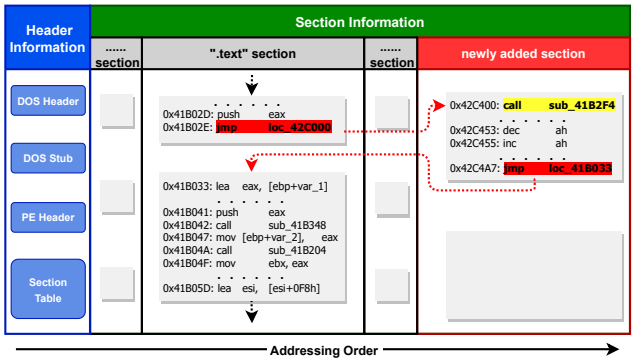


Figure 4: The conceptual layout of the reconstructed adversarial Windows malware file for the "LockBit 3.0" ransomware, highlighting the corresponding reconstruction operations for the one employed transformation in Fig. 3.

Again, taking one `call`-based `redividing` transformation that is employed in the "LockBit 3.0" ransomware in Fig. 3 as an example, we further present the conceptual layout of the corresponding adversarial malware file of "LockBit 3.0" in Fig. 4. In particular, the newly added section with red color is placed at the end of all original sections, and those instructions in between the address of $0x42C400$ and $0x42C4A7$ therein are the combination of $\mathbb{I}^{call}_k$, $\mathbb{I}^{semnops}_k$ and a new jmp instruction (with red color) to be injected by the determined `call`-based `redividing` transformation. Furthermore, for the transformed "call" instruction in the original ".text" section, we just replace it with a new "jmp" instruction of "jmp_[loc_42C000]" (with red color), in which $0x42C000$ is the starting address of this transformations in the new section. In short, it is worth mentioning that, just applying two optimized transformations of `call`-based `redividing` to the well-known "LockBit 3.0" ransomware could generate a corresponding adversarial LockBit, which can successfully bypass three state-of-the-art learning-based malware detection (*i.e.*, MalGraph, Magic, MalConv) and three mainstream anti-virus products (*i.e.*, Avast, AVG, Kaspersky).

4 EVALUATION

In this section, experiments are conducted to answer the following four research questions:

- **RQ1 (Attack Performance)**: What is the attack performance of `MalGuise` against learning-based Windows malware detection systems?
- **RQ2 (Impacting Factors)**: What could affect the attack performance of `MalGuise`?
- **RQ3 (Utility Performance)**: Can the adversarial malware generated by `MalGuise` preserve the original semantics?
- **RQ4 (Real-world Performance)**: Is `MalGuise` effective against real-world anti-virus products?

4.1 Evaluation Setup

4.1.1 Dataset. We evaluate the proposed `MalGuise` with a mixed wild dataset with over 200,000 Windows executables that contain both malware and goodware. In general, we follow the same data collection approach as it is in [47]. To be specific, all Windows malware is collected from the academic malware sample repository provided by VirusTotal [73]. It is worth noting that, to ensure the dataset quality and avoid possible biases in labeling malware, we double confirm that any provided suspicious executable is malware only if it is detected as malware by more than 2/3 of all anti-virus engines (*e.g.*, Macfee [52], Kaspersky [35]) in VirusTotal. On the other hand, for Windows goodware, as there are no publicly dataset available, we adopt the most commonly used data collection approach [33, 40, 45, 47, 60], which starts by deploying a clear virtual machine with a operating system of Windows 10 Pro and then utilizing the 360 software manger [1] to install as much inbuilt software as possible. As a result, we collect all these installed executables as Windows goodware in our datasets. Furthermore, to focus on the security vulnerability of the learning-based Windows malware detection model itself and exclude the influence of whether executables in the dataset are packed or not, we employ multiple standard unpacker tools (*e.g.*, UPX [69], PEiD [32], CAPE [37]) to unpack those malware and goodware in the dataset until they are no longer detected as packed [3, 47]. Overall, as summarized in Table 1, we finally obtain a mixed wild dataset of 210,251 Windows executables with 101,641 malware and 108,610 goodware, and split it into three disjoint training/validation/testing datasets.

Table 1: Summary statistics of the evaluating dataset.

Dataset	Training	Validation	Testing	Total
Malware	81,641	10,000	10,000	101,641
Goodware	88,610	10,000	10,000	108,610
Total	170,251	20,000	20,000	210,251

4.1.2 Target Windows Detection Systems. We demonstrate our proposed attack of `MalGuise` on three state-of-the-art learning-based Windows malware detection systems and seven real-world anti-virus products.

Learning-based Windows Malware Detection & Implementation. We first evaluate our proposed `MalGuise` on three state-of-the-art learning-based Windows malware detection systems, *i.e.*, MalGraph [47], Magic [75], and MalConv [60]. In particular, as previously introduced in § 2.1, MalGraph and Magic are two learning-based Windows malware detection on the basis of abstract graph representation. MalGraph [47] first represents an executable with a novel hierarchical abstract graph that combines function call graph and CFG, and then employs graph neural network to detect Windows malware. Magic [75] uses a similar framework of learning-based malware detection but only represents the executable with CFG. Differently, MalConv is the most representative Windows malware detection based on raw bytes, which directly represent the executable with a sequence of raw bytes in integers and then feed them into a CNN-based classifier.

Furthermore, to reduce possible evaluation biases, we directly adopt the publicly available implementation as well as the hyperparameters of the three malware detection systems in their own papers, and evaluate their detection performance with three commonly used metrics, *i.e.*, AUC [13], TPR/FPR [13], and balanced Accuracy (bACC) [71] (more details can be found in Appendix C.1). As shown in Table 2, in terms of AUC, the overall performance of all three target systems is higher than 99%. Moreover, even for the case where FPR is at an extremely low level of 0.1%, the accuracies of bACC are still as high as 96.36%, 94.59%, and 93.22% for MalGraph, Magic, and MalConv, respectively. In short, the above evaluations show similar experimental results to as them in the original paper, demonstrating excellent performance in detecting Windows malware.

Table 2: The overall performance of three learning-based Windows malware detection systems.

Target Models	AUC (%)	FPR = 1%		FPR = 0.1%	
		TPR (%)	bACC (%)	TPR (%)	bACC (%)
MalGraph	99.94	99.34	99.18	92.78	96.36
Magic	99.89	99.02	99.02	89.28	94.59
MalConv	99.91	99.22	99.12	86.54	93.22

Real-world Anti-virus Products. To further demonstrate the effectiveness of `MalGuise` in the real world, we additionally employ seven anti-virus tools (*i.e.*, McAfee [52], Comodo [21], Avast [10], AVG [11], Kaspersky [35], ClamAV [20], and Microsoft Defender ATP [54]) that are popular with both individual and enterprise users as the target Windows detection systems. Among them, McAfee, Comodo, Avast, AVG, and Kaspersky are five award-winning commercial anti-virus products recommended in [67]. ClamAV is the most popular open-sourced anti-virus engine that can be employed in all major operating systems, including Linux, Windows, and MacOS. Microsoft Defender ATP is an ML-based security protection tool that is widely used on the Windows platform [42, 55].

4.1.3 Baseline Adversarial Attacks. To demonstrate the effectiveness of our proposed `MalGuise` against Windows malware detection, we compare it with two baseline adversarial attacks, *i.e.*, Malware Makeover (MMO) [50] and SRL [78]. In particular, MMO actually is a white-box adversarial attack that uses a gradient-based optimization to guide binary-diversification tools for manipulating the raw bytes of Windows malware. SRL is a black-box adversarial attack that trains a deep agent of reinforcement learning to iteratively inject semantic NOPs into the CFG representation of

Windows malware until the modified CFG can bypass the target malware detection. It is worth noting that, SRL only generates the adversarial CFG features rather than adversarial malware executable, it thus is not applicable to MalConv which requires inputs of raw bytes. To facilitate fair comparisons with MalGuise, we follow the same experimental setting of both MMO and SRL in their original papers and give more details in Appendix C.2.

4.1.4 Evaluation Metrics of Attack Success. To evaluate the performance of our proposed MalGuise, we use two metrics of attack success as follows.

1) **Attack Success Rate (ASR)** is the most commonly used metric in evaluating adversarial attacks [44, 46, 47]. Suppose there is a set of candidate malware samples $\mathbf{Z}$ to be evaluated, ASR is defined as the ratio of generated adversarial malware that successfully bypasses the target system (*i.e.*, $f(z_{adv}) = 0$) among all evaluated malware samples (*i.e.*, $f(z) = 1$) in our evaluation as follows.

$$ASR = \frac{|f(z) = 1 \wedge f(z_{adv}) = 0|}{|f(z) = 1|}, \ \forall z \in \mathbf{Z} \tag{7}$$

in which $|\cdot|$ denotes the number of counts that meet the condition. The larger ASR is, the more effective the adversarial attack is.

2) **Semantics Preservation Rate (SPR)**. Although successfully bypassing the target system, the generated adversarial malware might not preserve the same semantics as the original malware, *i.e.*, cannot be executed or lose the original malicious behaviors. In particular, we define SPR as the ratio of adversarial malware with the original semantics preserved among all evaluated adversarial malware as follows.

$$SPR = \frac{|Sem(z, z_{adv}) = 1|}{|f(z) = 1 \wedge f(z_{adv}) = 0|}, \ \forall z \in \mathbf{Z} \tag{8}$$

in which $Sem(z, z_{adv}) = 1$ denotes z_{adv} and z have the same semantics (*i.e.*, z_{adv} preserves the same semantics as z), while $Sem(z, z_{adv}) = 0$ denotes z_{adv} does not preserve the same semantics. In § 4.2.3, we will elaborate how on to measure it empirically.

4.1.5 Implementation Details. Our MalGuise is primarily implemented in Python and evaluated on a PC equipped with 20 Intel Xeon 2.10GHz CPU, 128 GB memory, and 4 NVIDIA GeForce RTX 3090. To be specific, in the first phase (*i.e.*, adversarial transformation preparation) of MalGuise, we employ the IDAPython plugin in IDA Pro 6.4 [28] to disassemble Windows executables and represent them with CFGs. By default, in the MCTS optimization, we set the max length N to 6, set the computational budget C to 40, set the simulation number S to 1, and limit the size of injected semantic nops to no more than 5% of the original malware's size. To reconstruct adversarial malware, we mainly employ two Python libraries of pefile [15] and LIEF [59] to parse and manipulate Windows executables. It is worth noting that, those injected semantic nops can be generally divided into four categories: arithmetic instructions (*e.g.*, "add_eax, 1; sub_eax, 1"), logical instructions (*e.g.*, "add_eax, eax"), comparison instructions (*e.g.*, "cmp_eax, eax"), and data transfer instructions (*e.g.*, "push_eax; pop_eax").

4.2 Evaluation Results & Analysis

4.2.1 Answer to RQ1: The attack performance of MalGuise *against learning-based Windows malware detection systems.* To demonstrate the attack effectiveness of our proposed MalGuise, we empirically evaluate MalGuise by measuring and comparing the ASR performance on all 10,000 testing malware samples from our benchmark dataset. Table 3 presents the ASRs of MalGuise and two baseline adversarial attacks (*i.e.*, MMO and SRL) against three state-of-the-art learning-based Windows malware detection systems, *i.e.*, MalGraph, Magic, and MalConv. Recall in § 4.1.3, we follow the same experiment setting of baseline adversarial attacks in their original papers to facilitate fair comparisons. It is also worth noting that, MMO is essentially a white-box adversarial attack that presents an upper bound of the attack effectiveness of the possible corresponding black-box adversarial attack, and SRL is not applicable to MalConv as it does not generate the real adversarial malware in raw bytes.

Table 3: Attack performance of MalGuise and two baseline attacks against three learning-based Windows malware detection systems in terms of attack success rates (ASR).

Attacks	MalGraph		Magic		MalConv	
	FPR =1%	FPR =0.1%	FPR =1%	FPR =0.1%	FPR =1%	FPR =0.1%
MMO [50]	15.55%	52.30%	12.82%	40.13%	11.99%	39.66%
SRL [78]	2.39%	19.59%	25.38%	86.77%	-	-
MalGuise	**97.47%**	**97.77%**	**99.29%**	**99.42%**	**34.36%**	**97.38%**

For the two baseline adversarial attacks, it is clearly observed from Table 3 that, with the decrease of FPR of each target malware detection system, more adversarial malware files can be successfully generated for the attack purpose, presenting a higher ASR of each adversarial attack. The reason is evident that a lower value of FPR allowed by a malware detection system indicates a higher value of binary classification threshold. Hence, it is easier for the adversarial attack to achieve the purpose of reducing the predicted malicious probability of each original malware sample to a level that is less than the threshold. Moreover, for MMO, it shows similar and inferior attack effectiveness on the three target malware detection systems. Particularly, in the case of FPR=1%, the ASRs of MMO on three target systems are in the range of about 10% to 15%, while in the case of FPR=0.1%, the ASRs of MMO are in the range of about 40% to 50%. These observations suggest that, although MMO can theoretically be used against all three target malware detection systems, it exhibits inferior attack effectiveness overall. The main reason we conjecture is that, as MMO manipulates the entire raw bytes of malware with two binary diversification tools, it does not take into account the unique characteristics of different malware detection systems. Hence, MMO is not scalable as a more effective adversarial attack by exploiting the unique characteristics of each different malware detection. For SRL, it shows an obviously higher ASR against Magic than MalGraph in both cases of FPRs. This is mainly because SRL is specifically designed to attack Magic in their original paper which purely builds on the CFG representation. However, the hierarchical nature of MalGraph that combines both function call graph and CFG further weakens the attack effectiveness caused by SRL, as SRL only manipulates the node information of CFG and ignores the manipulations of edge information.

Compared with the two baseline adversarial attacks, it can be seen from Table 3 that our proposed MalGuise achieves the best attack effectiveness against all three learning-based malware detection systems in both cases of FPRs. In particular, when FPR=0.1%, the values of ASR achieved by MalGuise are as high as 97.77%, 99.42%, and 97.38% for MalGraph, Magic, and MalConv, respectively. Even when FPR=1%, the values of ASRs of MalGuise are still higher than 97% for MalGraph and Magic. Differently, for MalConv with FPR=1%, the ASR of MalGuise is 34.36%. However, when further investigating MalGuise in the subsequent § 4.2.2, we find that its attack effectiveness against MalConv is highly dependent on the injected semantic nops. Therefore, if using the 25 most frequently used semantic nops, MalGuise can achieve a high ASR of 97.76% against MalConv. All these evaluation results demonstrate that MalGuise can successfully bypass existing learning-based Windows malware detection systems with a high ASR.

Answer to RQ1: In short, previous adversarial attacks either do not offer satisfactory attack effectiveness, or fail to scale well to other types of Windows malware detection. However, our proposed MalGuise is agnostic to the target malware detection system with a high attack success rate of mostly over 97% achieved.

4.2.2 Answer to RQ2: Impacting factors that affect attack performance of MalGuise*?* Now, we further perform ablation studies to explore which impacting factors could affect the attack performance of MalGuise.

Impact of different types of semantic nops. We investigate the impact of different types of semantic nops that are generated to be injected in the transformation of call-based redividing. First of all, we divide the semantic nops generated by MalGuise into four categories (*i.e.*, arithmetic instructions, logical instructions, comparison instructions, and data transfer instructions), and present the occurrence frequency of different types of semantic nops that lead to successful evasions against the three learning-based Windows malware detection in Fig. 5, where all three malware detection are in the case of FPR=1%. We can observe that, arithmetic/logic/data transfer instructions are approximately the most effective instructions for MalGuise to be generated against MalGraph, Magic, and MalConv. However, for MalConv, the occurrence frequency of the generated comparison instructions (*e.g.* cmp eax,eax) is only 4.4%, which means comparison instructions have almost no effect to bypass MalConv.

Furthermore, we delve into the impact of different opcodes of semantic nops generated by MalGuise to perform adversarial attacks against MalGuise, and present the occurrence frequency of different opcodes of semantic nops that lead to successful evasions against MalConv in Fig. 6. It is clearly observed that, some opcodes (*e.g.*, dec/inc, xor, push/pop, and add/sub) of semantics nops occur with a relatively high frequency, while some other opcodes (*e.g.*, cmp and test) of semantics nops occur with a relative frequency. Next, when we limit MalGuise to using the 25 most frequently used semantic nops, the ASR of MalGuise against MalConv increases from 34.36% to 97.76%, which highlights the importance of different types of semantic nops in MalGuise. Therefore, we can conclude

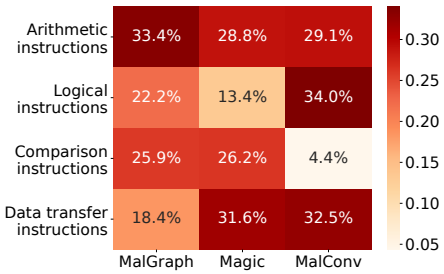


Figure 5: The occurrence frequency of different types of semantic nops **in MalGuise that leads to successful evasions against MalGraph, Magic, and MalConv.**

that the attack performance of MalGuise can be significantly improved by fine-tuning the types of semantic nops to be injected in the transformation of call-based redividing.

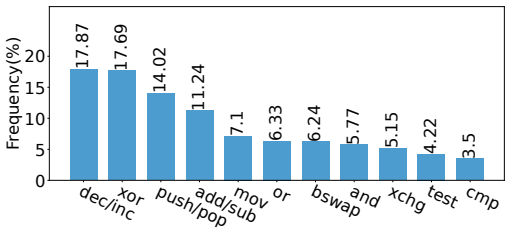
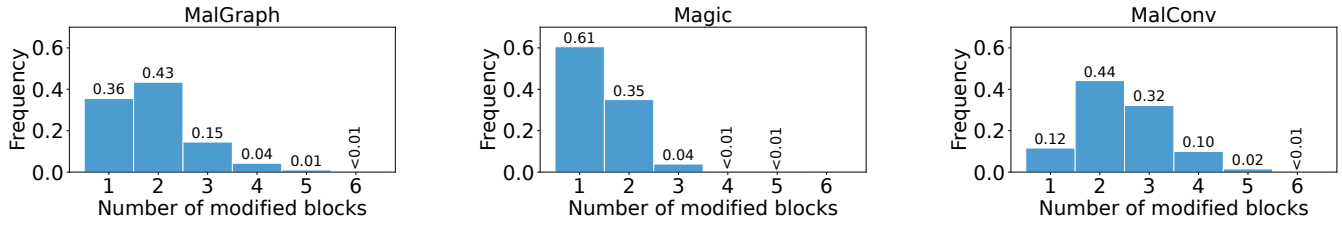


Figure 6: The occurrence frequency of different opcodes of semantic nops **in MalGuise that lead to successful evasions against MalConv.**

Impact of the numbers of modified blocks in the CFG representation. We further investigate the impact of the numbers of modified blocks in the CFG representation of all generated adversarial malware samples that successfully evade all three learning-based malware detection systems in the case of FPR=1%, and show their corresponding distribution frequencies in Fig. 7. It is clearly seen that over 98% of all adversarial malware samples that successfully evade the three target systems require only modifications of no more than four basic blocks of the corresponding CFG representation. In particular, MalGuise only needs to modify one basic block to make 61% of malware samples to evade Magic, and only needs to modify two basic blocks to make 79% and 56% of malware samples to evade MalGraph and MalConv, respectively. The above evaluation results indicate that, by only modifying a small number of basic blocks of the CFG representation, our proposed MalGuise offers excellent attack performance against all three target learning-based Windows malware detection systems.

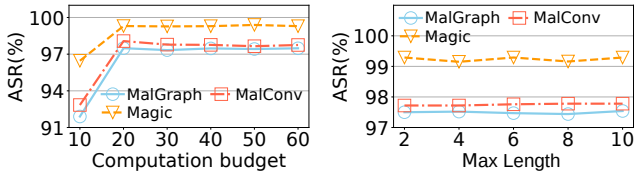
Impact of the MCTS optimization's hyper-parameters. We finally investigate how the hyper-parameters of the MCTS optimization (*i.e.*, the computation budget C and the max length N of the sequence of call-based redividing transformations) affect the attack performance of MalGuise. First of all, we follow the same implementation settings that have been detailed in § 4.1.5 and only vary the value of computation budget C to 10, 20, 30, 40, 50 and 60. The impact of the computation budget C is shown in Fig. 8(a), from which we can find that, with the increase of the



(a) MalGraph (b) Magic (c) MalConv

Figure 7: The distribution frequency of the number of modified basic blocks of the CFG representation for all generated adversarial malware samples that successfully evade the three learning-based Windows malware detection systems.

computational budget C form 10 to 20, the ASR of MalGuise rise sharply up to over 97% against all three learning-based Windows malware detection. When the computational budget C reaches 20, the attack performance of MalGuise tends to stabilize at a high value of ASR over 97%. Similarly, we only vary the max length N (*i.e.*, 2, 4, 6, 8, and 10) of the sequence of call-based redividing transformation employed in MalGuise and show the impact of max length N in Fig. 8(b). It is clearly observed that, when $N = 2$, the overall attack performance of MalGuise against all three target malware detection is pretty well, for which MalGuise achieves over 99% ASR against Magic and over 97% ASR against both MalGraph and MalConv. With the increase of N, the ASRs of MalGuise remain stable at a high value over 97% against all target malware detection systems. These observations indicate that our proposed MalGuise can achieve high attack performance even with no more than two transformations of call-based redividing employed. It is also noted that the impact of other factors on the attack performance of MalGuise is further discussed in Appendix D.1.



(a) Impact of computation budget C (b) Impact of max length N

Figure 8: Impact of different MCTS Optimization's hyper-parameters on the attack performance of MalGuise.

> **Answer to RQ2**: In summary, targeting at different malware detection systems, different factors of hyper-parameters in MalGuise have different impacts on the attack performance, and thereby we can fine-tune the hyper-parameters to further improve the attack performance.

4.2.3 Answer to RQ3: The utility performance of MalGuise *in terms of preserving the original semantics.* To verify whether the adversarial malware files generated by MalGuise could preserve the original semantics, we empirically evaluate the utility performance of MalGuise and two baseline adversarial attacks against three Windows malware detection systems in terms of semantics preservation rate (SPR). As defined in Eq. (7), the core of the calculation of SPR is $Sem(z, z_{adv})$ which needs to judge whether the adversarial malware z_{adv} has the same semantics as the original malware z (*e.g.*, can z_{adv} be executed successfully? does z_{adv} appear the same behaviors as z? *etc.*) Due to the inherent complexity of executables in computer systems, there is no solution to exactly judge whether the original semantics are preserved [7]. Therefore, we resort to an empirical verification to calculate $Sem(z, z_{adv})$ by collecting and comparing the two sequences of application programming interfaces (APIs) (*i.e.*, $\texttt{API}_{z_{adv}}$ and $\texttt{API}_z$) invoked by both z_{adv} and z when they are run on Cuckoo sandbox [68]. As shown in Eq. (9), to quantify the semantic difference between z_{adv} and z, we thus compute a normalized edit distance $dist_{norm}(z, z_{adv})$ [62] between the two API sequences of $\texttt{API}_{z_{adv}}$ and $\texttt{API}_z$ as follow.

$$dist_{norm}(z, z_{adv}) = \frac{Distance(\texttt{API}_z, \texttt{API}_{z_{adv}})}{Max\big(len(\texttt{API}_z), len(\texttt{API}_{z_{adv}})\big)} \in [0, 1] \quad (9)$$

in which $Distance(\texttt{API}_z, \texttt{API}_{z_{adv}})$ denotes the edit distance between two sequences and $len(\cdot)$ denotes the length of the sequence.

However, since malware may perform random actions during execution [34], the API sequences collected by running the same malware z *twice* in the same environment may be different, which means $dist_{norm}(z, z)$ almost can not take the value of 0. Therefore, we calculate value of $Sem(z, z) \in \{0, 1\}$ by comparing the $dist_{norm}(z, z)$ with a general distance threshold $dist_{threshold}$. In particular for determining $dist_{threshold}$, we first analyze all original malware samples in the same Cuckoo sandbox twice and then select the value at the 99.5-th percentile among all the corresponding $dist_{norm}(z, z)$ as $dist_{threshold}$. After that, as shown in Eq. (10), we can finally determine whether z_{adv} and z have the same semantics by comparing their normalized edit distance $dist_{norm}(z, z_{adv})$ with $dist_{threshold}$, and further evaluate the evaluation metrics of *SPR* according to Eq. (8).

$$Sem(z, z_{adv}) = \begin{cases} 1 & \text{if } dist_{norm}(z, z_{adv}) < dist_{threshold} \\ 0 & \text{otherwise.} \end{cases} \quad (10)$$

Table 4 present the overall utility performance of MalGuise and two baseline adversarial attack in terms of SPR, in which a higher value of SPR indicate the more effective the adversarial attack is. It should be noted that, due to the extreme resource and time consumption of the above evaluation process, we randomly select 10% from all adversarial malware that successfully bypasses the corresponding target system in § 4.2.1 for the subsequent evaluation. As for SRL, it is just a feature-space adversarial attack without generating executable adversarial malware files, not to mention evaluates its utility performance. Apparently, it can be observed

Table 4: Utility performance of MalGuise and two baseline attacks against three learning-based Windows malware detection systems in terms of semantics preservation rate (SPR).

Attacks	MalGraph		Maigc		MalConv	
	FPR =1%	FPR =0.1%	FPR =1%	FPR =0.1%	FPR =1%	FPR =0.1%
MMO [50]	41.8%	49.4%	39.6%	39.8%	39.2%	50.8%
SRL [78]	-	-	-	-	-	-
MalGuise	**91.84%**	**91.99%**	**93.45%**	**92.28%**	**92.67%**	**91.68%**

from Table 4 that, the SPRs achieved by MalGuise against three learning-based Windows malware detection are at a low level, *i.e.*, approximately ranging from 40% to 50%. It indicates that, only less than 50% of adversarial malware generated by MMO could preserve their original semantics before be attacking. However, our proposed MalGuise achieves the best utility performance of over 91% SPR for all three malware detection systems, which demonstrates the effectiveness of MalGuise in preserving the original semantics of the generated adversarial malware. Although more than 91% of adversarial malware generated by MalGuise can maintain their original semantics, it is worth noting that, approximately 8% still cannot maintain mainly due to the unique nature of different malware samples. We carefully detail the specific reasons in Appendix D.2, which can be used to improve the utility performance of MalGuise in preserving the original semantics.

Answer to RQ3: Previous adversarial attacks either only generate non-executable adversarial "feature", or generate a large proportion of adversarial malware that lose their original semantics. However, our proposed MalGuise exhibits the best utility performance of over 91% of the generated adversarial malware files with their original semantics preserved.

4.2.4 Answer to RQ4: The attack performance of MalGuise *against real-world anti-virus products.* To further understand the real threats of our proposed MalGuise against real-world anti-virus products, we empirically evaluate MalGuise against seven commercial anti-virus tools (*i.e.*, McAfee [52], Comodo [21], Avast [10], AVG [11], Kaspersky [35], ClamAV [20], and Microsoft Defender ATP [54]), and measure the ASR performance on 1,000 testing malware samples, which are randomly selected from the entire testing dataset. It should be noted that, the main reason for randomly selecting 1,000 testing malware samples for evaluation is that these seven anti-virus products are deployed remotely on another machine, and their processing and scanning speeds are much slower than learning-based Windows malware detection.

Table 5: The attack success rates (ASRs) of MalGuise against seven real-world anti-virus products.

Attacks	McAfee	ATP	ClamAV	Comodo	Avast	AVG	Kaspersky
MalGuise	48.81%	**70.63%**	31.94%	36.00%	6.13%	7.92%	11.29%
MalGuise (S)	52.49%	**74.97%**	32.33%	36.36%	5.64%	8.17%	13.36%
Increased ASR	+3.68%	+4.34%	+0.39%	+0.36%	-0.49%	+0.25%	+2.07%

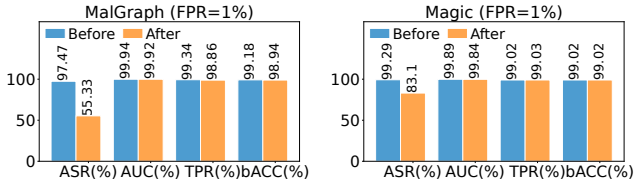
The attack success rates (ASRs) of MalGuise against seven real-world anti-virus products are shown in Table 5, in which ATP is short for Microsoft Defender ATP. It can be observed that for four (*i.e.*, McAfee, ATP, ClamAV, and Comodo) of the seven evaluated anti-virus products, our proposed MalGuise can achieve an ASR of more than 30%. Especially for ATP, the achieved ASR of MalGuise reach 70.63%. Similar to the discussion in § 4.2.2, we further explore the impact of inserted semantic nops on the attack performance of MalGuise against anti-virus products, and limit the semantic nops to be inserted in MalGuise to 25 most effective semantic nops that we selected We denote this variant of MalGuise as MalGuise (S) for brevity. Compared with the default MalGuise, it is observed from Table 5, the specific selection of semantics nops (*i.e.*, MalGuise (S)) can lead to overall improvement of attack performance against six anti-virus products except Avast. In particular, MalGuise (S) can increase the value of ASR against ATP by 4.34%, indicating MalGuise can be further improved by carefully fine-tuning its hyperparameters. These observations clearly demonstrate that MalGuise is highly effective against real-world anti-virus products under the strict black-box setting and poses potentially serious security threats to the computer system of users.

Answer to RQ4: Our proposed MalGuise is systematically and empirically demonstrated to be effective against seven commonly used real-world anti-virus products. In particular, MalGuise can achieve an attack success rate of more than 30% against four of them, *i.e.*, McAfee, ATP, ClamAV, and Comodo.

5 DISCUSSION

5.1 Potential Defenses

To further evaluate the attack performance of MalGuise in the worst-case scenario that the defender has full knowledge of it, we attempt to improve the robustness of the target learning-based Windows malware detection through adversarial training. As the training process is performed on the raw bytes of executables which are extremely costly to be generated, we take both MalGraph and Magic in the case of FPR=1% as two representative target systems, and re-train them from scratch with almost all parameter settings unchanged. The only difference is that, in the adversarial training process, our training samples not only include the original malware and goodware, but also additionally added the corresponding adversarial malware generated by MalGuise. Table 9 shows the performance comparison between the original target systems (*i.e.*, MalGraph and Magic) and the systems defended with adversarial training. It is clearly observed that, after adversarial training, the detection performance (*i.e.*, AUC, TPR, and bACC) of MalGraph and Magic remains basically unchanged, while the robustness performance (*i.e.*, the opposite of ASR) of them increases to some extent. Through adversarial training, it is quite reasonable that the robustness performance of existing learning-based Windows malware detection can be improved, but our proposed MalGuise can still achieve ASRs as high as 55.33% and 83.10% against MalGraph and Magic, respectively. Therefore, we can conclude that, even with the possible defense of adversarial training where the defender is supposed to have the full knowledge of MalGuise, MalGuise is still highly effective against existing malware detection.



(a) The target system is MalGraph. **(b) The target system is Magic.**

Figure 9: The performance comparison between the original target system (before) and the target system defended with adversarial training (after).

5.2 Related Work

5.2.1 Learning-based Windows malware detection. Due to the clearly considerable diversity of malware among different file formats (*e.g.*, PE, ELF, Mach-O, APK) in different operating systems (*e.g.*, Windows, Linux, macOS, Android), research shows that there is no malware analysis technique that is universally applicable to all different in different operating systems, and thereby all existing malware detection commonly points out which operating system and which file format of malware will be studied [48, 61, 70]. As for the Windows malware detection that this paper focuses on, with recent huge successes achieved in masses of different application domains, machine learning, and deep learning techniques have been extensively studied to increase both effectiveness and generalization in detecting newly emerging and previously unknown Windows malware, namely learning-based Windows malware detection. As previously introduced in § 2.1 and Fig. 1, learning-based Windows malware detection share a similar spirit that first performs somewhat feature engineering over the input software to extract a feature representation in numerical value, and then applies reasonable ML/DL models (*e.g.*, SVM [22], CNN [41], RNN [18], GNN [38]) to learn a malware classifier in discriminating between malware and goodware. To be specific, the feature representations extracted by feature engineering for Windows malware can be mainly divided into two categories, *i.e.*, static features and dynamic features [48]. In particular, static features refer to those features that can be obtained without running the input software, such as raw bytes, API calls, abstract graphs, *etc.* On the contrary, dynamic features refer to those features that can only be obtained by running the input software, such as CPU/Memory/File/Network status, *etc.* Due to the significant complexity and time consumption for the extraction of dynamic features, static features instead exhibit better scalability for malware detection and thereby being more ubiquitous to be deployed [48, 76]. Among existing static features, recent studies actually have demonstrated that the abstract graph representation of Windows executables, including control flow graph and function call graph, show the state-of-the-art performance of malware detection in practice [2, 5, 27, 30, 75].

5.2.2 Adversarial attacks against malware detection. Almost all existing research on adversarial attacks for the task of malware detection focuses on learning-based malware detection with static features. Owing to the vast diversity of feature representations employed by different kinds of learning-based malware detection, researchers have proposed different adversarial attacks accordingly. To be specific, to attack those malware detection methods based on API calls, a line of adversarial attacks since 2017 has been proposed via adding irrelevant API calls, which are selected by gradient-based optimizations or greedy algorithms [4, 17, 29, 72]. However, these adversarial attacks are impractical as they actually generate adversarial API calls rather than realistic executable adversarial malware that preserves the original semantics. As for malware detection based on raw bytes, especially for MalConv [60], many research efforts of adversarial attacks have been made by either partially modifying specific regions or globally modifying all raw bytes while preserving the same semantics. For instance, all of [6, 24, 39, 40, 66] rely on appending or injecting maliciously generated bytes at specific locations of the input malware, while MMO [50] entirely manipulates its raw bytes with binary-diversification tools. Nevertheless, we argue that those adversarial attacks designed are strictly limited to aim at raw-bytes-based malware detection, and thus cannot be scalable to other various kinds of malware detection. Furthermore, the most recent studies have begun to explore adversarial attacks against the latest and most advanced malware detection based on abstract graph representation. In 2022, Zhang *et al.* [78] propose an adversarial attack of SRL that sequentially injects semantic NOPs into the control-flow graph representation of the given malware under the guidance of reinforcement learning. Likewise, our work also aims at the most advanced malware detection based on CFG, but fundamentally differs from SRL in the following three aspects.

(1) SRL only employ a coarse-grained transformation that manipulates the nodes of CFG, however, we propose a novel and finer-grained transformation `call-based redividing` that manipulate both nodes and edges of CFG.
(2) As discussed in § 4.2.3, SRL is a feature-space adversarial attack that essentially generates adversarial "features", while our proposed `MalGuise` can successfully generate real adversarial malware that preserve the original semantics.
(3) Existing adversarial attacks including SRL only evaluate against specific malware detection systems, while our proposed `MalGuise` additionally other anti-virus tools in practice, thereby demonstrating better attacking generalizability.

6 CONCLUSION

In this paper, we first propose a novel semantics-preserving transformation of `call-based redividing` that concurrently manipulates both nodes and edges of the CFG representation of Windows executables, and further present an adversarial attack framework against learning-based windows malware detection in the strict black-box setting, namely `MalGuise`. Extensive evaluations demonstrate that MalGuise can not only successfully bypass state-of-the-art learning-based Windows malware detection with an attack success rate of mostly over 97%, but also can bypass seven representative commercial anti-virus products with an attack success rate in a range of 5.64% to 74.97%. Our findings indicate a clear and urgent need of strengthening the robustness of existing learning-based malware detection systems and commercial anti-viruses, or even designing new alternative solutions from scratch with robustness in mind.

REFERENCES

[1] 360 Total Security. 2020. https://www.360totalsecurity.com/. Online (last accessed Aug. 1, 2020).

[2] Ahmed Abusnaina, Mohammed Abuhamad, Hisham Alasmary, Afsah Anwar, Rhongho Jang, Saeed Salem, Daehun Nyang, and David Mohaisen. 2021. DL-FHMC: Deep learning-based fine-grained hierarchical learning approach for robust malware classification. *IEEE Transactions on Dependable and Secure Computing* (2021).

[3] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. 2020. When Malware is Packin'Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *NDSS*.

[4] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. 2018. Adversarial deep learning for robust detection of binary encoded malware. In *IEEE Security and Privacy Workshops*.

[5] Hisham Alasmary, Ahmed Abusnaina, Rhongho Jang, Mohammed Abuhamad, Afsah Anwar, DaeHun Nyang, and David Mohaisen. 2020. Soteria: Detecting adversarial examples in control flow graph-based malware classifiers. In *International Conference on Distributed Computing Systems*. IEEE, Singapore, 888–898.

[6] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. 2017. Evading machine learning malware detection. In *Black Hat USA*.

[7] Martin Apel, Christian Bockermann, and Michael Meier. 2009. Measuring similarity of malware behavior. In *2009 IEEE 34th Conference on Local Computer Networks*. IEEE, 891–898.

[8] Atlas VPN. 2022. Over 95% of all new malware threats discovered in 2022 are aimed at Windows. https://atlasvpn.com/blog/over-95-of-all-new-malware-threats-discovered-in-2022-are-aimed-at-windows. Online (last accessed December 25, 2022).

[9] AV-TEST Institute. 2022. Total Malware Statistics. https://www.av-test.org/en/statistics/malware/. Online (last accessed December 25, 2022).

[10] Avast. 2022. https://www.avast.com/. Online (last accessed Nov. 1, 2022).

[11] Avg. 2022. https://www.avg.com/. Online (last accessed Nov. 1, 2022).

[12] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition* 84 (2018), 317–331.

[13] Andrew P Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern recognition* 30, 7 (1997), 1145–1159.

[14] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 1–43.

[15] Ero Carrera. 2022. PEFILE. https://github.com/erocarrera/pefile. Online (last accessed December 25, 2022).

[16] Fabrício Ceschin, Heitor Murilo Gomes, Marcus Botacin, Albert Bifet, Bernhard Pfahringer, Luiz S Oliveira, and André Grégio. 2020. Machine Learning (In) Security: A Stream of Problems. (2020). arXiv preprint arXiv:2010.16045.

[17] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. 2017. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *European Intelligence and Security Informatics Conference*. IEEE, Athens, Greece, 99–106.

[18] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Empirical Methods in Natural Language Processing*. ACL, Doha, Qatar, 1724–1734.

[19] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Xiaodong Song, and Randal E. Bryant. 2005. Semantics-aware malware detection. *IEEE Symposium on Security and Privacy* (2005), 32–46.

[20] ClamAv. 2022. https://www.clamav.net/. Online (last accessed Nov. 1, 2022).

[21] Comodo. 2022. https://www.comodo.com/home/internet-security/antivirus.php. Online (last accessed Nov. 1, 2022).

[22] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (1995), 273–297.

[23] Rémi Coulom. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*. Springer, 72–83.

[24] Luca Demetrio, Scott E Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. 2020. Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection. *arXiv:2008.07125* (2020).

[25] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. 2012. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Commun. ACM* 55, 3 (2012), 106–113.

[26] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *ICLR*.

[27] Jerome Dinal Herath, Priti Prabhakar Wakodikar, Ping Yang, and Guanhua Yan. 2022. CFGExplainer: Explaining Graph Neural Network-Based Malware Classification from Control Flow Graphs. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 172–184.

[28] Hex-Rays. 2022. IDA Pro. https://hex-rays.com/ida-pro/. Online (last accessed Jan. 11, 2022).

[29] Weiwei Hu and Ying Tan. 2017. Generating adversarial malware examples for black-box attacks based on gan. *arXiv:1702.05983* (2017).

[30] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. 2009. Large-scale malware indexing using function-call graphs. In *CCS*.

[31] Christopher Jämthagen, Patrik Lantz, and Martin Hell. 2013. A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries. In *2013 Workshop on Anti-malware Testing Research*. IEEE, 1–9.

[32] Jibz and Qwerton and snaker and xineohP. 2020. PEiD: PE iDentifier. https://www.aldeid.com/wiki/PEiD. Online (last accessed Dec. 15, 2020).

[33] Yuan Junkun, Zhou Shaofang, Lin Lanfen, Wang Feng, and Cui Jia. 2020. Black-Box Adversarial Attacks Against Deep Learning Based Malware Binaries Detection with GAN. In *ECAI*.

[34] Takahiro Kasama, Katsunari Yoshioka, Daisuke Inoue, and Tsutomu Matsumoto. 2012. Malware detection method by catching their random behavior in multiple executions. In *2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet*. IEEE, 262–266.

[35] Kaspersky. 2022. https://www.kaspersky.com/. Online (last accessed Nov. 1, 2022).

[36] kaspersky. 2022. AI and Machine Learning in Cybersecurity — How They Will Shape the Future. https://www.kaspersky.com/resource-center/definitions/ai-cybersecurity. Online (last accessed December 25, 2022).

[37] Kevin O'Reilly. 2020. CAPE: Malware Configuration And Payload Extraction. https://github.com/kevoreilly/CAPEv2. Online (last accessed Dec. 15, 2020).

[38] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[39] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *EUSIPCO*.

[40] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. 2018. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv:1802.04528* (2018).

[41] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., Lake Tahoe, Nevada, United States, 1106–1114.

[42] Vasantha Lakshmi. 2019. Beginning Security with Microsoft Technologies. *Beginning Security with Microsoft Technologies* (2019).

[43] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. 2021. Arms Race in Adversarial Malware Detection: A Survey. *ACM Computing Surveys (CSUR)* 55, 1 (2021), 1–35.

[44] Jinfeng Li, Tianyu Du, Shouling Ji, Rong Zhang, Quan Lu, Min Yang, and Ting Wang. 2020. TextShield: Robust text classification based on multimodal embedding and neural machine translation. In *USENIX Security*.

[45] Xiang Li, Kefan Qiu, Cheng Qian, and Gang Zhao. 2020. An Adversarial Machine Learning Method Based on OpCode N-grams Feature in Malware Detection. In *DSC*.

[46] Xiang Ling, Shouling Ji, Jiaxu Zou, Jiannan Wang, Chunming Wu, Bo Li, and Ting Wang. 2019. DEEPSEC: A Uniform Platform for Security Analysis of Deep Learning Model. In *IEEE Symposium on Security and Privacy*. IEEE, San Francisco, USA, 673–690.

[47] Xiang Ling, Lingfei Wu, Wei Deng, Zhenqing Qu, Jiangyu Zhang, Sheng Zhang, Tengfei Ma, Bin Wang, Chunming Wu, and Shouling Ji. 2022. MalGraph: Hierarchical Graph Neural Networks for Robust Windows Malware Detection. In *IEEE Conference on Computer Communications (INFOCOM)*. IEEE, Virtual Event, 1998–2007.

[48] Xiang Ling, Lingfei Wu, Jiangyu Zhang, Zhenqing Qu, Wei Deng, Xiang Chen, Chunming Wu, Shouling Ji, Tianyue Luo, Jingzheng Wu, and Yanjun Wu. 2021. Adversarial Attacks against Windows PE Malware Detection: A Survey of the State-of-the-Art. *arXiv preprint arXiv:2112.12310* (2021).

[49] lockbit-3.0. 2022. https://www.avertium.com/resources/threat-reports/an-update-on-lockbit-3.0. Online (last accessed Dec. 18, 2022).

[50] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. 2021. Malware Makeover: breaking ML-based static analysis by modifying executable bytes. In *ACM Asia Conference on Computer and Communications Security*. 744–758.

[51] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. 2019. Towards adversarial malware detection: Lessons learned from PDF-based attacks. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–36.

[52] Mcafee. 2022. https://www.mcafee.com/. Online (last accessed Nov. 1, 2022).

[53] Alex McFarland. 2022. 10 Best Antivirus Programs of 2022 (AI Powered). https://www.unite.ai/10-best-antivirus-programs-of-2022-ai-powered/. Online (last accessed December 25, 2022).

[54] Microsoft Defender for Endpoint. 2022. https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/microsoft-defender-endpoint?view=o365-worldwide/. Online (last accessed Nov. 1, 2022).

[55] Microsoft Defender Security Research Team. 2022. Windows Defender ATP machine learning: Detecting new and unusual breach activity. https://www.microsoft.com/en-us/security/blog/2017/08/03/windows-

defender-atp-machine-learning-detecting-new-and-unusual-breach-activity/. Online (last accessed Nov. 1, 2022).
[56] Microsoft, Inc. 2022. PE Format. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. Online (last accessed August 20, 2022).
[57] Rémi Munos et al. 2014. From bandits to Monte-Carlo Tree Search: The optimistic principle applied to optimization and planning. *Foundations and Trends® in Machine Learning* 7, 1 (2014), 1–129.
[58] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing properties of adversarial ml attacks in the problem space. In *S&P*.
[59] Quarkslab. 2022. LIEF : Library to Instrument Executable Formats. https://lief-project.github.io/. Online (last accessed December 25, 2022).
[60] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. 2017. Malware detection by eating a whole EXE. *arXiv: 1710.09435* (2017).
[61] Edward Raff and Charles Nicholas. 2020. A Survey of Machine Learning Methods and Challenges for Windows Malware Classification. (2020). arXiv preprint arXiv:2006.09271.
[62] Eric Sven Ristad and Peter N Yianilos. 1998. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 5 (1998), 522–532.
[63] Takami Sato, Junjie Shen, Ningfei Wang, Yunhan Jia, Xue Lin, and Qi Alfred Chen. 2021. Dirty road can attack: Security of deep learning based automated lane centering under Physical-World attack. In *30th USENIX Security Symposium*. 3309–3326.
[64] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. 2021. Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers. In *30th USENIX Security Symposium (USENIX Security 21)*. 1487–1504.
[65] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.
[66] Octavian Suciu, Scott E Coull, and Jeffrey Johns. 2019. Exploring adversarial examples in malware detection. In *IEEE Security and Privacy Workshops*.
[67] The best antivirus protection. 2022. https://www.pcmag.com/picks/the-best-antivirus-protection. Online (last accessed Nov. 1, 2022).
[68] The Cuckoo Sandbox. 2022. https://cuckoosandbox.org/. Online (last accessed Nov. 1, 2022).
[69] The UPX Team. 2020. UPX: The Ultimate Packer for eXecutables. https://upx.github.io/. Online (last accessed Dec. 15, 2020).
[70] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. 2019. Survey of machine learning techniques for malware analysis. *Computers & Security* 81 (2019), 123–147.
[71] Digna R Velez, Bill C White, Alison A Motsinger, William S Bush, Marylyn D Ritchie, Scott M Williams, and Jason H Moore. 2007. A balanced accuracy function for epistasis modeling in imbalanced datasets using multifactor dimensionality reduction. *Genetic Epidemiology: the Official Publication of the International Genetic Epidemiology Society* 31, 4 (2007), 306–315.
[72] Sicco Verwer, Azqa Nadeem, Christian Hammerschmidt, Laurens Bliek, Abdullah Al-Dujaili, and Una-May O'Reilly. 2020. The Robust Malware Detection Challenge and Greedy Random Accelerated Multi-Bit Search. In *AISec*.
[73] VirusTotal.com. 2020. https://www.virustotal.com/gui/contact-us. Online (last accessed Aug. 1, 2020).
[74] Han Xu, Yao Ma, Hao-Chen Liu, Debayan Deb, Hui Liu, Ji-Liang Tang, and Anil K Jain. 2020. Adversarial attacks and defenses in images, graphs and text: A review. *IJAC* 17, 2 (2020), 151–178.
[75] Jiaqi Yan, Guanhua Yan, and Dong Jin. 2019. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *DSN*.
[76] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. 2017. A survey on malware detection using data mining techniques. *ACM Computing Surveys* 50, 3 (2017), 1–40.
[77] Oleh Yuschuk. 2022. OllyDbg. https://www.ollydbg.de/. Online (last accessed Nov. 1, 2022).
[78] Lan Zhang, Peng Liu, Yoonho Choi, and Ping Chen. 2022. Semantics-preserving Reinforcement Learning Attack Against Graph Neural Networks for Malware Detection. *IEEE Transactions on Dependable and Secure Computing* (2022).

A APPENDIX OF THE IMPLEMENTATION DETAILS OF MCTS

A.1 Algorithm of Selecting the Best Child

Algorithm 2 shows the procedure of `Selection`. As the possible paths in the MCTS game tree is infinite, exploring all the nodes will substantially increase the computational overhead. By employing the Upper Confidence Bounds algorithm [14], procedure `Selection` selects the child of MCTS node v with the highest score considering the trade off between the visit times and the reward (*line 4-6*). The selected child node is regarded as the most "promising" node and will be explored.

Algorithm 2: Procedure of `Selection`.

Input : MCTS node v.
Output: Selected child node $v_{selected}$ of v.

1 **Procedure** `Selection`(v)
2 $max\_score \leftarrow 0$;
3 **for** v_{child} *in children of* v **do**
4 $exploit \leftarrow v_{child}.reward/v_{child}.visits$;
5 $explore \leftarrow \sqrt{2\ln(v.visits)/v_{child}.visits}$;
6 $score \leftarrow exploit + \lambda \times explore$;
7 **if** $max\_score < score$ **then**
8 $max\_score \leftarrow score$;
9 $v_{selected} \leftarrow v_{child}$;
10 **end**
11 **end**
12 **return** $v_{selected}$.

A.2 Algorithm of Simulation

Algorithm 3 shows the procedure of `Simulation`, which finally returns the reward r. Based on the selected MCTS node $v_{selected}$, this procedure iteratively expands the MCTS game tree until the simulation number S is reached (*line 3-7*). In each iteration, the corresponding CFG representation x of the expanded node is input to the target malware detection system f, and the reward r will be calculated as $r = 1 - f(x)$ (*line 5*). After the procedure is terminated, it returns the reward r.

Algorithm 3: Procedure of `Simulation`.

Input : The selected node $v_{selected}$, target malware detection system f, simulation number S.
Output: Reward r.

1 **Procedure** `Simulation`($v_{selected}, f, S$)
2 $v' \leftarrow v_{selected}$;
3 **for** $i \leftarrow 1$ **to** S **do**
4 $v' \leftarrow$ `Expansion`(v');
5 $x \leftarrow v'.x$;
6 $r \leftarrow 1 - f(x)$;
7 **end**
8 **return** r.

B APPENDIX OF THE IMPLEMENTATION DETAILS OF MALWARE RECONSTRUCTION

B.1 Algorithm of Adding a New Section

Algorithm 4 shows the procedure of adding a new section. The size Δ_{new}^{align} of the new section is computed in *line 2-9*, where Δ_{new} is total size of all the instructions in all $\mathbf{V}_{mid}$ (as discussed in § 3.5.1). Let A_{last} and S_{last} represent the address and size of the last section, respectively. The starting address of the new section can be

computed as $A^{new} = A_{last} + \texttt{RoundUp}(S_{last}, page\_size)$ (*line 10-11*), where A_{last} and S_{last} can be both obtained from the header information of the malware, and *page_size* represents architecture's page size. After adding a new entry to the section table and adjusting the PE header (such as modifying the "number of sections" and "size of image", *etc*), the new section is successfully added to the malware. After the new section is added, procedure `AddSection` returns the starting address A^{new} of the new section.

Algorithm 4: Procedure of adding a new section.

Input : Original malware z, sequence of the transformations $\mathbf{T}$.
Output: Address of the new section A^{new}.

1 **Procedure** `AddSection`$(z, \mathbf{T})$
2 $\quad \Delta_{new} \leftarrow 0$;
3 $\quad$ **for** $k \leftarrow 1$ **to** n **do**
4 $\quad\quad \Delta_{new} \leftarrow \Delta_{new} + \texttt{GetSize}(\mathbb{I}_k^{semnops})$;
5 $\quad\quad \Delta_{new} \leftarrow \Delta_{new} + \texttt{GetSize}(\mathbb{I}_k^{call})$;
6 $\quad\quad \mathbb{I}_k^{jmp} \leftarrow \texttt{jmp}\_[A_k^{call} + \texttt{GetSize}(\mathbb{I}_k^{call})]$;
7 $\quad\quad \Delta_{new} \leftarrow \Delta_{new} + \texttt{GetSize}(\mathbb{I}_k^{jmp})$;
8 $\quad$ **end**
9 $\quad \Delta_{new}^{align} \leftarrow \texttt{RoundUp}(\Delta_{new}, page\_size)$;
10 $\quad$ Obtain the address A_{last} and the content size S_{last} of the last section from the PE header.;
11 $\quad A^{new} \leftarrow A_{last} + \texttt{RoundUp}(S_{last}, page\_size)$;
12 $\quad$ Add an entry to section table;
13 $\quad$ Adjust the PE header;
14 **return** A^{new}.

B.2 Algorithm of Windows Malware Reconstruction

Algorithm 5 shows the procedure of Windows malware reconstruction. After adding a new section (*line 2*), the transformations will be iteratively applied to the malware (*line 4-14*), thus generating an adversarial one. In the k-th iteration, the call instruction $\mathbb{I}_k^{call}$ is first replaced with the jmp instruction "$\texttt{jmp}\_[\texttt{A}_\texttt{k}^{\texttt{new}}]$" that transfers control to the address A_k^{new} in the new section (*line 5-6*). The instruction at address A_k^{new} will be set to the call instruction $\mathbb{I}_k^{call}$, followed by which is the semantic nops $\mathbb{I}_k^{semnops}$ and the jmp instruction "$\texttt{jmp}\_[A_k^{call} + \texttt{GetSize}(\mathbb{I}_k^{call})]$" that transfers control back to the next instruction of $\mathbb{I}_k^{call}$ in the original section (*line 7-12*). After all the N transformations are applied to the malware, the adversarial one z_{adv} is generated.

Algorithm 5: Reconstruction of Windows malware.

Input : original malware z, sequence of the transformations $\mathbf{T} = \{T_1, T_2, ..., T_N\}$.
Output: adversarial malware z_{adv}.

1 **Begin**
2 $\quad A^{new} \leftarrow \texttt{AddSection}(z, \mathbf{T})$;
3 $\quad A_1^{new} \leftarrow A^{new}$;
4 $\quad$ **for** $k \leftarrow 1$ **to** n **do**
5 $\quad\quad$ Get the address A_k^{call} of the `call` instruction $\mathbb{I}_k^{call}$;
6 $\quad\quad \texttt{Patch}(z, A_k^{call}, \texttt{jmp}\_[A_k^{new}])$;
7 $\quad\quad \texttt{Patch}(z, A_k^{new}, \mathbb{I}_k^{call})$;
8 $\quad\quad A_k^{new} \leftarrow A_k^{new} + \texttt{GetSize}(\mathbb{I}_k^{call})$;
9 $\quad\quad \texttt{Patch}(z, A_k^{new}, \mathbb{I}_k^{semnops})$;
10 $\quad\quad A_k^{new} \leftarrow A_k^{new} + \texttt{GetSize}(\mathbb{I}_k^{semnops})$;
11 $\quad\quad \mathbb{I}_k^{jmp} \leftarrow \texttt{jmp}\_[A_k^{call} + \texttt{GetSize}(\mathbb{I}_k^{call})]$;
12 $\quad\quad \texttt{Patch}(z, A_k^{new}, \mathbb{I}_k^{jmp})$;
13 $\quad\quad A_{k+1}^{new} \leftarrow A_k^{new} + \texttt{GetSize}(\mathbb{I}_k^{jmp})$;
14 $\quad$ **end**
15 $\quad z_{adv} \leftarrow z$;
16 **return** z_{adv}.

C DETAILS OF IMPLEMENTATIONS AND HYPER-PARAMETERS

C.1 Evaluation Metrics of Learning-based Windows Malware Detection.

The metrics for evaluating the performance of learning-based windows malware detection are introduced as follows.

1) **Area under the curve (AUC) [13]** represents the overall performance of the binary classification model. This metric is defined as the area under the entire ROC curve [13] and is independent of the chosen detection threshold. A higher value of AUC indicates a better performance of the malware detection.

2) **True positive rate (TPR) / false positive rate (FPR) [13] and balanced accuracy (bACC) [71]** are commonly used metrics to evaluate the effectiveness of the malware detector. Under the same value of false positive rate (FPR), the higher the value of TPR or bACC, the more effective the detector is in detecting malicious samples. In the experiment, the values of FPR are set to be 1% and 0.1%, and the corresponding two detection thresholds are then calculated. Given the calculated detection threshold, the values of TPR and bACC are computed as follows.

$$\begin{aligned} TPR &= TP/(TP+FN) \\ TNR &= TN/(FP+TN) \\ bACC &= (TNR+TPR)/2 \end{aligned} \tag{11}$$

Where $FPR = FP/(FP+TN)$, and TN, FN, TP, FP denote the number of true negative, false negative, true positive, and false positive, respectively.

C.2 Implementations and Hyper-parameters Details of Baseline Adversarial Attacks.

In this section, the implementations and hyper-parameters details of the two baseline adversarial attacks, *i.e.* Malware Makeover (MMO) [50] and SRL [78] is elaborated. In principle, we follow the same experimental settings as the baseline attacks in their original papers In the experiment, 5,000 Windows malwares are randomly selected from the testing dataset to evaluate the effectiveness of the two baseline attacks. MMO is implemented to attack the three learning-based malware detection models mentioned in § 4.1.2, the experiment is conducted in the white box setting, where the maximum number of iterations is set to be 200, and the increment of the malware's size is limited to 5% of the size of the original malware, which is the same experiment setup as in [50]. SRL is implemented to attack the two GNN based model: MalGraph and Magic. The

reason why MalConv is not used to evaluate SRL is that, SRL is designed for attacking the GNN based model whose input is the extracted ACFGs of executables, it can not be applied to attack MalConv model that takes the raw bytes as the input. Moreover, the same experiment setup as in [78] is used, where the maximum number of iterations, the injection budget, and the modified basic blocks are set to be 30, 5%, and 1250, respectively.

D APPENDIX OF EVALUATION RESULTS AND ANALYSIS

D.1 Ablation Study of the Size of Semantic nops for MalGuise

The impact of the size of the injected semantic nops on the performance of MalGuise is illustrated in Figure 10, where the three learning-based models are all at 1% FPR. The thresholds of the size of the injected semantic nops are set to be 1%, 3%, 5%, 7% and 9% of the original malware's size. Note that, the section alignment (*page_size*) of a Windows executable is 4096 bytes in Intel x86 architecture, which means the size of the new section containing the injected semantic nops must be a multiple of 4096 bytes [56]. In the experiment, the two cases where the threshold is rounded up and the threshold is not rounded up are both considered. In Figure 10(a), the threshold is rounded up to a multiple of 4096 bytes, while in Figure 10(b), the threshold is not rounded up. From Figure 10(a), it can be observed that, when the threshold is rounded up, the values of ASR obtained by MalGuise for MalGraph and Magic are always in a high level (higher than 97%) with the increase of the threshold. As shown in Figure 10(b), when the threshold is not rounded up, with the increase of the threshold from 1% to 9%, the ASRs achieved by MalGuise against MalGraph gradually rises from 81.99% to 95.64%, and the ASRs of MalGuise against Magic also increases from 90.26% to 98.99%. For MalConv, when all the semantic nops are implemented, the performance of MalGuise is inferior and unstable as it can only obtain an ASR near 36%. It can be explained that, most of the inserted semantic nops have little impact on the prediction results of MalConv. To improve the efficiency of MalGuise against MalConv, 25 semantic nops that are most frequently used by MalGuise are selected. As shown in Figure 10, when inserting only the selected 25 semantic nops, MalGuise offers excellent performance as the achieved ASR is higher than 96% even when the size of semantic nops is only 1% of the original malware's size. The result indicates that, the effectiveness of MalGuise against MalConv is highly dependent on the implemented semantic nops, and it's important to search for a few semantic nops that cause significant degradation in the detection performance of MalConv. In the rest of this paper, MalGuise only implements the selected 25 semantic nops for MalConv.

In addition, the experiment is conducted when the threshold of the size of the semantic nops takes the values of $n \times 4096$, where the values of n are set to be 1, 2, 3, 4, 5. The experimental results are presented in Table 6, and the three learning based models are all at 1% FPR. It can be seen that, that values of ASR attained by MalGuise are as high as 97.61%, 99.20% and 97.53% for MalGraph, Magic and MalConv, respectively. Even when $n = 1$, the ASRs of MalGuise are still higher than 90% against the three models. Moreover, when MalGuise is implemented to attack the Magic model, the obtained ASRs of MalGuise are always higher than 99% with the increase of n. The above experimental results indicate that, MalGuise can effectively deceive the learning-based models with a small size of the injected semantic nops.

Table 6: The attack success rates (ASRs) of MalGuise when the threshold of the size of semantic nops takes the value of $n \times 4096$.

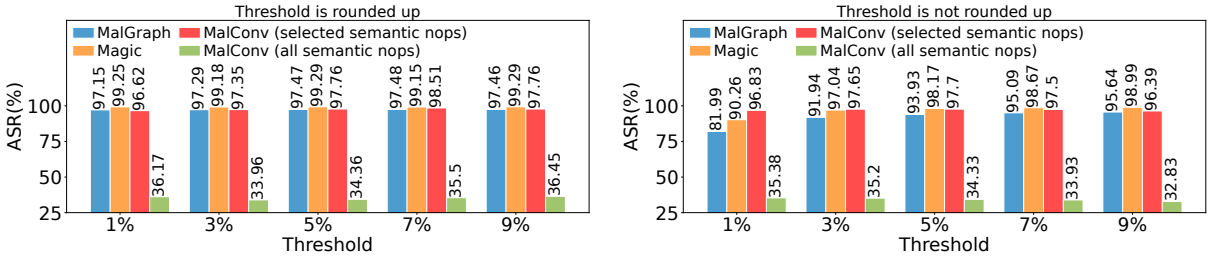
Models (FPR=1%)	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
MalGraph	96.31%	97.49%	97.60%	97.54%	97.61%
Magic	99.16%	99.15%	99.17%	99.17%	99.20%
MalConv	91.56%	96.34%	97.33%	97.48%	97.53%

D.2 The Detailed Reasons Why Adversarial Malware Generated by MalGuise cannot Preserve the Original Semantics

To explain why some malware samples can not keep their semantics after being modified by MalGuise, the malware samples with broken functionalities are analyzed manually via IDA Pro [28] and OllyDbg [77]. There are two main reasons for the dysfunction of the adversarial malware samples. The first reason is the malware sample contains overlay, which is the extra data that is not covered by the PE header. In the execution process of the malware, the overlay will be extracted to perform the malicious behavior. After adding a new section to the malware, the extraction of the overlay may be affected, thus influences the execution process of the malware. The second reason is the malware sample contains junk code, which is used for anti-disassembly [31]. Specifically, when the `call` instruction is executed, the return address of the `call` instruction will be pushed into the stack. After the execution of the `call` instruction, the return address that is pushed into the stack will be removed from the stack. In this case, when MalGuise is implemented, the return address of the `call` instruction will be changed, and the changed address will be pushed into the stack and removed from the stack. Thus, the stack will not be affected. However, if the malware contains junk code, the changed address will not be removed from the stack, which will affect the execution process of the malware and break its functionality.

D.3 Appendix of Evaluations of MalGuise against Real-world Anti-virus Products.

The impact of the inserted semantic nops on the performance of MalGuise is further explored. We select 25 semantic nops that are most frequently used by MalGuise when attacking the real-world anti-virus products. Figure 11 illustrates the occurrence frequency of the opcodes of the selected 25 semantic nops. As shown in Figure 11, `xchg`, `mov`, `test`, `push/pop`, `add/sub`, `test` and `add/sub` are the most effective opcodes for deceiving McAfee, Microsoft Defender ATP, ClamAV, Comodo, Avast, AVG and Kaspersky, respectively. Furthermore, when using only the selected 25 semantic nops, the ASRs of MalGuise are presented in Table 5. It can be observed that, the selection of the semantic nops leads to improvement in the performance of MalGuise for all the anti-virus products except Avast. Especially for McAfee, Microsoft Defender ATP and Kaspersky, the



(a) (b)

Figure 10: Impact of the size of the injected semantic nops on the performance of `MalGuise`. In Figure 10(a), the threshold will be rounded up to a multiple of 4096 bytes, while in Figure 10(b), the threshold will not be rounded up.

ASRs of `MalGuise` increased by 3.68%, 4.34% and 2.07%, respectively. In practice, an undetected malware sample can cause great damage to the computer system. Overall, above experimental results demonstrate the effectiveness of `MalGuise` against the anti-virus products deployed in the real world.

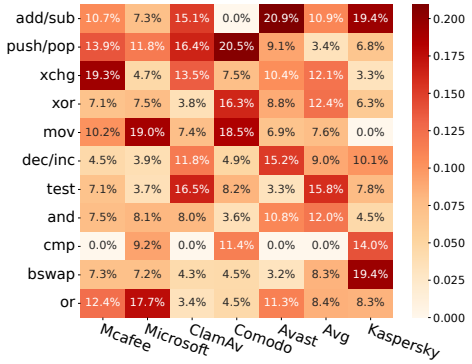


Figure 11: The occurrence frequency of the opcodes of the top 25 semantic nops that lead to evasion against real-world anti-virus products.

Impact of the Number of Modified Blocks. The distribution of the number of modified blocks in the adversarial malware samples that evade the anti-virus products is illustrated in Figure 12. As shown in Figure 12, for all the seven anti-virus products, the number of modified blocks in the adversarial malware samples are less than 6. Especially for McAfee, ClamAV and Comodo, more than 90% of the malware samples that successfully evade the three products have only one block modified (95%, 98% and 94% for McAfee, ClamAV and Comodo, respectively). In conclusion, `MalGuise` can be applied against the anti-virus products by modifying a small number of basic blocks.

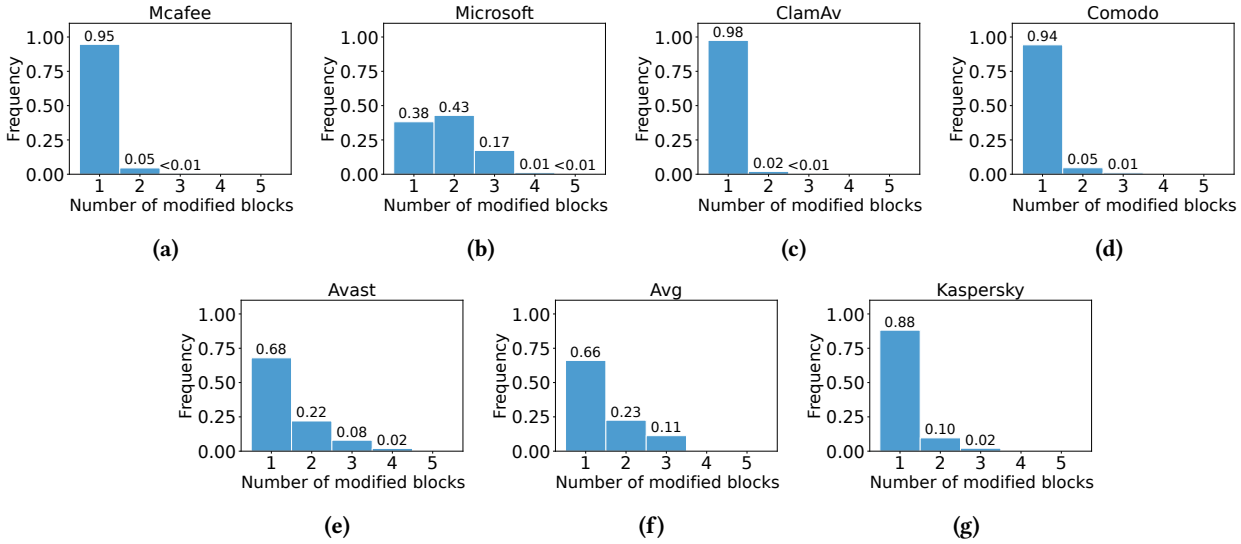


Figure 12: The distribution of the number of modified blocks for the malware samples that evade the anti-virus products.