



A Wolf in Sheep's Clothing: Practical Black-box Adversarial Attacks for Evading Learning-based Windows Malware Detection in the Wild

Xiang Ling, Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences; Key Laboratory of System Software (Chinese Academy of Sciences); State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences; Zhiyu Wu, Zhejiang University; Bin Wang, Zhejiang Key Laboratory of Artificial Intelligence of Things (AIoT) Network and Data Security; Hangzhou Research Institute, Xidian University; Wei Deng, Zhejiang University; Jingzheng Wu, Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences; Key Laboratory of System Software (Chinese Academy of Sciences); State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences; Shouling Ji, Zhejiang University; Tianyue Luo, Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences; Yanjun Wu, Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences; Key Laboratory of System Software (Chinese Academy of Sciences); State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences

<https://www.usenix.org/conference/usenixsecurity24/presentation/ling>

**This paper is included in the Proceedings of the
33rd USENIX Security Symposium.**

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

**Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.**

A Wolf in Sheep's Clothing: Practical Black-box Adversarial Attacks for Evading Learning-based Windows Malware Detection in the Wild

Xiang Ling^{1,2,3*}, Zhiyu Wu⁶, Bin Wang^{4,5}, Wei Deng⁶, Jingzheng Wu^{1,2,3}
Shouling Ji⁶, Tianyue Luo¹, and Yanjun Wu^{1,2,3}

¹Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences

²Key Laboratory of System Software (Chinese Academy of Sciences)

³State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences

⁴Zhejiang Key Laboratory of Artificial Intelligence of Things (AIoT) Network and Data Security

⁵Hangzhou Research Institute, Xidian University ⁶Zhejiang University

Abstract

Given the remarkable achievements of existing learning-based malware detection in both academia and industry, this paper presents MalGuise, a practical black-box adversarial attack framework that evaluates the security risks of existing learning-based Windows malware detection systems under the black-box setting. MalGuise first employs a novel semantics-preserving transformation of call-based redividing to concurrently manipulate both nodes and edges of malware's control-flow graph, making it less noticeable. By employing a Monte-Carlo-tree-search-based optimization, MalGuise then searches for an optimized sequence of call-based redividing transformations to apply to the input Windows malware for evasions. Finally, it reconstructs the adversarial malware file based on the optimized transformation sequence while adhering to Windows executable format constraints, thereby maintaining the same semantics as the original. MalGuise is systematically evaluated against three state-of-the-art learning-based Windows malware detection systems under the black-box setting. Evaluation results demonstrate that MalGuise achieves a remarkably high attack success rate, mostly exceeding 95%, with over 91% of the generated adversarial malware files maintaining the same semantics. Furthermore, MalGuise achieves up to a 74.97% attack success rate against five anti-virus products, highlighting potential tangible security concerns to real-world users.

1 Introduction

With the sustainable development of computer technology, the proliferation of malware, short for **malicious software**, has emerged as a grave security threat that performs malicious activities on computer systems. In particular, the widespread adoption of the Microsoft Windows family of operating systems (*i.e.*, Windows) has rendered it a primary target for malware attacks. According to AV-TEST [8], the first three quar-

ters of 2022 have witnessed approximately 59.58 million new instances of Windows malware, constituting over 95% of all recently identified malware samples during this period [7]. To defend against the ever-increasing Windows malware, considerable research efforts with cutting-edge technologies have been devoted to Windows malware detection [12, 40, 61]. Basically, Windows malware detection can trace its history back to signature-based malware detection in the 1990s, which mainly blacklists suspicious software based on a frequently updated database of previously collected malware signatures. Evidently, signature-based malware detection cannot detect new or previously unknown malware. In the past two decades, a variety of machine learning (ML) and deep learning (DL) models have been explored and employed for Windows malware detection, collectively referred to as *learning-based Windows malware detection* [40, 61] in this paper. Leveraging the high learning capacities of ML/DL models, learning-based Windows malware detection has demonstrated its ability to detect newly emerging and even zero-day malware, establishing itself as a pivotal component of contemporary mainstream anti-virus products in a fiercely competitive market [28, 43].

It is well known that signature-based malware detection can be easily evaded by traditional obfuscation techniques, such as compression, encryption, register reassignment, code virtualization, *etc.* However, with the widespread availability of de-obfuscation tools and the advanced capabilities of learning-based malware detection, traditional obfuscations are largely ineffective in evading learning-based malware detection [2, 30, 48, 63]. On the other hand, recent studies have unveiled that ML/DL models are inherently vulnerable to adversarial attacks [21, 34, 35, 37, 58] in various domains (*e.g.*, computer vision, natural language processing), by which the adversary maliciously creates adversarial examples as the input to trigger the target ML/DL model to make mistakes. Therefore, in addition to traditional obfuscations, this paper explores adversarial attacks with a specific focus on evading present-day learning-based Windows malware detection systems. It should be noted that this paper does not aim to substitute traditional obfuscations, but rather to complement

*Xiang Ling and Zhiyu Wu are the co-first authors. Bin Wang and Jingzheng Wu are the co-corresponding authors.

their limitations in evading advanced learning-based malware detection due to its widespread adoption and escalating prominence in both academia and industry [28, 43, 61].

In particular, this paper attempts to explore an adversarial attack under the realistic black-box setting for effectively and efficiently generating practical adversarial malware files, which are capable of evading learning-based Windows malware detection systems. Towards this, we identify two key challenges (**C#1** & **C#2**) that need to be addressed as follows.

- **C#1:** How to generate practical adversarial malware files that maintain the same semantics as the original ones while remaining less noticeable to possible defenders? Previous adversarial attacks mainly consist of ① adding irrelevant API calls [3, 13, 26, 54], ② manipulating raw bytes of malware partially or globally [5, 19, 31, 33, 41, 51], and ③ manipulating the control-flow graph (CFG) of malware by injecting semantic nops [63]. We argue that the first two types of adversarial attacks (① & ②) are either impractical as they generate adversarial features rather than files, or are strictly limited to specific malware detection like MalConv [48]. While the third type ③ exhibits improved scalability against CFG-based malware detection, it only considers coarse-grained transformations that manipulate CFG's nodes, rendering it quite noticeable and easily detectable by defenders. Thus, to tackle **C#1**, we propose a novel fine-grained transformation towards CFG, namely *call-based redividing*, which not only manipulates the nodes (*i.e.*, instruction blocks) but also the edges, *i.e.*, the control-flow relationships between two blocks.

- **C#2:** How to efficiently search in the vast and discrete space of malware under the black-box setting such that the optimized adversarial malware file can evade learning-based Windows malware detection? We investigate existing state-of-the-art black-box adversarial attacks, including gradient estimations with surrogate models [3, 31, 54], evolutionary algorithms [13, 41], and reinforcement learning [5, 63]. It is evident that those attacks based on gradient estimations heavily rely on prior information (*e.g.*, training data, model architecture) about the target system. Likewise, those attacks based on evolutionary algorithms and reinforcement learning are computationally expensive due to the vast and discrete space of malware. Thus, we address **C#2** by employing a Monte Carlo tree search (MCTS)-based optimization to efficiently search for the optimal adversarial malware that can successfully evade the target malware detection system.

To sum up, this paper proposes a practical adversarial attack framework, namely MalGuise, against learning-based Windows malware detection systems under the black-box setting. As depicted in Fig. 2, MalGuise first represents the input Windows malware as the CFG and introduces a novel semantics-preserving transformation of *call-based redividing* that can manipulate both nodes and edges of CFG, making it less noticeable compared to previous attacks. Then, we employ an MCTS optimization that could effectively and efficiently guide MalGuise to search for an optimal sequence of *call-*

based redividing transformations within the vast and discrete space of malware under the black-box setting. Finally, based on the optimized transformation sequence, MalGuise reconstructs the adversarial malware file while adhering to the constraints of Windows executables, thereby successfully evading the target Windows malware detection system while preserving the same semantics as the original malware.

We evaluate the attack effectiveness of MalGuise against three representative learning-based Windows malware detection systems (*i.e.*, MalGraph, Magic, and MalConv) compared with two state-of-the-art adversarial attacks on a large wild dataset containing hundreds of thousands of Windows malware and goodware samples. Evaluation results show that MalGuise is agnostic to the target learning-based Windows malware detection systems and consistently achieves a high attack success rate exceeding 95%. Meanwhile, we empirically verify that MalGuise can generate realistic adversarial malware files with a probability of over 91%, a significant improvement over previous adversarial attacks that achieved probabilities of less than 50% or failed completely. Furthermore, to understand the security risks of anti-virus products in the wild, we evaluate and observe that the attack success rate of MalGuise against five commonly used commercial anti-virus products can reach a range of 11.29% to 74.97%. To summarize, we highlight our key contributions as follows:

- To understand and assess the security risks of present-day learning-based Windows malware detection, we propose a practical black-box adversarial attack framework MalGuise that generates realistic adversarial malware files.
- To the best of our knowledge, MalGuise is the first to apply a more fine-grained transformation to the CFG of Windows malware, *call-based redividing*, which not only manipulates its nodes (*i.e.*, instruction blocks) but also its edges, *i.e.*, the control-flow relationship.
- Evaluations demonstrate that MalGuise not only effectively evades state-of-the-art learning-based Windows malware detection with attack success rates exceeding 95%, but also evades five commercial anti-virus products, achieving attack success rates ranging from 11.29% to 74.97%.

2 Preliminaries & Threat Model

2.1 Preliminaries on learning-based malware detection

Fig. 1 provides an overview of learning-based malware detection. First, as ML/DL models only operate on numeric data, the training and testing samples are pre-processed by feature engineering before inputting. Formally, feature engineering can be formulated as $\phi: \mathcal{Z} \rightarrow \mathcal{X}$, which produces a feature vector x 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, using training samples, the ML/DL model is employed and trained as the learning-based malware detection $f: \mathcal{Z} \rightarrow \mathcal{Y}$.

That is, given an executable $z \in \mathcal{Z}$, f can predict a corresponding label y in the *label-space* \mathcal{Y} (i.e., $y \in \mathcal{Y} = \{0,1\}$), such that $f(z) = y$, in which $y=0$ denotes goodwill while $y=1$ denotes malware. For ease of notations like [36, 40], we denote the learning-based malware detection that can return the malicious probability as $g: \mathcal{Z} \rightarrow \mathbb{R}$, in which $g(z)$ denotes the predicted malicious probability for z and the opposite benign probability is naturally inferred as $1-g(z)$.

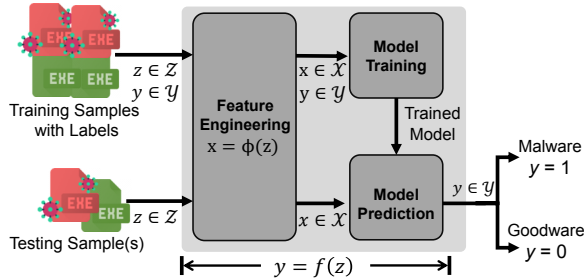


Figure 1: A overview of learning-based malware detection.

2.2 Threat Model

Following the widely used framework of modeling threats in adversarial machine learning [47], we report our threat model in terms of goal, knowledge, and capability as follows.

Adversary’s Goal. Aiming at evading learning-based malware detection, it is hugely profitable for the adversary to misclassify malware as goodwill, but not vice versa. Therefore, the primary goal of the adversary is to generate an adversarial malware file z_{adv} derived from an input malware $z \in \mathcal{Z}$ (i.e., $f(z) = 1$) with minimal efforts, such that $z_{adv} \in \mathcal{Z}$ can not only evade the target learning-based malware detection f (i.e., $f(z_{adv}) = 0$) but also preserve the same semantics as z [40].

Adversary’s Knowledge and Capability. We start with an adversary who intends to perform the classic *zero-knowledge black-box attack* [42, 47] against the learning-based malware detection system. It indicates that the adversary has no prior information on the target system in terms of training data, extracted feature sets, employed learning algorithms with parameters, and model architectures with weights. However, it should be clarified that the zero-knowledge black-box attack retains some minimal information about the target system, including the specific detection task (e.g., static or dynamic analysis), the employed feature type that represents executables (e.g., image, sequence, or graph), and the querying feedback. In this paper, we restrict the adversary’s knowledge to only knowing that our target system is based on static analysis that focuses on advanced abstract graph representations like CFG, and knowing the predicted label $f(z)$ along *with* or *without* its probability $g(z)$ after inputting z . Furthermore, to ensure that realistic adversarial malware files can be generated, the adversary has the capability of manipulating Windows executables while adhering to its standard specifications [45].

3 Design of MALGUISE

3.1 Problem Formulation

Targeting learning-based Windows malware detection, the primary goal of this paper is to generate a realistic adversarial malware file $z_{adv} \in \mathcal{Z}$ from a given malware z , such that z_{adv} is not only misclassified as goodwill but also maintains the same semantics as z . To this end, we explore the problem-space attack [47] that applies *semantics-preserving transformations* within the problem space that can transform z to z_{adv} step-by-step. In particular, as formulated in Eq. (1), we start to define the first black-box attack scenario with the malicious probability $g(\cdot)$ by reducing the probability that z_{adv} is predicted as malicious as much as possible.

$$\arg \max_{\mathbf{T}} g(z) - g(z_{adv}) \quad \blacktriangleright \text{with predicted probabilities} \quad (1)$$

$$\arg \min_{\mathbf{T}} f(z_{adv}) \quad \blacktriangleright \text{without predicted probabilities} \quad (2)$$

$$\text{s. t.: } f(z) = 1, f(z_{adv}) = 0, z_{adv} = \mathbf{T}(z) \in \mathcal{Z} \quad (3)$$

$$\mathbf{T} = T_1 \circ T_2 \circ \dots \circ T_n \in \mathbb{T} \quad (4)$$

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

Similarly, Eq. (2) defines another stricter black-box setting where the adversary can only obtain the predicted label $f(\cdot)$ without any predicted probabilities, which simply minimizes the predicted label of z_{adv} to 0 since 0 denotes goodwill.

3.2 Overview Framework of MALGUISE

As depicted in Fig. 2, the overview framework of MalGuisE mainly consists of three backbone phases, i.e., ① adversarial transformation preparation, ② MCTS-guided searching, and ③ adversarial malware reconstruction, which are elaborated in the following parts of §3.2.1, §3.2.2, and §3.2.3, respectively.

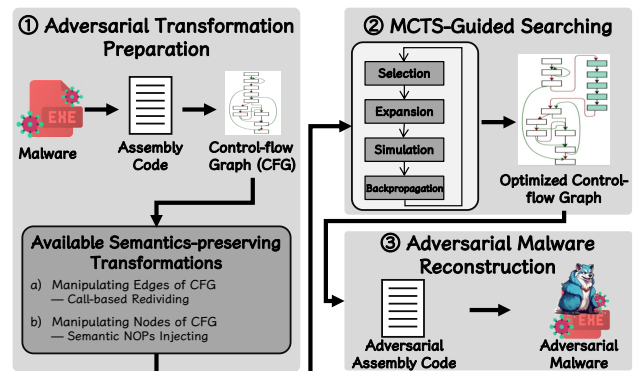


Figure 2: The overview framework of MalGuisE.

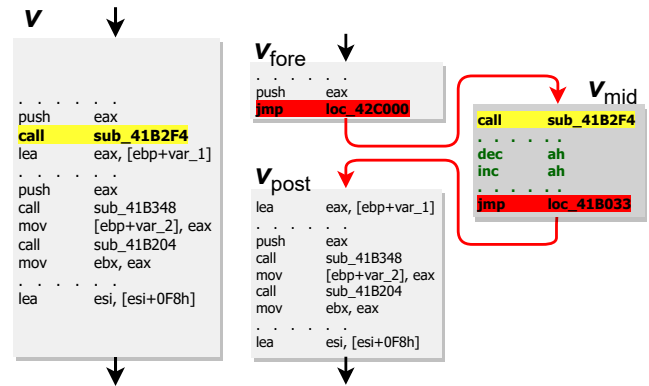
3.2.1 Adversarial Transformation Preparation

To prepare adversarial transformations that manipulate the given malware while preserving its semantics, we initially represent a malware sample as the CFG, in which each node denotes a basic block and each edge denotes a control-flow path between two basic blocks during execution. The main reason to represent malware with CFG is twofold. i) CFG encapsulates the intrinsic control flows during execution, containing rich semantic and structural information of assembly instructions. Therefore, if we can manipulate both CFG’s nodes and edges, we can change both the semantics and structures for generating possible adversarial malware files in a more fine-grained fashion, thereby making them less noticeable by possible defenders. ii) CFG has gained widespread adoption in various software analysis tasks [39], and CFG-based malware detection is extensively proven to be technically advanced and highly effective in both industry and academia [23, 38, 59]. Therefore, if we could successfully attack these advanced CFG-based malware detection as representative cases, it would demonstrate the state-of-the-art attack effectiveness of MalGuise.

However, if we transform one executable into another by directly manipulating its CFG, it is extremely easy to cause various unexpected issues like addressing or processing errors, rendering the transformed executable unable to execute properly or even crashing immediately. To circumvent these issues and avoid being noticeable by potential defenders, we propose a novel semantics-preserving transformation for Windows executables, namely *call-based redividing*. It *redivides* the basic blocks that contain at least one “call” instruction for concurrently manipulating both nodes and edges in the CFG.

To be specific, for a given executable, the *call-based redividing* first identifies and annotates all available basic blocks with “call” instruction(s). Then, supposing there is a basic block v containing a “call” instruction, it takes the “call” as the dividing line, and attempts to redivide the original basic block v into a combination of three consecutive basic blocks (*i.e.*, the fore-basic-block v_{fore} , the post-basic-block v_{post} , and the mid-basic-block v_{mid}). Finally, to avoid the basic block of v_{mid} being easily noticed by defenders due to having only two instructions of “call” and “jmp”, *call-based redividing* enriches the assembly instructions in v_{mid} by injecting semantic nops [14, 41] before the “call” or between the “call” and “jmp”, and employs the context-free grammar from [41] to generate the semantic nops diversely.

As illustrated in Fig. 3, we first show one basic block of the latest “LockBit 3.0”, an active and famous ransom gang, as a representative example in Fig. 3(a). After applying the transformation, Fig. 3(b) shows the transformed composite of three consecutive basic blocks, in which the ends of two basic blocks (*i.e.*, v_{fore} and v_{mid}) are two newly added “jmp” instructions, and those assembly instructions between “call” and “jmp” in v_{mid} are newly added semantic nops.



(a) Before transformation. (b) After applying a call-based redividing.

Figure 3: The *call-based redividing* 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)).

3.2.2 MCTS-Guided Searching

Recalling our adversarial attack formulated in § 3.1 and the *call-based redividing* transformation towards the CFG representation (*i.e.*, $x = \phi(z)$) of a given malware z in § 3.2.1, we decompose MalGuise into first finding an optimized transformation sequence T that consecutively transforms the original CFG x into an adversarial CFG of x_{adv} (detailed in this part of § 3.2.2), and then reconstructing the final adversarial malware file (*i.e.*, z_{adv}), which will be detailed in § 3.2.3.

In essence, the optimal solution we are solving here is an optimized sequence of transformations $T = T_1 \circ \dots \circ T_N$ of length N , and each $T_k = \{\mathbb{I}_k^{call}, \mathbb{I}_k^{s-nops}\}$ involves two decision-making processes: i) Selecting one of all available “call” instructions to be redivided, *i.e.*, \mathbb{I}_k^{call} , and it should be noted that \mathbb{I}_k^{call} can be *repeatedly* selected in a recursive manner; ii) Determining the proper semantic nops to be injected, namely \mathbb{I}_k^{s-nops} , and it can be generated *infinitely* by the employed context-free grammar [41]. In short, determining an optimal T in MalGuise requires exploring and optimizing in an *infinite* and *discrete* space with a limited computational budget under the *black-box* setting. To this end, our key idea is to employ an MCTS-guided searching algorithm [17] for two major reasons. First, MCTS has been widely and successfully used to solve the long-standing challenging problem of computer Go [17, 49] and other difficult optimization problems that require little or no domain knowledge [10]. Second, our task of optimizing T under the black-box setting is strictly limited to searching in an infinite and discrete space without prior knowledge. Therefore, we argue that an MCTS-guided searching algorithm aligns well with our task requirements.

Algorithm 1 presents the MCTS-guided searching algorithm, which inputs a given malware z with its CFG x and outputs the transformation sequence T . Firstly, we obtain all available instructions \mathbb{I}^{call} from x and initialize the MCTS’s root node z and T (line 2–3). Meanwhile, we limit the max-

Algorithm 1: MCTS-Guided Searching Algorithm.

Input : a given malware z with its CFG x , target system f , max length N , simulation number S , budget C .
Output : the transformation sequence \mathbf{T} .

```
1 Begin
2  $\mathbb{I}^{\text{call}} \leftarrow \text{GetAllCalls}(x)$ ;
3  $v, \mathbf{T} \leftarrow \text{InitMCTSRootNode}(x, \mathbb{I}^{\text{call}}, \emptyset$ ; //initialize
4 for  $i \leftarrow 1$  to  $N$  do //loop upto maximum length
5   for  $j \leftarrow 1$  to  $C$  do //loop upto computation budget
6     if  $\text{random}(0,1) < 0.5$  then //avoid unlimited expansion
7        $v_{\text{selected}} \leftarrow \text{Selection}(v)$ ;
8     else
9        $v_{\text{selected}} \leftarrow \text{Expansion}(v)$ ;
10       $\text{reward} \leftarrow \text{Simulation}(v_{\text{selected}}, f, S)$ ;
11       $\text{BackPropagation}(v_{\text{selected}}, \text{reward})$ ;
12       $v_{\text{node}} \leftarrow \text{ChildWithHighestReward}(v)$ ;
13       $\mathbf{T} \leftarrow \mathbf{T}.\text{append}(v_{\text{node}}.T)$ ;
14       $x_{\text{adv}} \leftarrow v_{\text{node}}.x$ ;
15      if  $\text{Evaded}(f, x_{\text{adv}}) == \text{True}$  then
16        return  $\mathbf{T}$ 
17       $v \leftarrow v_{\text{node}}$ 
```

imum length of the optimized transformation sequence to N (line 4–17) and limit the maximum number of iterations of MCTS to C , *i.e.*, the computational budget (line 5–11). As for the MCTS optimization process, we follow the four standard steps (*i.e.*, Selection, Expansion, Simulation, and Backpropagation) (line 6–11). It is noted that, as call-based redividing can be performed unlimitedly, the game tree of MCTS can be unlimitedly expanded downwards, *i.e.*, Expansion. Therefore, we force to select the most promising child node (*i.e.*, Selection) in the established game tree via a simple random sampling (line 6). After C iterations, we can thus obtain the child node v_{node} with the highest reward value, append its transformation into the transformation sequence \mathbf{T} , and update the adversarial CFG x_{adv} (line 12–14). Finally, if x_{adv} evades f , return the transformation sequence \mathbf{T} , otherwise, continue to use v_{node} as the root node for the next round until reaching the maximum length N . For simplicity, more implementation details can be found in Appendix A.

3.2.3 Adversarial Malware Reconstruction

Finally, we reconstruct the adversarial malware $z_{\text{adv}} = \mathbf{T}(z)$ based on the original malware file z and the optimized transformation sequence \mathbf{T} , which is briefly outlined in Algorithm 2. It is noted that each transformation $T_k = \{\mathbb{I}_k^{\text{call}}, \mathbb{I}_k^{\text{s-nops}}\}$ and we denote A_k^{call} as the address of $\mathbb{I}_k^{\text{call}}$. We first calculate the space required for all call-based redividing transformations in \mathbf{T} as Δ (line 2–6). Meanwhile, let S_{slack} and A_{slack} denote the size and address of the slack space in the “.text” section of z , respectively. Similarly, let A_{last} and S_{last} denote the address and size of the last section. If Δ is less than the size of slack space, we directly take A_{slack} as the

Algorithm 2: Adversarial Malware Reconstruction

Input : original malware z , the transformation sequence \mathbf{T} .
Output : the adversarial malware z_{adv} .

```
1 Begin
2  $\Delta \leftarrow 0$ ;
3 for  $k \leftarrow 1$  to  $N$  do
4    $\Delta \leftarrow \Delta + \text{GetSize}(\mathbb{I}_k^{\text{call}}) + \text{GetSize}(\mathbb{I}_k^{\text{s-nops}})$ ;
5    $\mathbb{I}_k^{\text{jmp}} \iff \text{jmp}_{\lfloor A_k^{\text{call}} + \text{GetSize}(\mathbb{I}_k^{\text{call}}) \rfloor}$ ;
6    $\Delta \leftarrow \Delta + \text{GetSize}(\mathbb{I}_k^{\text{jmp}})$ ;
7 if  $\Delta < S_{\text{slack}}$  then //inject into the slack space
8    $A^{\text{inj}} \leftarrow A_{\text{slack}}$ 
9 else //inject into a new section
10   $A^{\text{inj}} \leftarrow A_{\text{last}} + \text{RoundUp}(S_{\text{last}}, \text{page\_size})$ ;
11  take actions to meet the standard specifications;
12   $z_{\text{adv}} \leftarrow \text{Adv\_Patch}(z, \mathbf{T}, A^{\text{inj}})$ ; //refer to Algorithm 3
13  return  $z_{\text{adv}}$ .
```

injecting address A^{inj} (line 7–8). Otherwise, we will add a new section, whose starting injection address is computed as $A^{\text{inj}} = A_{\text{last}} + \text{RoundUp}(S_{\text{last}}, \text{page_size})$ (line 9–10). This is because, according to the standard Windows executable specifications [45], its section size must be a multiple of the architecture’s page size (*i.e.*, 4KB for x86 and MIPS), to prevent unexpected issues (*e.g.*, addressing errors) that may arise during execution. Apart from this, other subsequent actions should be taken to meet the Windows specifications, such as setting the size for the “.text” section or the newly added section and adjusting other fields (*e.g.*, “size of image”) in the header of Windows executables.

After obtaining the injecting address A^{inj} , we reconstruct z_{adv} with the procedure of Adv_Patch, which is mainly outlined Algorithm 3. In particular, we first replace the selected $\mathbb{I}_k^{\text{call}}$ instruction with a new jmp instruction (*i.e.*, “ $\text{jmp}_{\lfloor A_k^{\text{inj}} \rfloor}$ ”), which transfers the control-flow to the injecting address for the k -th transformation in the new destination, *i.e.*, A_k^{inj} (line 4). Secondly, starting from the injecting address A_k^{inj} of either the slack space or the newly added section, we deposit $\mathbb{I}_k^{\text{call}}$ and $\mathbb{I}_k^{\text{s-nops}}$ in order (line 5–7). Subsequently, we inject a new jmp instruction (*i.e.*, $\mathbb{I}_k^{\text{jmp}} \iff \text{jmp}_{\lfloor A_k^{\text{call}} + \text{GetSize}(\mathbb{I}_k^{\text{call}}) \rfloor}$), which transfers the control-flow backward to the next instruction of $\mathbb{I}_k^{\text{call}}$ in the original section (line 8–9). Finally, after all of N transformations are applied, we can reconstruct the final adversarial malware z_{adv} that preserves the same semantics as the original malware z .

Fig. 4 shows the conceptual layout of the generated adversarial “LockBit” after taking one call-based redividing transformation. It is important to note that, just applying two optimized transformations of call-based redividing to “LockBit 3.0”, the generated adversarial “LockBit” can successfully evade three state-of-the-art learning-based malware detection (*i.e.*, MalGraph, Magic, and MalConv) and the famous anti-virus product of Kaspersky in our evaluation.

Algorithm 3: Procedure of $\text{Adv_Patch}(z, \mathbf{T}, A^{inj})$

```

1 Begin
2    $z_{adv}, A_1^{inj} \leftarrow z, A^{inj};$ 
3   for  $k \leftarrow 1$  to  $N$  do
4      $z_{adv} \leftarrow \text{Patch}(z_{adv}, A_k^{call}, \text{jmp}_{\lfloor A_k^{inj} \rfloor});$ 
5      $z_{adv} \leftarrow \text{Patch}(z_{adv}, A_k^{inj}, \mathbb{I}_k^{call});$ 
6      $A_k^{inj} \leftarrow A_k^{inj} + \text{GetSize}(\mathbb{I}_k^{call});$ 
7      $z_{adv} \leftarrow \text{Patch}(z_{adv}, A_k^{inj}, \mathbb{I}_k^{s-nops});$ 
8      $A_k^{inj} \leftarrow A_k^{inj} + \text{GetSize}(\mathbb{I}_k^{s-nops});$ 
9      $z_{adv} \leftarrow \text{Patch}(z_{adv}, A_k^{inj}, \mathbb{I}_k^{jmp});$ 
10     $A_{k+1}^{inj} \leftarrow A_k^{inj} + \text{GetSize}(\mathbb{I}_k^{jmp});$ 
11  return  $z_{adv}$ 

```

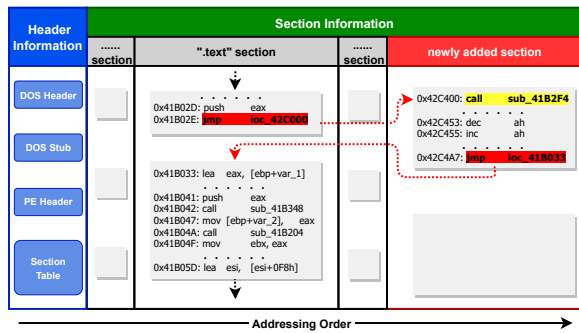


Figure 4: The conceptual layout of the reconstructed adversarial Windows malware file for the “LockBit 3.0” ransomware.

4 Evaluation

This section is dedicated to conducting evaluations aiming at answering the following five research questions:

- **RQ1 (Attack Performance):** What is the attack performance of MalGuise against the state-of-the-art learning-based Windows malware detection systems?
- **RQ2 (Impacting Factors):** What impacting factors affect the attack performance of MalGuise?
- **RQ3 (Utility Performance):** Does the adversarial malware generated by MalGuise maintain the original semantics?
- **RQ4 (Real-world Performance):** To what extent does MalGuise evade existing commercial anti-virus products?
- **RQ5 (Possible Defenses):** What is the attack performance of MalGuise against potential defenses?

4.1 Evaluation Setup

4.1.1 Benchmark Dataset

We utilize the same benchmark dataset as employed in prior studies [38]. This dataset is a balanced wild dataset of 210,251 Windows executables with 108,610 goodware and 101,641 malware, consisting of 848 different malware families. As summarized in Table 1, we split it into three disjoint train-

ing/validation/testing sets and more details are in Appendix B.

Table 1: Summary statistics of the benchmark 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 Systems with Detecting Performance

We evaluate the attack performance of MalGuise against two kinds of target systems (*i.e.*, learning-based malware detection systems and real-world anti-virus products) as follows.

• Learning-based Windows malware detection systems.

We first employ three SOTA learning-based malware detection (*i.e.*, MalGraph [38], Magic [59], MalConv [48]) from either top-tier academic conferences or highly cited publications. To avoid possible biases, we directly adopt their publicly available implementations with default hyper-parameters and evaluate their detecting performance with three commonly used metrics, *i.e.*, AUC, TPR/FPR, and balanced Accuracy (bACC). Table 2 shows that all of them show similar detecting performance as presented in their original publications, affirming their excellent performance in detecting malware.

Table 2: The detecting performance of three learning-based Windows malware detection systems in our testing dataset.

Target Systems	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

• **Anti-virus products.** We additionally employ five anti-virus products, *i.e.*, McAfee, Comodo, Kaspersky, ClamAV, and Microsoft Defender ATP (MS-ATP) [44], due to their widespread recognition in the security community of Windows malware detection. In particular, McAfee, Comodo, and Kaspersky are three award-winning commercial anti-viruses recommended in [52]. ClamAV [16] is the most popular open-sourced anti-virus engine, which has been extensively employed in both academia and industry. MS-ATP [44] is a learning-based security protection tool for Windows and is awarded a perfect 5-star rating by [4]. Notably, we do not employ VirusTotal as it strongly advises against using their anti-viruses for comparative evaluations [55].

4.1.3 Baseline Attacks

We compare MalGuise with two kinds of baseline attacks, *i.e.*, adversarial attacks and obfuscations. In principle, to facilitate

fair comparisons, we follow the same evaluation settings of all baseline attacks as their publications or implementations.

- **Adversarial attacks:** We employ two SOTA adversarial attacks: i) MMO [41] is a white-box adversarial attack that uses gradient-based optimizations to guide binary diversification tools to manipulate the raw bytes of Windows malware. Its maximum iteration number is 200 and the increment rate of adversarial malware size is limited to 5%. ii) SRL [63] is a black-box adversarial attack that employs reinforcement learning to iteratively inject semantic nops into the malware’s CFG. Its maximum iteration number, the injection budget, and the modified basic blocks are set to be 30, 5%, and 1250, respectively. Notably, SRL only generates the adversarial CFG features rather than realistic malware files, making it incompatible with MalConv, which requires raw bytes as inputs.

- **Obfuscations:** We employ three obfuscation tools that have been widely employed to obfuscate Windows executables and follow their default settings for evaluations. In particular, UPX [53] is an open-source and universal packing tool for executables by performing compression on the entire file; VMProtect [56] typically uses code virtualization for obfuscations via simulating a virtual machine executing the key part in executables; Enigma [20] employs a combination of multiple obfuscation techniques, *e.g.*, import table elimination, API simulation, code virtualization, *etc.*

4.1.4 Evaluation Metrics

We employ two kinds of evaluation metrics as follows.

- **Attack Success Rate (ASR).** ASR is the most common evaluation metric for adversarial attacks [37, 41, 63]. Given a candidate malware set \mathbf{Z} , ASR is defined as the ratio of generated adversarial malware that successfully evades the target system (*i.e.*, $f(z_{adv})=0$) among all malware (*i.e.*, $f(z)=1$).

$$ASR = \frac{|(f(z)=1) \wedge (f(z_{adv})=0)|}{|f(z)=1|}, \forall z \in \mathbf{Z} \quad (5)$$

where $|\cdot|$ counts the number that meets the condition.

- **Semantics Preservation Rate (SPR).** As the generated adversarial malware might not preserve the same semantics as the original malware, *i.e.*, it cannot be executed or lose the original malicious behaviors. To this end, we define SPR as the ratio of adversarial malware with the original semantics preserved among all adversarial malware as follows.

$$SPR = \frac{|Sem(z, z_{adv})=1|}{|(f(z)=1) \wedge (f(z_{adv})=0)|}, \forall z \in \mathbf{Z} \quad (6)$$

where $Sem(z, z_{adv})=1$ denotes z_{adv} and z maintain the same semantics. §4.2.3 will detail how to measure SPR empirically.

4.1.5 Implementation Details

MalGuise is primarily implemented with Python and evaluated on a computer equipped with 20 Intel Xeon CPUs, 128

GB memory, and 4 NVIDIA GeForce RTX 3090. Firstly, MalGuise uses IDAPython in IDA Pro 6.4 [24] to disassemble Windows executables and represent them as CFGs. In the MCTS-guided searching algorithm, by default, 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 size. To reconstruct the adversarial malware file, we mainly employ two Python libraries, `pefile`¹ and `LIEF`², to parse and patch Windows executables. It is worth noting that, those injected semantic nops can be generally divided into four categories: arithmetic (*e.g.*, “`add_eax, 1; sub_eax, 1`”), logical (*e.g.*, “`add_eax, eax`”), comparison (*e.g.*, “`cmp_eax, eax`”), and data transfer (*e.g.*, “`push_eax; pop_eax`”).

4.2 Evaluation Results & Analysis

4.2.1 Answer to RQ1 (Attack Performance)

To assess the attack performance of MalGuise, we evaluate it by comparing its ASR performance with all baseline attacks on all 10,000 testing malware samples from the benchmark dataset, which is illustrated in Table 3. Recall in §4.1.3, MMO is a white-box adversarial attack, serving as an upper bound to assess the attack performance of its black-box attacks. All three obfuscations only apply to the *w/o prob.* scenario as they do not require any feedback from the target system.

Table 3: The ASR performance (%) comparisons between MalGuise and baseline attacks against three target systems under two black-box scenarios, *i.e.*, *w/ prob.* and *w/o prob.*

Black-box Scenarios	Attacks	MalGraph		Magic		MalConv	
		FPR =1%	FPR =0.1%	FPR =1%	FPR =0.1%	FPR =1%	FPR =0.1%
<i>w/ prob.</i>	MMO	15.55	52.30	12.82	40.13	11.99	39.66
	SRL	2.39	19.59	25.38	86.77	—	—
	MalGuise	97.47	97.77	99.29	99.42	34.36 (97.76)	97.38 (99.77)
<i>w/o prob.</i>	MMO	3.73	27.83	3.41	25.46	2.46	20.72
	SRL	2.59	15.28	3.84	47.48	—	—
	UPX	0.55	4.43	3.30	39.80	0.31	9.32
	VMProtect	0	0	0.23	4.33	0	0
	Enigma	0.81	11.69	0	28.96	0	0.24
	MalGuise	96.84	96.49	99.27	99.07	31.41 (95.18)	88.02 (99.77)
	MalGuise	96.84	96.49	99.27	99.07	31.41 (95.18)	88.02 (99.77)

“—” means SRL does not apply to MalConv as it cannot generate real malware files.

For all attacks, Table 3 shows that lower FPR values for the three target systems correspond to higher ASRs achieved by each attack. The reason is evident that a lower value of

¹pefile: <https://github.com/erocarrera/pefile>

²LIEF: <https://lief-project.github.io>

FPR allowed by the binary classifier indicates it has a higher threshold. Hence, the adversary can more easily reduce the predicted malicious probability to a level below the threshold. Additionally, it is evident that each attack performs no better in the scenario of *w/o prob.* than in the scenario of *w/ prob.*

Adversarial attacks: MMO achieves low ASR performance against all three target systems in both attack scenarios. Particularly, in the case of FPR=1% for all target systems, the ASRs of MMO are below 16% and 4% in the scenarios of *w/ prob.* and *w/o prob.*, respectively. These imply that, although MMO theoretically can be used against all target systems, it shows inferior attack performance against them. The main reason we conjecture is that, since MMO manipulates the entire raw bytes of malware with binary diversification techniques in general, it does not take into account the discriminative features employed in different target systems.

For SRL, in both scenarios of *w/ prob.* and *w/o prob.*, it shows obviously higher ASRs against Magic than MalGraph in both FPRs. This is mainly because SRL is specifically designed to attack Magic which purely builds on CFG. However, the hierarchical nature of MalGraph that combines both the function call graph and CFG further weakens the attack performance of SRL, as SRL only manipulates nodes of CFG and neglects to manipulate its edges.

Obfuscations: It is observed that all three obfuscations show inferior attack performance, which once again validates that traditional obfuscation tools are largely ineffective against learning-based malware detection [2, 30, 48]. Specifically, VMProtect achieves the worst attack performance against all three target systems, while UPX and Enigma show slightly better attack performance, but remain unsatisfactory with all achieved ASRs not exceeding 40%. This is mainly because, VMProtect typically obfuscates only a small portion of the malware file [56, 57], while both UPX and Enigma can globally obfuscate the entire malware file, which slightly increases the likelihood of altering the discriminative semantic features. However, as UPX and Enigma remain unknown without the security experts' intervention, it is still challenging to purposefully manipulate the discriminative semantic features. More evaluations in terms of size alteration ratio are in Appendix C.

Compare MalGuise with all baseline attacks: Table 3 shows that MalGuise achieves the best attack performance against all three target systems in all scenarios and cases. When attacking MalGraph and Magic in all scenarios and cases, all ASR values achieved by MalGuise exceed 97%. More importantly, even in the strict attack scenario of *w/o prob.*, MalGuise still maintains its ASR performance nearly unchanged, *i.e.*, decrease by no more than 1%, compared to the scenario of *w/ prob.* When attacking MalConv, although the ASR performance of MalGuise is much better than that of all baseline attacks, its ASR is still relatively poor (*i.e.*, below 35%) in the case of FPR=1%. However, when investigating MalGuise in the subsequent §4.2.2, we find that its attack performance against MalConv is highly dependent on

the injected semantic nops. Thus, by incorporating the 25 most frequently used semantic nops, MalGuise is improved to achieve higher ASRs of 97.76% and 95.18% against MalConv in both *w/ prob.* and *w/o prob.* scenarios, respectively.

Answer to RQ1 (Attack Performance): Prior obfuscations and adversarial attacks either cannot provide satisfactory attack effectiveness or fail to scale well to different types of learning-based Windows malware detection. However, even in the strict black-box attack scenario (*w/o prob.*), MalGuise is agnostic to learning-based Windows malware detection with a high attack success rate exceeding 95% in most cases.

4.2.2 Answer to RQ2 (Impacting Factors)

We conduct a series of ablation studies to explore the impacting factors that affect the attack performance of MalGuise.

Impact of key parts in call-based redividing. The core of MalGuise is to apply the transformation of *call-based redividing*, which primarily involves two key parts: † *injecting semantic nops* and ‡ *redividing call instructions*, for manipulating the CFG's *nodes* and *edges*, respectively. To examine the impact of the above two parts, we compare the ASR performance of MalGuise with its two variants that apply only one of the key parts (*i.e.*, MalGuise[†] and MalGuise[‡]) under the attack scenario of *w/ prob.*

From Table 4, both MalGuise[†] and MalGuise[‡] show almost negligible attack performance (with ASRs below 5%) against both MalGraph and Magic in both FPRs. When attacking MalConv, MalGuise[‡] still shows inferior attack performance, while MalGuise[†] shows better attack performance, achieving ASRs of 23.04% and 79.86% in the cases of FPR=1% and FPR=0.1%, respectively. The reason is evident that the essence of MalGuise[†] is to inject semantic nops into malware, which can significantly and directly alter their raw bytes. Thus, MalGuise[†] can easily affect the detecting performance of MalConv which inputs the raw bytes of malware. Finally, comparing with the above two variants, MalGuise achieves considerably higher attack performance with over 97% ASR in all cases. More evaluations against anti-viruses are in Appendix C. All the above observations indicate that MalGuise with only manipulating either CFG's nodes or edges demonstrates inferior attack performance, and only by combining both can the attack performance of MalGuise be maximized.

Impact of the number of modified basic blocks in CFG. We investigate the impact of the number of modified basic blocks for all generated adversarial malware that successfully evade the three learning-based malware detection when FPR=1%, and show their corresponding frequencies in Fig. 5. It is observed that over 98% of all adversarial malware files that successfully evade the target systems only require modifying no more than four basic blocks in its CFG. In particular, MalGuise only needs to modify one basic block to make 61% of malware samples evade Magic, and only modify two basic blocks to make 79% and 56% of malware samples evade

Table 4: The ASR performance (%) of three MalGuise variants against three target systems under the scenario of *w/prob*.

MalGuise Variants	MalGraph		Magic		MalConv	
	FPR=1%	FPR=0.1%	FPR=1%	FPR=0.1%	FPR=1%	FPR=0.1%
MalGuise [†]	0.02	4.35	1.70	3.34	23.04	79.86
MalGuise [‡]	0.10	4.87	0.10	1.42	0.17	1.86
MalGuise	97.47	97.77	99.29	99.42	34.36	97.38
					97.76	99.77

MalGraph and MalConv, respectively. These evaluations indicate that, by only modifying a small number of basic blocks in CFG, MalGuise could offer excellent attack performance against all three learning-based malware detection systems.

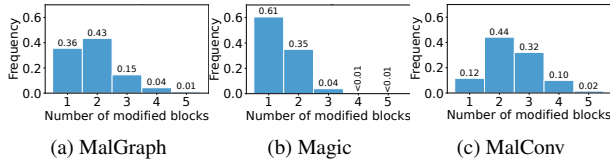


Figure 5: Frequency of the number of modified basic blocks of all adversarial malware that evades the three target systems.

Impact of different types of semantic nops. When performing MalGuise against MalConv, we examine the impact of different opcodes in the injected semantic nops by presenting the occurrence frequency of different opcodes that lead to successful evasions in Fig. 6. It is observed that, some opcodes (*e.g.*, `dec/inc`, `xor`) occur with a relatively high frequency, while some others (*e.g.*, `cmp` and `test`) occur with a lower frequency. Next, we limit MalGuise to use the 25 most frequently used opcodes, it is observed from Table 3 that the ASR performance of MalGuise ASR against MalConv has significantly increased, highlighting the importance of different types of semantic nops in MalGuise. To sum up, we can conclude that the attack performance of MalGuise can be significantly improved by fine-tuning the types of semantic nops to be injected. More evaluations about the impact of the size of semantic nops for MalGuise are discussed in Appendix C.

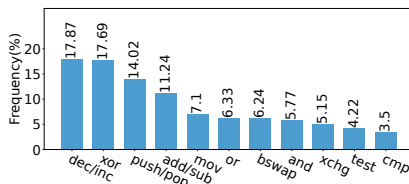


Figure 6: Occurrence frequency of different opcodes in semantic nops that lead to successful evasions against MalConv.

Impact of the hyper-parameters in MCTS. We investigate the impact of the hyper-parameters in MCTS (*i.e.*, C

and N in Algorithm 1) for MalGuise. First, for the computation budget C , we follow the same implementation settings detailed in §4.1.5 and only vary C to 10, 20, 30, 40, 50, and 60. From Fig. 7, it is observed that, as C increases from 10 to 20, the ASRs of MalGuise rise sharply up to over 97% against all three target systems. When C reaches 20, its ASR performance tends to stabilize at a high value of over 97%.

Similarly, for the max length N , we only vary N (*i.e.*, 2, 4, 6, 8, and 10) to show its impact on MalGuise in Fig. 8. It is observed that when $N=2$, the overall attack performance of MalGuise against all three target systems is pretty well, with ASRs exceeding 97%. With the increase of N , the ASRs of MalGuise remain stable at a high value of over 97% against all target systems. These observations indicate that MalGuise can achieve high attack performance even with no more than two transformations of call-based redividing employed.

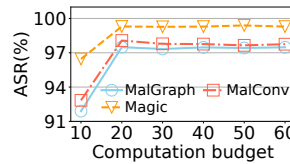


Figure 7: Impact of the computation budget C .

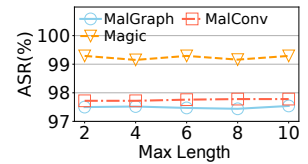


Figure 8: Impact of the max length N .

Answer to RQ2 (Impacting Factors): To sum up, targeting different learning-based malware detection systems, different hyper-parameters in MalGuise have different impacts on the attack performance, and thereby we can fine-tune its hyper-parameters to enhance its attack performance.

4.2.3 Answer to RQ3 (Utility Performance)

As previously defined in Eq. (6), we evaluate the utility performance of MalGuise and two baseline adversarial attacks in terms of semantics preservation rate (SPR), whose core is to judge whether the adversarial malware z_{adv} has the same semantics as the original malware z , *i.e.*, $Sem(z, z_{adv})$. Due to the inherent complexity of executables, there is no exact solution to judge $Sem(z, z_{adv})$ [6] and we resort to an empirical verification to judge it by collecting and comparing the two API sequences (*i.e.*, $API_{z_{adv}}$ and API_z) invoked by both z_{adv} and z when they are run on the Cuckoo sandbox [18]. As shown in Eq. (7), to quantify the semantic difference between z_{adv} and z , we thus compute a normalized edit distance $dist_{norm}(z, z_{adv})$ between the two API sequences as follows.

$$dist_{norm}(z, z_{adv}) = \frac{Distance(API_z, API_{z_{adv}})}{\max(l(API_z), l(API_{z_{adv}}))} \in [0, 1] \quad (7)$$

in which $Distance(API_z, API_{z_{adv}})$ denotes the edit distance between two sequences and $l(\cdot)$ denotes the sequence length.

However, since malware may perform random actions during execution [27], the API sequences collected by running

the same malware z twice in the same sandbox may be different, which means $dist_{norm}(z, z)$ almost can not take the value of 0. Therefore, we calculate the value of $Sem(z, z) \in \{0, 1\}$ by comparing the $dist_{norm}(z, z)$ with a general distance threshold $dist_{\Delta}$. To determine $dist_{\Delta}$, we first analyze all original malware samples in the same sandbox twice and then select the value at the 99.5-th percentile among all the corresponding $dist_{norm}(z, z)$ as $dist_{\Delta}$. After that, as shown in Eq. (8), 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 $dist_{\Delta}$, and further evaluate SPR according to Eq. (6).

$$Sem(z, z_{adv}) = \begin{cases} 1 & \text{if } dist_{norm}(z, z_{adv}) < dist_{\Delta} \\ 0 & \text{otherwise.} \end{cases} \quad (8)$$

Due to the extreme resource and time consumption of the above evaluation process, we randomly select 10% from all adversarial malware that successfully evades the corresponding target system in §4.2.1 for the subsequent evaluations and present the evaluation results in Table 5. Apparently, evaluating the utility performance of SRL is not applicable as it only generates adversarial features rather than realistic adversarial malware files. As for MMO, it is observed that the achieved SPR values against three target systems are at a low level, *i.e.*, approximately ranging from 40% to 50%. It indicates, only less than 50% of adversarial malware generated by MMO preserve their original semantics. However, MalGuise achieves the best utility performance of over 91% SPR for all three target systems, which demonstrates the best effectiveness of MalGuise in preserving their original semantics.

Table 5: The SPR (%) of MalGuise and two baseline adversarial attacks against three target systems.

Attacks	MalGraph		Maigc		MalConv	
	FPR=1%	FPR=0.1%	FPR=1%	FPR=0.1%	FPR=1%	FPR=0.1%
MMO	41.8	49.4	39.6	39.8	39.2	50.8
SRL	—	—	—	—	—	—
MalGuise	91.84	91.99	93.45	92.28	92.67	91.68

Causes for 9% failures. To investigate the causes behind the approximately 9% of failures in preserving the same semantics in the above evaluations, we conduct manual inspections using IDA Pro [24] and OllyDbg [62] and reveal two primary reasons for these failures. First, there are a few malware samples that contain *overlay*, which is not part of the official Windows executable format but is usually used to perform malicious behaviors in malware. Despite adhering to the standard Windows executable format specifications, the adversarial malware reconstruction phase in MalGuise might affect the overlay of those few malware samples, thereby failing to be executed identically or properly. Second, it is a standard process for Windows executables to push/pop the return

address onto/off the stack when handling the “call” instruction. However, there are few malware samples that contain *junk code*, which might not follow the above process, *e.g.*, not popping the return address. These exceptions might render MalGuise ineffective as MalGuise follows the standard Windows specifications to reconstruct the adversarial malware.

Answer to RQ3 (Utility Performance): Prior adversarial attacks either only generate non-executable adversarial “features”, or generate a large portion of adversarial malware losing their original semantics. However, MalGuise exhibits the best utility performance with over 91% of generated adversarial malware files preserving their original semantics.

4.2.4 Answer to RQ4 (Real-world Performance)

As discussed in §4.1.2, to further understand the real security threats of MalGuise against anti-virus products, we empirically evaluate MalGuise against five commercial anti-virus products by measuring their ASR performance on 1,000 testing malware samples, which are randomly selected from the testing dataset. It is noted that, the main reason for randomly selecting 1,000 testing malware samples is that all five anti-virus products are deployed remotely on another machine, and their processing and scanning speeds are much slower than the employed learning-based Windows malware detection.

Table 6: The ASR (%) of MalGuise against five anti-viruses.

Attacks	McAfee	Comodo	Kaspersky	ClamAV	MS-ATP
MalGuise	48.81	36.00	11.29	31.94	70.63
MalGuise(S)	52.49	36.36	13.36	32.33	74.97
Increased ASR	+3.68	+0.36	+2.07	+0.39	+4.34

We observe in Table 6 that, for four (*i.e.*, McAfee, Comodo, ClamAV, and MS-ATP) of the five evaluated anti-viruses, MalGuise achieves ASRs of more than 30%. Especially for MS-ATP, the achieved ASR of MalGuise reaches upto 70.63%. As discussed in §4.2.2, we further limit MalGuise to use the 25 most frequently used opcodes and denote this variant as MalGuise(S) for brevity. Compared with the default MalGuise, MalGuise(S) clearly leads to an overall improvement in the attack performance for all five anti-virus products. In particular, MalGuise(S) increases its ASR against MS-ATP by 4.34%, implying it can be further improved by carefully fine-tuning.

Impact of different types of semantic nops. We further investigate how the different opcodes in the injected semantic nops affect the attack performance of MalGuise against anti-virus products. Similarly, by limiting MalGuise to use the 25 most frequently used opcodes, Fig. 9 illustrates the occurrence frequency of different opcodes of all generated adversarial malware files that lead to successful evasions for five anti-viruses. It is clearly observed that different anti-virus products are sensitive to different semantic nops injected by MalGuise.

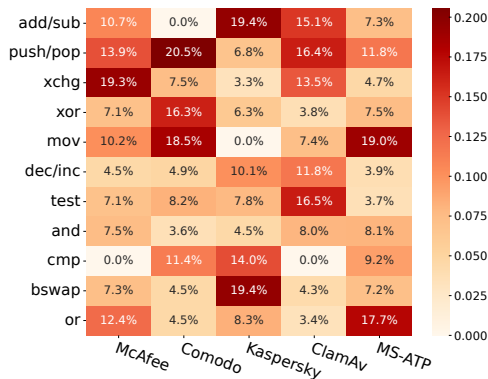


Figure 9: Occurrence frequency of different opcodes in semantic nops that lead to successful evasions for five anti-viruses.

Table 7: Distribution frequency (%) of the number of modified blocks for adversarial malware that evades anti-virus products.

# of blocks	McAfee	Comodo	Kaspersky	ClamAV	MS-ATP
1	96.66	94.28	88.17	97.54	38.21
2	4.58	4.71	9.68	2.05	42.88
3	0.76	1.01	2.15	0.41	17.35
4	0	0	0	0	1.17
5	0	0	0	0	0.39

Impact of the number of modified basic blocks. The distribution of the number of modified blocks in the adversarial malware that successfully evades the target anti-viruses is illustrated in Table 7. For all five anti-viruses, the number of modified blocks in all the adversarial malware is less than 6. Especially for McAfee, Comodo, and ClamAV, more than 90% of adversarial malware only need to modify one block, while the other two anti-virus products (*i.e.*, Kaspersky and MS-ATP) only need to modify two basic blocks. Therefore, we can conclude that MalGuise can be applied against anti-virus products by only modifying very few blocks in CFG.

Answer to RQ4 (Real-world Performance): MalGuise is empirically evaluated to be effective against five anti-virus products in the wild. In particular, MalGuise achieves attack success rates of over 30% against four of them, presenting potential tangible security concerns to real-world users.

4.2.5 Answer to RQ5 (Possible Defenses)

To understand the risks that potential artifacts in MalGuise might be noticed and identified by defenders, we evaluate MalGuise with four categories of possible defenses as follows.

(1) **Adversarial training:** Adversarial training is recognized as one of the most effective defenses against adversarial attacks [9]. Therefore, we evaluate the attack performance of MalGuise against the three target learning-based malware detection systems that have been defended using adversarial

training (*i.e.*, the defended systems.) To set up, we re-train them from scratch with nearly unchanged parameter settings, except for integrating corresponding adversarial malware generated by MalGuise during the training process.

Fig. 10 shows, after being defended by adversarial training, the detecting performance (*i.e.*, AUC, TPR, and bACC) of the three target systems remains basically unchanged, while the corresponding ASR values achieved by MalGuise have been decreased to some extent, indicating the defended systems’ robustness performance (*i.e.*, the opposite of ASR) has indeed improved. Nonetheless, it is clearly observed that, after being defended by adversarial training, MalGuise remains highly effective, achieving ASRs of 55.33%, 83.10%, and 50.68% against MalGraph, Magic, and MalConv, respectively.

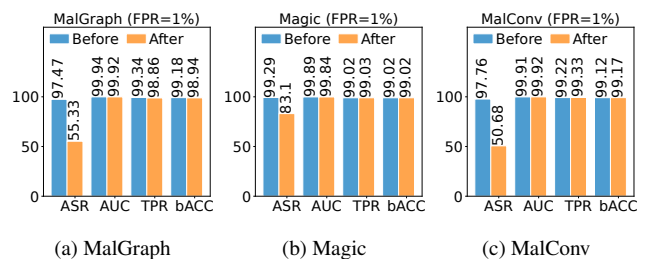


Figure 10: The overall performance comparisons between the original (before) and the defended (after) target systems.

(2) **Binary code optimizations:** Recalling that our proposed call-based redividing transformation involves injecting semantic nops and redividing “call” instructions to manipulate CFG, which might be noticed by potential defenders. Therefore, to evaluate MalGuise’s attack performance against possible defenses based on binary code optimizations (*e.g.*, dead code removal or CFG reduction), we employ the off-the-shelf IDA Pro plugin – Optimice³, which has won the Hex-Rays’ IDA Pro plugin contest [25]. To set up, we take Optimice’s binary code optimizations as the preliminary step for the three target learning-based malware detection systems and evaluate them under the black-box scenario of *w. prob.*

Table 8: The ASR performance (%) comparisons between the original target systems and the defended systems by Optimice.

Target Systems	Original System	the defended system by Optimice		
		only w/ dead code removal	only w/ CFG reduction	w/ both
MalGraph	97.47	82.25	82.43	80.26 (-17.21)
Magic	99.29	97.94	96.91	93.85 (-5.44)
MalConv	97.76	80.07	75.15	74.63 (-23.13)

Table 8 shows that, with either dead code removal or CFG reduction, or a combination of the above two, the ASR performance achieved by MalGuise has decreased, suggesting the

³ <https://code.google.com/archive/p/optimice/>

robustness performance of the defended systems has indeed increased accordingly. However, even when defended with binary code optimizations, MalGuise still achieves relatively high attack performance, with ASRs exceeding 74%.

(3) **Heuristic-based adversarial detection:** To evaluate the impact of possible defenders who are fully aware of the implementation details of MalGuise, we implement an adaptive defense of heuristic-based adversarial detection, which consists of two heuristic rules: \mathbb{R}_1 and \mathbb{R}_2 . In particular, \mathbb{R}_1 indicates, other than the “.text” section, there exists another or more sections with executable permission and fixed addresses inside. \mathbb{R}_2 indicates, there exists at least one basic block that contains a “call” instruction followed by a “jmp” instruction with semantic nops in between. It is noted that, as MalGuise iteratively generates semantic nops with the context-free grammar [41], determining whether a specific code fragment is semantic nops is well-known as non-trivial and undecidable [15, 50] and we thus employ Optimice [25] to detect the semantic nops as above. To set up, we first randomly select 15000 adversarial malware from all those that have successfully evaded three learning-based Windows malware detection systems and 15000 goodware from the benchmark dataset, and then employ the above adversarial detection to check whether the adversarial malware can be detected, and finally show the detection results in Table 9.

Table 9: The detecting performance (*i.e.*, TPR and FPR) of adversarial detection based on two heuristic rules

Adversarial Detection	TPR	FNR=(100%-TPR)	FPR
\mathbb{R}_1 -based	70.15	29.85	5.66
\mathbb{R}_2 -based	96.91	3.09	52.16
\mathbb{R}_1 & \mathbb{R}_2 -based	67.75	33.25	2.95

It is observed that both TPR and FPR of \mathbb{R}_1 -based adversarial detection are low, suggesting that most goodware indeed has only a “.text” section. However, there is a lot of adversarial malware with only a “.text” section as long as the semantic nops could be injected into the remaining slack space. Moreover, \mathbb{R}_2 -based adversarial detection shows the best TPR but the worst FPR. It indicates, although \mathbb{R}_2 can be used to detect almost all adversarial malware, there exists over 50% of goodware is misclassified as adversarial malware. Similar to \mathbb{R}_1 , the adversarial detection that combines both \mathbb{R}_1 and \mathbb{R}_2 shows a TPR of 67.7%, indicating that about 33.2% of adversarial is misclassified as goodware, *i.e.*, evading the corresponding the adversarial detection. Moreover, it also shows an FPR of 2.9%, which is higher than the upper limit of FPR (*e.g.*, 1% or 0.1%) that is tolerated by general malware detection systems.

To sum up, this adaptive defense of heuristic-based adversarial detection can detect adversarial malware generated by MalGuise to a certain extent as it is equipped with all the implementation details of MalGuise. However, it also shows the dilemma that TPR and FPR cannot be balanced simultane-

ously, *i.e.*, failing to offer satisfactory detecting performance.

(4) **Fuzzy hashing:** As MalGuise is optimized to use as few transformations as possible to preserve the same semantics, it also enables possible defenders to use fuzzy-hashing-based malware analysis (*i.e.*, ssdeep [32], TLSH [46], SpamSum [29], *etc*) for defenses, which has been extensively studied [11, 60]. To implement the fuzzy-hashing-based defenses, we first submit all malware samples in the training set to initialize *the malware database*, and then use the validation set to determine the thresholds satisfying FPR=1% and FPR=0.1% for fair comparisons. After that, we evaluate the detecting performance of fuzzy-hashing-based defenses within the testing set, which is shown in Table 10. By comparing Table 2 with Table 10, we find that all fuzzy-hashing-based defenses show less stable and worse detecting performance.

Table 10: The detecting performance (*i.e.*, AUC, TPR, bACC) and ASR performance of fuzzy-hashing-based defenses

Fuzzy-hashing-based defenses	AUC	FPR=1%			FPR=0.1%		
		TPR	bACC	ASR	TPR	bACC	ASR
ssdeep	83.92	57.61	78.70	52.94	10.16	55.31	100
TLSH	74.23	55.55	77.66	73.70	42.27	71.44	100
SpamSum	83.98	57.72	78.75	50.22	38.91	69.46	98.13

We further reinforce the three fuzzy-hashing-based defenses by submitting all malware samples in the testing set to the *the malware database*, indicating all these defenses have full knowledge of the original malware samples and can use fuzzy hashing for detection. Finally, we evaluate the ASR performance of MalGuise against those reinforced defenses, which is shown in the column named “ASR” in Table 10. It is clearly observed that, in the case of FPR=1%, the ASRs against all three reinforced defenses exceed 50%, and the ASRs are nearly 100% for FPR=0.1%, exhibiting a high attack success rate against fuzzy-hashing-based defenses.

Answer to RQ5 (Possible Defenses): MalGuise remains exceptionally effective against all three categories of possible defenses even though they are adaptively equipped with the knowledge of MalGuise.

5 Discussions

5.1 Related Work

Almost all existing studies on adversarial attacks against malware detection predominantly focus 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 tailored to these detection methods. For instance, to attack those malware detection methods based on API calls, a line of adversarial attacks

has been proposed via adding irrelevant API calls, which are selected by gradient-based optimizations or greedy algorithms [3, 13, 26, 54]. However, these adversarial attacks are impractical because they generate adversarial API calls rather than realistic executable malware files. As for these malware detection based on raw bytes, especially for MalConv [48], researchers have explored either partially modifying specific regions or globally modifying all raw bytes while preserving the same semantics. For instance, all of [5, 19, 31, 33, 51] rely on appending or injecting maliciously generated bytes at specific locations of the input malware, while MMO [41] globally manipulates its raw bytes with binary diversification techniques. However, we argue that those adversarial attacks are strictly limited to raw-bytes-based malware detection, and thus cannot be scalable to other malware detection.

More recently, studies have begun to explore adversarial attacks against more advanced malware detection based on abstract graph representations. In 2022, Zhang *et al.* [63] proposed SRL against CFG-based malware detection, which sequentially injects semantic nops into the CFG guided by reinforcement learning. Likewise, our work also aims to evade advanced malware detection based on CFGs, but fundamentally differs from SRL in three key aspects.

- **Fine-grained transformations.** SRL employs a coarse-grained transformation that only manipulates nodes of CFG. However, we propose a finer-grained transformation of `call-based redividing` that manipulates both nodes and edges, making it less noticeable to potential defenders.
- **Not adversarial “features”.** As discussed in §4.2.3, SRL is a feature-space adversarial attack that essentially generates adversarial “features”, while our MalGuise can successfully generate real adversarial malware files for evasions.
- **Attacking generalizability.** Existing adversarial attacks like SRL are primarily evaluated with limited learning-based malware detection models, while MalGuise additionally targets real-world anti-virus products in practice, thereby demonstrating better attacking generalizability.

5.2 Possible Use Cases

We discuss possible use cases of MalGuise as follows. First, MalGuise, a practical black-box adversarial attack, can complement the blue team’s toolkit, addressing the limitations of traditional obfuscations in attempts to evade present-day advanced learning-based malware detection. Second, MalGuise could serve as a means for the R&D team of anti-virus vendors to expose the underlying weaknesses of learning-based malware detection under development, and thus they can proactively and purposefully improve the robustness of anti-virus products. Third, comprehensive and impartial testing for anti-viruses is crucial for ensuring transparency and fostering trust between users and vendors. MalGuise offers a unique opportunity for third-party independent testing organizations like AV-TEST [8] to assess anti-viruses comprehensively, given

the exceptional evasion capability of adversarial malware.

5.3 Potential Ethical Concerns

The primary objective of this study is to assess the security risks of learning-based Windows malware detection with adversarial attacks, an approved topic with established precedence in earlier studies [22, 40, 41, 47, 63, 64], which is largely motivated by the concern for potential adversaries to craft Windows malware capable of evading detection, and by highlighting the necessity for more robust learning-based Windows malware detection methods. Even though our intent is strict about assessing the security risks of learning-based Windows malware detection with MalGuise, we recognize the potential ethical concerns associated with our study. Therefore, to strike a balance between avoiding potential ethical concerns and assisting the security community in enhancing the robustness of learning-based Windows detection, we limit our code sharing to *verified academic researchers only*, following the precedent established by previous studies [22, 47, 64].

6 Conclusion, Limitations and Future Work

This paper proposes a novel semantics-preserving transformation of `call-based redividing`, capable of concurrently manipulating both nodes and edges of the CFG and further presents an adversarial attack framework of MalGuise against learning-based Windows malware detection under the black-box scenario. Extensive evaluations demonstrate that MalGuise can not only effectively evade state-of-the-art learning-based Windows malware detection systems with an attack success rate of mostly exceeding 95%, but also can evade five commercial anti-virus products with an attack success rate of up to 74.97%. We believe our study raises public awareness about the security threats posed by adversarial attacks in the domain of Windows malware detection and calls for further studies to enhance the robustness of existing Windows malware detection. However, we recognize the limitations and outline potential future work as follows.

- **Verification of semantic preservation.** To verify if the generated adversarial malware preserves the original semantics, we have presented an automatic empirical verification in §4.2.3 and Eq. (7) by comparing their API sequences invoked when running on the same sandbox environment. While practical and reasonable, this empirical verification cannot guarantee the strict semantic equivalence between two malware in all possible environments since some malware may invoke random APIs in different environments.
- **Dynamic-analysis-based malware detection.** MalGuise targets learning-based Windows malware detection, which belongs to static analysis. Particularly, the core of MalGuise is to manipulate the CFG of executables via the `call-based redividing` transformation, which does not change their execution flow and thereby does not impact dynamic

analysis. Therefore, although dynamic analysis is less ubiquitously deployed due to excessive time and resource consumption, we leave generating adversarial malware against dynamic-analysis-based malware detection as future work.

- **Format-agnostic adversarial malware.** Since there is no malware analysis technique that is universally applicable to all types of malware with different file formats and operating systems, existing malware analysis normally points out the targeted file format and operating system. While in this paper, our MalGuise targets the Windows malware in the file format of portable executable, we will explore how to generate format-agnostic adversarial malware against all kinds of malware detection in future work.

Acknowledgments

We sincerely appreciate the shepherd and anonymous reviewers for their valuable comments to improve our work. We would also like to thank Bolin Zhou, Zhiqin Rui, and Yuhao Peng for their valuable contributions during the review process. This work was partly supported by the National Natural Science Foundation of China (62202457), the Major Research plan of the National Natural Science Foundation of China (92167203), the Open Source Community Software Bill of Materials (SBOM) Platform (E3GX310201), and YuanTu Large Research Infrastructure.

References

- [1] abuse.ch. MalwareBazaar. <https://bazaar.abuse.ch/sample/ab31092c90dbe2968d95d0ce959365ecd49ba4384c5f794ebcfb75bab83ab6b/>, 2024. Online (last accessed March 1, 2024).
- [2] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When malware is packin' heat; limits of machine learning classifiers based on static analysis features. In *NDSS*, 2020.
- [3] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *IEEE S&P Workshops*, 2018.
- [4] Alon Rosental. Microsoft defender atp awarded a perfect 5-star rating by sc media. <https://techcommunity.microsoft.com/t5/microsoft-defender-for-endpoint/microsoft-defender-atp-awarded-a-perfect-5-star-rating-by-sc/ba-p/1511340>, 2020. Online (last accessed April 22, 2024).
- [5] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading machine learning malware detection. In *Black Hat USA*, 2017.
- [6] Martin Apel, Christian Bockermann, and Michael Meier. Measuring similarity of malware behavior. In *IEEE Conference on Local Computer Networks*, 2009.
- [7] Atlas VPN. 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>, 2022. Online (last accessed December 25, 2022).
- [8] AV-TEST. About the av-test institute. <https://www.av-test.org/en/about-the-institute/>, 2024. Online (last accessed April 18, 2024).
- [9] Tao Bai, Jinqi Luo, Jun Zhao, Bihan Wen, and Qian Wang. Recent advances in adversarial training for adversarial robustness. In *IJCAI*, 2021.
- [10] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 2012.
- [11] Alan Cao and Brendan Dolan-Gavitt. What the fork? finding and analyzing malware in github forks. In *NDSS*, 2022.
- [12] Fabrício Ceschin, Heitor Murilo Gomes, Marcus Botacin, Albert Bifet, Bernhard Pfahringer, Luiz S Oliveira, and André Grégio. Machine learning (in) security: A stream of problems. arXiv:2010.16045, 2020.
- [13] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *IEEE European Intelligence and Security Informatics Conference*, 2017.
- [14] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. *IEEE S&P*, 2005.
- [15] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Xiaodong Song, and Randal E. Bryant. Semantics-aware malware detection. In *IEEE S&P*, 2005.
- [16] Cisco Talos Intelligence Group. ClamAV Documentation. <https://docs.clamav.net>, 2024. Online (last accessed March 1, 2024).
- [17] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, 2006.

- [18] Cuckoo Sandbox. <https://cuckoosandbox.org>, 2022. Online (last accessed Nov. 1, 2022).
- [19] Luca Demetrio, Scott E Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial EXEmPles: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *arXiv:2008.07125*, 2020.
- [20] Enigma Protector Developers Team. Enigma: A professional system for executable files licensing and protection. <https://enigmaprotector.com>, 2024. Online (last accessed March 14, 2024).
- [21] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.
- [22] Ping He, Yifan Xia, Xuhong Zhang, and Shouling Ji. Efficient query-based attack against ml-based android malware detection under zero knowledge setting. In *ACM CCS*, 2023.
- [23] Jerome Dinal Herath, Priti Prabhakar Wakodikar, Ping Yang, and Guanhua Yan. CFGExplainer: Explaining graph neural network-based malware classification from control flow graphs. In *IEEE DSN*, 2022.
- [24] Hex-Rays. IDA Pro. <https://hex-rays.com/ida-pro/>, 2022. Online (last accessed Jan. 11, 2022).
- [25] Hex-Rays. IDA Pro Plug-In Contest. https://hex-rays.com/contests_details/contest2011/, 2024. Online (last accessed March 1, 2024).
- [26] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. *arXiv:1702.05983*, 2017.
- [27] Takahiro Kasama, Katsunari Yoshioka, Daisuke Inoue, and Tsutomu Matsumoto. Malware detection method by catching their random behavior in multiple executions. In *IEEE/IPSJ International Symposium on Applications and the Internet*, 2012.
- [28] Kaspersky. AI and Machine Learning in Cybersecurity. <https://www.kaspersky.com/resource-center/definitions/ai-cybersecurity>, 2022. Online (last accessed December 25, 2022).
- [29] Russell Keith-Magee. pypamsum. <https://pypi.org/project/pypamsum/>, 2024. Online (last accessed March 2, 2024).
- [30] Jin-Young Kim and Sung-Bae Cho. Obfuscated malware detection using deep generative model based on global/local features. *Computers & Security*, 2022.
- [31] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *European signal processing conference*, 2018.
- [32] Jesse Kornblum. ssdeep-fuzzy hashing program. <https://ssdeep-project.github.io/ssdeep/index.html>, 2024. Online (last accessed March 2, 2024).
- [33] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv:1802.04528*, 2018.
- [34] Changjiang Li, Shouling Ji, Haiqin Weng, Bo Li, Jie Shi, Raheem Beyah, Shanqing Guo, Zonghui Wang, and Ting Wang. Towards certifying the asymmetric robustness for neural networks: quantification and applications. *TDSC*, 2021.
- [35] Changjiang Li, Ren Pang, Bochuan Cao, Zhaohan Xi, Jinghui Chen, Shouling Ji, and Ting Wang. On the difficulty of defending contrastive learning against backdoor attacks. *USENIX Security*, 2023.
- [36] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. Arms race in adversarial malware detection: A survey. *ACM Computing Surveys*, 2021.
- [37] Xiang Ling, Shouling Ji, Jiayu Zou, Jiannan Wang, Chunming Wu, Bo Li, and Ting Wang. DEEPSEC: A uniform platform for security analysis of deep learning model. In *IEEE S&P*, 2019.
- [38] Xiang Ling, Lingfei Wu, Wei Deng, Zhenqing Qu, Jiangyu Zhang, Sheng Zhang, Tengfei Ma, Bin Wang, Chunming Wu, and Shouling Ji. MalGraph: Hierarchical graph neural networks for robust Windows malware detection. In *IEEE INFOCOM*, 2022.
- [39] Xiang Ling, Lingfei Wu, Saizhuo Wang, Tengfei Ma, Fangli Xu, Alex X Liu, Chunming Wu, and Shouling Ji. Multilevel graph matching networks for deep graph similarity learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.
- [40] Xiang Ling, Lingfei Wu, Jiangyu Zhang, Zhenqing Qu, Wei Deng, Xiang Chen, Yaguan Qian, Chunming Wu, Shouling Ji, Tianyue Luo, Jingzheng Wu, and Yanjun Wu. Adversarial attacks against Windows PE malware detection: A survey of the state-of-the-art. *Computer & Security*, 2023.
- [41] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. Malware makeover: breaking ml-based static analysis by modifying executable bytes. In *ACM AsiaCCS*, 2021.

- [42] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. Towards adversarial malware detection: Lessons learned from pdf-based attacks. *ACM Computing Surveys*, 2019.
- [43] Alex McFarland. 10 Best Antivirus Programs of 2022 (AI Powered). <https://www.unite.ai/10-best-antivirus-programs-of-2022-ai-powered/>, 2022. Online (last accessed December 25, 2022).
- [44] Microsoft Defender for Endpoint. <https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/microsoft-defender-endpoint?view=o365-worldwide/>, 2022. Online (last accessed Nov. 1, 2022).
- [45] Microsoft, Inc. PE format. <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format>, 2022. Online (last accessed August 20, 2022).
- [46] Jonathan Oliver. TLSh - A Locality Sensitive Hash. <https://tlsh.org/index.html>, 2024. Online (last accessed March 2, 2024).
- [47] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *IEEE S&P*, 2020.
- [48] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole EXE. *arXiv: 1710.09435*, 2017.
- [49] 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. Mastering the game of Go with deep neural networks and tree search. *Nature*, 2016.
- [50] Jagsir Singh and Jaswinder Singh. Challenges of malware analysis: Obfuscation techniques. *International Journal of Information Security Science*, 2018.
- [51] Octavian Suciuc, Scott E Coull, and Jeffrey Johns. Exploring adversarial examples in malware detection. In *IEEE S&P Workshops*, 2019.
- [52] The best antivirus protection. <https://www.pcmag.com/picks/the-best-antivirus-protection>, 2022. Online (last accessed Nov. 1, 2022).
- [53] The UPX Team. Upx: The ultimate packer for executables. <https://upx.github.io>, 2024. Online (last accessed Feb. 15, 2024).
- [54] Sicco Verwer, Azqa Nadeem, Christian Hammerschmidt, Laurens Bliet, Abdullah Al-Dujaili, and Una-May O’Reilly. The robust malware detection challenge and greedy random accelerated multi-bit search. In *ACM AISec*, 2020.
- [55] VirusTotal Documentation Hub. Why don’t you have statistics comparing antivirus performance? <https://docs.virustotal.com/docs/antivirus-stats>, 2024. Online (last accessed Feb. 18, 2024).
- [56] VMProtect Software. VMProtect User Manual. <http://www.vdown.cn/vmpsoft/en/support/user-manual/>, 2024. Online (last accessed April 18, 2024).
- [57] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. Vmhunt: A verifiable approach to partially-virtualized binary code simplification. In *ACM CCS*, 2018.
- [58] Han Xu, Yao Ma, Hao-Chen Liu, Debayan Deb, Hui Liu, Ji-Liang Tang, and Anil K Jain. Adversarial attacks and defenses in images, graphs and text: A review. *IJAC*, 2020.
- [59] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *DSN*, 2019.
- [60] Wang Yang, Mingzhe Gao, Ligeng Chen, Zhengxuan Liu, and Lingyun Ying. Recmal: Rectify the malware family label via hybrid analysis. *Computers & Security*, 2023.
- [61] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys*, 2017.
- [62] Oleh Yuschuk. OllyDbg. <https://www.ollydbg.de>, 2022. Online (last accessed Nov. 1, 2022).
- [63] Lan Zhang, Peng Liu, Yoonho Choi, and Ping Chen. Semantics-preserving reinforcement learning attack against graph neural networks for malware detection. *TDSC*, 2022.
- [64] Kaifa Zhao, Hao Zhou, Yulin Zhu, Xian Zhan, Kai Zhou, Jianfeng Li, Le Yu, Wei Yuan, and Xiapu Luo. Structural attack against graph based android malware detection. In *ACM CCS*, 2021.

A More Implementation Detail in MALGUISE

Algorithm 4 outlines the main procedure of Selection. As the possible paths in the game tree of MCTS are infinite, exploring all the nodes in the game tree will substantially increase the computational overhead. By employing the Upper Confidence Bounds algorithm [10], Selection could select the child of the MCTS node v with the highest score considering the trade-off between the visit times and the reward value (line 4-6).

Algorithm 4: Procedure of $\text{Selection}(v)$.

```
1 Begin
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 return  $v_{selected}$  ; // the most promising node to be explored
```

Algorithm 5 outlines the main procedure of Simulation that returns the reward r . Based on the $v_{selected}$, this procedure iteratively expands the MCTS game tree until the simulation number S is reached (line 3–6). In each iteration, the corresponding CFG representation x of the expanded node is input to the target malware detection system f , and the returned reward is calculated as $reward = 1 - f(x)$ (line 6).

Algorithm 5: Procedure of $\text{Simulation}(v_{selected}, f, S)$

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

B More about the Benchmark Dataset

To fairly evaluate the effectiveness of learning-based malware detection systems in detecting previously unseen malware, we employ a time-based training/testing split. In particular, for all malware, we take all malware samples before May 6, 2020, as the training set and equally divide the remaining malware into the validation and testing set. For all goodwill, we randomly split them according to the percentages of the training/validation/testing sets of malware. It is reported that there is a total of 848 different malware families, in which its training/validation/testing sets have 780/366/375 different malware families, correspondingly. Fig. 11 illustrates the distribution percentages of the top 30 malware families.

C More Evaluations

The below includes four more evaluations as follows.

(1) More evaluations in terms of size alteration ratio.

In Table 11, we measure the “size alteration ratio (%)” of successfully evaded malware samples before and after the

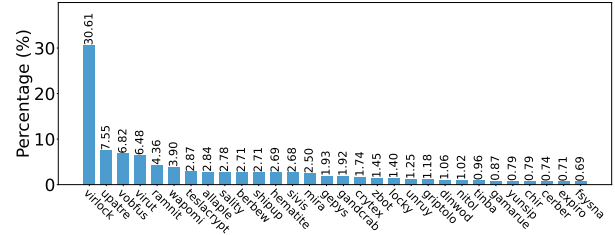


Figure 11: The distribution of top 30 malware families.

obfuscating process to observe the overhead introduced by the obfuscation tools. It is clearly observed that the malware size alteration ratio after being obfuscated by UPX is approximately $3 \times$ times that of MalGuise, while the size alteration ratios after being obfuscated by VMProtect and Enigma exceed $980 \times$ times and $330 \times$ times that of MalGuise, respectively.

Table 11: The “size alteration ratio (%)” performance comparisons between MalGuise and three baseline obfuscations.

Attacks	MalGraph		Magic		MalConv		Avg.
	FPR =1%	FPR =0.1%	FPR =1%	FPR =0.1%	FPR =1%	FPR =0.1%	
UPX	-35.8	-28.2	-30.4	-19.6	-11.1	-27.8	23.3
VMProtect	—	—	+822.7	+5343.8	—	—	7767.7
Enigma	+1737.3	+3850.5	—	+2089.7	—	+181.4	2613.1
MalGuise	+7.97	+8.03	+7.89	+7.08	+7.97	+7.72	7.87

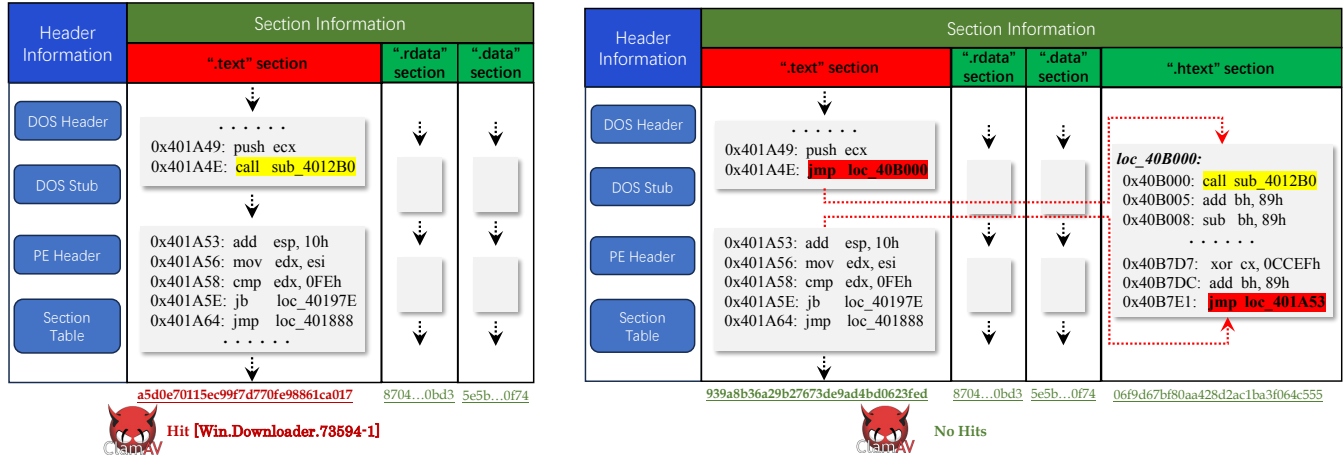
(2) **More evaluations of MalGuise variants against anti-virus products.** Table 12 summarizes the ASR performance of four MalGuise variants against five anti-virus products. In a similar vein, MalGuise and MalGuise(S) achieve the best attack performance than MalGuise[†] and MalGuise[‡] when evaluated by all employed real-world anti-virus products. This observation reconfirms that, only by concurrently applying [†] injecting semantic nops and [‡] redividing call instructions, the attack effectiveness of MalGuise can be maximized.

Table 12: The ASR performance (%) of four MalGuise variants against five anti-virus products.

Attacks	McAfee	Comodo	Kaspersky	ClamAV	MS-ATP
MalGuise [†]	28.82	30.96	4.15	27.22	—
MalGuise [‡]	44.03	33.41	6.05	29.75	—
MalGuise	48.81	36.00	11.29	31.94	70.63
MalGuise(S)	52.49	36.36	13.36	32.33	74.97

“—” indicates that we cannot evaluate MalGuise variants against MS-ATP as Microsoft has changed its authorization process since about 2024 and has not officially approved our application until now.

(3) **Impact of the size of semantic nops for MalGuise.** We also investigate the impact of the size of the injected se-



(a) The ransomware is originally detected as malware.

(b) The corresponding adversarial ransomware is mistakenly identified as goodware.

Figure 12: Illustration of a case study of how MalGuise manipulates one ransomware to evade the detection of ClamAV.

mantic nops for MalGuise against three target learning-based malware detection systems in the cases of FPR=1%. Recall from § 3.2.3 that, the size of a newly injected section in Windows executables must be a multiple of the architecture’s page size (e.g., 4KB for Intel x86). To this end, we limit the size of semantic nops generated by MalGuise to no more than $m \times 4KB$ (i.e., 1, 2, 3, 4, and 5) and present their attack performance against three target systems in Table 13. It is observed that, for both MalGraph and Magic, MalGuise requires only one time the page size ($m = 1$) to achieve a relatively high and stable ASR performance. Differently, MalGuise requires about three times the page size ($m = 3$) to achieve a high and stable ASR performance against MalConv. The main reason we conjecture is that MalConv purely relies on the raw bytes of malware, and MalGraph and Magic tend to rely on structural information like the function call relationship or control-flow information. Thus, altering the detecting output of MalConv (i.e., evading its detection), requires MalGuise to alter more raw bytes of the given malware, which is mainly accomplished by injecting semantic nops. All the above indicates that MalGuise can effectively evade learning-based malware detection with a small size of injected semantic nops.

Table 13: The ASR performance % of MalGuise against three target learning-based malware detection systems when the size of semantic nops to be injected is limited to $m \times 4KB$.

Target Systems	$m = 1$	$m = 2$	$m = 3$	$m = 4$	$m = 5$
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

(4) **Case study of how MalGuise evade ClamAV.** To unveil how MalGuise evades real-world anti-virus products, we

present a case study of how MalGuise manipulates one wild malware to evade ClamAV. We take ClamAV mainly because four of the five employed anti-virus products are proprietary and closed-source, and only ClamAV is open-sourced by Cisco [16]. Notably, for ClamAV, an executable is reported as malware if any of its signatures match those in the ClamAV Virus Database (CVD), which is continuously updated with the latest malware samples.

To set up our case study, we randomly select one newly emerged malware variant from the WannaCry ransomware attack [1], termed as WANNACRY⁴. Subsequently, as illustrated in Fig. 12(a), WANNACRY is submitted to ClamAV and deterministically detected as malware because the hash signature of the “.text” section (i.e., “a5d0e70115ec99f7d770fe98861ca017”) exactly matches the signature in CVD. Finally, as illustrated in Fig. 12(b), after applying only a single call-based redividing transformation with MalGuise, the hash signature of the “.text” section has been changed to “939a8b36a29b27673de9ad4bd0623fed”, which is no longer matched by any signature in CVD and is therefore mistakenly identified as goodware in the end.

⁴Its SHA256 hash is 043bc5f8da479077084c4ec75e5c1182254366d-135373059906bb6fed0bf5148