Optimization-Guided Binary Diversification to Mislead Neural Networks for Malware Detection

Mahmood Sharif†
NortonLifeLock Research Group
mahmood_sharif@symantec.com

Keane Lucas
Carnegie Mellon University
keanelucas@cmu.edu

Lujo Bauer
Carnegie Mellon University
lbauer@cmu.com

Michael K. Reiter
University of North Carolina at Chapel Hill
reiter@cs.unc.edu

Saurabh Shintre
NortonLifeLock Research Group
saurabh_shintre@symantec.com

arXiv:1912.09064v1 [cs.CR] 19 Dec 2019

***Abstract*—Motivated by the transformative impact of deep neural networks (DNNs) on different areas (e.g., image and speech recognition), researchers and anti-virus vendors are proposing end-to-end DNNs for malware detection from raw bytes that do not require manual feature engineering. Given the security sensitivity of the task that these DNNs aim to solve, it is important to assess their susceptibility to evasion.**

In this work, we propose an attack that guides binary-diversification tools via optimization to mislead DNNs for malware detection while preserving the functionality of binaries. Unlike previous attacks on such DNNs, ours manipulates instructions that are a functional part of the binary, which makes it particularly challenging to defend against. We evaluated our attack against three DNNs in white-box and black-box settings, and found that it can often achieve success rates near 100%. Moreover, we found that our attack can fool some commercial anti-viruses, in certain cases with a success rate of 85%. We explored several defenses, both new and old, and identified some that can successfully prevent over 80% of our evasion attempts. However, these defenses may still be susceptible to evasion by adaptive attackers, and so we advocate for augmenting malware-detection systems with methods that do not rely on machine learning.

I. Introduction

Adversarial examples, the artifacts created by test-time evasion attacks against machine-learning (ML) algorithms, have recently emerged as a threat to ML-based systems. For example, adversarial examples can enable attackers to impersonate users that are enrolled in face-recognition systems [91], [92], fool street-sign recognition algorithms into misclassifying street signs [30], and trick voice-controlled interfaces to misinterpret commands [20], [78], [88].

In particular, adversarial examples are a potential threat to malware detection—a fundamental computer-security problem that is increasingly addressed with the help of ML models (e.g., [4], [55], [80], [99]). In this domain, attackers are interested in altering programs to mislead ML-based malware detectors into misclassifying malicious programs as benign, or vice versa. In doing so, attackers face a non-trivial constraint: in addition to misleading the malware detectors, any alteration of a program must not change its original, intended, functionality. For example, a keylogger altered to evade being detected as malware should still carry out its intended function, including invoking necessary APIs, accessing sensitive files, and connecting to attackers' servers. This constraint is arguably more challenging than ones imposed by other domains (e.g., evading image recognition without making the changes conspicuous to humans [30], [91], [92]) as it is less amenable to being encoded into traditional frameworks for generating adversarial examples, and most changes to byte values are likely to break a program's syntax or semantics. In this work, we show that the constraint of preserving functionality can be incorporated into the process of generating adversarial examples to fool state-of-the-art deep neural networks (DNNs) for malware detection [55], [80].

Roughly speaking, malware-detection methods can be categorized as dynamic or static [11], [99]. Dynamic methods (e.g., [42]) execute programs to learn behavioral features that can be used for classification. In contrast, static methods (e.g., [4], [55]) classify programs using features that can be computed without execution. While potentially more accurate, dynamic methods are more computationally expensive, and, consequently, less ubiquitously deployed [11], [49]. Therefore, we focus on static methods.

Several attacks have been proposed to generate adversarial examples against DNNs for static malware detection [26], [50], [56], [97]. To fool the DNNs while preserving functionality, these attacks introduce adversarially crafted byte values in regions that do not affect execution (e.g., at the end of programs or between sections). These attacks can be defended against by masking out or removing the added content before classification (e.g., [57]); we confirm this empirically.

In this paper we show how binary-diversification tools—tools for transforming programs at the binary level to create diverse variants of the same program—that were originally proposed to defend against code-reuse attacks [54], [77] can be leveraged to evade malware-detection DNNs. While these tools preserve the functionality of programs after transformation by design, they are ineffective at evading malware detection when applied naïvely (e.g., functionality-preserving randomization). To address this, we propose optimization algorithms to guide the transformations of binaries to fool malware-detection DNNs, both in settings where attackers have access to the DNNs' parameters (i.e., white-box) and ones where they

†Work partially done as a Ph.D. student at Carnegie Mellon University.

have no access (i.e., black-box). The algorithms we propose can produce program variants that often fool DNNs in 100% of evasion attempts. Perhaps most worryingly, we find that the attack samples produced by the algorithms are also often effective at evading commercial malware detectors (in some cases with success rates as high as 85%). Because our attacks transform functional parts of programs, they are particularly difficult to defend against, especially when augmented with methods to deter static and dynamic analyses. We explore potential mitigations to the attacks that we propose (e.g., via preprocessing programs to normalize them before classification [3], [18], [103]), but conclude that attackers may adapt to circumvent these mitigations. This leads us to advocate against relying only on ML-based techniques for malware detection, as is becoming increasingly common [24].

In a nutshell, the contributions of our paper are as follows:

- We propose a novel functionality-preserving attack against DNNs for malware detection from raw bytes (Sec. III). The attack uses binary-diversification techniques in a novel way to prevent defenses applicable to prior attacks, and is applicable both in white-box and black-box settings.
- We evaluate and demonstrate the effectiveness of the proposed attack in different settings, including against commercial anti-viruses (Sec. IV). We also compare our attack with prior attacks, and show that it achieves comparable or higher success rates, while being more challenging to defend against.
- We explore the effectiveness of prior and new defenses against our proposed attack (Sec. V). While several defenses seem promising to defend against specific variants of the attack, we warn against the risk of adaptive attackers.

Next, we review some background and related work.

II. Background and Related Work

We start this section with background on DNNs for malware detection, which are the main target of our attacks. Then, we discuss research on attacking and defending ML algorithms generally, and malware detection specifically. We end the section with background on binary randomization and rewriting methods, which serve as building blocks for our attacks.

A. DNNs for Malware Detection

In this work, we study attacks targeting two DNN architectures for detecting malware from the raw bytes of Windows binaries (i.e., executables in Portable Executable format) [55], [80]. The main appeal of these DNNs is that they achieve state-of-the-art performance using automatically learned features, instead of manually crafted features that require tedious human effort (e.g., [4], [51], [44]). In fact, due to their desirable properties, computer-security companies use DNNs similar to the ones that we study (i.e., ones that operate on raw bytes and use a convolution architectures) for malware detection [23]. As these DNNs classify binaries without executing them, they fall under the category of static detection methods [11], [99].

The DNNs proposed by prior work follow standard convolutional architectures similar to the ones used for image classification [55], [80]. Yet, in contrast to image-based classifiers that classify inputs from continuous domains, the malware-detection DNNs classify inputs from discrete domains—byte values of binaries. To this end, the DNNs were designed with initial embedding layers that map each byte in the input to a vector in $\mathbb{R}^8$. Once the input is represented in a real vector space after the embedding, standard convolutional and non-linear operations are performed by subsequent layers.

B. Attacking and Defending ML Algorithms

Attacks on Image Classification Adversarial examples—inputs that are minimally perturbed to fool ML algorithms—have emerged as challenge to ML. The majority of attacks in prior work (e.g., [8], [10], [13], [33], [71], [75], [98], [105]) focused on DNNs for image-classification, and on finding adversarial perturbations that have small L_p-norm (p typically $\in \{0, 2, \infty\}$) that lead to misclassification when added to input images. By limiting perturbations to small L_p-norms, attacks aim to ensure that the perturbations are imperceptible to humans. Attacks are often formalized as optimization processes. For example, Carlini and Wagner [13] proposed the following formulation for finding adversarial perturbations that target a class c_t and have small L_2-norms:

$$\arg\min_{r} \; \mathit{Loss}_{cw}(x + r, c_t) + \kappa \cdot ||r||_2$$

where x is the original image, r is the perturbation, and κ is a parameter to tune the L_2-norm of the perturbation. Loss_{cw} is a function that, when minimized, leads $x+r$ to be (mis)classified as c_t. It is roughly defined as:

$$\mathit{Loss}_{cw}(x + r, c_t) = \max_{c \neq c_t}\{\mathbb{L}_c(x + r)\} - \mathbb{L}_{c_t}(x + r)$$

where $\mathbb{L}_c$ is the output for class c at the logits of the DNN—the output of the one-before-last layer. Our attacks use Loss_{cw} to mislead the malware-detection DNNs.

Attacks with Complex Objectives Most early attacks reduce the L_p-norms of adversarial perturbations to maintain perceptual similarity between the original images and their corresponding adversarial examples. However, small L_p-norms may be unnecessary or insufficient for maintaining perceptual similarity [29], [90]. Moreover, while reducing the L_p-norms is an objective that can easily integrated into processes for generating adversarial examples, it is unlikely by itself to result in attacks that affect systems in practice. Therefore, follow-up attacks proposed adversarial examples that satisfy objectives other than similarity as measured in L_p-norms. Similarly to the attacks that we explore in this work, which aim to preserve the functionality of transformed binaries, other attacks aim to preserve certain properties of the adversarial artifacts that are critical for their practicality. For example, in the realm of face and street-sign detection and recognition, researchers proposed ways to change the physical appearance of faces and street signs to mislead detection and recognition systems [14], [30], [91], [92]. In the speech-recognition domain, researchers showed how to modify audio signals slightly such that they would mislead speech-recognition systems when played [20], [78], [88]. Last, researchers also showed that text classification methods (e.g., for sentiment analysis) can be misled while

maintaining certain properties of the original text, such as its meaning [63], [76], [93].

Attacks on Malware Detection Multiple attacks were proposed to evade ML-based malware classifiers while preserving the (malicious) functionality of the malware. Some attacks (e.g., [25], [96], [102], [108]) tweak malware to mimic benign files (e.g., adding benign code-snippets to malicious PDF files). Other attacks (e.g., [1], [26], [35], [41], [50], [56], [97]) tweak malware using gradient-based optimizations or generative methods (e.g., to find which APIs to import). A third type of attacks uses a combination of mimicry and gradient-based optimizations [84].

Differently from some of the prior work (e.g., [1], [84], [102]) which studied attacks against dynamic ML-based malware detectors, we explore attacks that target DNNs for malware detection from raw bytes (i.e., static detection methods). Furthermore, the attacks we explore do not take advantage of weaknesses in the feature-extraction process by introducing adversarially crafted bytes to unreachable regions of the binaries [50], [56], [97] (which may be possible to detect and sanitize statically, see Sec. IV-C), or by mangling bytes in the header of binaries [26] (which can be stripped before classification [83]). Instead, the attacks we propose transform the original code of binaries in a functionality-preserving manner to achieve misclassification. Nonetheless, we still compare the evasion success rates of our attack with a representative prior attack (see Sec. IV-C), to ensure that the additional properties are not achieved at the expense of lower evasion success.

More traditionally, attackers use various obfuscation techniques to evade malware detection. Packing [11], [85], [99], [100]—encrypting binaries' code and data, and then decrypting them at run time—is commonly used to hide malicious content from static detection methods. As we explain later (Sec. III-A) we only consider unpacked binaries in this work, as is often the case for static analysis methods [11], [55]. Attackers also obfuscate binaries by substituting instructions with others or altering the control flow graphs of binaries [16], [17], [45], [99]. We demonstrate that such obfuscation methods do not fool the malware-detection DNNs when applied naïvely (see Sec. IV-B). To address this, our attacks leverage stochastic optimization techniques to guide the transformation of binaries and mislead malware detection.

Perhaps most closely related to our work is the recent work on misleading ML algorithms for authorship attribution [67], [79]. Meng et al. proposed an attack to mislead authorship attribution at the binary level [67]. Unlike the attacks we propose, Meng et al. leverage weaknesses in feature extraction and modify debug information and non-loadable sections to fool the ML models. Furthermore, their method leaves a conspicuous footprint that the binary was modified (e.g., by introducing multiple data and code sections to the binaries). While this is potentially acceptable for evading author identification, it may raise suspicion when evading malware detection. Quiring et al. recently proposed an attack to mislead authorship attribution from source code [79]. In a similar spirit to our work, their attack leverages an optimization algorithm to guide code transformations that change syntactic and lexical features of the code (e.g., switching between `printf` and `cout`) to mislead ML algorithms for authorship attribution.

Defending ML Algorithms Researchers in the area of adversarial ML are actively seeking ways to defend against adversarial examples. One line of work, called adversarial training, aims to train robust models largely by augmenting the training data with correctly labeled adversarial examples [33], [47], [46], [58], [64], [98]. Another line of work proposes algorithms to train certifiably (i.e., provably) robust defenses against certain attacks [21], [52], [61], [69], [109]. Unfortunately, these defenses are limited to specific types of perturbations (e.g., ones with small L_2- or L_∞-norms). Moreover, they often do not scale to large models that are trained on large datasets. As discussed in Sec. V, amongst other limitations, our evaluation shows that these defenses would also be too expensive to practically mitigate our attacks. Some defenses suggest that certain input transformations (e.g., quantization) can "undo" adversarial perturbations before classification [37], [62], [66], [86], [95], [106], [107]. In practice, however, it has been shown that attackers can adapt to circumvent such defenses [5], [6]. Additionally, the input transformations that have been explored in the image-classification domain cannot be applied in the context of malware detection. Prior work has also shown that adaptive attackers [12] can circumvent methods for detecting the presence of attacks (e.g., [31], [34], [66], [68]). We expect that such attackers can circumvent attempts to detect our attacks as well.

Prior work proposed ML-based malware-classification methods designed to be robust against evasion [27], [44]. However, these methods either have low accuracy [44], or target linear classifiers [27] which are unsuitable for detecting malware from raw bytes.

Fleshman et al. proposed to make malware-detection DNNs more robust by constraining the parameter weights in the last layer to non-negative values [32]. Their approach aims to prevent attackers from introducing additional features to malware to decrease its likelihood of being classified correctly. While this rationale holds for single-layer neural networks (i.e., linear classifiers), DNNs with multiple layers constitute complex functions where the addition of features at the input may correspond to the deletion of features in deep layers. As a result of the misalignment between the threat model and the defense, we found that DNNs trained with this defense are as vulnerable to prior attacks [56] as undefended DNNs. In contrast, Fleshman et al. report that their defense is effective against prior attacks. The failure of the attacks may have been caused by the obfuscated gradients phenomenon [6].

C. Binary Rewriting and Randomization

Software diversification is a technique developed to produce diverse binary versions of programs, all with the same functionality, to resist different kinds of attacks, such as memory-corruption, code-injection, and code-reuse attacks [59]. Diversification can be performed at the source-code level (via the development of multiple implementations), at compilation time (e.g., using a multicompiler), or after compilation (by rewriting and randomizing programs' binaries). In this work, we build on diversification techniques after compilation, at the binary level, as they have wider applicability (e.g., self-spreading malware can use them to evade detection without having access to the source code [72]), and are more efficient (producing the binary after a transformation does not

require recompilation). Nevertheless, we expect that this work can be extended to work with different diversification methods.

There is a large body of work on binary rewriting from the programming-languages, computer-architecture, and computer-security communities (e.g., [38], [54], [53], [65], [77], [87], [104]). Some of the rewriting methods aim to achieve higher-performing code via relatively expensive search through the space of equivalent programs [65], [87]. Other methods significantly increase the size of binaries, or may leave a conspicuous sign that rewriting took place [38], [104]. We build on binary-randomization tools that have little-to-no effect on the size or run time of the randomized binaries, thus helping our attacks remain stealthy [54], [77]. We present these tools and our extensions thereof in the following section.

III. Technical Approach

This section discusses the technical approach behind our attack. Before delving into the details, we initially lay down the threat model.

A. Threat Model

We assume that the attacker has white-box or black-box access to DNNs for malware detection that receive raw bytes of program binaries as input. In the white-box setting, the attacker has access to the DNNs' architectures and weights and is able to efficiently compute the gradients of loss functions with respect to the DNNs' input via forward and backward passes. On the other hand, the attacker in the black-box setting may only query the model with a binary and receive the probability estimate that the binary is malicious.

The weights of the DNNs are fixed and *cannot* be controlled by the attacker (e.g., by poisoning the training data). The attacker uses binary rewriting methods that are challenging to undo to manipulate the raw bytes of binaries and cause misclassification while keeping functionality intact. Attacks may seek to cause malware to be misclassified as benign or benign binaries to be misclassified as malware. The former type of attack may cause malware to circumvent defenses and be executed on a victim's machine. The latter may be useful to induce false positives, which may lead users to turn off or ignore the defenses [39].

We also assume that the binaries are unpacked, as is often the case for static malware-detection methods [11], [55]. Detecting packed binaries and unpacking them are problems orthogonal to ours that have been addressed by other researchers (e.g., [11], [15], [100]). Nonetheless, adversaries may still use our attacks simultaneously with packing: As packed binaries are usually unpacked before being classified by static methods [11], adversaries can use our attacks to modify binaries before packing them so that the binaries would be misclassified once unpacked.

As is standard for ML-based malware detection from raw bytes in particular (Sec. II-A), and for classification of inputs from discrete domains in general (e.g., [60]), we assume that the first layer of the DNN is an embedding layer. This layer maps each discrete token from the input space to a vector of real numbers via a function $\mathbb{E}(\cdot)$. When computing the DNN's output $\mathbb{F}(x)$ on an input binary x, one first computes the embeddings and feeds them to the subsequent layers. Thus, if we denote the composition of the layers following the embedding by $\mathbb{H}(\cdot)$, then $\mathbb{F}(x) = \mathbb{H}(\mathbb{E}(x))$. While the DNNs we attack contain embedding layers, our attacks conceptually apply to DNNs that do not contain such layers. Specifically, for a DNN function $\mathbb{F}(x) = \ell_{n-1}(\dots \ell_{i+1}(\ell_i(\dots \ell_0(x)\dots))\dots)$ for which the errors can be propagated back to the $(i+1)^{th}$ layer, the attack presented below can be executed by defining $\mathbb{E}(x) = \ell_i(\dots \ell_0(x)\dots)$.

B. Functionality-Preserving Attack

The attack we propose iteratively transforms a given binary x of class y (y=0 for benign binaries, and y=1 for malware) until misclassification occurs or a maximum number of iterations is reached. To keep the binary's functionality intact, the types of transformations are limited to ones that preserve functionality. (The transformation types that we consider in this work are detailed below.) In each iteration, the attack picks a transformation type at random for each function, and attempts to transform the function using it. For instance, if the transformation type can replace certain instructions within a function with functionally equivalent ones, a random subset of those instructions will be selected for replacement. The attempted transformation is applied only if the DNN becomes more likely to misclassify the binary.

Algorithm 1: White-box attack against malware detection.

Input : $\mathbb{F} = \mathbb{H}(\mathbb{E}(\cdot))$, $\mathbb{L}_\mathbb{F}$, x, y, $niters$
Output: $\hat{x}$

1 $i \leftarrow 0$;
2 $\hat{x} \leftarrow RandomizeAll(x)$;
3 **while** $\mathbb{F}(\hat{x}) = y$ *and* $i < niters$ **do**
4 $\quad \hat{e} \leftarrow \mathbb{E}(\hat{x})$;
5 $\quad g \leftarrow \frac{\partial \mathbb{L}_\mathbb{F}(\hat{x}, y)}{\partial \hat{e}}$;
6 $\quad$ **for** $f \in \hat{x}$ **do**
7 $\quad\quad o \leftarrow RandomTransformationType()$;
8 $\quad\quad \tilde{x} \leftarrow RandomizeFunction(\hat{x}, f, o)$;
9 $\quad\quad \tilde{e} \leftarrow \mathbb{E}(\tilde{x})$;
10 $\quad\quad \delta_f = \tilde{e}_f - \hat{e}_f$;
11 $\quad\quad$ **if** $g_f \cdot \delta_f > 0$ **then**
12 $\quad\quad\quad \hat{x} \leftarrow \tilde{x}$;
13 $\quad\quad$ **end**
14 $\quad$ **end**
15 $\quad i \leftarrow i + 1$;
16 **end**
17 **return** $\hat{x}$;

Alg. 1 presents the pseudocode of the attack in the white-box setting. The algorithm starts by transforming all the functions in the binary in an undirected way. Namely, for each function in the binary, a transformation type is selected at random from the set of available transformations. The transformation is then applied to that function. When there are multiple ways to apply the transformation to the function, one is chosen at random. The algorithm then proceeds to further transform the binary for up to $niters$ iterations. Each iteration starts by computing the embedding of the binary to a vector space, $\hat{e}$, and the gradient, g, of the DNN's loss function, $\mathbb{L}_\mathbb{F}$, with respect to the embedding (lines 4–5). The loss function we use is the Carlini and Wagner loss function ($Loss_{cw}$) presented in

Sec. II. Ideally, to move the binary closer to misclassification, we would manipulate the binary so that the difference of its embedding from $\hat{e} + \alpha g$ (for some scaling factor α) is minimized (see prior work for examples [50], [56]). However, if applied without proper care, such manipulation would likely change the functionality of the binary or cause it to become ill-formed. Instead, we transform the binary via functionality-preserving transformations. As the transformation types are stochastic and may have many possible outputs (in some cases, more than can be feasibly enumerated), we cannot estimate their impact on the binary a priori. Therefore, we transform each function, f, by attempting to apply a (randomly picked) functionality-preserving transformation type at random (once per iteration); we apply the transformation only if it shifts the embedding in a direction similar to g (lines 6–14). More concretely, if g_f is the gradient with respect to the embedding of the bytes corresponding to f, and δ_f is the difference between the embedding of f's bytes after the attempted transformation and its bytes before, then the transformation is applied only if the cosine similarity (or, equivalently, the dot product) between g_f and δ_f is positive. Other optimization methods (e.g., genetic programming [108]) and similarity measures (e.g., similarity in the Euclidean space) that we tested did not perform as well.

If the input were continuous, it would be possible to perform the same attack in a black-box setting after estimating the gradients by querying the model (e.g., [43]). In our case, however, it is not possible to estimate the gradients of the loss with respect to the input, as the input is discrete. Therefore, the black-box attack we propose follows a general hill-climbing approach (e.g., [96]) rather than gradient ascent. The black-box attack is conceptually similar to the white-box one, and differs only in the condition checking whether to apply attempted transformations: Whereas the white-box attack uses gradient-related information to decide whether to apply a transformation, the black-box attack queries the model after attempting to transform a function, and accepts the transformation only if the probability of the target class increases.

Transformation Types In this work, we consider two families of transformation types [54], [77], as well as their combination. For the first family, we adopt and extend the transformation types proposed in the in-place randomization (*IPR*) work of Pappas et al. [77]. Given a binary to randomize, Pappas et al. proposed to disassemble it and identify functions and basic blocks, statically perform four types of transformations that preserve the functionality of the code, and then update the binary accordingly from the modified assembly code. The four transformation types considered are: *1)* to replace instructions with equivalent ones of the same length (e.g., `sub eax,4` → `add eax,-4`); *2)* to reassign registers within functions or a set of basic blocks (e.g., swap all instances of `ebx` and `ecx`) if this does not affect code that follows; *3)* to reorder instructions, using a dependence graph to ensure that no instruction appears before another one it depends on; and *4)* to change the order in which register values are pushed to and popped from the stack to save them across function calls.

To maintain the semantics of the code, the disassembly and transformations are performed conservatively (e.g., speculative disassembly, a disassembly technique that has a relatively high likelihood of misidentifying code, is avoided). *IPR* does not alter binaries' sizes and has no measurable effect on their run time [77].

The original implementation of Pappas et al. could not be used to evade malware detection due to several limitations, including ones preventing it from producing the majority of functionally equivalent binary variants that are conceptually achievable under the four transformation types. Thus, we extend and improve the implementation in various ways. First, we enable the transformations to compose. In other words, unlike Pappas et al.'s implementation, our implementation allows us to iteratively apply different transformation types to the same function. Second, we apply transformations more conservatively to ensure that the functionality of the binaries is preserved (e.g., by not replacing `add` and `sub` instructions if they are followed by instructions that read the flags register). Third, compared to the previous implementation, our implementation can handle a larger number of instructions and additional function-calling conventions. In particular, our implementation can rewrite binaries containing additional instructions (e.g., `shrd`, `shld`, `ccmove`) as well as less common calling conventions (e.g., nonstandard returns via increment of `esp` followed by a `jmp` instruction) without impacting the binaries' functionality. Last, we fix bugs in the original implementation (e.g., incorrect checks for writes to memory after reads). Fig. 1 shows an example of transforming code using *IPR*.

The second family of transformation types that we build on is based on code displacement (*Disp*), proposed by Koo and Polychronakis [54]. Similarly to *IPR*, *Disp* begins by conservatively disassembling the binary. The original idea of *Disp* is to move code that can be leveraged as a gadget in code-reuse attacks to a new executable section in order to break the gadget. The original code to be displaced has to be at least five bytes in size so that it can be replaced with a `jmp` instruction that passes the control to the displaced code. If the displaced code contains more than five bytes, the bytes after the `jmp` are replaced with trap instructions that terminate the program; these would be executed if a code-reuse attack is attempted. In addition, another `jmp` instruction is appended immediately after the displaced code to pass the control back to the instruction that should follow. Of course, any displaced instruction that uses an address relative to the instruction-pointer (i.e., `IP`) register is also updated to reflect the new address after displacement. *Disp* has a minor effect on binaries' sizes ($\sim$2% increase on average) and causes a small amount of run-time overhead ($<$1% on average) [54].

We extend *Disp* in two primary ways. First, we make it possible to displace any set of consecutive instructions within the same basic block, not only ones that belong to gadgets. Second, instead of replacing the original instructions with traps, we replace them with *semantic nops*—sets of instructions that *cumulatively* do not affect the memory or register values and have no side effects [17]. These semantic nops get jumped to immediately after the displaced code is done executing. To create the semantic nops, we use the context-free grammar described in Fig. 2. At a high-level, a semantic nop can be an atomic instruction (e.g., `nop`), or recursively defined as an invertible instruction that is followed by a semantic nop and then by the inverse instruction (e.g., `push eax` followed by a semantic nop and then by `pop`

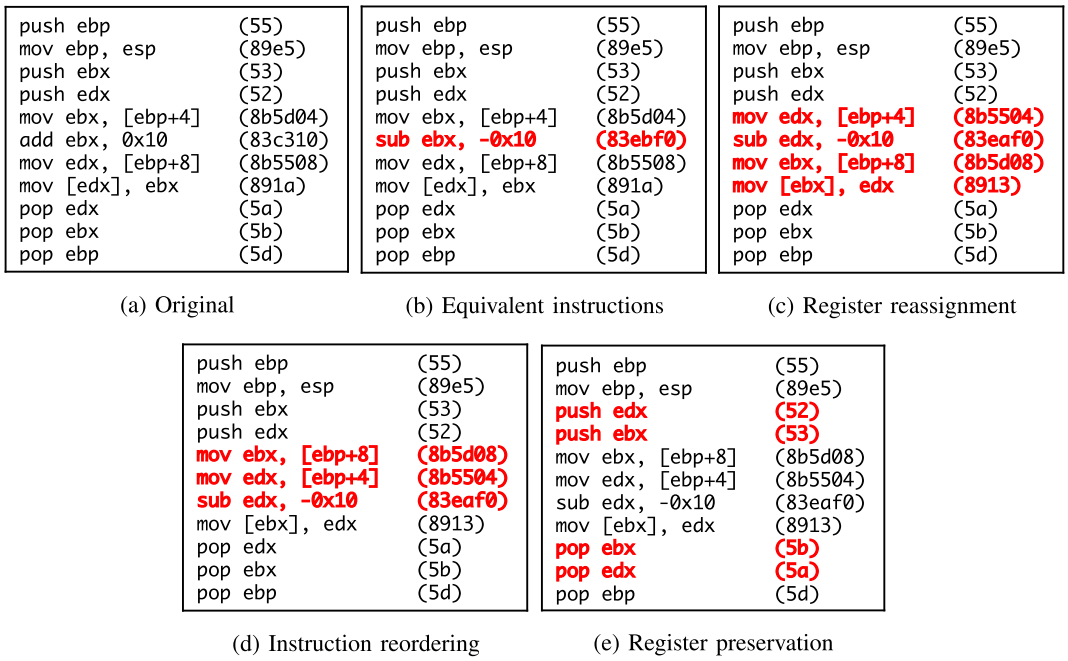


Fig. 1: An illustration of *IPR*. We show how the original code (a) changes after replacing instructions with equivalent ones (b), reassigning registers (c), reordering instructions (d), and changing the order of instructions that save register values (e). We provide the hex encoding of each instruction to its right. The affected instructions are boldfaced and colored in red.

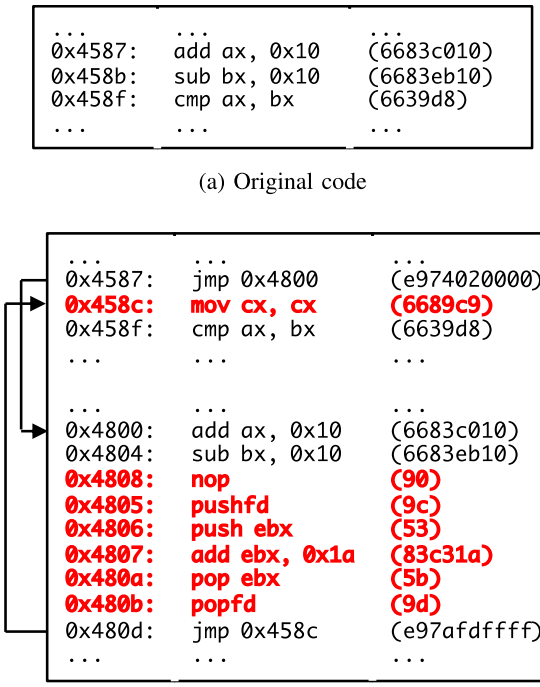
$$
\begin{aligned}
1\quad & S \rightarrow \mathit{Atom} \mid S \cdot S \mid \\
2\quad & \qquad \texttt{bswp r} \cdot S \cdot \texttt{bswp r} \mid \\
3\quad & \qquad \texttt{xchg rh, rl} \cdot S \cdot \texttt{xchg rh, rl} \mid \\
4\quad & \qquad \texttt{push r} \cdot S_r \cdot \texttt{pop r} \mid \\
5\quad & \qquad \texttt{pushfd} \cdot S_{ef} \cdot \texttt{popfd} \\
6\quad & \mathit{Atom} \rightarrow \Phi \mid \texttt{nop} \mid \texttt{mov r, r} \\
7\quad & S_r \rightarrow S \mid S_r \cdot S_r \mid \texttt{pushfd} \cdot S_{ef,r} \cdot \texttt{popfd} \\
8\quad & S_{ef} \rightarrow S \mid S_{ef} \cdot S_{ef} \mid \\
9\quad & \qquad \texttt{arth r, v} \cdot S_{ef} \cdot \texttt{invarth r, v} \mid \\
10\quad & \qquad \texttt{push r} \cdot S_{ef,r} \cdot \texttt{pop r} \\
11\quad & S_{ef,r} \rightarrow S \mid S_r \mid S_{ef} \mid S_{ef,r} \cdot S_{ef,r} \mid \\
12\quad & \qquad \texttt{arth r, v} \cdot S_{ef,r} \mid \\
13\quad & \qquad \texttt{logic r, v} \cdot S_{ef,r}
\end{aligned}
$$

Fig. 2: A context-free grammar for generating semantic nops. S is the starting symbol, Φ is the empty string, the symbol `arth` indicates an arithmetic operation (specifically, `add`, `sub`, `adc`, or `sbb`), `invarth` indicates its inverse, `logic` indicates a logical operation (specifically, `and`, `or`, or `xor`), and `r` and `v` indicate a register and a randomly chosen integer, respectively.

`eax`), or as two consecutive semantic nops. When the flags register's value is saved (i.e., between `pushfd` and `popfd` instructions), a semantic nop may contain instructions that affect flags (e.g., add and then subtract a value from a register), and when a register's value is saved too (i.e., between `push r` and `pop r`), a semantic nop may contain instructions that affect the register (e.g., decrement it by a random value). Using the grammar for generating semantic nops, for example, one may generate a semantic nop that stores the flags and `ebx` registers on the stack (`pushfd`; `push ebx`), performs an operation that might affect both registers (e.g., `add ebx, 0xff`), and then restores the registers (`pop ebx`; `popfd`).

When using *Disp*, our attacks start by displacing code up to a certain budget, to ensure that the resulting binary's size does not increase above a threshold (e.g., 1% above the original size). We first divide the budget (expressed as the number of bytes to be displaced) by the number of functions in the binary, and we attempt to displace exactly that number of bytes per function. If multiple options exist for what code in a function to displace, the code to be displaced is chosen at random. If a function does not contain enough code to displace, then we attach semantic nops (occupying the necessary number of bytes) after the displaced code to meet the per-function budget. In the rare case that the function does not have any basic block larger than five bytes, we skip that function. Fig. 3 illustrates an example of displacement where semantic nops are inserted to replace original code, as well as after the displaced code, to consume the budget. Then, in each iteration of modifying the binary to cause it to be misclassified, new semantic nops are chosen at random and used to replace the previously inserted semantic nops if that moves the binary closer to misclassification.

Some of the semantic nops contain integer values that can be set arbitrarily (e.g., see line 12 of Fig. 2). In a white-box setting, the bytes of the binary that correspond to these values can be set to perturb the embedding in the direction that is most similar to the gradient. Namely, if an integer value in the semantic nop corresponds to the i^{th} byte in the binary, we set this i^{th} byte to $b \in \{0, \ldots, 255\}$ such that the cosine similarity between $\mathbb{E}(b) - \mathbb{E}(\hat{x}_i)$ and g_i is maximized. This process is



(a) Original code

(b) After *Disp*

Fig. 3: An example of displacement. The two instructions staring at address `0x4587` in the original code (a) are displaced to to starting address `0x4800`. The original instructions are replaced with a `jmp` instruction and a semantic nop. To consume the displacement budget, semantic nops are added immediately after the displaced instructions and just before the `jmp` the passes the control back to the original code. Semantic nops are shown in boldface and red.

repeated each time a semantic nop is drawn to replace previous semantic nops in white-box attacks.

Prior work has suggested methods for detecting and removing semantic nops from binaries [18]. Such methods might appear viable for defending against *Disp*-based attacks, though as we discuss in Sec. V, attackers can leverage various techniques to evade semantic-nop detection and removal.

Limitations While our implementation extends prior implementations, it can still be further improved. For instance, our implementation does not displace code that has been displaced in early iterations. A more comprehensive implementation might apply displacements recursively. Furthermore, the composability of *IPR* and *Disp* transformations can be enhanced. Particularly, when applying both the *Disp* and *IPR* transformations to a binary, both types of transformations affect the original instructions of the binary. However, *IPR* does not affect the semantic nops that are introduced by *Disp*. Although there remains room for improvement, we did not pursue the remaining engineering endeavors because the attacks were successful despite the shortcomings of the implementation.

Group	Train	Val.	Test
Benign	10,349	3,848	5,337
Malicious	10,868	5,257	5,616

TABLE I: The number of benign and malicious binaries used to train, validate, and test the DNNs.

IV. Evaluation

In this section, we provide a comprehensive evaluation of our attack. We begin by providing details about the DNNs and data used for evaluation. We then show that naïve, random, transformations that are not guided via optimization do not lead to misclassification. Subsequently, we provide an evaluation of variants of our attack under a white-box setting, and compare with prior work. Then, we move to discuss evaluations of our attack in the black-box setting, both against the DNNs and commercial anti-viruses. We close the section with experiments to validate that the attacks preserve functionality.

A. Datasets and Malware-Detection DNNs

To train malware-detection DNNs, we used malicious binaries from a publicly available dataset that we augmented with benign binaries from standard software packages, as is standard (e.g., [51], [56]). In particular, we used malware binaries from nine malware families[1] that were published as part of a malware-classification competition organized by Microsoft [83]. This dataset contains raw binaries of malware samples targeting Windows machines. As such, the binaries adhere to the Portable Executable format (*PE*; the standard format for `.dll` and `.exe` files) [48]. However, to maintain sterility and prevent the binaries from executing, the curators removed their *PE* headers (which, among others, contain the entry points of the code). In total, the dataset contains 21,741 binaries that were partitioned into training and test sets by the dataset curators. We further partitioned the test set randomly into one group for validation (i.e., model and hyperparameter tuning) and another for final testing. Table I lists the number of binaries in the training, validation, and test sets.

Prior work [2], [55], [80] used larger malware datasets for training (in some cases containing two orders of magnitude more samples than the Microsoft dataset). Unfortunately, however, the raw binaries from prior work's datasets are proprietary. Consequently, we resorted to using a publicly available dataset. Nonetheless, the DNNs that we trained achieve comparable performance to those of prior work.

To collect benign binaries, we installed standard packages on a newly created 32-bit Windows 7 virtual machine and gathered the *PE* binaries pertaining to these packages. Specifically, we used the Ninite and Chocolatey[2] package managers to install 179 packages. The packages that we installed included popular ones that are commonly used by a variety of users (such as Chrome, Firefox, WinRAR, Spotify, ...), as well as packages that are likely to be used by specific user groups, such as developers (e.g., PyCharm), academics (e.g., MiKTeX),

[1] Gatak, Kelihos v1, Kelihos v3, Lollipop, Obfuscator ACY, Ramnit, Simda, Tracur, and Vundo.

[2] https://ninite.com/ and https://chocolatey.org/

DNN	Accuracy Train	Val.	Test	TPR @ 0.1% FPR
AvastNet	99.23%	98.29%	98.92%	80.28%
MalConv	99.96%	98.33%	99.15%	88.73%

TABLE II: The DNNs' performance. We report the accuracies on the different data partitions, as well as the TPR at the operating point where the FPR equals 0.1%.

Group	*AvastNet*	*MalConv*	*Endgame*
Benign	99	99	99
Malicious	72	95	86

TABLE III: The number of benign and malicious binaries used to test our attacks against the three DNNs.

and graphics designers (e.g., Gimp). This resulted in 19,534 binaries that we partitioned into training, validation, and test sets of comparable sizes to those for malware (see Table I). When partitioning, we placed binaries from the same packages in the same partitions to ensure that the DNNs learned to tell apart malicious and benign binaries rather to than to associate binaries of the same packages with each other.

Using the malicious and benign samples, we trained two malware-detection DNNs. Both DNNs receive binaries' raw bytes as inputs and output the probability that the binaries are malicious. The first DNN, proposed by Krčál et al. [55], receives inputs up to 512 KB in size. We refer to it by *AvastNet*, in reference to the authors' affiliation. The second DNN, proposed by Raff et al. [80], receives inputs up to 2 MB in size. We refer to this DNN by *MalConv*, as per the authors' naming. Except for the batch-size parameter, we used the same training parameters reported in the papers. We set the batch size to 32 due to memory limitations. In addition, when using benign binaries for training, we excluded the headers. This is both to remain consistent with the malicious binaries (which do not include headers), but also to ensure that the DNNs would not rely on header values that are easily manipulable for classification [26]. As the results below demonstrate, excluding the header leads to DNNs that are more difficult to evade.

The classification performance of the DNNs is reported in Table II. Both DNNs achieve test accuracy of about 99%. Even when restricting the false positive rates (FPRs) conservatively to 0.1% (as is often done by anti-virus vendors [55]), the true positive rates (TPRs) remain as high as 80–89% (i.e., 80–89% of malicious binaries are detected). The performance results that we computed are superior to the ones reported in the original papers both for classification from raw bytes and from manually crafted features [55], [80]. We believe the reason to be that our dataset was restricted to nine malware families, and expect the performance to slightly decrease when incorporating additional malware families.

In addition to the two DNNs that we trained, we evaluated the attacks using a publicly available DNN that was trained by Anderson and Roth [2]. We refer to this DNN by *Endgame*, in reference to the authors' affiliation. *Endgame* has a similar architecture to *MalConv*. The salient differences are that: *1) Endgame*'s input dimensionality is 1 MB (compared to 2 MB for *MalConv*); and *2) Endgame* uses the *PE* header for classification. On a dataset separate from ours that was curated by a computer-security company, *Endgame* achieved about 92% TPR when the FPR was restricted to 0.1% [2].

To evaluate attacks against the DNNs, we selected binaries according to three criteria. First, the binaries had to be unpacked. To this end, we used standard packer detectors (specifically, Packerid [89] and Yara [101]) and deemed binaries as unpacked only if none of the detectors exhibited a positive detection. This method is similar to the one followed by Biondi et al. [11].[3] While the data used to train and evaluate the performance of the DNNs included packed binaries (we could not exclude potentially packed binaries from the Microsoft dataset due to missing headers), the high accuracy of the DNNs on the test samples suggests that the DNNs' performance was not impacted by (lack of) packing. Second, the binaries had to be classified correctly and with high confidence by the DNNs that we trained. In particular, malicious (resp., benign) binaries had to be classified as malicious (resp., benign), and the estimated probability that they are malicious had to be above (resp., below) the threshold where the FPR (resp., false negative rate, FNR) is 0.1%. Consequently, our evaluation of the attacks' success is conservative: the attacks would be more successful for binaries that are initially classified correctly, but not with high confidence. Third, the binaries' sizes had to be smaller than the DNNs' input dimensionality. While the DNNs can classify binaries whose size is larger than the input dimensionality (as can be seen from the high classification accuracy on the validation and test sets), we avoided large binaries as a means to prevent evasion by displacing malicious code outside the input range of the DNNs.

Using these criteria, we selected 99 benign binaries from the test set to evaluate the attacks against each of the three DNNs. Leading malware detection to misclassify these benign samples can harm users' trust in the defense [39]. Unfortunately, we were unable to use the malicious binaries from the Microsoft dataset to evaluate our attacks, as they lack the *PE* headers, and so they cannot be disassembled as necessary for the *IPR* and *Disp* transformations. To this end, we used VirusShare [82]—an online repository of malware samples—to collect malicious binaries belonging to the nine families that are present in the Microsoft dataset (as indicated by the labels that commercial anti-viruses assigned the samples). Following this approach, we collected a variable number of binaries that were unseen in training to test the attacks against each one of the DNNs, as specified in Table III. The total number of samples the we collected to evaluate the attacks is comparable to that used in prior work on evading malware detection [50], [56], [96], [97].

B. Randomly Applied Transformations

We first evaluated whether naïvely transforming binaries at random would lead to evading the DNNs. To do so, for each binary that we used to evaluate the attacks we created 200 variants using the *IPR* and *Disp* transformations and classified them using the DNNs. If any of the variants was

[3] Biondi et al. used three packer-detection tools instead of two. Unfortunately, we were unable to get access to one of the proprietary tools.

misclassified by a DNN, we would consider the evasion attempt successful. We set *Disp* to increase binaries' sizes by 5% (i.e., the displacement budget was set to 5% of the binary's original size). We selected 200 and 5% as parameters for this experiment because our attacks were executed for 200 iterations at most, and achieved almost perfect success when increasing binaries' sizes by 5% (see below).

Except for a single benign binary that was misclassified by *MalConv* after being transformed, no other misclassification occurred. Hence, we can conclude that the DNNs are robust against naïve transformations, and that more principled approaches are needed to mislead them.

C. White-Box Attacks vs. DNNs

In the white-box setting, we evaluated seven variants of our attack. One variant, to which refer by *IPR*, relies on the *IPR* transformations. Three variants, *Disp*-1, *Disp*-3, and *Disp*-5, rely on the *Disp* transformations, where the numbers indicate the displacement budget as a percentage of the binaries' sizes (e.g., *Disp*-1 increases binaries' sizes by 1%). The last three attack variants, *IPR*+*Disp*-1, *IPR*+*Disp*-3, and *IPR*+*Disp*-5, use the *IPR* and *Disp* transformations combined. We executed the attacks up to 200 iterations and stopped early if the binaries were misclassified with high confidence. For malicious (resp., benign) binaries, this meant that they were misclassified as benign (resp., malicious) with an estimated probability that they are malicious below the probability where the FPR (resp., FNR) is 0.1%. We set 5% as the maximum displacement budget and 200 as the maximum number of iterations, as we empirically found that the attacks were almost always successful with these parameters.

In addition to our attacks, we implemented and evaluated an attack proposed by Kreuk et al. [56]. To mislead DNNs, the attack of Kreuk et al. appends adversarially crafted bytes to binaries. These bytes are crafted via an iterative algorithm that first computes the gradient g_i of the loss with respect to the embedding $\mathbb{E}(x_i)$ of the binary x_i at the i^{th} iteration, and then sets the adversarial bytes to minimize the L_2 distance of the new embedding $\mathbb{E}(x_{i+1})$ from $\mathbb{E}(x_i)+\epsilon sign(g_i)$, where ϵ is a scaling parameter. We tested three variants of the attack, denoted by *Kreuk*-1, *Kreuk*-3, and *Kreuk*-5, which increase the binaries' sizes by 1%, 3%, and 5%, respectively. As a loss function, we used $Loss_{cw}$. Similarly to our attacks, we executed Kreuk et al.'s attacks up to 200 iterations, stopping sooner if misclassification with high confidence occurred. Furthermore, we set ϵ=1, as we empirically found that it leads to high evasion success.

We measured the success rates of attacks by the percentage of binaries that were misclassified. The results of the experiment are provided in Fig. 4. One can immediately see that attacks using the *Disp* transformations were more successful than *IPR*. In fact, *IPR* was able to mislead only *Endgame*, achieving 62% success rate at misclassifying malicious binaries as benign, and 74% success rate at misclassifying benign binaries as malicious. *IPR* was unable to mislead *AvastNet* and *MalConv* in any attempt. This indicates that using binaries' headers for classification, similarly to *Endgame*, may lead to more vulnerable models than when excluding the header.

In contrast to *IPR*, other variants of our attack achieved considerable success. For example, *Disp*-5 achieved high-confidence misclassification in all attempts, except when attempting to mislead *AvastNet* to misclassify benign binaries, where 81% of the attempts succeeded. As one would expect, attacks with higher displacement budget were more successful. Specifically, attacks with 5% displacement budget were more successful than ones with 3%, and the latter were more successful than attacks with 1% displacement budget.

In addition to achieving higher success rates, another advantage of *Disp*-based attacks over *IPR*-based ones is their time efficiency. While displacing instruction at random from within a function with n instructions has $\mathcal{O}(n)$ time complexity, certain *IPR* transformations have $\mathcal{O}(n^2)$ time complexity. For example, reordering instructions requires building a dependence graph and extracting instructions one after the other. If every instruction in a function depends on previous ones, this process takes $\mathcal{O}(n^2)$ time. In practice, we found that while *Disp*-based attacks took about 159 seconds to run on average, *IPR*-based ones took 606 seconds.

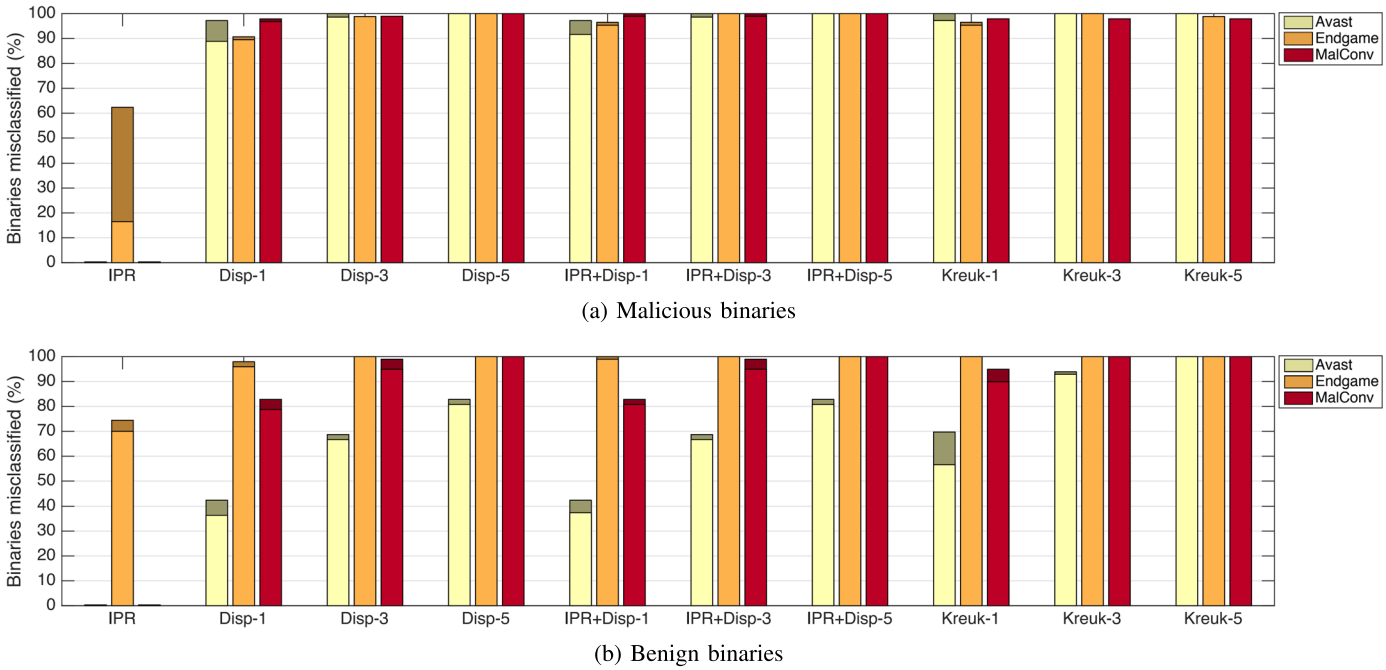
While *IPR* had limited success, combining *IPR* with *Disp* achieved higher success rates than respective *Disp*-only attacks with the same displacement budget. For example, *IPR*+*Disp*-1 had 6% higher success rate than *Disp*-1 when misleading *Endgame* to misclassify a malicious binary as benign (97% vs. 91% success rate). Thus, in certain situations, *Disp* and *IPR* can be combined to fool the DNNs while increasing binaries' sizes less than *Disp* alone.

The variants of Kreuk et al.'s attack achieved success rates comparable to variants of our attack. For example, *Kreuk*-5 was almost always able to mislead the DNNs—it achieved 99% and 98% success rate when attempting to mislead *Endgame* and *MalConv*, respectively, to misclassify malicious binaries, and 100% success rate in all the other attempts. One can also see that the success rate increased as the attacks increased the binaries' sizes. In particular, *Kreuk*-5 was more successful at misleading the DNNs than *Kreuk*-3, which, in turn, was more successful than *Kreuk*-1.

While Kreuk et al.'s attack achieved success rates that are comparable to ours, it is important to highlight that their attack is easier to defend against. As a proof of concept, we implemented a sanitization method to defend against the attack. The method finds all the sections that do not contain instructions (using the IDAPro disassembler [40]) and masks the sections' content with zeros. As Kreuk et al.'s attack does not introduce code to the binaries, the defense masks the adversarial bytes that it introduces. Consequently, the evasion success rates of the attack drop significantly. For example, the success rates of *Kreuk*-5 against *MalConv* drop to 1% and 11% for malicious and benign binaries, respectively. At the same time, the defense has little-to-no effect on our attacks. For example, *Disp*-5 achieves 91% and 100% success rates for malicious and benign binaries, respectively. Moreover, the classification accuracy remains high for malicious (99%) and benign (93%) binaries after the defense.

D. Black-Box Attacks vs. DNNs

As explained in Sec. III, because the DNNs' input is discrete, estimating gradient information to mislead them in



(a) Malicious binaries

(b) Benign binaries

Fig. 4: Attacks' success rates in the white-box setting. For each attack and DNN, we provide the percentage of misclassified malicious (a) and benign (b) binaries. The brightly colored bars show the percentage of binaries that were misclassified with high confidence.

a black-box setting is not possible. To this end, the black-box version of Alg. 1 uses a hill-climbing approach to query the DNN after each attempted transformation to decide whether to keep the transformation. Because querying the DNNs after each attempted transformation leads to a significant increase in run time of the attacks ($\sim$30$\times$ on a machine with GeForce GTX 980 GPU), we limited our experiments to *Disp* transformations with a displacement budget of 5%, and attempted to mislead the DNNs to misclassify malicious binaries. We executed the attacks up to 200 iterations and stopped early if misclassification occurred.

The attacks were most successful against *Endgame*, achieving a success rate of 97%. In contrast, 33% of the evasion attempts against *MalConv* succeeded, and none of attempts against *AvastNet* were successful. In cases of failure, we observed that the optimization process was getting stuck in local minimas. We believe that it may be possible to enhance the performance of the attacks in such cases by deploying methods for overcoming local minimas (e.g., the Metropolis algorithm [73], or Monte-Carlo tree search [94]).

Motivated by the universality property of adversarial examples [70] and the finding of Kreuk et al. that adversarial bytes generated for one binary may lead to evasion when appended to another binary [56], we wanted to see if the transformations applied to one binary can lead to evasion when applied to another. If so, attackers in a black-box setting may invest considerable effort to transform one binary to mislead malware-detection and then apply the same transformations to other binaries. We focused on *Disp* transformations and tested whether the semantic nops that were added to certain binaries via the *Disp*-5 attack in the white-box setting lead to evasion when added to other binaries. To do so, we developed a modified version of *Disp* that displaces code within binaries at random, but instead of drawing the semantic nops randomly, it borrows them from another previously transformed binary.

We tested this approach with malicious binaries and found that it leads to relatively high success rates. For *AvastNet* and *Endgame*, certain transformations led to evasion success rates as high as 75% and 86%, respectively, when borrowed from one binary and applied to other binaries (i.e., merely 14%–25% lower success rates than for white-box attacks). The success rates were more limited against *MalConv*, achieving a maximum of 24%.

E. Transferability of Attacks to Commercial Anti-Viruses

To assess whether our attacks affect commercial anti-viruses, we tested how the binaries that were misclassified by the DNNs with high confidence in the white-box setting get classified by anti-viruses available via VirusTotal [19]—an online service that aggregates the results of 68 commercial anti-viruses. Since anti-viruses often rely on ML for malware detection, and since prior work has shown that adversarial examples that evade one ML model often evade other models (a phenomenon called transferability) [33], [74], we expected that the malicious (resp., benign) binaries generated by our attacks would be classified as malicious by fewer (resp., more) anti-viruses than the original binaries.

As a baseline, we first classified the original binaries using the VirusTotal anti-viruses. As one would expect, all the malicious binaries were detected by several anti-viruses. The median number of anti-viruses that detected any particular

malware binary as malicious was 55, out of 68 total anti-viruses. In contrast, the original benign binaries were detected by a median of 0 anti-viruses, with a total of five false positives across all binaries and anti-viruses. To further gauge the accuracy of the commercial anti-viruses, we used them to classify binaries that were transformed at random using the *Disp* and *IPR* transformation types (in the same manner as Sec. IV-B). We found that certain anti-viruses were susceptible to such simple evasion attempts—the median number of anti-viruses that detected the malicious binaries correctly decreased to 42. At the same time, the median number of anti-viruses that detected benign binaries as malicious remained 0. Presumably, some anti-viruses were evaded by random transformations due to using fragile detection mechanisms, such as signatures.

Table IV summarizes the effect of our attacks on the number of positive detections (i.e., classification of binaries as malicious) by the anti-viruses. Compared to the original malicious binaries and ones that were transformed at random, the malicious binaries transformed by our attacks were detected as malicious by fewer anti-viruses. The median number of anti-viruses that correctly detected the malicious binaries decreased from 55 for the original binaries and 42 for ones transformed at random to 33–36, depending on the attack variant and the targeted DNN. According to a Kruskal-Wallis test, this reduction is statistically significant (p <0.01 after Bonferroni correction). In other words, the malicious binaries that were transformed by our attacks were detected by only 49%–53% of the VirusTotal anti-viruses in the median case.

The number of positive detections of benign binaries increased after they were transformed by our attacks: The median number of anti-viruses that detected the benign binaries as malicious was one or two, depending on the attack variant and the targeted DNN. In certain cases, the number of positive detections was as high as 19 (i.e., 28% of the VirusTotal anti-viruses reported the binary as malicious). Except for one attack (*IPR* targeting *Endgame*), the increase in the number of detections is statistically significant (p <0.01 after Bonferroni correction), according to a Kruskal-Wallis test.

Our attacks evaded a larger number of anti-viruses compared to random transformations, likely due to transferring from our DNNs to ML detectors that are used by the anti-viruses. A glance at the websites of the anti-viruses' vendors showed that 15 of the 68 vendors explicitly advertise relying on ML for malware detection. These anti-viruses were especially susceptible to evasion by our attacks. Of particular note, one vendor advertises that it relies solely on ML for malware detection. This vendor's anti-virus misclassified 78% of the benign binaries that were produced by one variant of our attack as malicious. In general, a median of 1–2 anti-viruses (from the 15 vendors) misclassified benign binaries that were processed by our attacks as malicious. Even more concerning, a popular anti-virus whose vendor reports to rely on ML misclassified 85% of the malicious binaries produced by a variant of our attack as benign. Generally, malicious binaries that were produced by our attacks were detected by a median number of 7–9 anti-viruses of the 15—down from 12 positive detections for the original binaries. All in all, while online advertising (or lack thereof) is a weak indicator of the nature of detectors used by anti-viruses (e.g., some prominent vendors do not explicitly advertise the use of ML), our results support that binaries that were produced by our attacks were able to evade ML-based detectors that are used by anti-virus vendors.

F. Correctness

A key feature of our attacks is that they transform binaries to mislead DNNs while preserving their functionality. We followed standard practices from the binary-diversification literature [53], [54], [77] to ensure that the functionality of the binaries was kept intact after being processed by our attacks. First, we transformed ten different benign binaries (e.g., `python.exe` of Python version 2.7, and Cygwin's[4] `less.exe` and `grep.exe`) with our attacks and manually validated that they functioned properly after being transformed. For example, we were still able to search files with `grep` after the transformations. Second, we transformed the `.exe` and `.dll` files of a stress-testing tool[5] with our attacks and checked that the tool's tests passed after the transformations. Using stress-testing tools to evaluate the correctness of binary-transformation methods is common, as such tools are expected to cover most branches affected by the transformations. Third, and last, we also transformed ten malware binaries and used the Cuckoo Sandbox [36]—a popular sandbox for malware analysis—to check that their behavior remained the same. All ten binaries attempted to access the same hosts, IP addresses, files, APIs, and registry keys before and after being transformed.

V. Discussion

Our proposed attacks achieved high success rates at fooling DNNs for malware detection in white-box and black-box settings. The attacks were also able to mislead commercial anti-viruses, especially ones that leverage ML algorithms. To protect users and their systems, it is important to develop mitigation measures to make malware detection robust against evasion by our attacks. Moreover, it is important to consider ways to extend our attacks to other settings to understand the weaknesses of other systems and help improve their security. Next, we discuss potential mitigations and extensions to our attacks.

A. Potential Mitigations

Prior Defenses We considered several prior defenses to mitigate our attacks, but, unfortunately, most showed little promise. For instance, adversarial training (e.g., [33], [58]) is infeasible, as the attacks are computationally expensive. Depending on the attack variant, it took an average of 159 or 606 seconds to run an attack. As a result, running just a single epoch of adversarial training would to take several weeks (using our hardware configuration), as each iteration of training requires running an attack for every sample in the training batch. Moreover, while adversarial training might increase the DNNs' robustness against attackers using certain transformation types, attackers using new transformation types may still succeed at evasion [29]. Defenses that provide formal guarantees (e.g., [52], [69]) are even more computationally expensive than adversarial training. Moreover, those defenses are restricted to adversarial perturbations that, unlike the ones

[4]https://www.cygwin.com/

[5]https://www.passmark.com/products/performancetest/

DNN	*IPR*	*Disp*-1	*Disp*-3	*Disp*-5	*IPR*+*Disp*-1	*IPR*+*Disp*-3	*IPR*+*Disp*-5
AvastNet	-	**36**	**35**	**36**	**36**	**35**	**36**
Endgame	**33**	**35**	**36**	**35**	**35**	**36**	**35**
MalConv	-	**36**	**35**	**36**	**36**	**35**	**36**

(a) Malicious binaries

DNN	*IPR*	*Disp*-1	*Disp*-3	*Disp*-5	*IPR*+*Disp*-1	*IPR*+*Disp*-3	*IPR*+*Disp*-5
AvastNet	-	**2**	**2**	**1**	**2**	**2**	**1**
Endgame	1	**2**	**2**	**2**	**2**	**2**	**2**
MalConv	-	**1**	**1**	**1**	**1**	**1**	**1**

(b) Benign binaries

TABLE IV: The median number of VirusTotal anti-viruses that positively detected (i.e., as malicious) malicious (a) and benign (b) binaries that were transformed by our white-box attacks (columns) to mislead the different DNNs (rows). The median number of anti-viruses that positively detected for the original malicious and benign binaries is 55 and 0, respectively. Cases in which the change in the number of detections is statistically significant are in bold.

produced by our attacks, have small L_∞- and L_2-norms. Prior defenses that transform the input before classification (e.g., via quantization [107]) are designed mainly for images and do not directly apply to binaries. Lastly, signature-based malware detection would not be effective, as our attacks are stochastic and produce different variants of the binaries after different executions.

Differently from prior attacks on DNNs for malware detection [50], [56], [97], our attacks do not merely append adversarially crafted bytes to binaries, or insert them between sections. Such attacks may be defended against by detecting and sanitizing the inserted bytes via static analysis methods (e.g., similarly to the proof of concept shown in Sec. IV-C, or using other methods [57]). Instead, our attacks transform binaries' original code, and extend binaries only by inserting instructions that are executed at run time at various parts of the binaries. As a result, our attacks are difficult to defend against via static or dynamic analyses methods (e.g., by detecting and removing unreachable code), especially when augmented by measures to evade these methods.

Binary normalization [3], [18], [103] is an approach that was proposed to enhance malware detection that seemed viable for defending against our attacks. The high-level idea of normalization is to employ certain transformations to map binaries to a standard form and thus undo attackers' evasion attempts before classifying the binaries as malicious or benign. For example, Christodorescu et al. proposed a method to detect and remove semantic nops from binaries before classification, and showed that it improves the performance of commercial anti-viruses [18]. To mitigate our *Disp*-based attacks, we considered using the semantic nop detection and removal method followed by a method to restore the displaced code to its original location. Unfortunately, we realized that such a defense can be undermined using *opaque predicates* [22], [72]. Opaque predicates are predicates whose value (w.l.g., assume true) is known a priori to the attacker, but is hard for the defender to deduce. Often, they are based on *NP*-hard problems [72]. Using opaque predicates, attackers can produce semantic nops that include instructions that affect the memory and registers only if an opaque predicate evaluates to false. Since opaque predicates are hard for defenders to deduce, the defenders are likely to have to assume that the semantic nops impact the behavior of the program. As a result, the semantic nops would survive the defenders' detection and removal attempts. As an alternative to opaque predicates, attackers can also use *evasive predicates*—predicates that evaluate to true or false with an overwhelming probability (e.g., checking if a randomly drawn 32-bit integer is equal to 0) [9]. In this case, the binary will function properly the majority of the time, and may function differently or crash once every many executions.

The normalization methods proposed by prior work would not apply to the transformations performed by our *IPR*-based attacks. Therefore, we explored methods to normalize binaries to a standard form to undo the effects of *IPR* before classification. We found that a normalization process that leverages the *IPR* transformations to map binaries to the form with the lowest lexicographic representation (where the alphabet contains all possible 256 byte values) is effective at preventing *IPR*-based attacks. Formally, if $[x]$ is the equivalence class of binaries that are functionally equivalent to x and that can be produced via the *IPR* transformation types, then the normalization process produces an output $norm(x) \in [x]$, such that, $norm(x) \leq x_i$ for every $x_i \in [x]$. Alg. 2 in App. A presents an algorithm that computes the normalized form of a binary when executed for a large number of iterations, and approximates it when executed for a few iterations. At a high level, the algorithm applies the *IPR* transformations iteratively in a way that attempts to reduce the lexicographic representation after every iteration. We found that executing the algorithm for ten iterations is sufficient to defend against *IPR*-based attacks. In particular, we executed the normalization algorithm using the malicious and benign binaries produced by the *IPR*-based attacks to fool *Endgame* in the white-box setting, and found that the success rates dropped to 3% and 0%, respectively, compared to 62% and 74% before normalization. At the same time, the classification accuracy over the original binaries was not affected by normalization. As our experiments in Sec. IV have shown, generating functionally equivalent variants of binaries via random transformations results in correct classifications almost all of the time. Normalization of binaries to the minimal lexicographic representation deterministically leads to the specific functionally equivalent variants that get correctly classified with high likelihood.

Instruction Masking While normalization was useful for defending against *IPR*-based attacks, it cannot mitigate the more pernicious *Disp*-based attacks that are augmented with opaque or evasive predicates. Moreover, normalization has the general limitations that attackers could use transformations that the normalization algorithm is not aware of or could obfuscate code to inhibit normalization. Therefore, we explored additional defensive measures. In particular, motivated by the fact that randomizing binaries without the guidance of an optimization process is unlikely to lead to misclassification, we explored whether masking instructions at random can mitigate attacks while maintaining high performance on the original binaries. The defense works by selecting a random subset of the bytes that pertain to instructions and masking them with zeros (a commonly used value to pad sections in binaries). While the masking is likely to result in an ill-formed binary that is unlikely to execute properly (if at all), the masking only occurs before classification, which does not require a functional binary. Depending on the classification result, one can decide whether or not to execute the unmasked binary.

We tested the defense on binaries generated via the *IPR*+*Disp*-5 white-box attack and found that it is effective at mitigating attacks. For example, when masking 25% of the bytes pertaining to instructions, the success rates of the attack decreases from 83%–100% for malicious and benign binaries against the three DNNs to 0%–20%, while the accuracy on the original samples was only slightly affected (e.g., it became 94% for *Endgame*). Masking less than 25% of the instructions' bytes was not as effective at mitigating attacks, while masking more than 25% led to a significant decrease in accuracy on the original samples.

Detecting Adversarial Examples To prevent binaries transformed with our attacks (i.e., adversarial examples) from fooling malware detection, defenders may attempt to deploy methods to detect them. In cases of positive detections of adversarial examples, defenders may immediately classify them as malicious (regardless of whether they were originally malicious or benign). For example, because *Disp*-based attacks increase binaries' sizes and introduce additional `jmp` instructions, defenders may train statistical ML models that use features such as binaries' sizes and the ratio between `jmp` instructions and other instructions to detect adversarial examples. While training relatively accurate detection models may be feasible, we expect this task to be difficult, as the attacks increase binaries' sizes only slightly (1%–5%), and do not introduce many `jmp` instructions (7% median increase for binaries transformed via *Disp*-5). Furthermore, approaches for detecting adversarial examples are likely to be susceptible to evasion attacks (e.g., by introducing instructions after opaque predicates to decrease the ratio between `jmp` instructions and others). Last, another risk that defenders should take into account is that the defense should be able to precisely distinguish between adversarial examples and non-adversarial benign binaries that are transformed by similar methods to mitigate code-reuse attacks [54], [77].

Takeaways While masking a subset of the bytes that pertain to instructions led to better performance on adversarial examples, it was still unable to prevent all evasion attempts. Although the defense may raise the bar to attackers, and make attacks even more difficult if combined with a method to detect adversarial examples, these defenses do not provide formal guarantees and so attackers may be able to adapt to undermine them. For example, attackers may build on techniques for optimization over expectations to generate binaries that would mislead the DNNs even when masking a large number of instructions, in a similar manner to how attackers can evade image-classification DNNs under varying lighting conditions and camera angles [7], [30], [91], [92]. In fact, prior work has already demonstrated how defenses without formal guarantees are often vulnerable to adaptive, more sophisticated, attacks [6]. Thus, since there is no clear defense to prevent attacks against the DNNs that we studied in this work, or even general methods to prevent attackers from fooling ML models via arbitrary perturbations, we advocate for augmenting malware-detection systems with methods that are not based on ML (e.g., ones using templates to reason about the semantics of programs [17]), and against the use of ML-only detection methods, as has become recently popular [24].

B. Potential Extensions

Our work focuses on attacks targeting DNNs for malware detection from raw bytes. Nevertheless, we believe that it can be extended to help study and improve the robustness of other malware-detection methods. For example, prior work studied the use of n-gram features for malware classification [51], [81]. By transforming binaries, our attacks can potentially change the n-gram statistics to evade malware detection.

Another potential extension to our work is to study and improve the robustness of clone-search methods (e.g., [28]) that are often used in reverse engineering for studying new malware, detecting patent infringements, or finding vulnerabilities in software. Ding et al. recently suggested the use of neural networks to map assembly code to vector representations that are similar for clones and different for non-clones [28]. Building on our attacks, we believe that attackers could manipulate the representations generated by such neural networks to make the representations of clones different (e.g., to make it difficult to study new malware), or make the representations of non-clones similar (e.g., to support a fake patent infringement case).

VI. Conclusion

Our work proposes evasion attacks on DNNs for malware detection. Differently from prior work, the attacks do not merely insert adversarially crafted bytes to mislead detection. Instead, guided by optimization processes, our attacks transform the instructions of binaries to fool malware detection while keeping functionality of the binaries intact. As a result, these attacks are challenging to defend against. We conservatively evaluated different variants of our attack against three DNNs under white-box and black-box settings, and found the attacks successful as often as 100% of the time. Moreover, we found that the attacks pose a security risk to commercial anti-viruses, particularly ones using ML, achieving evasion success rates of up to 85%. We explored several potential defenses, and found some to be promising. Nevertheless, adaptive adversaries remain a risk, and we recommend the deployment of multiple detection algorithms, including ones not based on ML, to raise the bar against such adversaries.

ACKNOWLEDGMENTS

We would like to thank Sandeep Bhatkar, Leyla Bilge, Yufei Han, Kevin Roundy, and Hugh Thompson for helpful discussions. This work was supported in part by the Multidisciplinary University Research Initiative (MURI) Cyber Deception grant; by NSF grants 1801391 and 1801494; by the National Security Agency under Award No. H9823018D0008; by gifts from Google and Nvidia, and from Lockheed Martin and NATO through Carnegie Mellon CyLab; and by a CyLab Presidential Fellowship and a NortonLifeLock Research Group Fellowship.

REFERENCES

[1] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth. Learning to evade static PE machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, 2018.

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

[3] S. E. Armoun and S. Hashemi. A general paradigm for normalizing metamorphic malwares. In *Proc. FIT*, 2012.

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. NDSS*, 2014.

[5] A. Athalye and N. Carlini. On the robustness of the CVPR 2018 white-box adversarial example defenses. *arXiv preprint arXiv:1804.03286*, 2018.

[6] A. Athalye, N. Carlini, and D. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proc. ICML*, 2018.

[7] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok. Synthesizing robust adversarial examples. In *Proc. ICML*, 2018.

[8] S. Baluja and I. Fischer. Adversarial transformation networks: Learning to generate adversarial examples. In *Proc. AAAI*, 2018.

[9] B. Barak, N. Bitansky, R. Canetti, Y. T. Kalai, O. Paneth, and A. Sahai. Obfuscation for evasive functions. In *Proc. TCC*, 2014.

[10] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Proc. ECML PKDD*, 2013.

[11] F. Biondi, M. Enescu, T. Given-Wilson, A. Legay, L. Noureddine, and V. Verma. Effective, efficient, and robust packing detection and classification. *Computers and Security*, 2018.

[12] N. Carlini and D. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proc. AISec*, 2017.

[13] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *Proc. IEEE S&P*, 2017.

[14] S.-T. Chen, C. Cornelius, J. Martin, and D. H. P. Chau. Shapeshifter: Robust physical adversarial attack on faster r-cnn object detector. In *Proc. ECML PKDD*, 2018.

[15] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J.-Y. Marion. Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In *Proc. CCS*, 2018.

[16] M. Christodorescu and S. Jha. Testing malware detectors. In *Proc. ISSTA*, 2004.

[17] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proc. IEEE S&P*, 2005.

[18] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical report, University of Wisconsin-Madison, 2005.

[19] Chronicle. Virustotal. https://www.virustotal.com/, 2004–. Online; accessed 17 June 2019.

[20] M. Cisse, Y. Adi, N. Neverova, and J. Keshet. Houdini: Fooling deep structured prediction models. In *Proc. NIPS*, 2017.

[21] J. M. Cohen, E. Rosenfeld, and J. Z. Kolter. Certified adversarial robustness via randomized smoothing. *arXiv preprint arXiv:1902.02918*, 2019.

[22] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, The University of Auckland, 1997.

[23] S. Coull and C. Gardner. What are deep neural networks learning about malware? https://www.fireeye.com/blog/threat-research/2018/12/what-are-deep-neural-networks-learning-about-malware.html, 2018. Online; accessed 1 July 2019.

[24] Cylance Inc. Cylance: Artificial intelligence based advanced threat prevention. https://www.cylance.com/en-us/index.html, 2019. Accessed on 06-17-2019.

[25] H. Dang, Y. Huang, and E.-C. Chang. Evading classifiers by morphing in the dark. In *Proc. CCS*, 2017.

[26] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583*, 2019.

[27] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! A case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[28] S. H. Ding, B. C. Fung, and P. Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proc. IEEE S&P*, 2019.

[29] L. Engstrom, D. Tsipras, L. Schmidt, and A. Madry. A rotation and a translation suffice: Fooling CNNs with simple transformations. In *Proc. NeurIPSW*, 2017.

[30] I. Evtimov, K. Eykholt, E. Fernandes, T. Kohno, B. Li, A. Prakash, A. Rahmati, and D. Song. Robust physical-world attacks on machine learning models. In *Proc. CVPR*, 2018.

[31] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*, 2017.

[32] W. Fleshman, E. Raff, J. Sylvester, S. Forsyth, and M. McLean. Non-negative networks against adversarial attacks. *arXiv preprint arXiv:1806.06108*, 2018.

[33] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *Proc. ICLR*, 2015.

[34] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*, 2017.

[35] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial examples for malware detection. In *Proc. ESORICS*, 2017.

[36] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser. The Cuckoo Sandbox. https://cuckoosandbox.org/, 2012. Online; accessed 21 June 2019.

[37] C. Guo, M. Rana, M. Cisse, and L. van der Maaten. Countering adversarial images using input transformations. 2018.

[38] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, 2005.

[39] C. Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proc. NSPW*, 2009.

[40] Hex-Rays. IDA: About. https://www.hex-rays.com/products/ida/. Online; accessed 13 September 2019.

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

[42] W. Huang and J. W. Stokes. MtNet: A multi-task neural network for dynamic malware classification. In *Proc. DIMVA*, 2016.

[43] A. Ilyas, L. Engstrom, and A. Madry. Prior convictions: Black-box adversarial attacks with bandits and priors. In *Proc. ICLR*, 2019.

[44] I. Incer, M. Theodorides, S. Afroz, and D. Wagner. Adversarially robust malware detection using monotonic classification. In *Proc. IWSPA*, 2018.

[45] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-llvm–software protection for the masses. In *Proc. IWSP*, 2015.

[46] H. Kannan, A. Kurakin, and I. Goodfellow. Adversarial logit pairing. *arXiv preprint arXiv:1803.06373*, 2018.

[47] A. Kantchelian, J. Tygar, and A. D. Joseph. Evasion and hardening of tree ensemble classifiers. In *Proc. ICML*, 2016.

[48] J. Kennedy, D. Batchelor, C. Robertson, M. Satran, and M. LeBLanc. PE format. https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format, 2019. Accessed on 06-03-2019.

[49] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. USENIX Security*, 2009.

[50] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *Proc. EUSIPCO*, 2018.

[51] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7(Dec):2721–2744, 2006.

[52] J. Z. Kolter and E. Wong. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proc. ICML*, 2018.

[53] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted code randomization. In *Proc. IEEE S&P*. IEEE, 2018.

[54] H. Koo and M. Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proc. AsiaCCS*, 2016.

[55] M. Krčál, O. Švec, M. Bálek, and O. Jašek. Deep convolutional malware classifiers can learn from raw executables and labels only. In *Proc. ICLRW*, 2018.

[56] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet. Adversarial examples on discrete sequences for beating whole-binary malware detection. In *Proc. NeurIPSW*, 2018.

[57] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. USENIX Security*, 2004.

[58] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial machine learning at scale. In *Proc. ICLR*, 2017.

[59] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Proc. IEEE S&P*, 2014.

[60] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proc. ICML*, 2014.

[61] M. Lecuyer, V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana. Certified robustness to adversarial examples with differential privacy. In *Proc. IEEE S&P*, 2019.

[62] F. Liao, M. Liang, Y. Dong, T. Pang, J. Zhu, and X. Hu. Defense against adversarial attacks using high-level representation guided denoiser. In *Proc. CVPR*, 2018.

[63] D. Lowd and C. Meek. Adversarial learning. In *Proc. KDD*, 2005.

[64] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *Proc. ICLR*, 2018.

[65] H. Massalin. Superoptimizer: A look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126, 1987.

[66] D. Meng and H. Chen. Magnet: A two-pronged defense against adversarial examples. In *Proc. CCS*, 2017.

[67] X. Meng, B. P. Miller, and S. Jha. Adversarial binaries for authorship identification. *arXiv preprint arXiv:1809.08316*, 2018.

[68] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff. On detecting adversarial perturbations. In *Proc. ICLR*, 2017.

[69] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *Proc. ICML*, 2018.

[70] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard. Universal adversarial perturbations. In *Proc. CVPR*, 2017.

[71] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. DeepFool: a simple and accurate method to fool deep neural networks. In *Proc. CVPR*, 2016.

[72] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. ACSAC*, 2007.

[73] P. S. Oliveto, T. Paixão, J. P. Heredia, D. Sudholt, and B. Trubenová. How to escape local optima in black box optimisation: When non-elitism outperforms elitism. *Algorithmica*, 80(5):1604–1633, 2018.

[74] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proc. AsiaCCS*, 2017.

[75] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *Proc. IEEE Euro S&P*, 2016.

[76] N. Papernot, P. McDaniel, A. Swami, and R. Harang. Crafting adversarial input sequences for recurrent neural networks. In *Proc. MILCOM*, 2016.

[77] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proc. IEEE S&P*, 2012.

[78] Y. Qin, N. Carlini, I. Goodfellow, G. Cottrell, and C. Raffel. Imperceptible, robust, and targeted adversarial examples for automatic speech recognition. In *Proc. ICML*, 2019.

[79] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *Proc. USENIX Security*, 2019.

[80] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas. Malware detection by eating a whole exe. In *Proc. AAAIW*, 2018.

[81] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, 14(1):1–20, 2018.

[82] M. Roberts. Virusshare. https://virusshare.com/, 2012. Online; accessed 18 June 2019.

[83] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*, 2018.

[84] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *Proc. RAID*, 2018.

[85] K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR)*, 46(1):4, 2013.

[86] P. Samangouei, M. Kabkab, and R. Chellappa. Defense-GAN: Protecting classifiers against adversarial attacks using generative models. In *Proc. ICLR*, 2018.

[87] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proc. ASPLOS*, 2013.

[88] L. Schönherr, K. Kohls, S. Zeiler, T. Holz, and D. Kolossa. Adversarial attacks against automatic speech recognition systems via psychoacoustic hiding. In *Proc. NDSS*, 2019.

[89] M. Sconzo. Packer yara ruleset. https://github.com/sooshie/packerid, 2014. Online; accessed 18 June 2019.

[90] M. Sharif, L. Bauer, and M. K. Reiter. On the suitability of lp-norms for creating and preventing adversarial examples. In *Proc. CVPRW*, 2018.

[91] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proc. CCS*, 2016.

[92] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter. Adversarial generative nets: Neural network attacks on state-of-the-art face recognition. *arXiv preprint arXiv:1801.00349*, 2017.

[93] R. Shetty, B. Schiele, and M. Fritz. A4nt: Author attribute anonymity by adversarial training of neural machine translation. In *Proc. USENIX Security*, 2018.

[94] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

[95] V. Srinivasan, A. Marban, K.-R. Müller, W. Samek, and S. Nakajima. Counterstrike: Defending deep learning architectures against adversarial samples by langevin dynamics with supervised denoising autoencoder. *arXiv preprint arXiv:1805.12017*, 2018.

[96] N. Srndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proc. IEEE S&P*, 2014.

[97] O. Suciu, S. E. Coull, and J. Johns. Exploring adversarial examples in malware detection. In *Proc. AAAIW*, 2018.

[98] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *Proc. ICLR*, 2014.

[99] P. Szor. *The Art of Computer Virus Research and Defense*. Pearson Education, 2005.

[100] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proc. IEEE S&P*, 2015.

[101] VirusTotal. Packer yara ruleset. https://github.com/Yara-Rules/rules/tree/master/Packers, 2016. Online; accessed 18 June 2019.

[102] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proc. CCS*, 2002.

[103] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. In *Proc. SCAM*, 2006.

[104] S. Wang, P. Wang, and D. Wu. Uroboros: Instrumenting stripped binaries with static reassembling. In *Proc. SANER*, 2016.

[105] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song. Generating adversarial examples with adversarial networks. In *Proc. IJCAI*, 2018.

[106] C. Xie, Y. Wu, L. van der Maaten, A. Yuille, and K. He. Feature denoising for improving adversarial robustness. *arXiv preprint arXiv:1812.03411*, 2018.

[107] W. Xu, D. Evans, and Y. Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *Proc. NDSS*, 2018.

[108] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Proc. NDSS*, 2016.

[109] H. Zhang, Y. Yu, J. Jiao, E. P. Xing, L. E. Ghaoui, and M. I. Jordan. Theoretically principled trade-off between robustness and accuracy. *arXiv preprint arXiv:1901.08573*, 2019.

APPENDIX

A. In-Place Normalization

In this section, we present a normalization process to map binaries to a standard form and undo the effect of the *IPR* transformations on classification. Specifically, the normalization process maps binaries to the functionally equivalent variant with the lowest lexicographic presentation that is achievable via the *IPR* transformation types. For each transformation type, we devise an operation that would decrease a binary's lexicographic representation when applied: *1)* instructions would be replaced with equivalent ones only if the new instructions are lexicographically lower (*Eqv*); *2)* registers in functions would be reassigned only if the byte representation of the first impacted instruction would decrease (*Regs*); *3)* instructions would be reordered such that each time we would extract the instruction from the dependence graph with the lowest byte representation that does not depend on any of the remaining instructions in the graph (*Ord1*); and *4)* `push` and `pop` instructions that save register values across function calls would be reordered to decrease the lexicographic representation while maintaining the last-in-first-out order (*Ord2*). Fig. 5 depicts an example of replacing one instruction with an equivalent one via *Eqv* to decrease the lexicographic order of code.

Unfortunately, as shown in Fig. 6, when the different types of transformation types are composed, applying individual normalization operations does not necessarily lead to the binary's variant with the minimal lexicographic representation, as the procedure may be stuck in a local minima. To this end, we propose a stochastic algorithm that is guaranteed to converge to binaries' normalized variants if executed for a sufficiently large number of iterations.

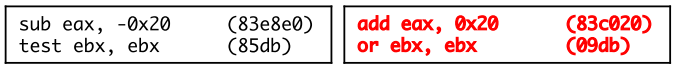


Fig. 5: An example of normalizing code via *Eqv*. The original code (a) is transformed via *Eqv* (b) to decrease the lexicographic order.

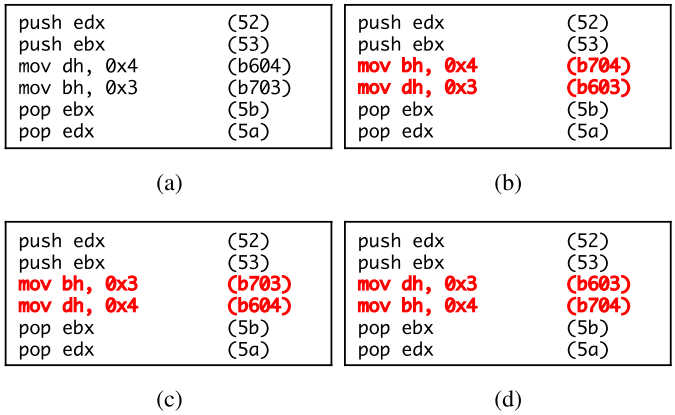


Fig. 6: The normalization process can get stuck in a local minima. The lexicographic order of the original code (a) increases when reassigning registers (b) or reordering instrcutions (c). However, composing the two transformation (d) decreases the lexicographic order.

Alg. 2 presents a psuedocode of the normalization algorithm. The algorithm receives a binary x and the number of iterations $niters$ as inputs. It begins by drawing a random variant of x, by applying all the transformation types to each function at random (line 1). The algorithm then proceeds to apply each of the individual normalization operations to decrease the lexicographic representation of the binary (lines 14–24), while self-supervising the normalization process. Specifically, the algorithm keeps track of the last iteration an operation decreased the binary's representation (line 19). If none of the four operations affect any of the functions, we deduce that the normalization process is stuck in a (global or local) minima, and a random binary is drawn again by randomizing all functions (limes 10–12), and the normalization process restarts.

When $niters \rightarrow \infty$ (i.e., the number of iterations is large enough), Alg. 2 would eventually converge to a global minima. Namely, it would find the variant of x with the minimal lexicographic representation. In fact, we are guaranteed to find $norm(x)$ even if we simply apply the transformation types at random x for $niters \rightarrow \infty$ iterations. When testing the algorithm with two binaries of moderate size, we found that $niters$=2,000 was sufficient to converge for the same respective variants after every run. These variants are likely to be the global minimas. However, executing the algorithm for 2,000 iterations is computationally expensive, and impractical within the context of a widely deployed malware-detection system. Hence, for the purpose of our experiments, we set $niters$=10,

Algorithm 2: Binary normalization.

Input : x, $niters$
Output: x_{min}

$\hat{x} \leftarrow RandomizeAll(x)$;
$i, last_{update} \leftarrow 0, 0$;
$x_{min} \leftarrow \hat{x}$;
while $i < niters$ **do**
  if $i\%4 = 0$ **then**
    `// Normalization operations`
    $ops \leftarrow \{\}$;
    for $f \in \hat{x}$ **do**
      $ops[f] = \{Eqv, Regs, Ord1, Ord2\}$;
    end
    `// If stuck, randomize and restart`
    if $i - last_{update} \geq 4$ **then**
      $\hat{x} \leftarrow RandomizeAll(x)$;
    end
  end
  for $f \in \hat{x}$ **do**
    $o \leftarrow pop(ops[f])$;
    $\tilde{x} \leftarrow Minimize(\hat{x}, f, o)$;
    if $\hat{x} \neq \tilde{x}$ **then**
      $\hat{x} \leftarrow \tilde{x}$;
      $last_{update} \leftarrow i$;
      if $\hat{x} < x_{min}$ **then**
        $x_{min} \leftarrow \hat{x}$;
      end
    end
  end
  $i \leftarrow i + 1$;
end
return x_{min};

which we found to be sufficient to successfully mitigate the majority of attacks.