Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes

Keane Lucas
keanelucas@cmu.edu
Carnegie Mellon University

Mahmood Sharif
mahmoods@vmware.com
Tel Aviv University and VMware

Lujo Bauer
lbauer@cmu.edu
Carnegie Mellon University

Michael K. Reiter
michael.reiter@duke.edu
Duke University

Saurabh Shintre
saurabh.shintre@nortonlifelock.com
NortonLifeLock Research Group

ABSTRACT

Motivated by the transformative impact of deep neural networks (DNNs) in various domains, researchers and anti-virus vendors have proposed DNNs for malware detection from raw bytes that do not require manual feature engineering. In this work, we propose an attack that interweaves binary-diversification techniques and optimization frameworks to mislead such DNNs while preserving the functionality of binaries. Unlike prior attacks, 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- and black-box settings, and found that it often achieved 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 foil over 80% of our evasion attempts. However, these defenses may still be susceptible to evasion by attacks, and so we advocate for augmenting malware-detection systems with methods that do not rely on machine learning.

CCS CONCEPTS

• **Security and privacy** → **Malware and its mitigation**; • **Computing methodologies** → **Supervised learning**.

KEYWORDS

adversarial machine learning; malware; neural networks; security

ACM Reference Format:
Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K. Reiter, and Saurabh Shintre. 2021. Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3433210.3453086

1 INTRODUCTION

Modern malware detectors, both academic (e.g., [4, 44]) and commercial (e.g., [25, 90]), increasingly rely on machine learning (ML) to classify executables as benign or malicious based on features such as imported libraries and API calls. In the space of static malware detection, where an executable is classified prior to its execution, recent efforts have proposed deep neural networks (DNNs) that detect malware from binaries' raw byte-level representation, with effectiveness similar to that of detectors based on hand-crafted features selected through tedious manual processing [54, 76].

As old techniques for obfuscating and packing malware (see Sec. 4) are rendered ineffective in the face of static ML-based detection, recent advances in adversarial ML might provide a new opening for attackers to bypass detectors. Specifically, ML algorithms, including DNNs, have been shown vulnerable to adversarial examples—modified inputs that resemble normal inputs but are intentionally designed to be misclassified. For instance, adversarial examples can enable attackers to impersonate users that are enrolled in face-recognition systems [85, 86], fool street-sign recognition algorithms into misclassifying street signs [30], and trick voice-controlled interfaces to misinterpret commands [21, 74, 83].

In the malware-detection domain, the attackers' goal is to alter programs to mislead ML-based malware detectors to misclassify malicious programs as benign or vice versa. In doing so, attackers face a non-trivial constraint: in addition to misleading the malware detectors, alterations to a program must not change its 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 exfiltrating information. This constraint is arguably more challenging than ones imposed by other domains (e.g., evading image recognition without making changes conspicuous to humans [30, 85, 86]) as it is less amenable to being encoded into traditional frameworks for generating adversarial examples, and most changes to a program's raw binary are likely to break a program's syntax or semantics. Prior work proposed attacks to generate adversarial examples to fool static malware detection DNNs [27, 49, 55, 72, 89] by adding adversarially crafted byte values in program regions that do not affect execution (e.g., at the end of programs or between sections). These attacks can be defended against by eliminating the added content before classification (e.g., [56]); we confirm this empirically.

In contrast, we develop a new way to modify binaries to both retain their functionality and mislead state-of-the-art DNN-based static malware detectors [54, 76]. We leverage binary-diversification tools—originally proposed to defend against code-reuse attacks by transforming program binaries to create diverse variants [53, 71]—to evade malware-detection DNNs. While these tools preserve the



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs International 4.0 License.

ASIA CCS '21, June 7–11, 2021, Hong Kong, Hong Kong.
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8287-8/21/06.
https://doi.org/10.1145/3433210.3453086

functionality of programs by design (e.g., functionality-preserving randomization), their naïve application is insufficient to evade malware detection. 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 have no access (i.e., black-box). The algorithms we propose can produce program variants that often fool DNNs in 100% of evasion attempts and, surprisingly, even evade some commercial malware detectors (likely over-reliant on ML-based static detection), 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 complementary methods to further deter static or dynamic analysis (as our methods alone should have no effect on dynamic analysis). We explore potential mitigations to our attacks (e.g., by normalizing programs before classification [3, 18, 98]), but identify their limitation in thwarting adaptive attackers.

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

- We repair and extend prior binary-diversification implementations to iteratively yield candidate transformations. We also reconstruct them to be composable, more capable, and resource-efficient. The code is available online.[1]
- We propose a novel functionality-preserving attack on DNNs for static malware detection from raw bytes (Sec. 3). The attack precisely composes the updated binary-diversification techniques, evades defenses against prior attacks, and applies to both white- and black-box settings.
- We evaluate and demonstrate the effectiveness of the proposed attack in different settings, including against commercial malware detectors (Sec. 4). We show that our attack effectively undermines ML-based static analysis, a significant component of state-of-the-art malware detection, while being robust to defenses that can thwart prior attacks.
- We explore the effectiveness of prior and new defenses against our proposed attack (Sec. 5). While some defenses seem promising against specific variants of the attack, none explored neutralize our most effective attack, and they are likely vulnerable to adaptive attackers.

2 BACKGROUND AND RELATED WORK

We first discuss previous work on DNNs that detect malware by examining program binaries. We then discuss research on attacking and defending ML algorithms generally, and malware detection specifically. Finally, we provide background on binary randomization methods, which serve as building blocks for our attacks.

2.1 DNNs for Static Malware Detection

We study attacks targeting two DNN architectures for detecting malware from the raw bytes of Windows binaries (i.e., executables in Portable Executable format) [54, 76]. 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, 43, 50]). Due to their desirable properties, computer-security companies use DNNs similar to the ones we study (i.e., ones that operate on raw bytes and use a convolution architectures) for malware detection [24].

The DNNs proposed by prior work follow standard convolutional architectures similar to the ones used for image classification [54, 76]. Yet, in contrast to image classifiers that classify continuous inputs, malware-detection DNNs classify discrete inputs—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 $\mathbb{R}^8$. After the embedding, standard convolutional and non-linear operations are performed by subsequent layers.

2.2 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. Most prior attacks (e.g., [9, 11, 14, 33, 70, 91]) 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; e.g., Carlini and Wagner [14] proposed the following formulation for finding adversarial perturbations that target a class c_t and have small L_2-norms:

$$\arg\min_r \; Loss_{cw}(x + r, c_t) + \kappa \cdot ||r||_2$$

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:

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

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 on Static Malware Detection Modern malware detection systems often leverage both dynamic and static analyses to determine maliciousness [8, 25, 44, 90, 93]. While in most cases an attacker would hence need to adopt countermeasures against both of these types of analyses, in other situations, such as potential attacks on end-user systems protected predominantly through static-analysis based anti-virus detectors [20, 95], defeating a static malware detector could be sufficient for an attacker to achieve their goals. Even when a combination of static and dynamic analyses is used for detecting malware, fooling static analysis is necessary for an attack to succeed. Here we focus on attacks that target ML-based static analyzers for detecting malware.

Multiple attacks were proposed to evade ML-based malware classifiers while preserving the malware's functionality. Some (e.g., [26, 88, 97, 102]) tweak malware to mimic benign files (e.g., adding benign code-snippets to malicious PDF files). Others (e.g., [1, 27, 35, 41, 49, 55, 72, 89]) tweak malware using gradient-based optimizations or generative methods (e.g., to find which APIs to import). Still others combine mimicry and gradient-based optimizations [79].

Differently from some prior work (e.g., [1, 79, 97]) that studied attacks against dynamic ML-based malware detectors, we explore attacks that target DNNs for malware detection from raw bytes

[1] https://github.com/pwwl/enhanced-binary-diversification

(i.e., static detection methods). Furthermore, the attacks we explore do not introduce adversarially crafted bytes to unreachable regions of the binaries [49, 55, 89] (which may be possible to detect and sanitize statically, see Sec. 4.4), or by mangling bytes in the header of binaries [27] (which can be stripped before classification [78]). Instead, our attacks transform actual instructions of binaries in a functionality-preserving manner to achieve misclassification.

More traditionally, attackers use various obfuscation techniques to evade malware detection. Packing [12, 80, 92, 94]—compressing or encrypting binaries' code and data, and then uncompressing or decrypting them at run time—is commonly used to hide malicious content from static detection methods. As we explain later (Sec. 3.1) we mostly consider unpacked binaries in this work, as is typical for static analysis [12, 54]. Attackers also obfuscate binaries by substituting instructions or altering their control-flow graphs [16, 17, 45, 92]. We demonstrate that such obfuscation methods do not fool malware-detection DNNs when applied naïvely (see Sec. 4.3). To address this, our attacks guide the transformation of binaries via stochastic optimization techniques to mislead malware detection.

Pierazzi et al. formalized the process of adversarial example generation in the problem space and used their formalization to produce malicious Android apps that evade detection [73]. Our attack fits the most challenging setting they describe, where mapping the problem space to features space is non-invertible and non-differentiable.

Most closely related to our work is the recent work on misleading ML algorithms for authorship attribution [65, 75]. Meng et al. proposed an attack to mislead authorship attribution at the binary level [65]. 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 [75]. 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 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, 46, 47, 57, 62, 91]. Another line of work proposes algorithms to train certifiably (i.e., provably) robust defenses against certain attacks [22, 51, 60, 67, 103], though 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 we show in Sec. 5, amongst other limitations, 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, 61, 64, 81, 87, 100, 101]. 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 attackers [13] can circumvent methods for detecting the presence of attacks (e.g., [31, 34, 64, 66]). We expect that such attackers can circumvent attempts to detect our attacks too.

Prior work proposed ML-based malware-classification methods designed to be robust against evasion [28, 43]. However, these methods either have low accuracy [43] or target linear classifiers [28], which are unsuitable for detecting malware from raw bytes.

Fleshman et al. proposed to harden malware-detection DNNs by constraining 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 feature addition at the input may correspond to feature deletion 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 [55] as undefended DNNs.

2.3 Binary Rewriting and Randomization

Software diversification is an approach 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 [58]. Diversification can be performed on source code, during compilation, or by rewriting and randomizing programs' binaries. In this work, we build on binary-level diversification techniques, as they have wider applicability (e.g., self-spreading malware can use them to evade detection without source-code access [68]). Nevertheless, we expect that this work can be extended to work with different diversification methods.

Binary rewriting takes many forms (e.g., [38, 52, 53, 63, 71, 82, 99]). Certain methods aim to speed up code via expensive search through the space of equivalent programs [63, 82]. Other methods significantly increase binaries' sizes, or leave conspicuous signs that rewriting took place [38, 99]. We build on binary-randomization tools that have little-to-no effect on the size or run time of randomized binaries, thus helping our attacks remain stealthy [53, 71]. We present these tools and our extensions thereof in Sec. 3.2.

3 TECHNICAL APPROACH

Here we present the technical approach of our attack. Before delving into the details, we initially describe the threat model.

3.1 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 can 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 DNNs' weights are fixed and *cannot* be controlled by attackers (e.g., by poisoning the training data). The attackers use binary

rewriting to manipulate the raw bytes of binaries and cause misclassification while keeping functionality intact. Namely, attackers aim mislead the DNNs while ensuring that the I/O behavior of program and the order of syscalls remain the same after rewriting. In certain practical settings (e.g., when both dynamic and static detection methods are used [92]) evading static detection techniques as the DNNs we study may be insufficient to evade the complete stack of detectors. Nonetheless, evading the static detection techniques in such settings is *necessary* for evading detection overall. In Sec. 4.6, we show that our attacks can evade commercial detectors, some of which may be using multiple detection methods.

Attacks may seek to cause malware to be misclassified as benign or benign binaries to be misclassified as malware. The former may cause malware to circumvent defenses and be executed on a victim's machine. The latter induces false positives, which may lead users to turn off or ignore the defenses [39]. Our methods are applicable to transform binaries in either direction, but we focus on transforming malicious binaries in this paper.

As is common for static malware detection [12, 54], we assume that the binaries are unpacked. While adversaries may attempt to evade detection via packing, our attack can act as an alternative or a complementary evasion technique (e.g., once packing is undone). Such a technique is particularly useful as packer-detection (e.g., [12]) and unpacking (e.g., [15]) techniques improve. In fact, we found that packing with a popular packer increases the likelihood of detection for malicious binaries (see Sec. 4.6), thus further motivating the need for complementary evasion measures.

As is standard for ML-based malware detection from raw bytes in particular (Sec. 2.1), and for classification of inputs from discrete domains in general (e.g., [59]), 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 $\mathbb{E}(x) = \ell_i(\dots \ell_0(x)\dots)$.

3.2 Functionality-Preserving Attack

The attack we propose iteratively transforms a 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, only functionality preserving transformations are used. In each iteration, the attack determines the subset of transformations that can be safely used on each function in the binary. The attack then randomly selects a transformation from each function-specific subset and enumerates candidate byte-level changes. Each candidate set of changes is mapped to its corresponding gradient. The changes are only applied if this gradient has positive cosine similarity with the target model's loss gradient.

Alg. 1 presents the pseudocode of the attack in the white-box setting. The algorithm starts with a random initialization. This is

Algorithm 1: White-box attack.

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

manifested 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 and applied to that function without consulting loss-gradient similarity. 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 using our gradient-guided method 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. Particularly, we use the $Loss_{cw}$, presented in Sec. 2, as loss function. Because the true value of g is affected by any committed function change and could be unreliable after transforming many preceding functions in large files, it is recalculated prior to transforming each function (lines 5–6).

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 [49, 55]). However, if applied naively, such manipulation would likely cause the binary to be ill-formed or change its functionality. Instead, we transform the binary via functionality-preserving transformations. As the transformations are stochastic and may have many possible outcomes (in some cases, more than can be feasibly enumerated), we cannot precisely estimate their impact on the binary a priori. Therefore, we implement the transformation of each function, f, as the acceptance or denial of candidate functionality-preserving transformations we iteratively generate throughout the function, where we apply a transformation only if it shifts the embedding in a direction similar to g (lines 5–13). 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 each small candidate 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 [102]) and similarity measures (e.g., similarity in the Euclidean space) that we tested did not perform as well.

If the input was 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., [42]). 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., [88]) rather than gradient ascent. The black-box attack is conceptually similar to the white-box one, and differs only in the method of 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 We consider two families of transformation types [53, 71]. As the first family, we adopt and extend transformation types proposed for in-place randomization (*IPR*) [71]. 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 functionality, and then update the binary accordingly from the modified assembly. The four transformation types are: *1)* replacing instructions with equivalent ones of the same length (e.g., `sub eax,4` → `add eax,-4`); *2)* reassigning registers within functions or sets of basic blocks (e.g., swapping all instances of `ebx` and `ecx`) if this does not affect code that follows; *3)* reordering instructions using a dependence graph to ensure that no instruction appears before one it depends on; and *4)* altering the order in which register values are pushed to and popped from the stack to preserve them across function calls. To maintain the semantics of the code, the disassembly and transformations are performed conservatively (e.g., speculative disassembly, which is likely to misidentify code, is avoided). *IPR* does not alter binaries' sizes and has no measurable effect on their run time [71]. Fig. 1 shows examples of transforming code via *IPR*.

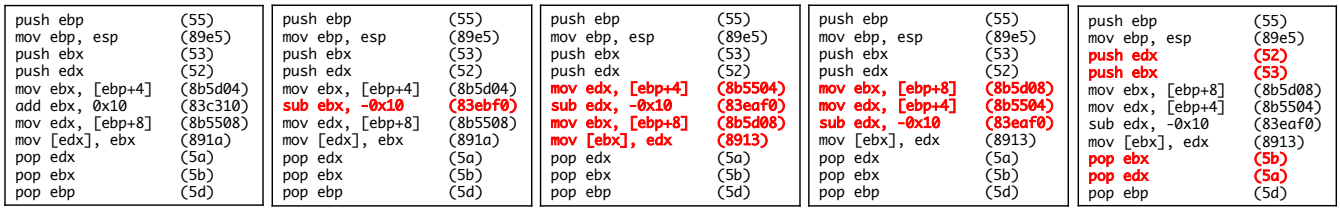
The original implementation of Pappas et al. was unable to produce the majority of functionally equivalent binary variants that should be achievable under the four transformation types. Thus, we extended and improved the implementation in various ways. First, we enabled the transformations to compose: 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, ours handles a larger number of instructions and function-calling conventions. In particular, our implementation can rewrite binaries containing additional instructions (e.g., `shrd`, `shld`, `ccmove`) and less common calling conventions (e.g., nonstandard returns via increment of `esp` followed by a `jmp` instruction). Last, we fixed significant bugs in the original implementation. These bugs include incorrect checks for writes to memory after reads, as well as memory leaks which required routine experiment restarts.

The second family of transformation types that we build on is based on code displacement (*Disp*) [53]. Similarly to *IPR*, *Disp* begins by conservatively disassembling the binary. The original idea of *Disp* is to break potential gadgets that can be leveraged by code-reuse attacks by moving code to a new executable section. 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 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 to the displaced code to pass control back to the instruction that should follow. 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 (~2% increase on average) and causes a small amount of run-time overhead (<1% on average) [53].

We extend *Disp* in two main ways. First, we allow it to displace any set of consecutive instructions within a 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.

While nops can be defined atomically (e.g., by a `nop` instruction), initial failures to mislead malware detection indicated that a rich semantic nop language is needed for successful attacks. Such a language enables the attack to search through a large set of functionally equivalent programs to evade DNNs. Therefore, we developed a context-free grammar to create diverse semantic nops (see Fig. 2). At a high level, a semantic nop is an atomic instruction; or 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 eax`); or 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 (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 divide the budget (expressed as the number of bytes to be displaced) by the number of functions in the binary and attempt to displace exactly that number of bytes per function. If multiple options exist for what code in a function to displace, we choose at random. If a function does not contain enough code to displace, then we attach semantic nops 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 displaced code, to consume

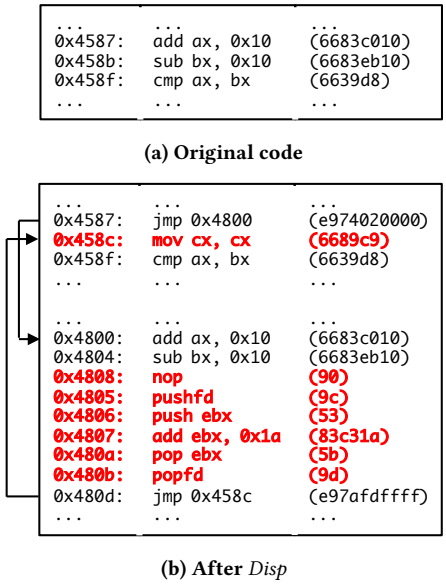


(a) Original (b) Equivalent instructions (c) Register reassignment (d) Instruction reordering (e) Register preservation

Figure 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 preserve register values (e). We provide the hex encoding of each instruction to its right. The affected instructions are boldfaced and colored in red.

```
1  S → Atom | S · S |
2        bswp r · S · bswp r |
3        xchg rh, rl · S · xchg rh, rl |
4        push r · S_r · pop r |
5        pushfd · S_ef · popfd
6  Atom → Φ | nop | mov r, r
7  S_r → S | S_r · S_r | pushfd · S_ef,r · popfd
8  S_ef → S | S_ef · S_ef |
9          arth r, v · S_ef · invarth r, v |
10         push r · S_ef,r · pop r
11 S_ef,r → S | S_r | S_ef | S_ef,r · S_ef,r |
12         arth r, v · S_ef,r |
13         logic r, v · S_ef,r
```

Figure 2: A context-free grammar for generating semantic nops. S is the starting symbol; Φ the empty string; `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.



(a) Original code

(b) After *Disp*

Figure 3: An example of displacement. The two instructions staring at address `0x4587` in the original code (a) are displaced 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` that passes the control back to the original code. Semantic nops are shown in boldface and red.

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

Known methods [18] for detecting and removing semantic nops from binaries might appear viable for defending against *Disp*-based attacks. However, as we discuss in Sec. 5, attackers can leverage various techniques to evade semantic-nop detection and removal.

Limitations Our implementation leaves room for improvement. For instance, it does not displace code that has been displaced in earlier iterations. A better implementation might apply displacements recursively. Furthermore, the composability of *IPR* and *Disp* transformations could be improved. In particular, when applying both *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*. Despite room for improvement, our implementation is already sufficient to generate successful attacks (see below).

4 EVALUATION

In this section, we comprehensively evaluate our attack. We first detail 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 evaluate variants

VTFeed	Train	Val.	Test
Benign	111,258	13,961	13,926
Malicious	111,395	13,870	13,906

Table 1: The number of benign and malicious binaries used to train, validate, and test the DNNs.

	Accuracy			TPR @
	Train	Val.	Test	0.1% FPR
AvastNet	99.89%	98.59%	98.60%	94.78%
MalConv	99.97%	98.67%	98.53%	96.08%

Table 2: The DNNs' accuracy and the TPR at the operating point where the FPR equals 0.1%.

of our attack in the white- and black-box setting and compare with prior work. We then evaluate our attack against commercial anti-viruses and close the section with experiments to validate that the attacks preserve functionality.

4.1 Datasets and Malware-Detection DNNs

4.1.1 Dataset composition. Our dataset, *VTFeed*, 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]. Overall, we use significantly more samples than similar prominent prior work (e.g., [4, 50]).

VTFeed was collected by sampling the VirusTotal feed for PE binaries, representing binaries encountered in practice by anti-virus vendors. Collection took around two weeks and was restricted to binaries first seen in 2020, to ensure recency, and smaller than 5 MB. Following prior work [2], binaries were filtered and labeled as benign (resp., malicious) if they were classified as malicious by 0 (resp., over 40) antivirus vendors as aggregated by VirusTotal. The dataset contains 278,316 binaries with a roughly even distribution between benign and malicious binaries. We sampled training, test, and validation sets at a ratio of 80%, 10%, and 10% respectively. Exact numbers can be seen in Table 1.

4.1.2 DNN Training. Using the malicious and benign samples, we trained two malware-detection DNNs. All DNNs receive binaries' raw bytes as inputs and output the probability that the binaries are malicious. The first DNN (henceforth, *AvastNet*), proposed by Krčál et al. [54], receives inputs up to 512 KB in size. The second DNN (henceforth, *MalConv*), proposed by Raff et al. [76], receives inputs up to 2 MB in size. Except for the batch size (set to 32 due to memory limitations), we used the same training parameters reported in prior work. When using binaries for training, we excluded their headers so the DNNs would not rely on header values, which are easily manipulable, for classification [27].

Each DNN achieves test accuracy of about 99% (see Table 2). Even when restricting the false positive rates (FPRs) conservatively to 0.1% (as is often done by anti-virus vendors [54]), the true positive rates (TPRs) remain as high as 94–96% (i.e., 94–96% of malicious binaries are detected). These results are superior to those reported in the original papers both for classification from raw bytes and from manually crafted features [54, 76]. This is likely because *VTFeed* was sampled over a narrow time span, and expect the performance would slightly decrease if we increased the diversity of the dataset.

In addition to the two DNNs that we trained, we evaluated our attacks using a publicly available DNN (henceforth, *Endgame*) trained by Anderson and Roth [2]. *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 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. We used standard packer detectors, Packerid [84] and Yara [96], and deemed binaries as unpacked only if no detector exhibited a positive detection. This method is similar to the one followed by Biondi et al. [12].[2] We also filtered out binaries labeled as packed in their VirusTotal metadata. While the data used to train and test the DNNs included packed binaries, 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 binaries had to be classified as malicious and the estimated probability that they are malicious had to be above the threshold where the FPR 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. We further restricted the binaries' sizes to be smaller than smallest input dimensionality of our DNNs (*AvastNet* at 512 KB). 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 100 malicious binaries from the test set to evaluate the attacks against each of the three DNNs.

The total number of samples we collected is comparable to that used in prior work on evading malware detection [49, 55, 88, 89].

4.2 Attack-Success Criteria

We executed the attacks for up to 200 iterations, stopping early if the binaries were misclassified at the operating point where the FPR equals 0.1%. For malicious binaries, this meant that they were misclassified as benign with a probability higher than a model-specific threshold set to achieve 0.1% FPR. This follows the threshold typically used by antivirus vendors (e.g., [54]). We also found that attack success on the same binary, given identical experiment parameters, was often stochastic. Therefore, we repeated each attack 10 times to get a reliable measure of attack success.

We compared the overall success of attacks in two ways: by the percentage of binaries that were misclassified in *at least* 1 of the 10 repeated attacks on them (**coverage**); and the overall percentage of

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

attacks that were successful across all attacked binaries (**potency**). Coverage is a measure of what percentage of binaries our attack *can* be successful on whereas potency is a measure of the how often a single attack trial succeeds. As a result of this definition, coverage will always be higher than potency.

4.3 Randomly Applied Transformations

We first evaluated whether naïvely transforming binaries at random would lead to evading the DNNs. 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. We transformed the binaries sequentially and at random. Namely, starting from the original variant, we created the next variant by transforming every function using a randomly picked transformation type that was applied at random. If any of the variants were misclassified by a DNN given the 1% FPR threshold, 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). This technique was most effective when attempting to misclassify malware as benign on *Endgame*, where four binaries evaded detection. However, for all other attempts to evade, no more than three binaries were successful.

Hence, we conclude the DNNs are robust to naïve transformations and more principled approaches are needed to mislead them.

4.4 White-Box Attacks vs. DNNs

In the white-box setting, we evaluated seven variants of our attack. One variant, to which we refer to as *IPR*, relies on the *IPR* transformations only. Three variants, *Disp*-1, *Disp*-3, and *Disp*-5, rely on the *Disp* transformations only, 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*-{1,3,5}, use the *IPR* and *Disp* transformations combined.

We set 5% as the maximum displacement budget and 200 as the maximum number of iterations, as the attacks were almost always successful with these parameters.

The results of the experiments are provided in Fig. 4 where the lighter part of the bar represents potency and the darker part represents coverage. One can immediately see that attacks using the *Disp* transformations were more successful than *IPR*. While showing some effectiveness in evading *Endgame*, *IPR* at best achieves a coverage of 52% while *Disp* of all budgets on all three models are able to cause at least 92% of binaries to be misclassified.

Moreover, *Disp*-5 achieved misclassification on all binaries except one on *AvastNet*. As one would expect, attacks with higher displacement budgets were more successful than attacks with lower displacement budgets. However, the main difference we see is in the potency of the attack, whereas the coverage only differs by a single missed binary between *Disp*-3 and *Disp*-5.

In addition to achieving higher coverage and potency, another advantage of *Disp*-based attacks over *IPR*-based ones is their time efficiency. While displacing instructions at random from within a function with n instructions has $O(n)$ time complexity, certain *IPR* transformations have $O(n^2)$ 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 $O(n^2)$ time. In practice, we found that *IPR*–based, *Disp*–based, and *IPR+Disp*–based attacks took 4424, 283, and 961 seconds on average respectively.[3]

Combining *IPR* with *Disp* achieved noticeably better results in fewer iterations than respective *Disp*-only attacks when the budget for *Disp* is low. For example, *IPR+Disp*-1 had 11% higher potency than *Disp*-1 when misleading *Endgame* to misclassify a malicious binary as benign (61% vs. 50% potency). Thus, in certain situations, *Disp* and *IPR* can be combined to fool the DNNs while increasing binaries' sizes less than *Disp* alone.

For our most performant attack, *IPR+Disp*-5, we re-executed the attacks with significantly more difficult success criteria. We changed the threshold for attack success to *MalConv* and *AvastNet*'s FNR of 0.01%. Beating this threshold means that a transformed binary must appear less malicious than the least malicious 0.1% of malware in the dataset. For *MalConv*, our potency drops from 97% to 92%, while coverage drops from 100% to 99%. For *AvastNet*, potency drops from 95% to 90% and coverage from 100% to 95%. These results demonstrate our attack's ability to evade more cautious ML detectors, even though this threshold is unlikely to be used as it would flag roughly a third of benign binaries as malware.

In Fig. 5, we averaged and plotted the classification output of the models and the resultant misclassifications of the binaries over the iterations of each attack. As shown, the majority of successful attacks that incorporate *Disp* succeeded in a single iteration, with almost all successful attacks occurring within ten iterations. We also examined the performance of the attacks as a function of the number of modifiable functions in a binary. On average, 89% of functions in a binary were modifiable. As expected, attacks were less likely to succeed when the binaries had few functions to modify (Fig. 6a). Consistent with that finding, as the number of modifiable functions (and number of functions overall) in a binary increased, the average number of iterations required for an attack to succeed decreased (Figs. 6b–6d). This trend held across different types of attacks, but was more pronounced for less successful attacks (*IPR*) than more successful ones (*Disp*), as the vast majority of the latter completed within a small number of iterations.

Finally, we compared the evasion success rates of our attack with a representative prior attack proposed by Kreuk et al. [55]. To mislead DNNs, this attack 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 which increase the binaries' sizes by 1%, 3%, and 5%. We used $Loss_{cw}$ as the loss function. As with our attacks, we executed Kreuk et al.'s attacks for up to 200 iterations, stopping sooner if misclassification occurred. We set ϵ=1, as we empirically found it leads to high evasion success.

[3]Times were computed on four machines: one with 2.2GHz AMD Opteron CPU and 128GB RAM, one with 3.4GHz Intel-i7 CPU and 24GB RAM, one with 2.2Ghz AMD Ryzen 3900X and 64GB RAM, and one with 2.7GHz Intel-i5 CPU and 24GB RAM.

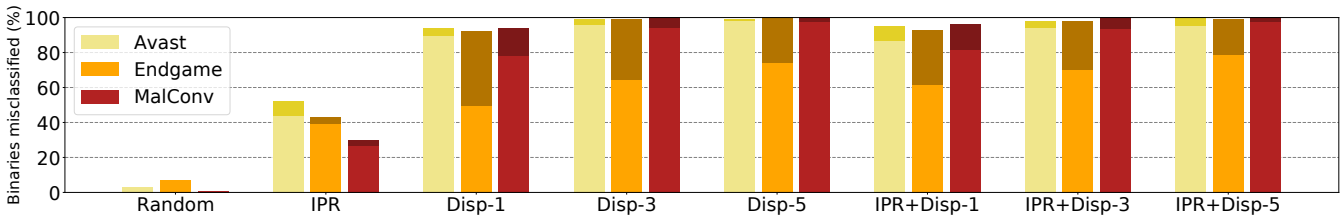


Figure 4: Attack success rates in the white-box setting. We show potency as the lighter bars and coverage as the darker bars.

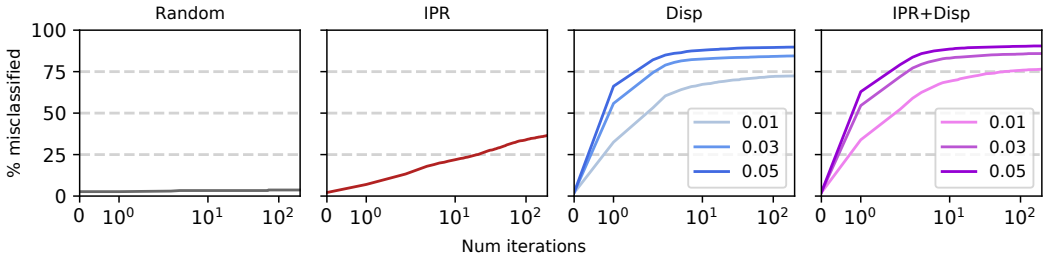
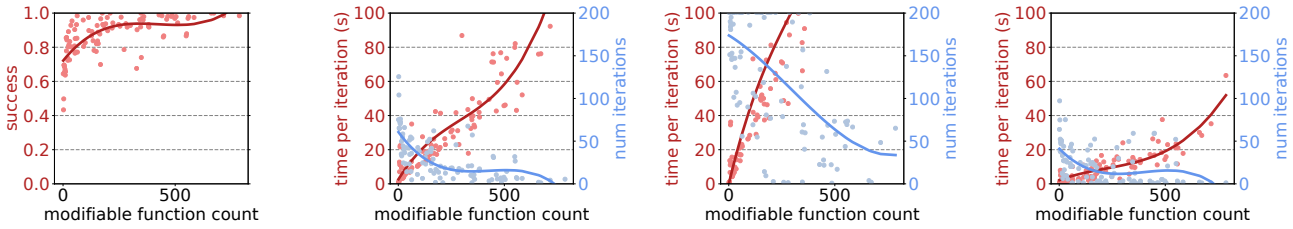


Figure 5: A contrasting view showing the potency over iteration for the whitebox attacks.



(a) Success (all attacks) **(b) Time (all attacks)** **(c) Time (*IPR*)** **(d) Time (*Disp*)**

Figure 6: As the number of modifiable functions increased, the average number of iterations to success decreased, while the time to execute an iteration increased. The lines in each plot are the best fit degree-3 polynomials.

The variants of Kreuk et al.'s attack achieved success rates comparable to our attack. *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 other attempts. Also similar to our attacks, the success rates increased as the attacks increased the binaries' sizes. However, as described in App. A, their attack is easier to defend against by sanitizing bytes (specifically, by masking with zeros) in sections that do not contain instructions.

4.5 Black-Box Attacks vs. DNNs

As explained in Sec. 3, because the DNNs' input is discrete, estimating gradient information to mislead them in a black-box setting is not possible. So, the black-box version of Alg. 1 uses hill climbing to query the DNN after each attempted transformation to decide whether to keep the transformation. Because querying the DNNs after each attempted transformation significantly increased the run time of the attacks (~30× on a machine with GeForce GTX 980 GPU), we limited our experiments to *Disp* transformations with a displacement budget of 5%. We executed the attacks up to 200 iterations, stopped early if misclassification occurred, and repeated them three times each to account for stochasticity.

The attacks were most successful against *MalConv*, achieving a coverage of 95% and potency of 92%. *AvastNet* and *Endgame* were only slightly more robust with attack coverages of 92% and 59% and potencies of 87% and 56% respectively. These results show our attack remains effective even in a black-box setting.

4.6 Commercial Anti-Viruses

To assess whether our attacks affect commercial anti-viruses, we tested the malicious transformed binaries that were misclassified by the DNNs in the white-box setting on the anti-viruses available via VirusTotal [19]—a service that aggregates the results of 68 commercial anti-viruses. Since anti-viruses often rely in part on static analysis, with increasing integration of ML, we expected that the malicious binaries generated by our attacks would be detected by fewer anti-viruses than the original binaries.

Due to contractual constraints, we were unable to perform this experiment with our previously described dataset. Thus, we resorted to using binaries taken from other sources. In this alternate dataset, we used 21,741 malicious binaries belonging to seven malware families that were published by Microsoft as part of a malware-classification competition [78]. We complemented these binaries with 19,534 benign binaries collected by installing standard packages (browsers, productivity tools, etc.) on a newly created 32-bit Windows 7 virtual machine.[4] After splitting the binaries for training (21,217), testing (9,105), and validation (10,953), we trained variants of *MalConv* and *AvastNet* that achieved 99.15% and 98.92% test accuracy, respectively. Subsequently, we collected 95 malicious binaries from VirusShare [77] that pertain to the seven malware families from the Microsoft competition. We then transformed these malicious binaries using our white-box attack to evade the DNNs we trained as well as *Endgame*, and tested how often the transformed binaries were detected by anti-viruses on VirusTotal.

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.

Random Transformations To further gauge the efficacy of our guided attack over random diversification, we used commercial anti-viruses to classify binaries that were transformed at random using the *Disp* and *IPR* transformations (as described in Sec. 4.3). We found that certain anti-viruses were susceptible to such simple evasion attempts, presumably due to using fragile detection mechanisms such as signatures. The median number of anti-viruses that correctly detected the malicious binaries decreased from 55 to 43.

Packing We tested whether anti-viruses were susceptible to evasion via packing. We used UPX [69], one of the most popular packers [80], and packed binaries using the highest compression ratios. Interestingly, packing malicious binaries was counterproductive for evading anti-viruses. Packed malicious binaries were more likely to be detected as malware—the median number of anti-viruses that correctly detected malicious binaries increased from 55 for the original binaries to 59 after packing.

Our Attacks Compared to the original malicious binaries and randomly transformed ones, the malicious binaries transformed by our attacks were detected 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. Table 3 in App. B summarizes each attack variant's effect on the number of positive detections by anti-viruses.

Because our attack should not affect any dynamic analysis (due to the desired attack property of functional invariance), these results indicate some anti-viruses may be over-reliant on static analyses and/or ML. We also highlight these results cannot only be attributed to breaking signature-based defenses, as the randomly transformed binaries (which were transformed for an equal number of iterations) would have been equally likely to evade anti-viruses as our attacks.

Furthermore, several anti-virus vendors that were misled by our attacks advertise the use of ML detectors. Evading the ML detectors of those vendors was necessary to mislead their anti-viruses. A glance at vendors' websites 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. Even more concerning, a popular and highly credible anti-virus whose vendor claims to rely on ML misclassified 85% of the malicious binaries produced by one of our attacks 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, our results support that binaries that were produced by our attacks were able to evade ML-based static detectors that are used by anti-virus vendors.

4.7 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 [52, 53, 71] 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[5] `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[6] with our attacks and checked that the tool's tests passed after the transformations. Using stress-testing tools to evaluate binary-transformation correctness 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.

5 POTENTIAL MITIGATIONS

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. Our efforts to explore mitigations, however, have met with limited success.

Due to space limitations, we defer a complete discussion of mitigations we have explored to App. C, providing only a brief summary here. We found adversarial training (e.g., [33, 57]) too expensive in this domain to be practical. Leveraging static or dynamic analysis methods to "cleanse" binaries is challenging because our attacks transform binaries' original code (vs. inserting unreachable code) and can be confounded through the insertion of opaque [23, 68]

[4]Specifically, we used the Ninite and Chocolatey (https://ninite.com/ and https://chocolatey.org/) package managers to install 179 packages.

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

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

or evasive [10] predicates. We found masking random subsets of the binary prior to classification promising as a defense in some cases, but it is far from comprehensive and likely a small obstacle to an adaptive attacker. Finally, detecting adversarial samples based on binary size or `jmp` instructions seems both difficult (our attacks increase neither substantially) and ultimately evadable. As further discussed in App. C, we thus advocate that ML-based static malware detection be augmented with methods not based on ML.

6 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 Leyla Bilge, Sandeep Bhatkar, Yufei Han, and Kevin Roundy for helpful discussions. This work was supported in part by the Multidisciplinary University Research Initiative (MURI) Cyber Deception grant under ARO award W911NF-17-1-0370; by NSF grants 1801391 and 2113345; by the National Security Agency under award H9823018D0008; by gifts from Google and Nvidia, and from Lockheed Martin and NATO through Carnegie Mellon CyLab; by a CyLab Presidential Fellowship and a NortonLifeLock Research Group Fellowship; and by a DoD National Defense Science and Engineering Graduate fellowship.

REFERENCES

[1] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. 2017. Evading machine learning malware detection. *Black Hat* (2017).
[2] H. S. Anderson and P. Roth. 2018. Ember: An Open Dataset for Training Static PE Malware Machine Learning Models. *arXiv preprint arXiv:1804.04637* (2018).
[3] Seyed Emad Armoun and Sattar Hashemi. 2012. A general paradigm for normalizing metamorphic malwares. In *Proc. FIT*.
[4] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *Proc. NDSS*.
[5] Anish Athalye and Nicholas Carlini. 2018. On the Robustness of the CVPR 2018 White-Box Adversarial Example Defenses. *arXiv:1804.03286* (2018).
[6] Anish Athalye, Nicholas Carlini, and David Wagner. 2018. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proc. ICML*.
[7] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. 2018. Synthesizing Robust Adversarial Examples. In *Proc. ICML*.
[8] Avast Software. 2020. Avast Malware detection and blocking. https://www.avast.com/en-us/technology/malware-detection-and-blocking. Accessed 12/09/2020.
[9] Shumeet Baluja and Ian Fischer. 2018. Adversarial Transformation Networks: Learning to Generate Adversarial Examples. In *Proc. AAAI*.
[10] Boaz Barak, Nir Bitansky, Ran Canetti, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. 2014. Obfuscation for evasive functions. In *Proc. TCC*.
[11] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Proc. ECML PKDD*.
[12] Fabrizio Biondi, Michael Enescu, Thomas Given-Wilson, Axel Legay, Lamine Noureddine, and Vivek Verma. 2018. Effective, Efficient, and Robust Packing Detection and Classification. *Computers and Security* (2018).
[13] Nicholas Carlini and David Wagner. 2017. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proc. AISec*.
[14] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *Proc. IEEE S&P*.
[15] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. 2018. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proc. CCS*.
[16] Mihai Christodorescu and Somesh Jha. 2004. Testing malware detectors. In *Proc. ISSTA*.
[17] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. 2005. Semantics-aware malware detection. In *Proc. IEEE S&P*.
[18] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. 2005. *Malware normalization*. Technical Report. U. Wisconsin-Madison.
[19] Chronicle. 2004–. VirusTotal. https://www.virustotal.com/. Online; accessed 17 June 2019.
[20] Cisco. 2020. ClamAV: Creating signature for ClamAV. https://www.clamav.net/documents/creating-signatures-for-clamav. Accessed 12/10/2020.
[21] Moustapha Cisse, Yossi Adi, Natalia Neverova, and Joseph Keshet. 2017. Houdini: Fooling deep structured prediction models. In *Proc. NIPS*.
[22] Jeremy M Cohen, Elan Rosenfeld, and J Zico Kolter. 2019. Certified adversarial robustness via randomized smoothing. *arXiv preprint arXiv:1902.02918* (2019).
[23] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report. The University of Auckland.
[24] Scott Coull and Christopher Gardner. 2018. 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. Online; accessed 1 July 2019.
[25] Cylance Inc. 2019. Cylance: Artificial Intelligence Based Advanced Threat Prevention. https://www.blackberry.com/us/en/cylance. Accessed 6/17/2019.
[26] Hung Dang, Yue Huang, and Ee-Chien Chang. 2017. Evading classifiers by morphing in the dark. In *Proc. CCS*.
[27] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. 2019. Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries. *arXiv preprint arXiv:1901.03583* (2019).
[28] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. 2017. Yes, machine learning can be more secure! A case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing* (2017).
[29] Logan Engstrom, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. 2017. A Rotation and a Translation Suffice: Fooling CNNs with Simple Transformations. In *Proc. NeurIPSW*.
[30] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. 2018. Robust Physical-World Attacks on Machine Learning Models. In *Proc. CVPR*.
[31] Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. 2017. Detecting Adversarial Samples from Artifacts. *arXiv:1703.00410* (2017).
[32] William Fleshman, Edward Raff, Jared Sylvester, Steven Forsyth, and Mark McLean. 2018. Non-Negative Networks Against Adversarial Attacks. *arXiv preprint arXiv:1806.06108* (2018).
[33] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *Proc. ICLR*.
[34] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. 2017. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280* (2017).
[35] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial examples for malware detection. In *Proc. ESORICS*.
[36] Claudio Guarnieri, Allessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. 2012. The Cuckoo Sandbox. https://cuckoosandbox.org/. accessed 6/21/2019.
[37] Chuan Guo, Mayank Rana, Moustapha Cisse, and Laurens van der Maaten. 2018. Countering adversarial images using input transformations. (2018).
[38] Laune C Harris and Barton P Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News* 33, 5 (2005), 63–68.
[39] Cormac Herley. 2009. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proc. NSPW*.
[40] Hex-Rays. [n.d.]. IDA: About. https://www.hex-rays.com/products/ida/. Online; accessed 13 September 2019.
[41] Weiwei Hu and Ying Tan. 2017. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983* (2017).

[42] Andrew Ilyas, Logan Engstrom, and Aleksander Madry. 2019. Prior convictions: Black-box adversarial attacks with bandits and priors. In *Proc. ICLR*.

[43] Inigo Incer, Michael Theodorides, Sadia Afroz, and David Wagner. 2018. Adversarially Robust Malware Detection Using Monotonic Classification. In *Proc. IWSPA*.

[44] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. 2019. Neurlux: Dynamic malware analysis without feature engineering. In *Proc. ACSAC*.

[45] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM–software protection for the masses. In *Proc. IWSP*.

[46] Harini Kannan, Alexey Kurakin, and Ian Goodfellow. 2018. Adversarial Logit Pairing. *arXiv preprint arXiv:1803.06373* (2018).

[47] Alex Kantchelian, JD Tygar, and Anthony D. Joseph. 2016. Evasion and Hardening of Tree Ensemble Classifiers. In *Proc. ICML*.

[48] John Kennedy, Drew Batchelor, Colin Robertson, Michael Satran, and Mark LeBLanc. 2019. PE Format. https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format. Accessed on 06-03-2019.

[49] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables. In *Proc. EUSIPCO*.

[50] J Zico Kolter and Marcus A Maloof. 2006. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research* (2006).

[51] J Zico Kolter and Eric Wong. 2018. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *Proc. ICML*.

[52] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted Code Randomization. In *Proc. IEEE S&P*.

[53] Hyungjoon Koo and Michalis Polychronakis. 2016. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proc. AsiaCCS*.

[54] Marek Krčál, Ondřej Švec, Martin Bálek, and Otakar Jašek. 2018. Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only. In *Proc. ICLRW*.

[55] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. 2018. Adversarial Examples on Discrete Sequences for Beating Whole-Binary Malware Detection. In *Proc. NeurIPSW*.

[56] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *Proc. USENIX Security*.

[57] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2017. Adversarial machine learning at scale. In *Proc. ICLR*.

[58] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *Proc. IEEE S&P*.

[59] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proc. ICML*.

[60] Mathias Lecuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. 2019. Certified robustness to adversarial examples with differential privacy. In *Proc. IEEE S&P*.

[61] Fangzhou Liao, Ming Liang, Yinpeng Dong, Tianyu Pang, Jun Zhu, and Xiaolin Hu. 2018. Defense against adversarial attacks using high-level representation guided denoiser. In *Proc. CVPR*.

[62] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards deep learning models resistant to adversarial attacks. In *Proc. ICLR*.

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

[64] Dongyu Meng and Hao Chen. 2017. MagNet: A Two-Pronged Defense against Adversarial Examples. In *Proc. CCS*.

[65] Xiaozhu Meng, Barton P Miller, and Somesh Jha. 2018. Adversarial Binaries for Authorship Identification. *arXiv preprint arXiv:1809.08316* (2018).

[66] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. 2017. On detecting adversarial perturbations. In *Proc. ICLR*.

[67] Matthew Mirman, Timon Gehr, and Martin Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *Proc. ICML*.

[68] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of static analysis for malware detection. In *Proc. ACSAC*.

[69] Markus Oberhumer, Laszlo Molnar, and John Reiser. [n.d.]. UPX: The Ultimate Packer for Executables. https://upx.github.io/. Online; accessed 1/13/2020.

[70] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *Proc. IEEE Euro S&P*.

[71] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proc. IEEE S&P*.

[72] Daniel Park, Haidar Khan, and Bulent Yener. 2019. Generation Evaluation of Adversarial Examples for Malware Obfuscation. In *Proc. ICMLA*. 1283–1290.

[73] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. 2020. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *Proc. IEEE S&P*.

[74] Yao Qin, Nicholas Carlini, Ian Goodfellow, Garrison Cottrell, and Colin Raffel. 2019. Imperceptible, Robust, and Targeted Adversarial Examples for Automatic Speech Recognition. In *Proc. ICML*.

[75] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading Authorship Attribution of Source Code using Adversarial Learning. In *Proc. USENIX Security*.

[76] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. 2018. Malware detection by eating a whole exe. In *Proc. AAAIW*.

[77] Michael Roberts. 2012. VirusShare. https://virusshare.com/. Online; accessed 18 June 2019.

[78] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. 2018. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135* (2018).

[79] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2018. Generic Black-Box End-to-End Attack Against State of the Art API Call Based Malware Classifiers. In *Proc. RAID*.

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

[81] Pouya Samangouei, Maya Kabkab, and Rama Chellappa. 2018. Defense-GAN: Protecting classifiers against adversarial attacks using generative models. In *Proc. ICLR*.

[82] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proc. ASPLOS*.

[83] Lea Schönherr, Katharina Kohls, Steffen Zeiler, Thorsten Holz, and Dorothea Kolossa. 2019. Adversarial Attacks Against Automatic Speech Recognition Systems via Psychoacoustic Hiding. In *Proc. NDSS*.

[84] Mike Sconzo. 2014. Packer YARA Ruleset. https://github.com/sooshie/packerid. Online; accessed 18 June 2019.

[85] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. 2016. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proc. CCS*.

[86] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. 2017. Adversarial Generative Nets: Neural Network Attacks on State-of-the-Art Face Recognition. *arXiv preprint arXiv:1801.00349* (2017).

[87] Vignesh Srinivasan, Arturo Marban, Klaus-Robert Müller, Wojciech Samek, and Shinichi Nakajima. 2018. Counterstrike: Defending Deep Learning Architectures Against Adversarial Samples by Langevin Dynamics with Supervised Denoising Autoencoder. *arXiv preprint arXiv:1805.12017* (2018).

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

[89] Octavian Suciu, Scott E Coull, and Jeffrey Johns. 2018. Exploring Adversarial Examples in Malware Detection. In *Proc. AAAIW*.

[90] Symantec. 2019. How does Symantec Endpoint Protection use advanced machine learning? https://techdocs.broadcom.com/us/en/symantec-security-software/endpoint-security-and-management/endpoint-protection/all/Using-policies-to-manage-security/preventing-and-handling-virus-and-spyware-attacks-v40739565-d49e172/how-does-use-advanced-machine-learning-v120625733-d47e275.html. Accessed on 01-12-2020.

[91] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *Proc. ICLR*.

[92] Peter Szor. 2005. *The Art of Computer Virus Research and Defense*. Pearson Education.

[93] TrendMicro. 2020. TrendMicro Machine Learning. https://www.trendmicro.com/vinfo/us/security/definition/machine-learning. Accessed 12/09/2020.

[94] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proc. IEEE S&P*.

[95] Vipre. 2020. Vipre Android Security. https://www.vipre.com/vipre-android-security/. Accessed 12/10/2020.

[96] VirusTotal. 2016. Packer YARA Ruleset. https://github.com/Yara-Rules/rules/tree/master/Packers. Online; accessed 18 June 2019.

[97] David Wagner and Paolo Soto. 2002. Mimicry attacks on host-based intrusion detection systems. In *Proc. CCS*.

[98] Andrew Walenstein, Rachit Mathur, Mohamed R Chouchane, and Arun Lakhotia. 2006. Normalizing metamorphic malware using term rewriting. In *Proc. SCAM*.

[99] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. Uroboros: Instrumenting stripped binaries with static reassembling. In *Proc. SANER*.

[100] Cihang Xie, Yuxin Wu, Laurens van der Maaten, Alan Yuille, and Kaiming He. 2018. Feature denoising for improving adversarial robustness. *arXiv preprint arXiv:1812.03411* (2018).

[101] Weilin Xu, David Evans, and Yanjun Qi. 2018. Feature squeezing: Detecting adversarial examples in deep neural networks. In *Proc. NDSS*.

[102] Weilin Xu, Yanjun Qi, and David Evans. 2016. Automatically evading classifiers. In *Proc. NDSS*.

[103] Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric P Xing, Laurent El Ghaoui, and Michael I Jordan. 2019. Theoretically Principled Trade-off between Robustness and Accuracy. *arXiv preprint arXiv:1901.08573* (2019).

A COMPARISON TO KREUK ET AL. AND SUCCESS AFTER SANITIZATION

While Kreuk et al.'s attack achieved success rates comparable to ours, their attack is easier to defend against. As a proof of concept, we implemented a sanitization method to defend against the attack using our alternate dataset described in Sec. 4.6. The method finds all the sections in a binary 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 add functional instructions to the binaries, the defense masks the adversarial bytes that the attack introduces. Consequently, the evasion success rates of the attack drop significantly. In fact, except for when attempting to mislead the *Endgame* DNN with malicious binaries, the success rates of the *Kreuk* attacks dropped below 15%. This defense had little-to-no effect on our attacks, however: e.g., *Disp*-5 still achieved 92% and 100% success rates against *MalConv* for malicious and benign binaries, respectively. Moreover, the classification accuracy remained high both for malicious (99%) and benign (93%) binaries after the defense. Fig. 7 in App. A presents the full results of the impact of sanitization on attacks' success on the *Kaggle* dataset.

Fig. 7 shows the success rates of attacks when sanitizing bytes in sections that do not include instructions. In particular, we replaced byte values in such sections with zeros, as described in Sec. 4.4. Our attacks maintained high success rates after sanitization (e.g., >90% for *Disp*-5), whereas the success rates of the *Kreuk* attacks dropped below 15% in most cases.

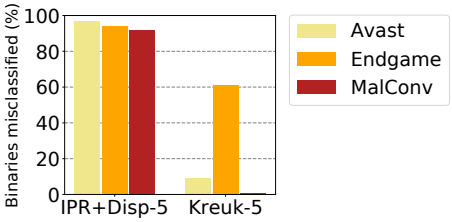


Figure 7: Attacks' success rates (measured by the percentage of misclassified binaries) in the white-box setting when masking out bytes in sections that do not include instructions before classification.

B OUR ATTACKS' TRANSFERABILITY TO COMMERCIAL ANTI-VIRUSES

Table 3 summarizes the effect of different attack variants on the number of positive detections (i.e., classification of binaries as malicious) by the anti-viruses featured on VirusTotal. Sec. 4.6 describes the experiment and explains the results.

C POTENTIAL MITIGATIONS

In this appendix we summarize our efforts to provide mitigations for our attacks.

C.1 Prior Defenses

We considered several prior defenses to mitigate our attacks, but, unfortunately, most showed little promise. For instance, adversarial training (e.g., [33, 57]) is currently infeasible, as the attacks are computationally expensive. Depending on the attack variant, it took an average of 283 to 4424 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., [51, 67]) are even more computationally expensive than adversarial training. Moreover, those defenses are restricted to adversarial perturbations that, unlike the ones produced by our attacks, have small L_∞- and L_2-norms. Prior defenses that transform the input before classification (e.g., via quantization [101]) 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 [49, 55, 89], 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. 4.4, or using other methods [56]). 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, 98] is a defense that initially 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* [23, 68]. 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 [68]. 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) [10]. In this case,

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**

Table 3: The median number of VirusTotal anti-viruses that positively detected (i.e., as malicious) malicious 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 binaries is 55. Cases in which the change in the number of detections is statistically significant are in bold.

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 produce the form with the lowest lexicographic representation (where the alphabet contains all possible 256 byte values) prevented *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 $x_i \in [x]$. App. D 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 an effort to reduce the lexicographic representation after every iteration. We found that executing the algorithm for ten iterations was 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. 4 have shown, generating functionally equivalent variants of binaries via random transformations results in correct classifications almost all of the time. Normalization of binaries deterministically led to specific variants that were correctly classified with high likelihood.

C.2 Masking Random Instructions

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 on *Kaggle* and found that it was effective at mitigating attacks. For example, when masking 25% of the bytes pertaining to instructions, the success rates of the attack decreased 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.

C.3 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 [53, 71].

C.4 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 for 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, 85, 86]. 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 [25].

D 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. 8 depicts an example of replacing one instruction with an equivalent one via *Eqv* to decrease the lexicographic order of code.

```
sub eax, -0x20      (83e8e0)
test ebx, ebx       (85db)
```

(a)

```
add eax, 0x20       (83c020)
or ebx, ebx         (09db)
```

(b)

Figure 8: An example of normalizing code via *Eqv*. The original code (a) is transformed via *Eqv* (b) to decrease the lexicographic order.

Unfortunately, as shown in Fig. 9, 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.

```
push edx            (52)
push ebx            (53)
mov dh, 0x4         (b604)
mov bh, 0x3         (b703)
pop ebx             (5b)
pop edx             (5a)
```

(a)

```
push edx            (52)
push ebx            (53)
mov bh, 0x4         (b704)
mov dh, 0x3         (b603)
pop ebx             (5b)
pop edx             (5a)
```

(b)

```
push edx            (52)
push ebx            (53)
mov bh, 0x3         (b703)
mov dh, 0x4         (b604)
pop ebx             (5b)
pop edx             (5a)
```

(c)

```
push edx            (52)
push ebx            (53)
mov dh, 0x3         (b603)
mov bh, 0x4         (b704)
pop ebx             (5b)
pop edx             (5a)
```

(d)

Figure 9: 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.

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. The algorithm then proceeds to apply each of the individual normalization operations to decrease the lexicographic representation of the binary while self-supervising the normalization process. Specifically, the algorithm keeps track of the last iteration an operation decreased the binary's representation. 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 and the normalization process restarts.

When $niters \rightarrow \infty$ (i.e., the number of iterations is large enough), This algorithm 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, which we found to be sufficient to successfully mitigate the majority of attacks.