Adversarial Training for Raw-Binary Malware Classifiers

Keane Lucas
Carnegie Mellon University

Samruddhi Pai
Carnegie Mellon University

Weiran Lin
Carnegie Mellon University

Lujo Bauer
Carnegie Mellon University

Michael K. Reiter
Duke University

Mahmood Sharif
Tel Aviv University

Abstract

Machine learning (ML) models have shown promise in classifying raw executable files (binaries) as malicious or benign with high accuracy. This has led to the increasing influence of ML-based classification methods in academic and real-world malware detection, a critical tool in cybersecurity. However, previous work provoked caution by creating variants of malicious binaries, referred to as *adversarial examples*, that are transformed in a functionality-preserving way to evade detection. In this work, we investigate the effectiveness of using adversarial training methods to create malware-classification models that are more robust to some state-of-the-art attacks. To train our most robust models, we significantly increase the efficiency and scale of creating adversarial examples to make adversarial training practical, which has not been done before in raw-binary malware detectors. We then analyze the effects of varying the length of adversarial training, as well as analyze the effects of training with various types of attacks. We find that data augmentation does not deter state-of-the-art attacks, but that using a generic gradient-guided method, used in other discrete domains, does improve robustness. We also show that in most cases, models can be made more robust to malware-domain attacks by adversarially training them with lower-effort versions of the same attack. In the best case, we reduce one state-of-the-art attack's success rate from 90% to 5%. We also find that training with some types of attacks can increase robustness to other types of attacks. Finally, we discuss insights gained from our results, and how they can be used to more effectively train robust malware detectors.

1 Introduction

Determining if a piece of software is malicious is an important cybersecurity task, which is currently accomplished via an ensemble of automated analyses. A recent addition, due to their high accuracy, is machine learning (ML) algorithms trained to infer maliciousness from the compiled bytes of an executable binary [32, 46, 47]. However, it has become evident that these trained malware classifiers can be fooled by *adversarial examples* [2, 18, 29, 33, 36, 53].

Finding adversarial examples for neural networks, along with efforts to make neural networks more robust and trustworthy, has become a well-documented cat-and-mouse game, where increasingly sophisticated attacks [5, 7, 9, 13, 23, 41, 55] (i.e., an evasive binary, adversarial example) lead to increasingly sophisticated defenses [13, 23, 37, 50, 60] and vice versa. Defending against such attacks is becoming increasingly important as ML-based systems take on more complex and safety-critical tasks (e.g., driving cars).

While there has been promising progress in defending against [36] raw-binary adversarial examples that modify non-executable portions of binaries [33], recent attacks remain effective at evading otherwise accurate malware classifiers [36]. In this work, we experiment with adversarial training strategies to produce raw-binary malware classifiers that resist three state-of-the-art evasion methods, referred to as *In-Place Replacement* (*IPR*) [36], *Displacement* (*Disp*) [36], and *Kreuk* [33]. Our strategies include augmenting the training data by (1) applying random (i.e., unguided) transformations of the same type used in attacks (e.g., *IPR* [42] and *Disp* [31]); (2) using modified versions of *IPR* and *Disp* adversarial examples [36]; (3) directly training with *Kreuk* adversarial examples; and (4) using a perturbation method inspired by similar perturbations from other discrete domains [61, 62]. Furthermore, we study the effects on robustness as we vary the number of batches of adversarial examples used in training, the number of iterations for which transformations and attacks are executed during training, and the percentage of bytes (i.e., budget) a transformation or attack can modify during training.

To make these strategies possible, we address several challenges. Training on *IPR* and *Disp* adversarial examples had not been attempted before, as using these transformations for evasion is new [36] and has previously been too computationally expensive (see Sec. 3). Also, to be eligible for transformation, *IPR* and *Disp* require that each candidate binary be unpacked and previously disassembled. To enable these experiments, we detail several improvements and optimizations, in-

cluding vastly increasing the number of binaries eligible to be turned into adversarial examples (200 → 126,009); code-level optimizations to the adversarial-example-generation code to speed up each individual attack (Sec. 3.3); a distributed system of over 140 workers across thirteen servers to produce adversarial examples in parallel (Sec. 3.1); and training with less effective but computationally cheaper versions (Sec. 3.3) of the originally proposed attacks [33, 36].

Our findings include the following:

- Adversarial training using data-augmentation techniques is ineffective in defending against adversarial examples in the domain of raw-binary classification.
- In contrast, adversarial training using state-of-the-art attacks can yield robust models (90% → 5%, 26% → 6%, and 84% → 30% attack success rate for *Disp*-, *IPR*-, *Kreuk*-based attacks, respectively) when evaluated against the same type of attack used during training. We also find that these models can reduce success for attacks they were not trained on by up to 65%. We investigate how changing parameters of the attacks used in training (e.g., number of attack iterations, permitted file size increase, type of attack) affects the robustness of the resulting model.
- We construct, train with, and evaluate a perturbation strategy inspired by other discrete domains [61, 62], referred to as *Greedy*, which modifies the discrete bytes of the input binary without regard to the underlying structure, to insert evasive bytes in the most *important* locations. We find that this approach is effective in inducing some robustness against all attacks.
- Finally, we adversarially train a classifier with adversarial examples created by three different attacks, and find that this classifier exhibits close to best-case robustness against every attack type.

2 Background and Related Work

Our work builds on research using deep neural networks (DNNs) for malware detection from raw binaries [32, 46], along with work generating adversarial examples in the malware domain [18, 19, 33, 36, 52] and other domains [5, 7, 9, 23, 41, 55]. Next, we cover the background of these topics, and describe the state of the art in creating adversarial examples for malware detectors and defending against them.

2.1 DNNs for Malware Detection

Various feature types have been proposed for malware detection. Expert-designed features (e.g., [3, 4, 20, 22, 26, 30, 49]) include vectors describing imported libraries, library or API function calls, network addresses contacted, number of system calls, byte entropy, whether specific strings are present, etc. Hand-crafting these features is time-consuming, but their use in DNNs has comparable results to DNNs that learn directly from raw bytes [3, 32, 36, 46]. In this paper, we only consider DNNs that infer maliciousness directly from raw bytes.

MalConv [46] and *AvastNet* [32] are examples of DNN architectures that use raw-bytes as input. These architectures, used in many related works [18, 21, 36, 52], achieved 98.5% and 98.6% test accuracies respectively in discriminating unseen executables [36].

Real-world malware detection consists of an ensemble of detection mechanisms, not just including the static analysis these DNNs perform [56]. Other analyses include dynamic analyses of files to infer maliciousness based on their execution behavior. However, if an adversary is trying to evade this detection, they will need to evade each of these components, including static analysis, which often serves as the first line of defense [56]. Hence, our work's focus on mitigating evasion attacks against static analysis remains critical.

2.2 Attacking DNNs

Some popular methods of fooling DNN classifiers use the gradients of the input with respect to the classification *loss* to perturb a benign input toward greater loss until the DNN misclassifies the perturbed input [23, 37]. This general method succeeds in creating what is called an *adversarial example*: a perturbed datapoint that the DNN classifies differently.

More formally, we define a *perturbation set* $P(x)$ as some neighborhood of a datapoint x in which all members are, to a human, in the same class (e.g., perturbations such as adding static or small rotations to an image or functionality preserving transformations on a compiled binary as described in Sec. 2.3). Given an otherwise accurate classifier f, it is generally easy to find $x' \in P(x)$ where $f(x) \neq f(x')$, i.e., the classifier thinks x' belongs to a different class than x.

The following optimization function describes optimal classifier parameters θ given a dataset of inputs X and labels Y:

$$\min_{\theta} \sum_{x \in X, y \in Y} L(f(x, \theta), y) \tag{1}$$

where L is the loss function. The following optimization function then describes the adversaries' goal when creating an adversarial example x' by searching the maximum possible loss within a perturbation set $P(x)$:

$$\max_{x' \in P(x)} L(f(x', \theta), y) \tag{2}$$

Previously established attacks that use the classifier's first-order gradients and gradient ascent to directly approach high loss regions include Fast Gradient Sign Method (FGSM) [23, 35] and Projected Gradient Descent (PGD) [37].

2.3 Attacking DNNs for Malware Detection

Crafting attacks against malware-detection DNNs includes unique constraints on the allowable perturbation set $P(x)$. In the case that a DNN is using expert-designed features, prior work crafting adversarial perturbations to these features [22] must ensure the modifications are plausible. For example, the file size could be tweaked to be larger, the presence of strings can be changed, and an adversary could choose to import libraries or make syscalls that are unnecessary. However, this work focuses on creating adversarial examples by modifying the raw bytes of compiled binaries.

Attacking DNNs That Use Raw-binaries In creating adversarial examples out of executable binaries (i.e. in the *problem space* [44]), an attacker must first define what is an allowable adversarial example in this domain. In both this work and related work, it is presumed that an attacker will only perturb the original binary in ways that preserve its original behavior when executed [11, 12, 19, 33, 36]. If a single byte is changed without taking careful account of all the other bytes, the entire binary would likely be rendered useless.

The input space for binaries consists of a variable length sequence of bytes, each with a value in $[0, 255]$. For *MalConv*, with an input size of 2MB, this corresponds to a feature space of 2×2^{20} *categorical* features. We also cannot naïvely follow the target model's loss gradients to produce adversarial examples, as we need to ensure that any transformations result in a valid binary. Prior work refers to this as the *inverse feature-mapping problem* [44]. Previous research has overcome these issues and created adversarial examples by modifying or adding bytes to non-executable regions of binaries, resulting in the attacks introduced by Kreuk et al. (referred to as *Kreuk* attacks [33]), and other attacks [2, 19, 29, 53]. Other work focused on modifying the portable executable (PE) structure [2, 18, 19], or by leveraging functionality preserving transformations to the assembly code [36] such as In-Place Randomization (*IPR*) [42] and Displacement (*Disp*) [31].

***Kreuk* Attacks** *Kreuk* attacks leverage the structure of PE files to modify or append bytes in non-executable areas [33]. To select the most evasive bytes, *Kreuk* uses FGSM [23, 35] on the embedded representation of the binary to perturb the embedding to increase the target model's loss. Then, the perturbed embedding is mapped to the bytes that are closest to it in Euclidean distance. Since changing any byte could affect the gradient of another byte, *Kreuk* uses multiple iterations of FGSM. This attack was shown to successfully transform 99% of tested binaries to evade correct classification [33]. Later work showed that *Kreuk* can be mitigated by masking out non-executable bytes before the binary is classified [36]. Still, an attacker could overcome this defense by obscuring which bytes are executable.

***IPR* and *Disp* Attacks** Using transformations developed for code diversification [31, 42], prior work applied only those code transformations that resulted in changes in the binary embedding that had a positive cosine similarity with the target model's pre-computed embedding loss gradient. While this technique proved effective at creating evasive adversarial examples, such attacks are computationally expensive [36]. Generally, this high computation cost comes from the sequential, scan-and-modify nature of the process. For each binary, the attacks consider each function one at a time, generating functionality-preserving transformations and keeping or generating more transformations based on the aforementioned gradient cosine. For large binaries or binaries with complex functions, this process can take very long (e.g., an average of 4424 seconds for *IPR* [36]). For more detail, see App. C.

Problem-Space Constraints In the context of the *problem-space constraints* formalized by Pierazzi et al. [44], the available transformation for *Kreuk* attacks is byte addition at the end of the file. For *Disp* attacks, the available transformations include editing and adding bytes to displace instructions and filling the resultant gaps with semantic nops. For both *Kreuk* and *Disp* attacks, these byte additions must not affect the functionality of the binary when executed. For both of these attacks, constraints on *plausibility* and *robustness to preprocessing* are not explicitly specified. However, prior work has constrained these attacks by a specified *budget*, the percentage by which the file size may increase [36]. For *IPR* attacks, the available transformations are limited to editing bytes via four functionality-preserving transformations. However, *IPR* is not constrained by a budget, but instead only by the fact that the transformations may not be able to change some bytes and preserve the binary's functionality.

2.4 Defending Raw-Binary Malware Detectors

While many evasion defenses have been explored in other domains, defenses in the malware-detection space remain sparse. For attacks that adversarially manipulate bytes in non-executable parts of a binary [29, 33, 53], zeroing out all non-executable bytes before classification significantly reduces attack success while maintaining high benign accuracy [36].

Prior work also found that *normalizing* a binary using *IPR* transformations to the lowest lexicographic representation using an alphabet consisting of the 256 possible byte values effectively defends against *IPR* [36]. However, such normalization requires access to all transformations used to produce a potentially adversarial binary. Furthermore, such normalization is computationally expensive as it requires, like the attack, disassembling the binary and applying many iterations of transformations. Considering that this overhead would be incurred at run time, prior to every classification attempt, this defense may not be viable in practice.

For *Disp*, prior work found that removing the adversarial changes would not work [36] because of the possible use of *opaque* [16, 39] or *evasive predicates* [6], instead of semantic nops, to fill in the gap left by displacement. Opaque predicates are sets of instructions whose functionality is known to the attacker (e.g., a branch condition that the attacker creates to always return true) but is difficult for the defender to infer statically, while evasive predicates are similar instructions that add randomness but will almost always perform one action known to the attacker (e.g., a branch condition that returns true 99.9% of the time). A defender would find it difficult to identify opaque and evasive predicates [36].

However, prior work did find that randomly masking a percentage of executable bytes in a binary before classification does provide some robustness to *IPR* and *Disp*, but at a cost to natural accuracy [36]. We implemented this defense and found that, at 25% masking (as suggested in prior work), the evasiveness of *Disp* attacks was reduced (86% $\rightarrow$ 65%), but at a steep cost (96% $\rightarrow$ 56%) to the True Positive Rate (TPR) at 0.1% False Positive Rate (FPR), which is a standard metric for assessing the performance of malware detectors [3, 32, 36].

Given the weaknesses of current defenses in raw-binary classification, we turn to adversarial training, a defense commonly used in other domains [23, 27, 28, 34, 37, 50, 60].

2.5 Adversarial Training

As we introduced in Sec. 2.2 and Sec. 2.3, adversaries can generate adversarial examples by using the gradients of the input with respect to the classification loss either directly [5, 7, 9, 23, 41, 55] or by ensuring positive cosine similarity [36].

A common defense against adversarial examples in the image domain is *adversarial training*: training the DNN to correctly classify the adversarial examples that are generated to target the current version of the model, along with benign training data [23, 37, 50, 60]. Adversarial training aims to find parameters θ that minimizes a weighted sum of the model's loss for classifying benign examples and the model's loss for classifying adversarial examples:

$$\min_{\theta} \sum_{x \in X, y \in Y} \lambda * (\max_{x' \in P(x)} L(f(x', \theta), y)) + (1 - \lambda) * L(f(x, \theta), y) \tag{3}$$

where $\lambda \in [0, 1]$ is a weighting parameter commonly set as 0.5 [23]. It has been shown that adversarial training substantially enhances DNN robustness to adversarial examples. Early works in adversarial training focused on images with the perturbation set as the ℓ_p ball on pixel RGB values [25, 59]. The intuition is that by restricting the norm sufficiently, all images within the perturbation set will, as perceived by a person, be in the same class. These works solved Eqn. 2 by using techniques such as FGSM and PGD [35].

However, these defense techniques rely on the availability of explicit first-order gradients in the model and a continuous input space where small perturbations are unlikely to produce a human-perceptible difference. In malware classification, as noted in Sec. 2.3, these assumptions are not applicable.

Adversarial training also requires creation of many adversarial examples to train with. For this reason, adversarial training in the raw-binary classification domain has been too computationally expensive. As such, previous attempts at adversarial training for malware classifiers have been constrained to classifiers and perturbations that operate on handcrafted binary features [22, 57], not the raw binaries themselves. Our research solves these technical challenges.

2.6 Threat Model

Following the standard threat modeling framework outlined in prior work [8, 44], we outline our assumptions about the attacker's goals, knowledge, and capabilities. We assume the attacker is trying to cause an *integrity violation* by causing a malware classifier to incorrectly classify a piece of malware as benign. The attacker has perfect knowledge of the malware classifier (*whitebox*), and can alter whichever bytes they want in the input binary using known functionality-preserving binary transformations (*Disp*, *IPR*, or *Kreuk*) [33, 36]. The attacker cannot modify or influence the trained model in any way except by changing the input.

Accordingly, all success scores in Sec. 4 are from whitebox attacks executed in the *problem space* [44], where the attacks are directly modifying the input binaries. The transformations we consider [33, 36] only work on unpacked 32-bit PEs. For this reason we also assume that the adversarial examples we defend against fit these requirements.

Finally, we emphasize that most attacker constraints in our threat model—such as the binaries being unpacked, 32-bit, and disassembled—are due to the constraints of the state-of-the-art attacks that our work endeavors to defend against. In practical use, the DNNs we train are only used for static analysis, and they could be paired with a good unpacker and used in combination with other defenses. This threat model mirrors the one used by the attacks we defend against [33, 36], lending credibility to the defenses we propose. As the general method we use, adversarial training is used in many domains, and we expect lessons from our investigation will also be relevant to situations in which these constraints do not apply.

3 Technical Approach

In this section, we present our technical approach. We describe our scaling infrastructure (Sec. 3.1), dataset and models (Sec. 3.2), and the methods to create the raw-binary adversarial examples that we experiment with (Sec. 3.3).

3.1 Experimental Infrastructure

Even after several optimizations to speed up creating adversarial examples (see Sec. 3.3), creating a single adversarial

VTFeed	Train	Val.	Test
Benign	111,258	13,961	13,926
Attack Eligible Benign	31,421	3,866	3,855
Malicious	111,395	13,870	13,906
Attack Eligible Malicious	69,743	8,548	8,576

Table 1: *VTFeed* dataset with more attack eligible binaries.

	Accuracy			TPR @
	Train	Val.	Test	0.1% FPR
MalConv	99.97%	98.67%	98.53%	96.08%

Table 2: Performance of the original *MalConv* architecture pre-trained on the *VTFeed* dataset (training partition) [36].

example could take between 30 seconds and a few hours. Consequently, a sequential training process could take months, as each training batch requires multiple adversarial examples. To make adversarial training practical, we built a distributed system to create multiple adversarial examples in parallel via multiple workers distributed among several servers. The system is depicted in Fig. 6 in App. C. Each worker in the system continuously checks for *jobs*, each consisting of a binary, parameters for creating an adversarial example, and information for where to send it. Upon job arrival, the worker updates its own version of the target model weights to the latest version. Next, the worker generates the adversarial example according to the parameters it received. On completion, the adversarial binary is sent to a central training process that collects them into batches and updates the target model weights. We used over 140 workers distributed among thirteen servers. While this system did not speed up individual attacks, the parallelism decreased the time it takes to complete an adversarial training epoch from several months to between one and ten days.

3.2 Dataset and Model

We used the *VTFeed* 32-bit Portable Executable (PE) dataset, obtained from the authors of previous work [36], which contains 278,316 binaries up to 5 MB in size, partitioned into training, validation, and test sets, with roughly balanced numbers of malicious and benign binaries.

Prior work restricted binaries eligible for attack to have a file size less than or equal to 512 KB [36]. We remove this restriction to increase the number of binaries that can be transformed. Otherwise, our requirements match prior work: each binary must be an unpacked, 32-bit PE file that is classified correctly with *high confidence* by a standard malware detector (*MalConv*) and is able to be disassembled by IDA Pro [24]. In particular, we determine that a binary labeled as malicious (resp. benign) is classified with high confidence if it is classified as malicious with confidence higher (resp. lower) than the 0.1% FPR (resp. 0.1% False Negative Rate (FNR)) threshold of the target classifier. Similar to prior work [36], we select binaries correctly classified with high confidence to ensure that the transformations we make to the binary are the cause of misclassification. The sizes of different partitions and the numbers of eligible binaries are shown in Table 1.

Prior work disassembled and transformed 200 binaries to show the effectiveness of their attacks [36]. In contrast, we disassembled 126,009 binaries across the training, validation, and test groups of the *VTFeed* dataset, and transformed each of them many times in order to adversarially train our models.

We adversarially trained raw-byte malware detectors based on the *MalConv* [46] and *AvastNet* architectures (see App. B for results on *AvastNet*). To ensure that any reduction in attack success is due to our adversarial training and not due to a change in dataset or model architecture, we started each training process with the model from prior work pre-trained on the *VTFeed* dataset used as a target for the state-of-the-art attacks [36]. This model's accuracy is shown in Table 2. Although a newer version of the *MalConv* architecture has been recently proposed [47], its primary difference from the model we used is better memory efficiency.

3.3 Adversarial Examples for Raw-Binary Malware Detectors

As mentioned in Sec. 2.3, adversarial examples in the raw-binary domain have unique constraints because of the nature of the input space (categorical raw bytes). We first describe how we implement data augmentation using functionality-preserving transformations (*IPR* and *Disp*). We then describe adaptations of gradient-guided versions of *IPR* and *Disp* (previously demonstrated as state-of-the-art adversarial examples [36]) and the parameters we vary during adversarial training to induce robustness while maintaining high TPR on the original data. Finally, we describe our implementation of *Kreuk* attacks and the *Greedy* perturbation.

Unguided Transformations In other domains, data augmentation is often used to improve DNN robustness to common or expected perturbations [17, 51]. In the image domain, augmentation could include random cropping, rotating, and brightening. In essence, augmentation randomly applies realistic perturbations that do not change the class to make the model more robust to those perturbations. Thus, it is reasonable to expect that, in the malware-detection domain, data augmentation using the functionality preserving transformations used by state-of-the-art adversarial examples would provide more resilience to those adversarial examples. To investigate, we train DNNs with binaries, with either *IPR* or *Disp* transformations applied to them randomly (i.e., unguided), using the setup as described in Sec. 4.1. The only difference between

these transformations and the *IPR* and *Disp* attacks we seek to defend against is that the attacks are guided and only apply the transformations that follow the targeted model's loss gradient. We execute each transformation up to 5 iterations (a cap shown to reduce attack performance when using guided transformations in other experiments). We then evaluate the trained models as described in Sec. 4.1.

To simulate an "unguided" analog to *Kreuk* attacks, we append randomly chosen bytes (not optimized to be evasive), up to a certain budget, to the end of the binary and train with them. The only difference between guided and unguided *Kreuk* adversarial examples is that guided versions optimize the appended bytes to be more evasive (see Sec. 2.3).

***IPR* and *Disp* Attacks** As mentioned in Sec. 3.1, adversarial training requires many adversarial examples. In prior work, *IPR* attacks were time consuming [36], taking an average of $4,424$ seconds to complete per binary. A full epoch of training using our dataset requires 55K batches, each of four adversarial examples. Cumulatively, these attacks would take $4,424$ seconds $\times$ 4 adversarial examples $\times$ $55,000$ batches ≈ 31 years. However, in prior work these attacks were given a cap of 200 iterations to succeed. We can first reduce the time by capping attacks to 1, 3, 5, or 10 iterations. We chose these numbers since the earliest iterations are, by far, the most evasive; most *Disp* attacks succeeded in a single iteration and about a third of successful *IPR* attacks within 5 iterations [36]. We can then use our previously described parallelization techniques to compute over 140 attacks at a time.

We also discovered that creating adversarial examples from our dataset was taking longer than reported in prior work. Investigation revealed this was because our binaries were up to 5 MB in size, rather than the 512 KB in prior work. We flagged attack instances that took much longer than average, and identified corner cases responsible for attacks that took days. By addressing several of these corner cases, and reducing unnecessary computation in the attack method, we sped up *IPR* attacks (capped at 5 iterations) from an average of $2,116$ to 132 seconds ($16\times$ faster) and *Disp* attacks (of up to 5 iterations) from an average of 233 to 63 seconds.

Finally, we discovered that data structures relied on by *IPR* and *Disp* had duplicate references to the same executable bytes in multiple objects. This occasionally led to small numbers of bytes being transformed more than once per iteration, possibly breaking the functionality of some binaries. We applied a fix, and verified through evaluation attacks on the original model that there was no noticeable decrease in attack success from either this fix or our code optimizations.

***Kreuk* Attacks** We implemented the end-of-file injection version of the *Kreuk* attack (see Sec. 2.3), which inserts evasive bytes in a new section, appended to the end of the file. Matching our experiments with *Disp*, we limited the file-size increase by a pre-defined percentage (1, 3, or 5%). Executing *Kreuk* was faster than *IPR* and *Disp* attacks.[1] Hence, we were able to execute the FGSM embedding perturbation and byte remapping process until the attack succeeded (as defined in Sec. 4.1), until the deterministic iterative process cycled to a previously found byte configuration (forming an infinite loop), or a maximum of 200 iterations.

3.4 *Greedy* Perturbation

Inspired by the enhanced robustness against all three attacks attained by *Disp*- and *Kreuk*-based training (Fig. 3d and Fig. 3f), we hypothesized that the benefit stemmed from training on examples containing bytes that had been explicitly optimized to be evasive, as happens in *Kreuk* and *Disp*.

For example, *Kreuk* attacks, as described in Sec. 3.3, can set multiple bytes to values identified as the most evasive, while *Disp* can insert the most evasive combination of semantic nop instructions. Both of these methods result in a set of bytes that are explicitly optimized to have an outsized effect on the binary's classification. If we could efficiently train a model to pay less attention to bytes that are optimized to have an outsized effect, this model may be more robust to *Kreuk* and *Disp* attacks. Moreover, with only this training goal in mind, we could avoid the expensive constraint of only creating training examples that perserve binary functionality.

We leveraged prior work from non-malware domains that create adversarial examples for discrete spaces [61, 62] to create the *Greedy* perturbation for binaries. Like prior work [61], we use Integrated Gradients (IG) [54] to find the most *important* features (i.e., bytes) that most influence a given binary's classification (benign or malicious). IG requires a baseline input, for which we used a binary with file size 0 (represented to the DNN as a sample made up of all padding bytes). Also, because IG provides an importance score for each channel in each byte's embedding (an 8-channel vector for *MalConv*), we chose to estimate the importance of each byte by summing its embedding channels' scores. We then select the most important features (i.e., bytes) up to a given percentage of the file (e.g., 1, 3, or 5%), and assign the most evasive value to the features (found using FGSM on the embedding), following the procedure outlined in prior work [62]. This process is similar to the *Kreuk* attack, except that the *Greedy* perturbation—we call it a *perturbation* because the result is not a well-formed executable binary and so is not an *attack*—can change any byte to any value, up to a certain percentage.

4 Evaluation

In this section, we report on the performance of models we adversarially trained under different configurations (e.g., methods for generating adversarial examples), eventually showing

[1] Average of 1–5 seconds, depending on the attack's file-size budget

how adversarial training can markedly limit the effectiveness of *Disp*-, *IPR*-, and *Kreuk*-based attacks.

4.1 Evaluation Setup

This section first describes the adversarial training parameters varied to produce robust models, followed by our method of evaluating and contrasting these models.

Varying the File-Size Budget of *Disp* and *Kreuk* Attacks In prior work, *Disp* attacks were the most successful, with success rates increasing as the displacement budget increased (i.e., the percent increase in file size allowed) [36]. In this work, to specify if an attack uses non-default budgets, it will be referred to as <attack>-<budget> (e.g., *Disp*-0.01). Otherwise, *Disp* and *Kreuk* refers to attacks with all three budgets (0.01, 0.03, 0.05). These three budgets mirror those used in prior work, where they were sufficient for nearly 100% of binaries to evade detection [36].

These different budgets raise the question on whether adversarial training with examples of lower budget will provide protection from *Disp* or *Kreuk* attacks of a higher budget. To answer this question, we train and compare three DNNs for each attack (six total); one DNN trained on only low budget versions of these attacks (i.e *Disp*-0.01 or *Kreuk*-0.01 adversarial examples), one DNN trained with 0.01 and 0.03 budget adversarial examples, and one with 0.01, 0.03, and 0.05 budget adversarial examples. To keep the comparison fair, the *Disp* training adversarial examples were all executed up to 3 iterations. *Kreuk* attacks (training and evaluation) were executed as discussed in Sec. 3.3. Like all other models, these trained models were evaluated with *Disp*-0.01, *Disp*-0.03, *Disp*-0.05, *Kreuk*-0.01, *Kreuk*-0.03, *Kreuk*-0.05, and *IPR* attacks.

Varying the Number of Attack Iterations In previous work, it was found that when transforming a binary to evade detection, the first several attack iterations (where an 'iteration' is a single pass of attempted transformations through a binary) often resulted in the largest changes in maliciousness. In other words, while an attack may be given up to 200 iterations to generate evasive samples, often the first 3–5 iterations were sufficient, especially for *Disp* [36]. Therefore, one of our modifications to the *IPR* and *Disp* attacks demonstrated in prior work was to only run the adversarial example generation process for 1–10 iterations and use the resultant example to train with, regardless of its success in evading the target model. By capping the process at these lower iterations we avoid the costly potential of spending 20–200 times as long for only marginally more evasive adversarial examples.

To examine the tradeoff between faster example generation and adversarial robustness, we compare eight separate training configurations: four with *IPR* adversarial examples capped at 1, 3, 5, and 10 iterations and four with *Disp* adversarial examples capped at 1, 3, 5, and 10 iterations. One of these configurations, which uses *Disp* adversarial examples to 3 iterations, was mentioned in the previous section about varying *Disp* budgets, meaning that we are now considering a total of $6+7=13$ adversarial training configurations.

Varying the Number of Training Batches We train with each of the previously mentioned 13 configurations up to 55K batches (equal to one epoch of the training dataset) and take checkpoints at 10K, 25K, and 55K batches for each training configuration. We capped our adversarial training at one epoch because of the high time cost (discussed in Sec. 3.3) and the diminishing marginal increase in robustness and natural TPR as training continued, which suggested further training would not provide more insight. We evaluate at 10K and 25K batches ($\sim$20% and $\sim$50% of the training data) to observe early- and mid-training behavior, such as an increase in robustness and decrease in natural accuracy. This results in $13 \times 3 = 39$ adversarially trained models that are then evaluated by the suite of attacks used in prior work [33, 36].

Comparing Adversarially Trained Models In adversarially training raw-binary malware detectors, we emulated prior work in other domains as described in Sec. 2.5 [23]. We chose a batch size of eight, with half the batch size consisting of the original examples from the training partition of the *VTFeed* dataset, and the other half being adversarial examples created from sampled binaries from the portion of the training partition that is eligible for attack.

To compare different configurations of adversarial training in our domain, we evaluate adversarially trained models against *Disp*, *IPR*, and *Kreuk* attacks using the same 100 malicious binaries used in prior work [36], making our results directly comparable to the original, non-adversarially-trained, model. As in prior work, these evaluation binaries are unseen by the model from the test partition of *VTFeed*, and are all below a file size of 512 KB to ensure that a *Disp* attack cannot evade a classifier by simply displacing the malicious bytes outside of the input size of the classifier, which for prior work was a smaller model (*AvastNet*) [36]. Also, in line with prior work, we allow attacks to attempt evasion for up to 200 iterations and calculate the percentage of final adversarial examples that successfully reduce their maliciousness below the 0.1% FPR threshold for the target model (i.e., the threshold at which the target model empirically misclassifies 0.1% of benign binaries as malicious in the entire original test partition of the dataset). This maliciousness threshold is calculated for each target model before attacks are executed. As these attacks are non-deterministic and the results could be noisy, we repeat the attacks 5 times to reduce noisiness of results.

We visualize these comparisons in plots showing the percentage of final adversarial examples that successfully evade detection. (by getting below their respective target model's 0.1% FPR threshold). Note that when *Disp* or *Kreuk* attack success is a single percentage, or appears as a single line in a

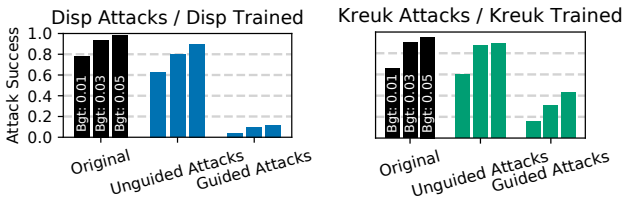


Figure 1: Attack success rates for *Disp* and *Kreuk* attacks up to 200 iterations where each group of three bars includes one bar per attack budget $\in \{0.01, 0.03, 0.05\}$. This figure compares the original model and models trained with unguided *Disp* adversarial examples (up to 5 iterations with budgets 0.01, 0.03, and 0.05) or unguided *Kreuk* adversarial examples (with a budget of 0.01).

plot, it refers to the mean success rate across all budgets evaluated $\{0.01, 0.03, 0.05\}$, whereas in barplots summarizing *Disp* or *Kreuk* attack success, budgets are shown distinctively in groups of three bars (e.g., Fig. 2).

Finally, we also measure how natural accuracy (on unmodified binaries) is affected by different adversarial training configurations. To this end, we test each adversarially trained model original test set of *VTFeed*.

4.2 Unguided Augmentation

As described in Sec. 3.3, we wanted to emulate the equivalent of data augmentation for this domain to see if it provided any robustness to state-of-the-art adversarial attacks. We trained with unguided *Disp* adversarial examples given up to 5 iterations, using all *Disp* budgets, on a full epoch of the training dataset (55K batches). We also trained another model with unguided *Kreuk*-0.01 adversarial examples described in Sec. 3.3. We constrained these examples budget to 0.01 as that is what gave us the best results for guided *Kreuk* training (Sec. 4.3).

Fig. 1 shows the attack success rate on these trained networks after unguided training compared to the original model, and the same configuration using guided attacks. As we can see, unguided *Disp* attacks do help some with robustness to *Disp* attacks (90% $\rightarrow$ 78% success rate), but using guided *Disp* attacks helps much more (90% $\rightarrow$ 13% success rate). Similarly, training on unguided analogs of *Kreuk*-0.01 reduce *Kreuk* attack success rate some (84% $\rightarrow$ 79%) but much less than training on evasive *Kreuk*-0.01 attacks (84% $\rightarrow$ 30%). Considering that we found guided *Disp* and *Kreuk* attacks empirically only take up to 12-20% longer to create than their unguided analogs, we conclude that using guided attacks is the better option for adversarial training.

We found that unguided *IPR* attacks up to 5 iterations take 478 seconds to complete on average, imputing around 11 days to complete an epoch of training. For this reason, we only trained on 25K batches to see if there was any effect. Through *IPR* attack evaluations, we found that 26% of *IPR* attacks still succeeded, which is equal to the success rate against the original model, implying no robustness gain from unguided *IPR* attacks. Moreover, we found that guided *IPR* attacks (up to 5 iterations) took much less time at an average of 132 seconds, nearly four times faster than unguided *IPR* attacks (after our optimizations mentioned in Sec. 3.3). Similarly to unguided *Disp* and *Kreuk*, we conclude that unguided *IPR* attacks are simply too inefficient and ineffective at inducing robustness to be suitable for adversarial training.

4.3 *Disp* and *Kreuk* Adversarial Training with Varying Budgets

Our experiments show, for both *Disp* and *Kreuk* attacks, that adversarially training solely with low budget attacks (0.01) provides some robustness to higher budget (3-5x) attacks (0.03, 0.05).

For *Disp* attacks, when we train on all budgets, the neural network is either no more or only slightly more robust to *Disp* attacks for all evaluation budgets. This trend is shown in Fig. 2 where the bar plots trained only on *Disp* attacks with a budget of 0.01 (Low Budget) have similar robustness compared to otherwise identical adversarial training configurations that also train on adversarial examples with budgets of 0.03 (Low+Mid Budget) and 0.05 (All Budgets).

Also, for *Disp*, the resulting TPR at a 0.1% FPR after 55K batches of training for Low, Low+Mid, and All Disp Budgets is 95.5%, 95.0%, and 95.0%, respectively. Interestingly, low budget attacks have an empirical 0.5% improvement over the other budgets, which may be due to the fact that these low budget adversarial examples are the most similar to the original non-adversarial examples trained on, and so training results in less of a dramatic shift in data distribution.

This effect is more pronounced for *Kreuk* attacks, where training with those higher budget *Kreuk* attacks degraded natural accuracy (96% $\rightarrow$ 17% TPR at 0.1% FPR), which made the 0.1% FPR attack success threshold for the resultant model easier to achieve, increasing the success rate of attacks. This is shown in Fig. 2, where the models trained only on *Kreuk* attacks with a budget of 0.01 (Low) or 0.01 and 0.03 (Low+Mid) reduce *Kreuk* attack success rates of all budgets, but training 0.01, 0.03, and 0.05 (All) results in increasing *Kreuk* attack success rates. As a result, we use the model only trained with *Kreuk*-0.01 (Low Budget) attacks to compare with other models in the rest of the paper.

Similar to *Disp* attacks, we found that training on low budget *Kreuk* attacks (0.01) provided some robustness to higher budget attacks (0.03, 0.05). For this reason, the most notable trend is that training with low budget attacks seems to give nearly equal or better robustness to higher budget attacks of the same class as training with those same higher budget attacks. This hints that the protection of *Disp* or *Kreuk* adversarial training cannot be easily defeated by simply increasing the budget of *Disp* or *Kreuk* attacks.

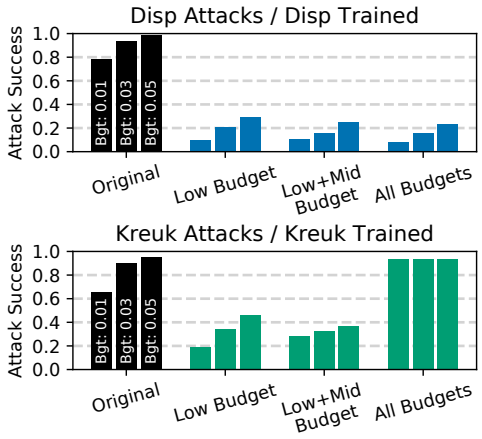


Figure 2: Comparing attack success rates for *Disp* and *Kreuk* attacks up to 200 iterations between models trained with varying budgets of *Disp* and *Kreuk* adversarial examples up to 3 and 200 iterations, respectively. Each group of three bars includes one bar per budget $\in \{0.01, 0.03, 0.05\}$.

4.4 *IPR* and *Disp* Adversarial Training with More Attack Iterations

In almost every instance of *IPR* and *Disp* adversarial training, DNNs became more robust to the attacks they were trained on, resulting in the attack generating processes needing more iterations to be able to successfully evade the strengthening target network. Adversarial examples in earlier batches of adversarial training were often successful in their first few iterations, but later training adversarial examples (of both *Disp* and *IPR*) targeting a now more robust model would almost always take up to the capped amount of iterations allowed for that particular adversarial training configuration. This is visualized in Fig. 7 in App. C. Since the time taken to generate an attack is roughly linearly dependent on how many iterations of the attack are executed, allowing a larger cap in iterations directly results in longer adversarial training times.

Disp **Training with More Iterations** Completing 1, 3, 5, or 10 iterations of *Disp* takes an average of 28, 45, 63, and 101 seconds respectively. However, as shown in Fig. 7, training with a single *Disp* iteration achieves most of the robustness benefit (90% $\rightarrow$ 30% *Disp* attack success rate) when compared to training with 3 iterations (90% $\rightarrow$ 16%), 5 iterations (90% $\rightarrow$ 13%), and 10 iterations (90% $\rightarrow$ 9%). This robustness from 1 iteration is accomplished while taking less than a third of the time as 10 iterations. Moreover, as visualized in Fig. 3a, the marginal benefit of completing more iterations of *Disp* quickly reduces, especially past 3 iterations. This trend is supported by the observation that later iterations of both *IPR* and *Disp* attacks are less effective at achieving evasion than the initial iterations, so much time can be saved by only training on attacks capped at lower numbers of iterations.

Disp-training's robustness against *IPR* and *Kreuk* attacks is visualized in Fig. 3d. Training with a single *Disp* iteration reduces *IPR* attack success rates by 26%, but only reduces *Kreuk* attack success rates by 6%. However, increasing the iterations executed of *Disp* training attacks from 1 to 10 results in further reducing both *IPR* and *Kreuk* attack success rates by at least another 20%. This suggests that the initial code displacement operation executed by *Disp* may help increase robustness to *IPR*, as it is made up of similar code manipulation techniques. However, this displacement does not help much against *Kreuk* attacks until extra bytes inserted by *Disp* are optimized to be more evasive. For more details on how *Disp* works, please see App. C.

For TPR at 0.1% FPR when varying iterations, at 55K batches, the difference was not significant when considering 1, 3, 5, and 10 iterations with a TPR of 94.8%, 95.0%, 94.8%, and 94.6%, respectively. For this reason, we conclude that *Disp* training at different iterations does not seem to affect TPR when training up to 55K batches (a full epoch).

IPR **Training with More Iterations** When we aggregate the vulnerability of *IPR*-trained models to explore the effect of increasing the iterations of *IPR* attacks trained on, we see that a single iteration of *IPR* does provide more robustness to *IPR* attacks (26% $\rightarrow$ 18% attack success). This is similar to the robustness to *IPR* gained from training on a single iteration of *Disp* attacks (26% $\rightarrow$ 19%). Increasing the number of *IPR* iterations from 1 to 10 notably makes DNNs more robust to *IPR* (18% $\rightarrow$ 7%). These trends are illustrated in Fig. 3b.

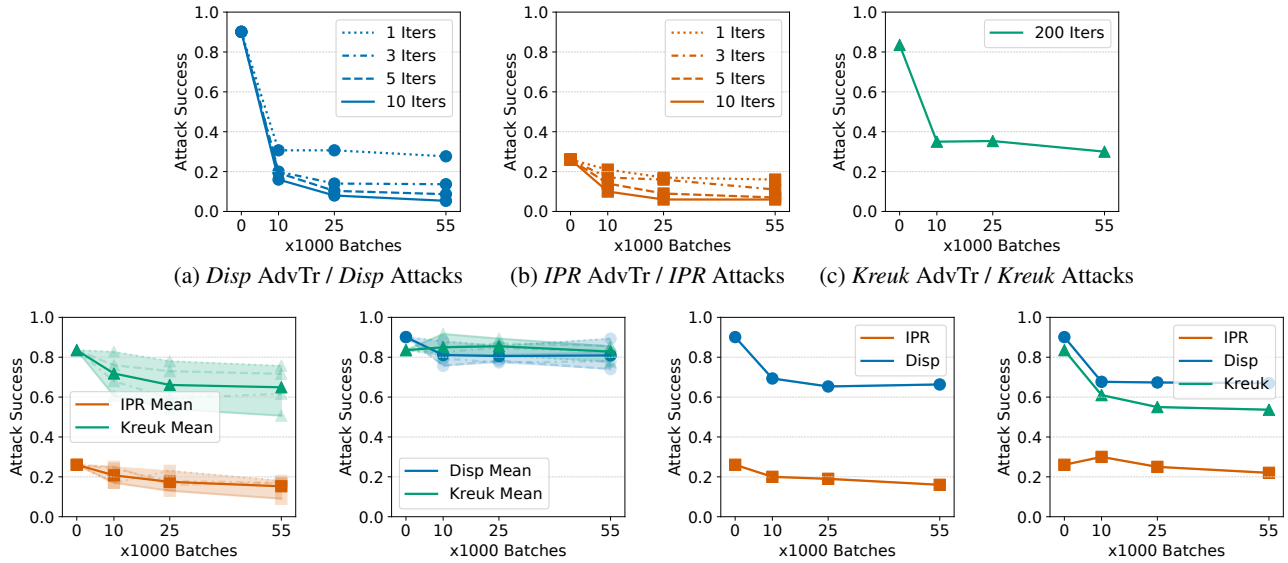
As shown in Fig. 3e, one iteration of *IPR* does little to make DNNs more robust to *Disp* or *Kreuk* attacks. However, we do see an extra 6% reduction in *Disp* attack success as DNNs are trained on 10 iteration *IPR* attacks. This is likely because *IPR* consists of four different transformations that compose together given multiple iterations over a binary, creating combination transformations not seen in one *IPR* iteration.

When considering how *IPR* training with more iterations affects TPR at 0.1% FPR at 55K batches, we find that the difference between 1, 3, 5, and 10 iterations is not significant at 95.2%, 95.6%, 95.7%, and 95.2%, respectively.

4.5 Adversarial Training with More Batches

To measure how the number of adversarial examples trained on affects robustness, we average our model evaluations at 10K, 25K, and 55K batches.

Disp **Training with More Batches** For *Disp* training, we observe a trend similar to the one described in Sec. 4.4: training on just 10K batches results in the largest robustness gain. This is shown in Fig. 3a. Training with 10K batches reduces *Disp* attack success rate from 90% to 21%. Training to 55K batches (i.e., 5.5$\times$ the work) further reduces *Disp* attack success, but only to 14%, a comparatively small effect.



(a) *Disp* AdvTr / *Disp* Attacks (b) *IPR* AdvTr / *IPR* Attacks (c) *Kreuk* AdvTr / *Kreuk* Attacks

(d) *Disp* AdvTr / Other Attacks (e) *IPR* AdvTr / Other Attacks (f) *Kreuk* AdvTr / Other Attacks (g) *Greedy* Tr / Other Attacks

Figure 3: Comparing attack success rates for *Disp*, *IPR*, and *Kreuk* attacks up to 200 iterations between models trained with varying number of adversarial examples and training attack iterations. Training on these attacks substantially increases model robustness to the same attack, as shown in Figs. 3a–3c. The downward trends in Fig. 3d and Fig. 3f show that training on *Disp* or *Kreuk* attacks reduces model vulnerability to other attacks by a smaller amount. "{*IPR*, *Disp*, *Kreuk*} Mean" is the mean attack success rate over multiple adversarially trained models (e.g., 1-, 3-, 5-, 10-iteration *Disp*-adversarially-trained models in Fig. 3d).

Similar trends hold on robustness to *IPR* and *Kreuk* attacks, and are shown in Fig. 3d. Training on 5.5× more adversarial examples decreases *IPR* attack success from 21% to 15%, similar to the *IPR* success decrease after training on 10K *Disp* batches (26% → 21%). The additional decrease in *Kreuk* attack success from 5.5× more training is also small (72% → 65%). This is in contrast to the larger robustness gains (against *IPR* and *Kreuk*) from increasing the number of iterations of *Disp* training attacks from 1 to 10 (Sec. 4.4).

However, the metric that seems most sensitive to the number of batches trained on is the natural TPR at 0.1% FPR. At 10K, 25K, and 55K batches, the mean TPR (averaged across 1, 3, 5, and 10 iterations) is 89.8%, 93.7%, 94.8%, respectively. It could be difficult to justify the significant decrease in TPR at 10K batches when compared to the TPR of the original model of 96.1%, despite the significant increase in adversarial robustness. For this reason, it seems training on more adversarial batches may be necessary.

***IPR* Training with More Batches** We coincidentally found that training on only 10K batches of *IPR* gave the same *IPR* robustness increase when compared to the original model as training on 55K batches of *Disp* (26% → 15% *IPR* attack success) and similar robustness as training on 55K batches of *Kreuk* (26% → 16%). This is not surprising, as we would expect training on *IPR* attacks would give more robustness to *IPR* attacks. However, similar to training with *Disp* attacks for *Disp* training, training on 5.5× more batches of *IPR* gives only marginally more robustness to *IPR* attacks (15% → 10% *IPR* attack success), as seen in Fig. 3b.

Fig. 3e illustrates that training with more than 10K batches of *IPR* attacks does not increase robustness to *Disp* and *Kreuk* attacks, indicating that any non-*IPR* robustness gained from training on *IPR* attacks is gained early in training.

In regard to how *IPR* training with more batches effects TPR, we observe a similar effect as in *Disp*-training, but less dramatic changes: At 10K, 25K, and 55K batches, the mean TPR (averaged across 1, 3, 5, 10 iterations) is 92.4%, 94.4%, and 95.4%, respectively. When comparing same number of batches, *IPR* training results in slightly higher TPR than *Disp* training. We suspect this is likely because *Disp* attacks are far more likely to result in a successful evasion, and therefore in larger weight updates, especially in earlier batches of training.

***Kreuk* Training with More Batches** Like *Disp*-training, *Kreuk*-0.01-training gets most of its robustness gain to *Kreuk* attacks early (within 10K batches) as shown in Fig. 3c. However, unlike the other attacks, this early high robustness gain (and subsequent low gain for further batches) also holds when evaluating with *IPR* and *Disp* attacks, as shown in Fig. 3f. This could be because *Kreuk* attacks include the addition of highly evasive bytes (Sec. 2.3), and learning to ignore clusters of highly evasive bytes may not take as many examples to learn compared to more complex *IPR* and *Disp* attacks.

However, *Kreuk*-0.01-training hurts TPR more than *Disp*- and *IPR*-training. At 10K, 25K, and 55K batches, the TPR is

84.4%, 87.0%, and 90.1%, respectively. Training with higher budget *Kreuk* attacks results in even lower TPR (Sec. 4.3).

4.6 *Greedy* Adversarial Training

Despite not creating viable binaries (Sec. 3.4), Fig. 3g shows that training with *Greedy* perturbed examples results in reduced attack success for all three attacks. *Disp*, *IPR*, and *Kreuk* attack success is reduced by 26%, 16%, and 36%, respectively after 55K batches of training with a budget of 0.01 (1% of the file size). We tried training with higher budgets of 0.03 and 0.05, but this resulted in much lower natural accuracy, and actually made the attacks more successful, similar to training with higher budgets of *Kreuk* attacks (Sec. 4.3).

However, even at a 0.01 budget, *Greedy*-training hurts TPR more than using the attacks (96% → 86%). This is likely because *Greedy* perturbations produce the largest distribution change as they do not result in valid binaries.

4.7 *IPR-Disp-Kreuk* Adversarial Training

Finally, we wanted to see if training simultaneously on *IPR*, *Disp*, and *Kreuk*-0.01 adversarial examples would result in a model that is more robust to all three attacks. We trained on 55K batches composed of adversarial examples generated evenly between *IPR* (up to 5 iterations), *Disp* (up to 5 iterations), and *Kreuk*-0.01 attacks. The average time to one of these training attacks was 66 seconds. We evaluated the model by executing *IPR*, *Disp*, and *Kreuk* attacks on it after 10K, 25K, and 55K batches, as is done for all adversarial training runs in previous experiments.

Fig. 4 shows the success rate of these evaluation attacks, compared with the performance of other adversarial training methods. The attack success rate (after 55K batches) against this combined-trained model from *Disp* (14%) and *IPR* (13%) attacks are notably higher than the level observed when training on only, respectively, *Disp* (9%) or *IPR* (7%) attacks. In contrast, the combined model's vulnerability to *Kreuk* attacks (21%) was lower than when we only trained with *Kreuk*-0.01 (30%) attacks. This supports two earlier findings: (1) that DNNs learn to be robust to *Kreuk* with fewer batches than *IPR* and *Disp* (Sec. 4.5), and (2) that DNNs gain robustness to *Kreuk* from *Disp*-training (Sec. 4.5).

For natural TPR at 0.1% FPR, we find that training on all attacks results in a lower TPR than training with *Disp* or *IPR*, but higher than training on *Kreuk*. At 10K, 25K, and 55K batches, the TPR is 83.8%, 89.1%, and 91.2%, respectively. This is likely because *Kreuk*-0.01 training attacks are especially hurtful for TPR, as first noted in Sec. 4.3.

4.8 Results Summary

We summarize our results here. Also, Table 3 gives a visual summary and Tables 4–7 (App. A) give a numeric summary.

Attack (α)	Low Effort	More robustness to α over LET by increasing: # Batches	*Disp* Iters.	*Disp* Budget
Disp	◉	+⊙	+◉	+◯
IPR	◉	+◉	+◉	+⊙
Kreuk	◉	+⊙	+◉	+◯

(a) Adversarial training with *Disp*.

Attack (α)	Low Effort	More robustness to α over LET by increasing: # Batches	*IPR* Iters.
Disp	⊙	+◯	+⊙
IPR	◉	+◯	+⊙
Kreuk	◯	+◯	+◯

(b) Adversarial training with *IPR*.

Attack (α)	Low Effort	More robustness to α over LET by increasing: # Batches	*Kreuk* Budget
Disp	◉	+◯	+◯
IPR	◉	+⊙	+◯
Kreuk	●	+⊙	+◯

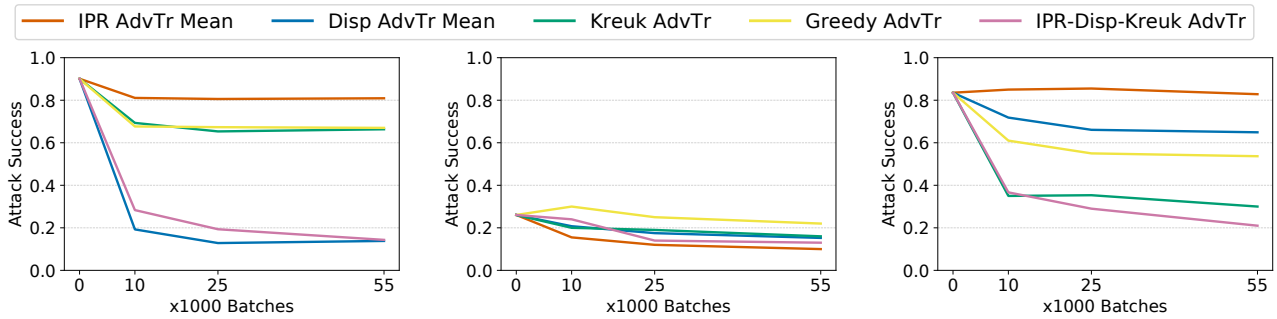
(c) Adversarial training with *Kreuk*.

Attack (α)	Low Effort	More robustness to α over LET by increasing: # Batches	*Greedy* Budget
Disp	◉	+◯	+◯
IPR	◯	+⊙	+◯
Kreuk	◉	+⊙	+◯

(d) Adversarial training with *Greedy*.

Table 3: Robustness gain from *Disp* (a), *IPR* (b), *Kreuk* (c), and *Greedy* (d) training to attack type $\alpha \in \{IPR, Disp, Kreuk\}$. ◯ = no gain, ⊙ = small gain, ◉ = moderate gain, and ● = large gain. Low Effort values are detailed in App. A.

- It is important to ensure the transformations in *Disp*, *IPR*, and *Kreuk* attacks are optimized to be highly evasive instead of randomly chosen (Sec. 4.2).
- High-budget *Disp* and *Kreuk* attacks are likely to see reduced success on lower budget adversarially trained models, an encouraging finding for defending against unseen larger-budget versions of these attacks (Sec. 4.3).
- Optimization of *IPR* and *Disp* adversarial examples to be more evasive beyond one iteration helps DNNs become more robust, but not dramatically (Sec. 4.4). For *IPR* training, more iterations are valuable as combinations of different *IPR* transformations can occur via multiple iterations over a binary, evidenced by more pronounced robustness to *Disp* attacks (90% → 76% in one case).
- Adversarial training with only 10K batches gains most of the robustness benefits to *IPR*, *Disp*, and *Kreuk* attacks, but natural TPR suffers early in training. Training on more batches is important to increase the TPR closer to the original model's performance (Sec. 4.5).



(a) *Disp* Attacks (b) *IPR* Attacks (c) *Kreuk* Attacks

Figure 4: Attack success rates for *Disp*, *IPR*, and *Kreuk* attacks up to 200 iterations against different types of adversarial training. *IPR-Disp-Kreuk* training represents a model trained with a combination of *IPR* (up to 5 iterations), *Disp* (up to 5 iterations), and *Kreuk* attacks. Training with all three of these attacks induces near best-case robustness benefit to all attacks simultaneously.

- Combining all attacks in training can result in more broadly robust models (Sec. 4.7). However, as shown in Fig. 4a and Fig. 4b, our most robust models from *Disp* and *IPR* attacks came from training solely on those same attacks. Therefore, the better system may be an ensemble of models trained on different attacks.

5 Discussion

In this section, we discuss lessons learned, useful features of our work, limitations, and suggest future directions.

Generalizing Lessons Learned to Other Attacks Our results suggest that adversarial training is a promising approach to defend against each attack we evaluated (*Disp*, *IPR*, and *Kreuk*). More importantly, we showed that it is not necessary to train on the highest-effort attacks to still gain substantial robustness to them. For example, training with 1- to 10-iteration *Disp* and *IPR* severely reduced success rates for attacks of up to 200 iterations (Sec. 4.4). Similarly, when considering attacks that scale their effectiveness by adding more or fewer evasive bytes (*Kreuk* and *Disp*), training on versions that modified only 1% of the binary increased robustness against attacks that modified up to 5% of the binary (Sec. 4.3).

Additionally, we showed that training on some attacks can increase robustness to other attacks that use similar techniques, such as for *Disp* and *Kreuk*, which both optimize the evasiveness of contiguous blocks of bytes. We observe this cross-robustness even though *Disp* modifies binaries by introducing only semantic `nop` instructions and *Kreuk* and *Disp* modify different parts of the binary. This effect is especially evident in *Greedy*-training (Sec. 4.6), where all attack success rates are reduced (*IPR*, *Disp*, *Kreuk* by -16%, -26%, -36%, respectively) without training on any of the attacks, and the *Greedy* perturbation does not produce viable binaries. However, *Greedy*-training is the most harmful to natural TPR at -11%, suggesting this method should not be preferred over other adversarial training strategies.

Finally, most of our adversarially trained models gained the majority of their robustness within the first 10K batches of training (Sec. 4.5). This suggests that lengthy adversarial training runs are not necessary to understand how robust a model can be to a particular attack. However, we also found that this early adversarial training harmed natural TPR, and that training with more batches is important to recover accuracy on the original data (Sec. 4.5 and Table 7).

These trends all hold when we repeat experiments with a different architecture (see App. B), and suggest that when adversarially training in the malware-detection domain, it is less important that each example trained on is the most effective version of an attack, than it is to train on an attack similar in approach to those we seek to defend against.

Also, to understand if *VTFeed*'s file size cap of 5 MB (discussed in Sec. 3.2) could have affected our results, we analyzed another popular dataset, *Ember2018*, which was collected by an antivirus company and whose constituent binaries were not constrained to be under 5 MB in size [3]. We found that 95% of binaries in that dataset were smaller than 5 MB, indicating that *VTFeed*'s file size cap only excluded around 5% of binaries on collection. Furthermore, prior work found that there was "no appreciable difference" in test accuracy between a *MalConv* model trained on *Ember2018* that truncated files to only analyze the first 2 MB (*MalConv*'s input size [46]) and a modified version of *MalConv* that did not truncate larger files (and instead inferred over the entire binary) [47, Sec 5.2]. As processing larger binaries did not affect the accuracy of *MalConv*, and because relatively few files were excluded from *VTFeed*, we believe our results are similarly unaffected by *VTFeed*'s exclusion of larger files.

Robustness to Other Binary Transformations To understand if our adversarial training affected robustness to other binary transformations, we used recent work on creating ad-

versarial examples from an ensemble of techniques [52]. We evaluate how the combined training by *IPR*, *Disp*, and *Kreuk* (Sec. 4.7) affects model vulnerability to these techniques, including section renaming, section appending, section adding, and checksum breaking. We modify prior work's code [52] to transform the same 100 malicious binaries used for evaluation in Sec. 4, targeting both our adversarially trained model and the original model. We find that our adversarially trained model's robust accuracy to this ensemble attack is an average of 49% whereas the original model's robust accuracy is an average of 23%, showing that the adversarially trained model is more robust to these unseen binary transformations. We also note that, as reported in the original work [52], five of the seven of these transformations occasionally rendered binaries invalid, limiting their utility in practical attacks.

Benefits from Transforming a Larger Set of Binaries As mentioned in Sec. 1 and Sec. 3.2, one of the challenges we overcame was disassembling 126,009 binaries from *VTFeed*, expanding the number of attackable binaries by over 500× compared to prior work's 200 binaries tested [36]. Doing so carried two primary benefits. First, executing prior work's code [36] on a much larger set of binaries revealed bugs that could lead transformations to break binary validity and corner cases where the code became very slow. We fixed these issues (Sec. 3.3), resulting in more correct and faster transformations. We will release the updated code upon publication.

Also, the increased scale of transform-able binaries allowed us to avoid overfitting the classifier's adversarial robustness to a small number of source binaries. For example, if we instead only trained on many adversarial versions of a small number of binaries, the classifier could memorize constant bytes (e.g., a function or set of bytes that none of the transformations can change) and learn to correctly classify all 'adversarial examples' just by recognizing those unchanging pieces from the small amount of source binaries. By having a much larger set of binaries to transform, we avoid this potential issue and expose the detector to a larger pool of adversarial examples.

Limitations We note here some limitations of this work. This work only trains and evaluates malware detection DNNs that operate on static features, and cannot detect malicious behavior only present at runtime, such as downloading a malicious payload. We also assume the attacker's perturbations do not change the binary's functionality and uses known adversarial example generation techniques [33, 36, 52].

Our adversarial training transformed only unpacked binaries that could be disassembled by IDA Pro, as mentioned in Sec. 2.6, as this was a requirement for the attacks we considered [36]. However, DNNs are adept at detecting malicious behavior even when the binary is packed, as shown in App. D.

Future Work An avenue for future work is to adversarially train on 200 iteration *IPR* and *Disp* attacks, to understand what a model's robustness would be if the long computation time were not an issue, and to fully understand the tradeoff made by training with lower iteration attacks as done in this work. However, with current capabilities, this could take several months or years to train (as calculated in Sec. 3.3).

Another area of improvement could be in having the budget, as used in the *Disp*, *Kreuk*, and *Greedy* transformations, be optimized instead of being a pre-defined value as in this work. This could be useful in a different threat model that considered that the attacker may want to reduce the distribution difference between their evasive binary and the original binary. Another reason to optimize this parameter could be that minimizing this distribution difference could help preserve benign accuracy during adversarial training. In that case, the techniques discussed by Pintor et al. [45] could be leveraged.

6 Conclusion

This work showed that it is possible to defend against adversarial examples in the raw-binary malware-detection domain via adversarial training. This was achieved via improvements on multiple fronts of this previously impractical process. Improvements includes speeding up adversarial example creation by using fewer iterations, code optimization, parallelization, and increasing the pool of attack-eligible binaries. Together with our findings that training on binary transformations yields a tradeoff between time taken to adversarially train a classifier and resultant TPR and robustness, our results suggest that a TPR of 90–95% at 0.1% FPR and separately trained best-case robustness to *Disp* (5% success rate), *IPR* (6% success rate), and *Kreuk* (30% success rate) can be achieved after 55K batches (one epoch) of training. Given that malware detection exists in an adversarial environment, this work represents an encouraging development and guide for how to create more robust raw-binary malware classifiers less susceptible to evasion attacks.

Acknowledgments

We thank Michael Stroucken for technical help. This work was supported in part by the U.S. Army Research Office under MURI Grants W911NF-17-1-0370 and W911NF-21-1-0317; by NSF grants 1801391 and 2113345; by the National Security Agency under award H9823018D0008; by the Defence Science and Technology Agency (DSTA); by a DoD National Defense Science and Engineering Graduate fellowship; by Len Blavatnik and the Blavatnik Family foundation; by a Maof prize for outstanding young scientists; and by the Neubauer Family foundation.

References

[1] M. Abadi et al. TensorFlow: A system for large-scale machine learning. In *Proc. OSDI*, 2016.

[2] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth. Evading machine learning malware detection. *Black Hat*, 2017.

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

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

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

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

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

[8] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

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

[10] E. Carrera Ventura. pefile. https://github.com/erocarrera/pefile, 2022.

[11] F. Ceschin, M. Botacin, H. M. Gomes, L. S. Oliveira, and A. Grégio. Shallow security: on the creation of adversarial variants to evade machine learning-based malware detectors. In *Proc. ROOTS*, Nov 2019.

[12] F. Ceschin, M. Botacin, G. Lüders, H. M. Gomes, L. Oliveira, and A. Gregio. No need to teach new tricks to old malware: Winning an evasion challenge with xor-based adversarial samples. In *Proc. ROOTS*, Nov 2020.

[13] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay. Adversarial attacks and defences: A survey. *arXiv preprint 1810.00069*, 2018.

[14] F. Chollet et al. Keras. https://keras.io, 2015.

[15] Chronicle. Virustotal. https://www.virustotal.com/, 2004–. Accessed 6/17/2019.

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

[17] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le. AutoAugment: Learning augmentation policies from data. In *Proc. CVPR*, 2019.

[18] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. In *Proc. ITASEC*, 2019.

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

[20] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab. A review on feature selection in mobile malware detection. *Digit. Investig.*, 13(C):22–37, Jun 2015.

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

[22] M. Galovič, B. Bosanský, and V. Lisý. Improving robustness of malware classifiers using adversarial strings generated from perturbed latent representations. In *Proc. NeurIPSW*, 2021.

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

[24] Hex-Rays. IDA: About. https://www.hex-rays.com/products/ida/. Accessed 9/13/2019.

[25] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry. Adversarial examples are not bugs, they are features. In *Proc. NeurIPS*. 2019.

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

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

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

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

[30] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 2006.

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

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

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

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

[35] X. Liu, J. Zhang, Y. Lin, and H. Li. ATMPA: Attacking machine learning-based malware visualization detection methods via adversarial examples. In *Proc. IWQoS*, 2019.

[36] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre. Malware makeover: Breaking ML-based static analysis by modifying executable bytes. In *Proc. AsiaCCS*, 2021.

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

[38] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

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

[40] T. pandas development team. pandas-dev/pandas: Pandas, Feb. 2020.

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

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

[43] J. Pereyda. jtpereyda/libdasm, Jul 2022.

[44] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *Proc. IEEE S&P*, 2020.

[45] M. Pintor, F. Roli, W. Brendel, and B. Biggio. Fast minimum-norm adversarial attacks through adaptive norm constraints. In *Proc. NeurIPS*, 2021.

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

[47] E. Raff, W. Fleshman, R. Zak, H. S. Anderson, B. Filar, and M. McLean. Classifying sequences of extreme length with constant memory applied to malware detection. In *Proc. AAAI*, 2021.

[48] Redis. Redis. https://redis.io/, 2009.

[49] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo. Data mining methods for detection of new malicious executables. In *Proc. IEEE S&P*, May 2001.

[50] A. Shafahi, M. Najibi, A. Ghiasi, Z. Xu, J. Dickerson, C. Studer, L. S. Davis, G. Taylor, and T. Goldstein. Adversarial training for free! In *Proc. NeurIPS*, 2019.

[51] C. Shorten and T. M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48, July 2019.

[52] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin. MAB-Malware: A reinforcement learning framework for blackbox generation of adversarial malware. In *Proc. AsiaCCS*, 2022.

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

[54] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. *Proc. ICML*, 2017.

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

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

[57] L. Tong, B. Li, C. Hajaj, C. Xiao, N. Zhang, and Y. Vorobeychik. Improving robustness of ml classifiers against realizable evasion attacks using conserved features. In *Proc. USENIX Security*, 2019.

[58] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

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

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

[61] H. Wu, C. Wang, Y. O. Tyshetskiy, A. Docherty, K. Lu, and L. Zhu. Adversarial examples for graph data: Deep insights into attack and defense. In *Proc. IJCAI*, 2019.

[62] P. Yang, J. Chen, C.-J. Hsieh, J.-L. Wang, and M. I. Jordan. Greedy attack and gumbel attack: Generating adversarial examples for discrete data. *Journal of Machine Learning Research*, 21(43):1–36, 2020.

A Quantitative Summary of Results

Tables 4–6 show the success of *Disp*, *IPR*, and *Kreuk* attacks for the original baseline and how their success is affected by varying different parameters of adversarial training. Table 7 shows the TPR at 0.1% FPR given these same variations. These tables also show the progression of using a **Low Effort** value and a **High Effort** value for the parameter being varied in that row. For varying *Disp* and *Kreuk* budgets, a budget of 0.01 is the Low Effort value, while training with all three budgets of 0.01, 0.03, and 0.05 is the High Effort value. For varying attack iteration cap, the Low / High Effort values are 1 iteration / 10 iterations. For varying batches, the Low / High Effort values are 10K / 55K batches.

Metric	original	**Low Effort**		**High Effort**	
	success	success	Δ	success	Δ
Disp: unguided	0.90			0.78	-14%
Disp: budget	0.90	0.20	-78%	0.16	-82%
Disp: attack iters	0.90	0.30	-67%	0.10	-89%
Disp: batches	0.90	0.21	-76%	0.14	-85%
IPR: batches	0.90	0.81	-10%	0.81	-10%
IPR: attack iters	0.90	0.88	-3%	0.82	-9%
Kreuk: budget	0.90	0.67	-26%	0.94	5%
Kreuk: batches	0.90	0.69	-23%	0.66	-26%
Greedy: batches	0.90	0.68	-25%	0.67	-26%

Table 4: Change in *Disp* attack success when varying each training parameter. The most robust configuration against *Disp* attacks was training on 10-iter *Disp* attacks, for 55K batches, with all budgets (90% → 5% success rate).

Metric	original	**Low Effort**		**High Effort**	
	success	success	Δ	success	Δ
Disp: unguided	0.26			0.28	7%
Disp: budget	0.26	0.22	-14%	0.20	-25%
Disp: attack iters	0.26	0.19	-26%	0.13	-50%
Disp: batches	0.26	0.21	-20%	0.15	-41%
IPR: batches	0.26	0.15	-40%	0.10	-62%
IPR: attack iters	0.26	0.18	-31%	0.07	-72%
Kreuk: budget	0.26	0.18	-30%	0.94	262%
Kreuk: batches	0.26	0.20	-23%	0.16	-39%
Greedy: batches	0.26	0.30	15%	0.22	-16%

Table 5: Change in *IPR* attack success when varying each training parameter. The most robust configuration against *IPR* attacks was training on 10 iteration *IPR* attacks, for 55K batches (26% → 6% success rate).

Metric	original	**Low Effort**		**High Effort**	
	success	success	Δ	success	Δ
Disp: unguided	0.84			0.80	-4%
Disp: budget	0.84	0.64	-24%	0.63	-25%
Disp: attack iters	0.84	0.79	-6%	0.55	-34%
Disp: batches	0.84	0.72	-14%	0.65	-22%
IPR: batches	0.84	0.85	2%	0.83	-1%
IPR: attack iters	0.84	0.80	-4%	0.89	6%
Kreuk: budget	0.84	0.33	-60%	0.94	13%
Kreuk: batches	0.84	0.35	-58%	0.30	-64%
Greedy: batches	0.84	0.61	-27%	0.54	-36%

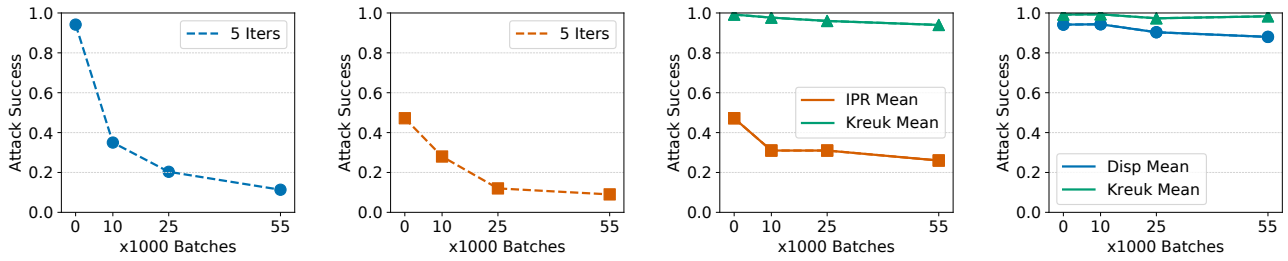
Table 6: Change in *Kreuk* attack success when varying each training parameter. The most robust configuration against *Kreuk* attacks was training on *Kreuk*-0.01 attacks for 55K batches (84% → 30% success rate).

B Adversarial Training Results on *AvastNet*

To investigate the generality of the trends found in this work, we repeated *IPR* and *Disp*-training on another architecture, *AvastNet* [32]. The accuracy of this model pre-trained on *VTFeed* is 99.89%, 98.59%, and 98.60% for the train, test, and validation sets, respectively. The TPR at 0.1% FPR is 94.78% on the test set [36]. Results are shown in Fig. 5.

A notable difference is that all attacks (*Disp*, *IPR*, and *Kreuk*) are more successful in attacking *AvastNet* than *MalConv*. This may be due to *AvastNet* having a smaller input space (512 KB) than *MalConv* (2 MB), allowing the attack's modifications to binaries to take a larger percent of the total input space of *AvastNet*, which may give the modified bytes more influence in determining the classification of the input.

Regardless, the trends observed in Sec. 4 based on experiments on the *MalConv* architecture were also observed when adversarially training *AvastNet*. Training on *IPR* or *Disp* attacks reduces the attack success rate of the same attack the most, while providing some more robustness to other attacks. *Disp*-training provides slightly more robustness to *Kreuk* attacks than *IPR*-training does, as can be seen by comparing Fig. 5c and Fig. 5d. Due to time constraints and the large amount of computation need to adversarially train a network and evaluate its performance at 10K, 25K, and 55K batches with all attacks, we did not experiment with a *Kreuk*-0.01-trained *AvastNet*. However, given that the interaction between *IPR*- and *Disp*-training and *IPR*, *Disp*, and *Kreuk* attacks on *AvastNet* mirrored the behavior of the same experiments on *MalConv*, we believe results for *Kreuk*-training would likely continue to follow previously observed trends.



(a) *Disp* AdvTr / *Disp* Attacks (b) *IPR* AdvTr / *IPR* Attacks (c) *Disp* AdvTr / Other Attacks (d) *IPR* AdvTr / Other Attacks

Figure 5: Plots show results for *AvastNet*. Comparing attack success rates for *Disp*, *IPR*, and *Kreuk* attacks up to 200 iterations between *AvastNet* models trained with varying number of adversarial examples and training attack iterations. Training on either *IPR* or *Disp* attacks substantially increases model robustness to the same attack, as shown in Figs. 5a–5b. The downward trends in Fig. 5c show that training on *Disp* attacks reduces model vulnerability to other attacks by a smaller amount.

Metric	original	**Low Effort**		**High Effort**	
	TPR	TPR	Δ	TPR	Δ
Disp: unguided	0.96			0.95	-1%
Disp: budget	0.96	0.94	-2%	0.94	-2%
Disp: attack iters	0.96	0.93	-3%	0.92	-4%
Disp: batches	0.96	0.90	-7%	0.95	-1%
IPR: batches	0.96	0.92	-4%	0.95	-1%
IPR: attack iters	0.96	0.94	-2%	0.94	-2%
Kreuk: budget	0.96	0.87	-9%	0.17	-82%
Kreuk: batches	0.96	0.84	-12%	0.90	-6%
Greedy: batches	0.96	0.76	-21%	0.86	-11%

Table 7: Change in model TPR at 0.1% FPR when changing each training parameter.

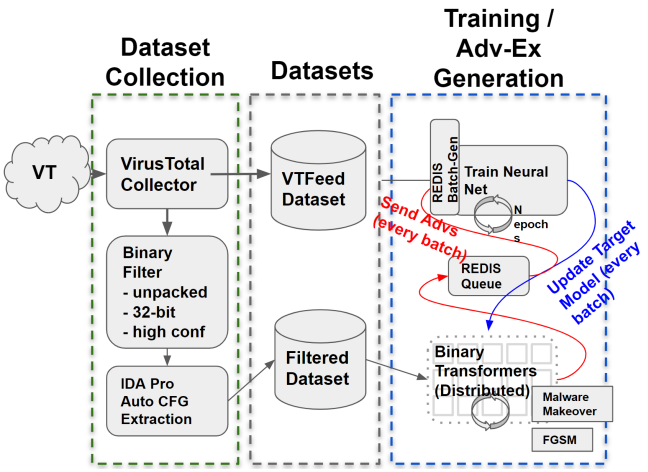


Figure 6: Distributed system for adversarially training DNNs.

C Implementation Details

Machine Learning Framework and Distributed System We use the Tensorflow [1] and Keras [14] libraries to train, evaluate, and store the DNN model checkpoints.

We use Docker [38] containers to execute workers across multiple Linux-based servers[2]. These workers read and write log files, using the pandas [40] library, to a Network File System (NFS) server. We use Redis [48] to distribute and deconflict jobs. The workers are coded in Python [58], as are the scripts used to generate all adversarial examples (*IPR*, *Disp*, and *Kreuk*) attacks. The workers use the CPU rather than GPU for all computations due to resource availability.

The attack implementations built on attacks used by Lucas et al. [36], which themselves built on code from other work [31, 42]. We describe our improvements in Sec. 3.3.

For every binary to be transformed, the *IPR* and *Disp* attack implementations use files outputted by IDA Pro [24] that describe the control flow graph of the binary. The *IPR*, *Disp*, and *Kreuk* attack implementations use the pefile [10] and libdasm [43] Python libraries to parse compiled binaries.

***IPR* and *Disp* Attacks** The attacks referred to as *IPR* and *Disp* in this paper were created by prior work. We describe them in more detail here for convenience, but for the most detailed description please see the original paper [36].

Both *IPR* and *Disp* attacks are based on prior work in binary diversification [31,42], which consists of transformations that modify the bytes, and the underlying instructions they represent, in such a way that the functionality is unchanged.

IPR (In-Place Replacement) transformations [42] include:

- **equiv**: This transformation replaces instructions with equivalent instructions. For example, the instruction `add eax, 0x18` can be replaced with `sub eax, -0x18`.
- **reorder**: This transformation reorders instructions that do not rely on a specific order of operations. For example,

[2] 3 servers with 64 GB RAM and AMD Ryzen 9 3900Xs, ; 2 servers with 256 GB RAM and AMD Ryzen Threadripper PRO 3975WXs; a server with 64 GB RAM and AMD Opteron 6274s; a server with 3 TB RAM and Intel Xeon E7-4850s; a server with 24 GB RAM and Intel i7-4770s; a server with 24 GB RAM and Intel i7-11700Ks; a server with 32 GB RAM and an Intel i7-2600; and 3 servers with 16 GB RAM and an Intel i7-2600

the instructions `add eax, 0x1` → `xor ebx, 0x2` can be reordered to `xor ebx, 0x2` → `add eax, 0x1`.

- **preserv**: When pushing register values to the stack to preserve their values, the order they are pushed must match the order in which they are restored, but is otherwise arbitrary. The `preserv` transformation alters this push-pop order considering this constraint.

- **swap**: This transformation swaps registers in a block of instructions. For example, if `eax` is being used as a loop counter and `ebx` holds an address, this transformation may switch their roles in the block by replacing every occurence of `eax` with `ebx` and vice versa.

To make these *IPR* transformations evade a target DNN classifier, prior work added a mechanism that calculated, for each atomic transformation (e.g., an `equiv` replacement of an instruction), if the change in byte values would perturb the targeted model's embedding of the binary in a direction that had a positive cosine similarity with the loss gradient of the targeted classifier [36]. If so, the transformation was applied. If not, the transformation was discarded.

The *Disp* (Displacement) transformation works by moving (*displacing*) a block of instructions to another part of the binary and connecting the instruction flow via `jmp` instructions. This leaves an open space in the original location of the block, which is then filled with *semantic nop* instructions. The semantic nop instructions do not necessarily need to be `nop` instructions, but can be any instruction that does not change the state of the program. For example, semantic nop instructions can be `add ebx, 0x0` or `mov eax, eax`. Because of this flexibility, the *Disp* attack can choose the *semantic nop* instructions that are most useful for evading correct classification by the currently targeted model.

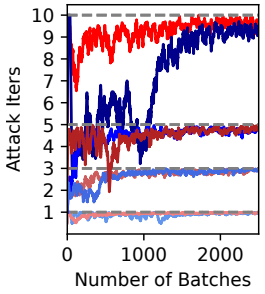


Figure 7: Number of attack iterations (smoothed over 100 batches) executed in different adversarial training runs. Red, blue lines represent *IPR*, *Disp* attacks, resp. Attacks quickly take up to their allowed iter cap, slowing adv training.

Both *IPR* and *Disp* attacks are implemented as an iterative process that considers and applies a transformation to each function[3] in a binary. We call one round of this process an *iteration*. Over multiple iterations, a single function can be transformed by multiple transformation types (e.g., `equiv` and `swap`). Increasing the number of iterations can lead to more evasive binaries, at the cost of more compute time.

As depicted in Fig. 7 and first mentioned in Sec. 4.4, attacks executed to create binaries for adversarial training are typically capped to a small number of iterations and execute for that number of iterations, unlike attacks used to test a trained classifier, which typically use a much larger iteration cap and hence often find an evasive binary and stop iterating well before reaching the cap. Hence, the time it takes to adversarially train a model is linearly dependent on the cap multiplied by the number of adversarial examples desired.

D Dataset Details

The dataset we use for our experiments, *VTFeed*, was collected and labeled by Lucas et al. [36]. We briefly describe the collection and labeling next.

VTFeed consists of 278,316 binaries. Each binary is a 32-bit Portable Executable (PE) file up to 5 MB in size, first seen in 2020, and classified by either 0 or 40+ antiviruses (AVs) as malware (as shown in the VirusTotal metadata for the file). Collection consisted of filtering the VirusTotal [15] feed for binaries that met these criteria, downloading them, and labeling them as malicious or non-malicious based on if they were categorized as malware by 40+ AVs or 0 AVs, respectively. *VTFeed* includes both `.dlls` and `.exes`.

The test set of *VTFeed* contains 7,571 packed and 20,261 unpacked files. The TPR (at 0.1% FPR) of the pre-trained *MalConv* model for packed and unpacked files separately, is 93.00% and 97.16%, respectively. This shows that, on average, binary packing did not significantly degrade trained DNN malware detectors. The *VTFeed* dataset contains 188,198 EXEs and 90,118 DLLs. The dataset was not constrained to specific malware families. The most common malware families reported by AVs after classifying *VTFeed* binaries were: `Virlock`, `Dharma`, `Upatre`, `Coinminer`, `GandCrab`, and `Kryptik`. A distribution of malware descriptions given by Malwarebytes is shown in Fig. 8.

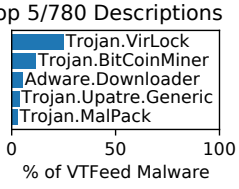


Figure 8: Distribution of malware descriptions for *VTFeed* given by Malwarebytes. Most prevalent is `Trojan.VirLock` at 25%.

The VirusTotal feed gives real-time access to the files being uploaded for analysis by VirusTotal, and as such is a good representation for the kind of files that would likely be analyzed by the static malware detectors this work considers.

[3] Function as defined by the IDA Pro disassembly, which may not correspond to functions in the original source code.