Training Robust ML-based Raw-Binary Malware Detectors in Hours, not Months

Keane Lucas
kjlucas@alumni.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Weiran Lin
weiranl@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Lujo Bauer
lbauer@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Michael K. Reiter
michael.reiter@duke.edu
Duke University
Durham, North Carolina, USA

Mahmood Sharif
mahmoods@tauex.tau.ac.il
Tel Aviv University
Tel Aviv, Israel

Abstract

Machine-learning (ML) classifiers are increasingly used to distinguish malware from benign binaries. Recent work has shown that ML-based detectors can be evaded by adversarial examples, but also that one may defend against such attacks via adversarial training. However, adversarial training, and subsequent robustness evaluation, is computationally expensive in the raw-binary malware-detection domain because it requires producing many adversarial examples for both **training** and **evaluation**. Prior work found that *Greedy-training*, a faster robust training technique that forgoes using adversarial examples, showed some promise in producing robust malware detectors. However, *Greedy*-training was far less effective in inducing robustness than the more expensive adversarial training, and it also severely hurt natural accuracy (i.e., accuracy on the original data). To faster **train** models, this work presents *GreedyBlock*-training, an enhanced version of *Greedy*-training that we empirically show achieves not only state-of-the-art robustness in malware detectors, exceeding even adversarial training, but also retains natural accuracy better than adversarial training. Furthermore, as it does not require creating adversarial (or functional) examples, *GreedyBlock*-training is significantly faster than adversarial training. Specifically, we show that *GreedyBlock*-training can produce more robust (+54% on average), more naturally accurate (+7% on average), and more efficiently trained (-91% average computation) malware detectors than prior work. To faster **evaluate** models, we also develop methods to faster gauge the robustness of ML-based raw-binary malware detectors by introducing *robustness proxies*, which can be used either to predict which models are likely to be the most robust, thus helping prioritize which detectors to evaluate with expensive attacks, or aiding in deciding which detectors are worthwhile to continue training. Experimentally, we show these proxy measures can find the most robust detector in a pool of detectors while using only ~20-50% of the computation that would otherwise be required.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690208

CCS Concepts

• **Security and privacy** → **Malware and its mitigation**; • **Computing methodologies** → *Feature selection*; *Neural networks*; *Supervised learning*.

Keywords

machine learning, adversarial robustness, malware detection

ACM Reference Format:
Keane Lucas, Weiran Lin, Lujo Bauer, Michael K. Reiter, and Mahmood Sharif. 2024. Training Robust ML-based Raw-Binary Malware Detectors in Hours, not Months. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690208

1 Introduction

ML-based malware detection is an important component of modern cybersecurity systems. Unfortunately, previous work shows that these detectors can be evaded by *adversarial examples*, regardless of whether the input is expert-derived features or the raw bytes of the binary [2, 13, 22, 26, 28, 40].

When using binaries' raw bytes as input, recent work shows that it is possible to defend against adversarial examples via adversarial training (i.e., using adversarial examples as training data) to create more robust malware-detection deep neural nets (DNNs) [27]. Unfortunately, adversarial training is costly: Many adversarial examples are needed both to train and to evaluate models, and in previous work each adversarial example took between 5 and 4424 seconds to create. We estimate[1] that prior work's training approach would take between one and 40 million CPU-seconds (12–463 days) per model, depending on the type of attack, how many iterations the attack is optimized for, and how many batches are trained on. Training the best-performing model suggested by previous work, which shows increased robustness to several types of attacks, would take around 15 million CPU-seconds (174 days). Furthermore, the time required to evaluate each model's robustness would take another 40,000 to three million CPU-seconds (1–35 days).

[1]Based on previous work's description of how many adversarial examples were used

In this paper we propose new, efficient methods for both **training** robust malware detectors and **evaluating** the resulting models. Regarding **training**, we enhance a previously ineffective data-augmentation method (*Greedy*-training [27]) by addressing its major weaknesses: (1) it significantly reduces the original model's natural accuracy (i.e., accuracy on original data), and (2) it produces models that are less robust than adversarially trained models. We address the harm to natural accuracy by reducing the distribution shift between the original and augmented data and the low robustness by better imitating the attacks used in adversarial training and disincentivizing the model from putting too much weight on a small number of bytes. We then comprehensively evaluate the model's performance in natural accuracy, the computation time it took to create, and robustness to attacks via a hyperparameter sweep. We also compare the resultant models to the state-of-the-art (more expensive) adversarial training models [27]. We show that *GreedyBlock*-training can produce more robust (+54% on average) and more naturally accurate (+7% on average) malware detectors than those reported in prior work [27], via a faster training process (−91% average computation time)[2].

Motivated by the need to **evaluate** a large number of models when seeking to identify the best-performing one, we also contribute a new method for more efficiently finding the most robust raw-binary malware detectors. The method consists of *robustness proxy measures* (or `robustness proxies`), along with processes for using these robustness proxies to find the most robust detector in a pool of detectors. The proposed robustness proxies allow efficient and accurate prediction of malware detectors' robustness. These robustness proxies are designed to capture aspects of ML models that have been shown in prior work to differ between robust and non-robust models. We draw upon prior work's findings that robust models generally have smoother, more interpretable decision boundaries [12, 45]; that models robust to one type of attack in the raw-binary malware detection domain are often also more robust to other types of attacks [27]; and that conditions for attack success in the raw-binary domain depend on the performance of the model on the test set (e.g., the 0.1% FPR maliciousness threshold at which attacks are evaluated in prior work [27, 28]). These findings motivate our measurement of decision boundary smoothness by using the L_2-norm, L_∞-norm, and mean of a classifier's gradients and attributions (i.e., explanations), along with measuring the smoothness of the attributions themselves, across both neighboring bytes and the whole binary. We also measure the attack success of simple and fast raw-binary attacks [26], and the model's performance on the test set (e.g., test accuracy). These measures can then be used as features in a predictor (e.g., linear regressor) to estimate the robustness of a model without the need to execute computation-intensive attacks.

In Sec. 5, we outline processes for using these robustness proxies to both predict individual detectors' robustness and, given a pool of detectors, rank them by their predicted robustness to quickly select the most promising ones. Specifically, we first train a predictor to use robustness proxies as features to predict a malware detector's robustness (i.e., 1—highest attack success rate (ASR)) using a pool of malware detectors (also referred to as *models*) with associated robustness proxies and the (computationally expensive) ASR already measured (i.e., training pool). To evaluate how well this process works, we use that predictor to predict the ASR for a pool of unseen models (i.e., test pool) and compare the predicted ASR to the models' true ASR. We then demonstrate through three use cases how this malware detector robustness prediction can allow us to avoid expensive attack execution while still identifying the most robust malware detectors.

Overall, we find that using these robustness proxies produces robustness predictions (on unseen models) that are highly correlated with the computationally expensive ASR-derived robustness of the model (with Pearson's correlation coefficient of $r \geq 0.90$), which can eliminate the need to attack most models when searching for the most robust model, reducing computational expense by ∼50–80%. Using these robustness proxies to predict robustness has benefits beyond just time savings: reduced computation time lowers monetary cost, as well as power consumption and carbon emissions [6]. The ability to more quickly retrain a robust model can also help keep it accurate on the latest binaries. This is important, as previous work has shown that models trained on older data can become less accurate over time due to data drift [10].

In summary, the contributions of this work are:

- We enhance *Greedy*-training [27] to develop *GreedyBlock*-training, a significantly more effective process for creating robust ML-based raw-binary malware detectors (Sec. 4.1). Through comprehensive experiments, we show that *GreedyBlock*-training can produce malware detectors that reduce the strongest attack's success from a worst case of 100% to less than 20%, retain a natural accuracy of over 95% TPR at 0.1% FPR, all while taking significantly less computation than the previous methods on average (Sec. 4.2.2).
- We propose *robustness proxies*, which allow efficient and accurate prediction of a raw-binary malware detector's robustness, alleviating much of the computational burden of the attacks used for evaluation. Using these measures with a feature selection algorithm and predictor (Sec. 5.1), we show high correlation of a model's predicted robustness to its computation-intensive ASR-derived robustness with $r \geq 0.90$.
- Finally, we demonstrate how robustness proxies can accelerate evaluation and training in three separate scenarios (Sec. 5.2.2). Two scenarios provide experimental evidence that robustness proxies can be used to identify a set of candidate models likely to be robust; evaluating only those models with real attacks takes only 10% of the time it would take to evaluate all models with real attacks. The third scenario instead uses the predictions of robustness proxies as a filter to decide whether to continue training a model based on its current estimated robustness; this speeds up *training* by up to 40% compared to training all models to completion while still resulting in similarly robust models (Table 2).

We have provided code at https://doi.org/10.1184/R1/26322505.

[2]This speed comparison excludes *Kreuk*- and *Greedy*-training, as they amount to strawman solutions and are not competitive in robustness.

2 Background and Related Work

This section introduces relevant background. We start with existing ML techniques to detect malware (Sec. 2.1). Then, we elaborate on attacks to evade ML classifiers, with a focus on malware detectors (Sec. 2.2). Additionally, we present adversarial training, a defense against evasion attacks, and its application in the raw-binary malware detection domain (Sec. 2.3). We then conclude by discussing attributions of ML models and how they relate to known characteristics of robust ML models (Sec. 2.5).

2.1 Malware Detection DNNs

Deep neural networks (DNNs) have been shown to be capable of detecting malware by using the raw bytes of an executable as input [3, 25, 34]. Compared to using expert-designed features which are usually constructed via some pre-processing of a raw-binary and therefore require time and effort to develop [3, 4, 15, 17, 21, 23, 35], using ML classifiers to detect malware from the raw compiled bytes of an executable empirically achieves similar performance [3, 25, 28, 34] and does not require expert feature engineering.

2.2 Evading Malware Detection DNNs

ML classifiers in various domains have been shown to be vulnerable to *adversarial examples*—slightly perturbed inputs that cause misclassification [5, 7, 9, 18, 30, 43]. To find adversarial examples, a common approach is to use the loss gradients to search for inputs that maximize the loss within some distance constraint from the original (unperturbed) input, so as to ensure the perturbed input remains recognizable to humans as the same class. A common distance metric used is an L_p norm, which was designed to mimic human imperceptibility of perturbations in the image domain [5, 7, 9, 18, 30]. The ability of models to resist being fooled by adversarial examples is referred to as *robustness*.

Evasion attacks have most often been considered in the image domain, but our concern here is evasion of raw-binary malware classifiers. Evasion attacks in the raw-binary domain differ markedly from their counterparts in the image domain. In the image domain, a slight perturbation to an image is still likely to be perceived as the same image by humans. In contrast, as binaries are a long sequence of discrete byte values that represent an underlying program, perturbing even a single byte's value, even a random one, could change the underlying functionality or simply make it no longer executable. This obstacle is called the *inverse feature-mapping problem* [33].

Additionally, unlike images, binaries are less often perceived visually by humans. Thus, instead of achieving human imperceptibility, evasion attacks in the raw-binary domain aim to maintain samples' functionality. After adding the perturbation, the binary should behave as if there were no perturbations [24, 26, 31].

IPR, Disp, and Kreuk attacks. Existing works take different approaches to preserve functionality when making changes to a binary. One of these approaches is to add or modify bytes in non-executable regions of binaries, (e.g., *Kreuk* attacks [26]). Another approach is to replace instructions with equivalent alternatives using existing functionality-preserving transformations (*IPR* attacks [28, 31]). There is also previous work that moves locations of instruction chunks and links them with `jmp` instructions (*Disp* attacks [24]). The gaps left behind by moving these instructions can then be filled with instructions that do nothing (no-op instructions) but are optimized to be evasive to the targeted classifier [28].

A key aspect of these raw-binary evasion attacks relevant for this work is that *IPR* and *Disp* attacks are computationally expensive to generate, as they require detailed analysis of the underlying assembly code to create candidate transformations that preserve functionality, and then a filtering process to keep only the transformations that are evasive to the targeted classifier. In our experiments, we found that *IPR* and *Disp* attacks take an average of 908 and 487 seconds, respectively, to generate a single adversarial example when using the same settings as prior work used for evaluation (i.e., up to 200 iterations) [27, 28]. *Kreuk* perturbations, on the other hand, are computationally much cheaper (~5 seconds) to generate, as they require much less analysis of the underlying code, and instead simply append non-executable bytes to the end of the binary. While this makes *Kreuk* attacks easier to defend against [28], it also makes them a good candidate for quickly figuring out if a model is robust or not, as discussed in Sec. 5.1.

Our work aims to hinder these three attacks (*IPR*, *Disp*, and *Kreuk*). Therefore, we also use them to evaluate the robustness of malware classifiers, as recent works do [20, 27, 28].

2.3 Adversarial Training Raw-Binary Malware Detectors

A common defense against evasion attacks is adversarial training [18, 29, 36, 48]. By training ML models on the adversarial examples generated to fool them, ML models can learn to be more robust. However, this can sometimes come at the cost of correctly classifying the original, unperturbed data (i.e., natural accuracy). In the image domain, adversarial training is one of the most widely used defenses against evasion attacks. Nonetheless, a key requirement of adversarial training is generation of many new adversarial examples to train on [29, 36, 48]. Techniques for more efficient adversarial training have been proposed that make more efficient use of gradient calculations [36] or use weaker attacks [48], but cannot be used in the raw-binary malware-detection domain due to the inverse feature-mapping problem discussed in Sec. 2.2.

There has been some prior work on adversarially training malware detectors that rely on hand-crafted features [14, 17, 44]. However, as we introduced in Sec. 2.2, generating adversarial examples against raw-binary malware detectors is computationally costly. Thus, adversarially training these malware detectors is even more costly. Prior work has relied on scale, code efficiency, and training with weaker versions of attacks to accomplish adversarial training with raw-binaries. However, this approach still required up to a year of computation to train a single malware detector [27].

Another expense that is especially relevant in the raw-binary malware detection domain is the cost to evaluate the robustness of a model after it has been trained. As discussed in Sec. 2.2, generating adversarial examples against raw-binary malware detectors is costly, and in order to measure the robustness of a model against several attacks, we need to generate adversarial examples for many binaries against the model for each attack. Moreover, *Disp* and *Kreuk* attacks can be executed with different *budgets*, which determines how much the attacker can increase the size of the binary due

to the attack, leading to more variations that must be tested. Prior work evaluated models by transforming 100 malicious binaries to look benign using each attack and attack variant, repeating the attack five times (to account for stochasticity), and repeating this process for 39 different model checkpoints. This validation phase amounts to an average of around 1.5 million seconds (~17 days) of computation time per malware detector, a large part of the cost of obtaining adversarially robust models.

In this paper, we show how to reduce the time cost for training and evaluating robust raw-binary malware detectors.

2.4 *Greedy*-Training

Greedy-training is a technique that was introduced in prior work [27] to reduce the computation cost of adversarial training for raw-binary malware detectors. The technique is based on the observation that most of the computational cost of raw-binary adversarial training comes from generating adversarial examples that are meticulously transformed to be functionally identical and executable, and that training with some types of attacks (i.e., *Kreuk*) can provide some robustness to more expensive attacks (i.e., *Disp*).

Therefore, *Greedy*-training forgoes the requirement of training with valid and executable adversarial examples, and instead simply trains with binary inputs where a random subset of selected bytes have been perturbed to be the most evasive to the targeted classifier. As a result, these inputs are no longer valid executables, and this technique qualifies as a data augmentation, rather than adversarial training. This technique was shown to be much faster than adversarial training, but also much less effective in inducing robustness [27], along with significantly hurting natural accuracy. This work addresses these issues.

2.5 ML Attributions

Some of the tools we use in this paper are derived from ML-explanation techniques. Generally, ML explanations attempt to explain the predictions of ML models, some by giving attribution scores to the input features that represent how important the feature is to the model's prediction. Examples of techniques that provide attribution scores include using the gradient of the input with respect to the model's prediction [38] and Integrated Gradients [42]. Another set of techniques we use are built on top of these attribution scores, originally developed for the purpose of evaluating how well an attribution technique is performing [47]. These techniques, called Necessity Ordering (N-Ord) and Sufficiency Ordering (S-Ord), order the input features by their attribution scores, and then remove (resp., add) the features one by one. The area under (resp., over) the curve of how the model's prediction changes as features are removed (resp., added) is then used to compare attribution methods' ability to explain the model's prediction. We adapt these techniques to instead measure a trained model's tendency to change predictions when important features (as determined by the attribution method) are removed or added to the input. The averages of the N-Ord or S-Ord scores using different attribution methods over several binaries are used as a robustness proxy for the model's robustness to adversarial examples, discussed more in Sec. 5.1.

ML attributions have also been used to characterize robust DNNs. In the image domain, robust DNNs have been found to have smoother decision boundaries and more accurate attributions (according to human-created bounding boxes) than non-robust DNNs [12, 46]. For example, a robust model's attributions may be more likely to highlight the pixels representing the body of a dog in an image as contributing to the classification of "dog". In the feature-based malware detection domain, attributions have been used to identify malware detector input features that are more vulnerable to adversarial manipulation [41]. We leverage these findings to construct our robustness proxies (Sec. 5.1).

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

Table 1: *VTFeed* dataset from prior work [28].

3 Threat Model and Dataset

As the purpose of our work is to improve upon the state of the art in adversarially training raw-binary malware detectors [27], we adopt the same threat model, dataset, and malware detection models.

3.1 Threat Model

We describe our threat model using the framework outlined in prior work [8, 33]. We assume that the attacker's *goal* is to evade being detected by (or possibly erode trust in) the malware detector by causing it to misclassify malware as benign, or vice versa. To achieve this, the attacker has the *capability* to modify the input binary in any functionality-preserving way, having *full knowledge* of the model's architecture and parameters (i.e., whitebox access). While prior work has also shown that the ML attributions used by some of our robustness proxies can be misled by an adversary [19, 50], our work is unaffected as the attacker has no control over the training data or training process in which ML attributions are used.

The raw-binary malware detector is a DNN that takes a binary as input and outputs a prediction of whether the binary is malware or benign. This method of detection is a form of *static analysis*, and neither prior work's attack or adversarial training, nor this work's enhanced data augmentation methods require the binary to be executed. As the attacks of interest in this work require the binary to be unpacked, we also assume that the binary is unpacked before being fed into the detector.

3.2 Dataset and Malware Detection DNNs

We use the same dataset and DNN architectures as prior work [20, 25, 27, 28]: the *VTFeed* dataset [20, 27, 28] and the *MalConv* DNN architecture [34]. We additionally conduct smaller-scale experiments with the *AvastNet* architecture [25], which we report on in App. A. *VTFeed* contains 278,316 32-bit Portable Executables (PEs) that are less than 5MB in size (both DLLs and EXEs), with a roughly even amount of benign and malicious binaries. Labeling is done by aggregating the results of anti-virus engines (AVs) from VirusTotal [11]. If a binary is classified as malicious by 40+ AVs, then it is labeled as malicious, and if it is classified as malicious by 0 AVs, it is labeled as benign. Binaries labeled as malicious by 1–39 AVs are excluded.

To evaluate model robustness, we execute attacks using the same 100 malicious binaries from *VTFeed* that prior work uses to compare models and evaluate attack success [27, 28]. We use the same dataset to ensure all comparisons with prior work are fair. These files were selected to be less than 512 KB (to ensure any change in size of the file would still fit within the model input size) and classified with *high confidence*.[3]

4 GreedyBlock-training

To accelerate training of robust malware detectors, this section introduces *GreedyBlock*, our proposed data augmentation that can produce robust and naturally accurate detectors more efficiently than the training methods reported in prior work [27]. First, we detail the design of *GreedyBlock* and how we use it as part of the process for training robust malware detectors (Sec. 4.1). Then, we compare the performance of *GreedyBlock*-trained malware detectors to the adversarially trained detectors described in previous work and *Greedy*-trained detectors (Sec. 4.2), showing that the *GreedyBlock*-trained detectors are more robust, more naturally accurate, and faster to train than those prior work's detectors [27].

4.1 *GreedyBlock* Technical Approach

Adversarially training a malware detector requires a large amount of computation [27]. In an effort to significantly reduce the computation required, we demonstrate a fast data-augmentation-based method of training malware detectors by both training on specially modified inputs and including dropout layers in the architecture. We call this method *GreedyBlock*-training, as it is based on *Greedy*-training (proposed in prior work) [27]. In Sec. 4.2, we show that this new training method can train malware detectors to be more accurate and more robust than detectors described in prior work [27] while requiring an order of magnitude lower computation.

Greedy-training modified half of the inputs to the model during training to include randomly placed evasive bytes. Overall, this resulted in a slightly more robust model, but it also hurt the model's natural accuracy more than other training [27].

We enhance *Greedy*-training as follows:

(1) We modify the *Greedy* perturbation (described in Sec. 2.4) to group evasive bytes together in one or more contiguous blocks, rather than randomly distributing them throughout the binary. This better mimics the behavior of *Disp* and *Kreuk* attacks, in which evasive bytes also occur contiguously. We hypothesize that this incentivizes DNNs to look for signs of maliciousness or benign-ness based on larger portions of the binary instead of placing too much significance on individual bytes. This in turn may imply that attacks need to change more bytes to affect classification, which has been shown to make raw-binary malware detectors more robust to attacks [20].
(2) We add dropout [39] to input and dense layers to further disincentivize DNNs from relying on small numbers of bytes. Dropout randomly sets a fraction of the input dimensions to zero during training, which should help the DNN learn to rely on more bytes in the binary to make decisions.
(3) We augment only one example per batch, rather than augmenting half of the batch. Compared to *Greedy*-training, this allows us to complete *GreedyBlock*-training over the entire dataset (e.g., complete epochs) faster, decreases the number of augmented examples required, and decreases the likelihood of hurting natural accuracy (since the augmented examples have less weight in the batch's loss).
(4) Finally, to further protect natural accuracy, we also experiment with lower budgets (i.e., fewer evasive bytes) and train for more epochs (e.g., three instead of one as reported in previous work [27]).

Algorithm 1: *GreedyBlock* augmentation

Input : binary, budget, num_blocks, target model
Output : augmented binary

```
num_bytes_to_perturb ← budget × size(binary)
block_size ← num_bytes_to_perturb / num_blocks
integrated_gradients ← getIG(binary); // byte attribution values
top_n_block_indices ←
  chooseTopNBlockIndices(integrated_gradients, block_size,
  num_blocks); // choose most important indices for classification
random_values ←
  initializeBlockIndices(top_n_block_indices); // randomly
  initialize the values at chosen block indices
setByteValues(top_n_block_indices, random_values)
for iteration in range(num_iterations) do
    binary_embedding ←
      getEmbeddingValues(top_n_block_indices); // get the
      embedding values at block indices
    loss_gradients ←
      getLossGradients(top_n_block_indices, target_model);
      // get loss gradients for block indices
    perturbed_embedding ← binary_embedding + α×
      loss_gradients; // perturb embedding towards increasing loss
    adversarial_byte_values ← getClosestValidByteEmbed-
      ding(perturbed_embedding)setByteValues(top_n_block_indices,
      adversarial_byte_values)
```

Pseudocode for *GreedyBlock*-training is shown in Alg. 1. We randomly choose the number of blocks (i.e., groups of contiguous evasive bytes) to be between one and five, and we set the maximum number of iterations (i.e., optimization steps to choose more evasive byte values) to 10. We also implemented the same early-stop threshold as described in prior work, which for malware (resp., benign) binaries is when the inferred maliciousness is below (resp., above) the 0.1% FPR threshold for the model being trained [27, 28].

GreedyBlock-trained model pool. To evaluate the performance of *GreedyBlock*-trained models, we created a large pool of malware detectors, each trained with a different hyperparameter configuration. Specifically, we trained a model for each possible combination of the following hyperparameter settings:

- ***GreedyBlock* budget**: {0%, 0.25%, 0.5%, 1%, 3%}
- **Dense-layer Dropout**: {0%, 10%, 30%, 50%, 75%}
- **Input Dropout**: {0%, 5%, 25%, 50%, 75%}
- **Number of epochs**: {1, 2, 3}

[3] A binary is classified as malicious with *high confidence* if it receives a maliciousness score greater than the 0.1% FPR threshold of the classifier on the *VTFeed* test set.

This gave us a total of $5 \times 5 \times 5 \times 3 = 375$ models. These hyperparameter values were chosen to cover a broad possible range of values, include a value in which the associated component has no effect (e.g., 0%), and, for the budget[4], include values that prior work used [27, 28].

We also needed to attack these models with (computationally expensive) *IPR*, *Disp*, and (inexpensive) *Kreuk* to find their attack success rates and the robustness of each model, a process that took several months. (We show in Sec. 5.2 how to significantly cut down on evaluation time using robustness proxies.)

We next compare the performance of these *GreedyBlock*-trained models to the adversarially trained models described in previous work [27], showing that *GreedyBlock*-trained models are more robust, have higher natural accuracy, and are trained faster.

4.2 *GreedyBlock* Results

In this section, we compare the robustness, natural accuracy, and training time of *GreedyBlock*-trained models to the state-of-the-art models reported in prior work. Sec. 4.2.1 details how we measure robustness, natural accuracy, and training time for both the *GreedyBlock*-trained models and the state-of-the-art. Sec. 4.2.2 then empirically compares and discusses these measurements.

4.2.1 Evaluation Setup for Comparing Robust Models. To compare the performance of adversarially trained models, we measure their robustness, natural accuracy, and training time.

Robustness is measured as one minus the attack success rate of the most successful attack (i.e., the attack with the highest attack success rate), where the attacks used are *IPR*, *Disp* [28], and *Kreuk* [26], the same attacks used in prior work [20, 27]. *Disp* and *Kreuk* attacks are executed with budgets of 0.01, 0.03, and 0.05, which also mirrors parameters used in prior work. This results in seven attacks (1 *IPR*, 3 *Disp*, 3 *Kreuk*) targeting each model, each trying to cause 100 malicious binaries misclassify as benign (as mentioned in Sec. 3.2). For simplicity, we report (and predict) the attack success for *IPR* and the mean attack success for *Disp* and *Kreuk* attacks across all budgets unless specified otherwise.

Natural accuracy is measured as the true positive rate (TPR) at a false positive rate (FPR) of 0.1% on the test set of *VTFeed*, mirroring prior work [27, 28].

Training time is measured as the cumulative time taken to train the model. For the baseline robust models described in prior work [27], this is calculated by totaling the time taken to generate the model-training attacks, as generating these attacks was the reported bottleneck[5]. We disregard any computation time spent actually calculating the gradients and updating DNN weights, making our estimates of computation time conservative for the previous work's models. Measuring the training time of *GreedyBlock*-trained models is simpler, as *GreedyBlock*-training is executed in a single process and does not require any attack generation. We use the creation timestamps of the last 10 model checkpoints to estimate a per-checkpoint training time and then multiply that per-checkpoint time by the number of checkpoints leading to our evaluated model. We only use the preceding 10 checkpoints because, in some cases, training is paused between epochs of a model, and only using the last 10 checkpoints guarantees exclusion of these pauses. We calculate these training times on the same hardware as that used for previous work. Supporting the fairness of this comparison, we found that executing the *IPR*, *Disp*, and *Kreuk* attacks on this same hardware roughly matched previous work's reported timing.

4.2.2 Comparison with Prior Work Models. This section compares the performance of *GreedyBlock*-trained models to the state of the art robust models reported in previous work [27]. We find that the best *GreedyBlock*-trained models outperform the state of the art in most measures.

Fig. 1 shows the results of attacking *GreedyBlock*-trained models compared to the state of the art [27]. The *GreedyBlock*-trained models can have higher or lower natural accuracy (TPR at 0.1% FPR) or robustness. However, because of their faster training method, almost all of the *GreedyBlock* models were trained in an order of magnitude of time less than the state-of-the-art models. Note that the time axis is logarithmic. The best *GreedyBlock* models, as shown in Fig. 1, are gathered near the corners of each plot representing ideal performance, showing that they simultaneously achieve high robustness, high TPR, and low computation time (Sec. 5.2 demonstrates that robustness proxies allow us to find the highly performing models and validate their robustness in a fraction of the time it would take to attack all trained models).

For a comprehensive numerical comparison, Table 2 provides the success rate of each attack executed for evaluation of the most robust *GreedyBlock*-trained model and its natural accuracy as well as the corresponding reported results (and our time estimations) for the most robust of prior work's reported models [27]. We label the attack success of *Disp* and *Kreuk* attacks that use different budgets with "-budget" following the attack name. For example, *Disp*-0.01 is the attack success of *Disp* attacks with a budget of 0.01. In Table 2, we group the attack success columns of different budgets of *Disp* and *Kreuk* (i.e., *Disp*-0.01, -0.03, -0.05 is the same as *Disp*-0.01, *Disp*-0.03, *Disp*-0.05).

Prior work's reported models' performance. The top half of Table 2 shows the attack success against prior work's reported most robust models produced via adversarial training: training with *IPR* attacks, *Disp* attacks, *Kreuk* attacks, or with all three at once. In each case, the resultant model becomes more robust against that attack compared. For example, the attack success of *IPR* attacks against *IPR*-trained models is 0.07, which is lower than the *IPR* attack success against all other models (including the combined training of *IPR-Disp-Kreuk*).

GreedyBlock performance. As can be seen in the *Best GreedyBlock* row of Table 2, the most robust *GreedyBlock*-trained model has the highest overall robustness to *IPR*, *Disp*, and *Kreuk* attacks compared to models that were adversarially trained with *IPR*, *Disp*, and *Kreuk* attacks. Specifically, while the best *IPR*-trained model is able to accomplish a lower *IPR* attack success rate (0.07) than the best *GreedyBlock* model (0.1), the highest attack success rate against the most robust *IPR*-trained model is 0.94, resulting in a robustness of 0.06. In contrast, *GreedyBlock* achieves a remarkable

[4]The *budget* is how many bytes (as percentage of the binary) that an attack can change or add

[5]These calculations are based off of numbers reported for average training adversarial example creation time in the original work [27]. This work used 13 servers with 16 to 256 GB of RAM, and Intel {i7-2600, i78-117700K, i7-4770, Xeon E7-4850}, AMD {Opteron 6274, Ryzen Threadripper PRO 3975WX, Ryzen 9 3900X}.

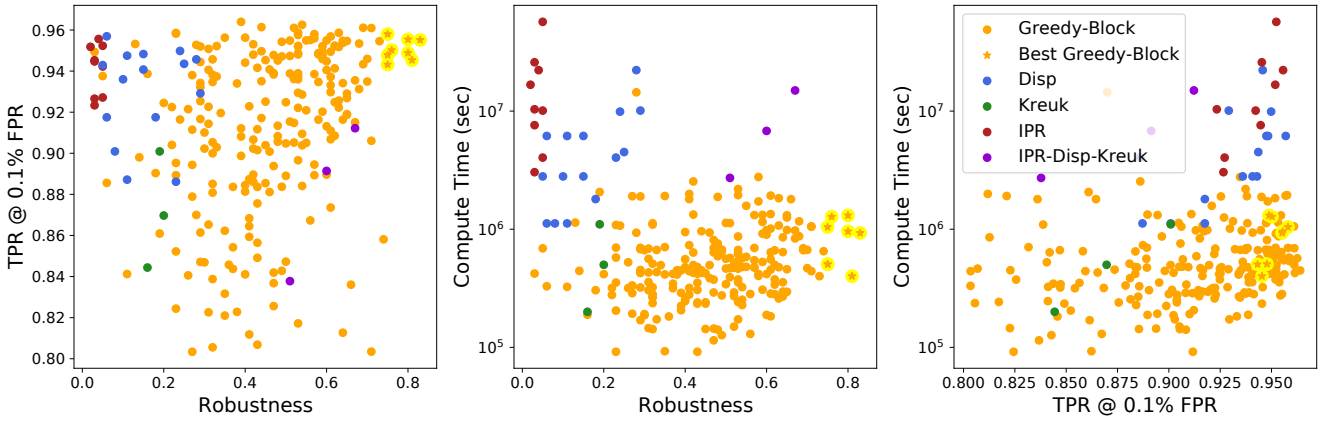


Figure 1: These plots show the relationship between *GreedyBlock*-trained models and prior work's models [27]. Robustness = 1 − [highest ASR]. The left plot shows that the best *GreedyBlock*-trained models have higher robustness than prior work at a competitive natural accuracy (TPR at 0.1% FPR). The center plot shows that the best *GreedyBlock*-trained models have higher robustness than prior work while taking less time to train. Finally, the right plot shows that the best *GreedyBlock*-trained models retain competitive natural accuracy, while taking less time to train.

Most Robust Results for Training Approach	Attack Success Rate (ASR)							Robustness ↑	TPR ↑	Time (s) ↓
	IPR ↓	*Disp*-0.01 ↓	-0.03 ↓	-0.05 ↓	*Kreuk*-0.01 ↓	-0.03 ↓	-0.05 ↓			
Original	0.26	0.78	0.94	0.99	0.65	0.90	0.95	0.01	0.96	–
IPR-training	**0.07**	0.57	0.77	0.88	0.70	0.93	0.94	0.06	**0.96**	29480K
Disp-training	0.13	**0.05**	**0.08**	**0.11**	0.36	0.55	0.71	0.29	0.93	10100K
Kreuk-training	0.19	0.50	0.66	0.80	0.19	0.37	0.50	0.20	0.87	500K
IPR-Disp-Kreuk-training	0.13	0.10	0.12	0.21	0.09	0.21	0.33	0.67	0.91	14960K
Greedy-training	0.30	0.55	0.69	0.79	0.49	0.64	0.70	0.21	0.76	**200K**
GreedyBlock-training	0.10	**0.05**	0.10	0.16	**0.08**	**0.15**	**0.17**	**0.83**	**0.96**	929K

Table 2: Prior work's best (i.e. most robust) models [27] vs. best *GreedyBlock* model. Robustness is 1 − highest ASR. Arrows show if lower/higher is better.

robustness of 0.83, which is higher than the most robust of all of the models described in prior work, *IPR-Disp-Kreuk*-training (0.67) [27]. The main takeaway is that the best *GreedyBlock*-trained model's decreased attack success rates are achieved *simultaneously*, whereas the models described in previous work are primarily only robust to the attack they were trained with [27].

This superior robustness is complemented by a superior natural accuracy (measure described in Sec. 4.2.1). The second-to-last column of Table 2 shows that the best *GreedyBlock*-trained model has a natural accuracy of 0.96 TPR, which is the same as prior work's most naturally accurate model (*IPR*-training) [27].

Regarding training time, *Kreuk*-training and *Greedy*-training is faster than *GreedyBlock*-training on average, and *GreedyBlock*, *Greedy*, and *Kreuk*-training are significantly faster than any other prior work training. This is because *Kreuk*-training relies on creating *Kreuk* attacks, which only take around 5 seconds each, as discussed in Sec. 2.2. Prior work's *Greedy*-training similarly is fast because *Greedy* augmentations can be generated faster and were only trained for 1 epoch [27]. However, as shown in all of the other columns concerning *Kreuk* and *Greedy*-training, the resultant models are not competitively robust or accurate with any of the models described in prior work, and they fall well short of the robustness and accuracy results of *GreedyBlock*-trained models [27].

As explained in Sec. 4.2.1, our estimation of prior work's reported model's training time is conservative, as it only accounts for the creation time of attacks, whereas *GreedyBlock*'s training time is directly measured via timestamps. This means that the training time of prior work's described models is likely even longer than we report, while the training time of *GreedyBlock*-trained models has been more precisely measured. As shown in Fig. 1, *GreedyBlock* training times vary from under 100K seconds to over 1M seconds. This variability is revelaed due to our method of measuring training time for *GreedyBlock*-training via the creation timestamps of model checkpoints (Sec. 4.2.1), allowing greater precision than the models' training-time estimates reported in prior work [27]. A primary cause of this variability is that if a model is becoming more robust, then the *GreedyBlock* augmentations often take longer to produce (as the models are becoming harder to fool).

This section has shown that *GreedyBlock* can be used to train more robust and accurate models than prior work, and that these models can be trained faster. However, verifying the robustness of these models by attacking all of them took considerable effort. The next section demonstrates how we could have avoided a significant portion of this effort by using robustness proxies to predict the most robust models without attacking all of them.

5 Robustness Proxies

To accelerate the evaluation of robust malware detectors by alleviating the burden of running expensive attacks against every detector, we propose creating and using *robustness proxies*. Sec. 5.1 details how we use ML explanation methods (see Sec. 2.5), training measures, and cheaper attacks to create robustness proxies for estimating the robustness of a malware detector. In Sec. 5.2, we evaluate the robustness proxies and measure the time savings they can produce if used in training robust malware detectors.

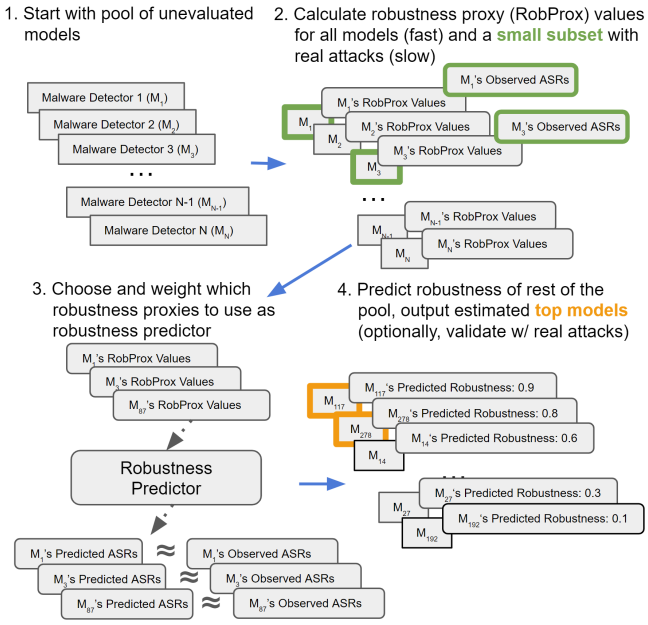


Figure 2: The general process for using robustness proxies to rank malware detectors and/or predict robustness of a malware detector without executing expensive attacks.

5.1 Robustness Proxies Technical Approach

We next detail how we construct robustness proxies (Sec. 5.1.1) and evaluate their ability to speed up the construction of robust malware-detection models (Sec. 5.1.2).

5.1.1 Proxy Measures of Model Robustness. We use measures based on ML attributions, simpler and faster raw-binary domain attacks, and the model's performance on the test set to estimate robustness.

Using ML attributions to estimate robustness. In domains such as image classification, prior work has found that robust models give better explanations (i.e., are more interpretable, have more robust attributions) and have smoother decision boundaries than non-robust models [12, 46]. As the gradients with respect to the input are used to construct these attributions, we hypothesized that the aggregated statistics of these gradients and explanations (e.g., L_2-norm, L_∞-norm) over many typical inputs could provide a proxy measure for the robustness of a model.

Robustness Proxy	Description
`[grad|IG]_l2_norm`	ℓ_2 norm of the attributions of the model's output with respect to the input.
`[grad|IG]_linf_norm`	ℓ_∞ norm of the attributions of the model's output with respect to the input.
`[grad|IG]_nord`	N-Ord value. Area under the curve created by iteratively padding out the most important bytes according to the attributions. A smaller number indicates it took fewer bytes padded out to change the classification.
`[grad|IG]_sord`	S-Ord value. Area under the curve created by iteratively adding the most important bytes according to the attributions. A smaller number indicates it took fewer bytes added to a baseline input to change the classification.
`[grad|IG]_mean`	Mean of the attributions of the model's output with respect to the input.
`[grad|IG]_std`	Standard deviation of the attributions of the model's output with respect to the input.
`[grad|IG]_smoothness`	Average of the absolute value of the difference between each byte's attribution and its direct neighbor's attribution.
`kreuk001`	*Kreuk* attack success with attack budget = 0.01.
`kreuk003`	*Kreuk* attack success with attack budget = 0.03.
`kreuk005`	*Kreuk* attack success with attack budget = 0.05.
`kreukmean`	Mean *Kreuk* attack success with attack budgets $\in \{0.01, 0.03, 0.05\}$.
`thresh`	Threshold for the model's output at the 0.1% false positive rate.
`TPR`	True positive rate at 0.1% false positive rate.

Table 3: Robustness proxies used to predict *IPR* success, *Disp* success, and robustness (i.e., 1−highest attack success), separated by those obtained using ML attributions, *Kreuk* attacks, and test set performance.

The two attribution methods we use are the gradients with respect to the input, and Integrated Gradients (IG) [42]. Each of these attribution methods can be used to give each byte in a binary a score on how important the byte is to the model's prediction, and toward which class (malware or benign) the byte was swaying the model. We aggregate these scores in different ways to compute a single score for the model.

The aggregation methods we use include:

- L_p-norms ($p = 2, \infty$): a scalar value calculated by taking the L_p-norm of the entire vector of attribution scores for all bytes in the binary;
- statistical measures: the mean and standard deviation of the attribution scores for all bytes in the binary;
- smoothness: the mean of the absolute value of the difference between each byte and its direct neighbor;
- Necessity and Sufficiency Ordering (N-Ord and S-Ord [47]): N-Ord is the area under (or over) the maliciousness curve created when iteratively masking out the most important bytes, as determined by the attribution score for each byte. S-Ord works the same way, except byte values are iteratively added to a baseline input instead of masked out.

Using simpler and faster attacks to estimate robustness. As shown in previous work, a model robust to one type of attack (e.g., *Disp*) may also be robust to another type of attack (e.g, *Kreuk*) [27]. Because *Kreuk* attacks can be completed with much less computation, as it does not consider the validity of the executable bytes and instead simply appends non-executable evasive bytes to the end of the binary, we use *Kreuk* attack success as a robustness proxy to estimate the robustness of a detector.

Using the model's performance on the test set to estimate robustness. Finally, we also use the model's natural accuracy and maliciousness threshold at 0.1% false positive rate as robustness proxies. The full list of robustness proxy measures is in Table 3.

5.1.2 Predicting Robustness. First, we define how we calculate *robustness*. To simplify comparisons between models that could have different levels of resistance to different types of attacks (e.g., *Disp*, *IPR*, *Kreuk*) and to ensure robustness is a measure of how well a model can withstand evasion attacks, we define the *robustness* of a model as one minus the attack success rate of the most successful attack against that model (as introduced in Sec. 4.2.1).

With this in mind, we attempt to predict this value using the proxy measures described above. We first select the robustness proxies that give us the best performance using a feature-selection algorithm, Sequential Feature Selection [1]. This algorithm first selects the one feature that provides the best performance when used to predict robustness (i.e., when used by some predictor, like a linear regressor), then adds the feature that provides the best performance when used with the previously selected features, and so on. This process continues until the performance of the robustness predictor does not improve with the addition of any of the remaining features. We use this algorithm to ensure that every robustness proxy selected has a demonstrated positive effect on prediction performance. Then, we train a predictor to predict the robustness of a model given the selected robustness proxies.

To measure prediction performance, we use the standard Pearson r correlation coefficient [16]. In our case, r measures the linear correlation between predicted robustness (or an attack's success rate) and the true robustness (or an attack's success rate). We experimented with the best performing types of prediction models for small datasets [16]. We decided to use linear regression because it is simple and interpretable, and because it performed as well as any other predictor type we tried. To train our linear regressor, we use `scikit-learn`'s `ElasticNetCV` with default parameters [32]. This regressor is trained with both L_1 and L_2 regularization, and we use cross validation (across different splits of the training data) to determine the best regularization parameters.

Using this methodology, we execute experiments to predict the robustness of malware detectors given three scenarios in Sec. 5.2.

5.2 Robustness Proxies Results

As discussed in Sec. 5.1, robustness proxies allow us to predict the robustness of a model without having to execute expensive attacks against it. Sec. 5.2.1 describes how we measure the performance of the robustness proxies in predicting robustness of, and *IPR* and *Disp* attack success against, the *GreedyBlock*-trained models described in Sec. 4.1. In Sec. 5.2.2, we then analyze three different ways, referred to as scenarios, in which robustness proxies can be used to find the most robust models in the pool, and we report the portion of the pool we could avoid evaluating with (expensive) attacks if we stopped evaluating models after finding the most robust *GreedyBlock* model found in Sec. 4.

5.2.1 Evaluation Setup. The purpose of robustness proxies is to predict the robustness of a model to *IPR*, *Disp*, and *Kreuk* attacks without having to actually attack it with the expensive *IPR* and *Disp* attacks. We use Pearson's correlation coefficient to measure the correlation between predicted robustness (by proxies) and the measured robustness (against real attacks).

Without robustness proxies, all candidate models would have to be evaluated by expensive attacks. With robustness proxies, the models could be ordered according to their predicted robustness, and then evaluated in descending order (i.e., the model predicted to be most robust is evaluated first, etc.). We report the percentage of models we could theoretically avoid evaluating with expensive attacks if we stopped after finding the most robust model possible. Albeit, we would not know if we had found the most robust model without evaluating all models, but we provide this measure and describe its practicality in Sec. 5.2.2.

For example, if there are 100 malware detectors to evaluate in a pool, and if the predictor (derived from robustness proxies) predicts the most robust possible model in the pool as the second to evaluate, then we only needed to evaluate two models. Therefore, the portion of the pool we avoided attacking is $(100 - 2)/100 = 98\%$. We note that there are other ways of measuring the benefits of a predictor, but we chose this method as it can be directly related to how much computation it is possible to save.

5.2.2 Finding Robust GreedyBlock-trained Models Using Robustness Proxies. Now, we use several example scenarios to illustrate how robustness proxies can be used to find and evaluate the most robust models in a pool of trained models, created as described in Sec. 4.1.

To predict the robustness of a model, we need training data for our predictor, which requires attacking at least some models. The general process on how we do this is shown in Fig. 2. To illustrate how a balance can be struck between time spent getting training data and time saved using robustness proxies, we consider three scenarios. We treat each scenario as an independent method of using robustness proxies to evaluate a pool of malware detectors. Hence, we select features and train and evaluate a predictor separately for each scenario.

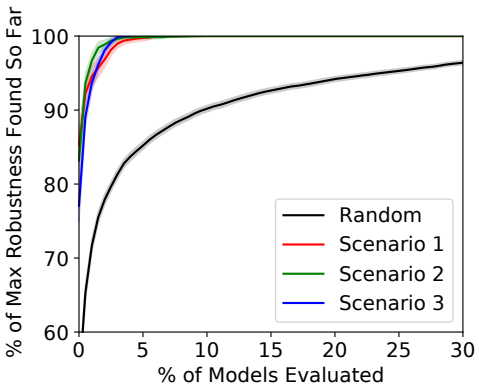


Figure 3: This plot compares the fraction of the model pool required to be evaluated to find the most robust model in each scenario in Sec. 5.2 compared to the baseline of evaluating all models in the pool in a random order.

Scenario 1: hyperparameter-sweep. The first scenario considers a simple training data selection method where we choose models trained with a subset of the hyperparameter values of the overall pool of models. Specifically, we select three out of the five hyperparameter values for each hyperparameter (enumerated in Sec. 4.1), and use all models trained with those hyperparameters as the predictor training set.

This gives us $3 \times 3 \times 3 \times 3 = 81$ models to train a predictor on. We then use the trained predictor to estimate the robustness of the remaining $375 - 81 = 294$ models.

We now discuss the robustness prediction results on these remaining 294 test models. As part of measuring robustness, Fig. 4a and Fig. 4b show how well the trained predictor predicts the success of *IPR* and *Disp*[6] attacks, respectively. Fig. 4c shows that using robustness proxies for robustness prediction successfully ranked the most robust model within the top 10% most robust in the pool (and therefore one of the first to evaluate). Fig. 4a and Fig. 4b further show that robustness proxies can predict *IPR* and *Disp* attack success.

Fig. 3 shows that, over 100 experiment trials (choosing different training data each time), using robustness proxies to determine the order of attacking models in scenario 1 resulted in finding the most robust model after attacking less than 10% of the test models in the pool.

Using the calculations described in Sec. 5.2.1, these results correspond to the ability to avoid attacking 95.9% of the models in the test pool in order to find the best possible model. The feature-selection method selected different individual robustness proxies to use in the predictors shown in each of the plots. The robustness proxies selected for the predictors in Fig. 4 are shown in Table 5 in App. B. Notably, some form of *Kreuk* attack success is chosen for every predictor, indicating that *Kreuk* attacks are a good indicator of *IPR* and *Disp* attack success, and of overall robustness.

[6]This *Disp* attack success is the mean of executing *Disp* attacks with budgets 0.01, 0.03, and 0.05, the same budgets previous work used [27, 28].

Scenario 2: admissibility. One issue with the hyperparameter-sweep scenario is that it does not account for models that are obviously inaccurate. For example, if the test accuracy of a model is around 0.5 (which matches the base rate of the dataset), then we know that the model has not learned anything useful. Not only would it be a waste of time to attack such models, but including them in the training data for predictors could be harmful, as these bad classifiers' gradients, attributions, etc. might not match the trends of better classifiers.

Therefore, we can save time by disregarding these models. In scenario 2, we define an *admissibility* criterion for models that we consider to be competitive in accuracy with previous state-of-the-art models [27]. Specifically, we consider a model to be admissible if it has a natural accuracy of at least 80% TPR at 0.1% FPR. With this criterion, only 46 of scenario 1's 81 models that used for training data would be admissible.

Out of the 375 models trained, we find that 246 are admissible. We randomly select 20% of these to be used as training data for our linear regressors, and predict the robustness of the remaining 80% of models. This gives us 49 models (20%) to attack with the expensive *IPR* and *Disp* attacks, and 197 models for which we estimate robustness in order to determine the evaluation order.

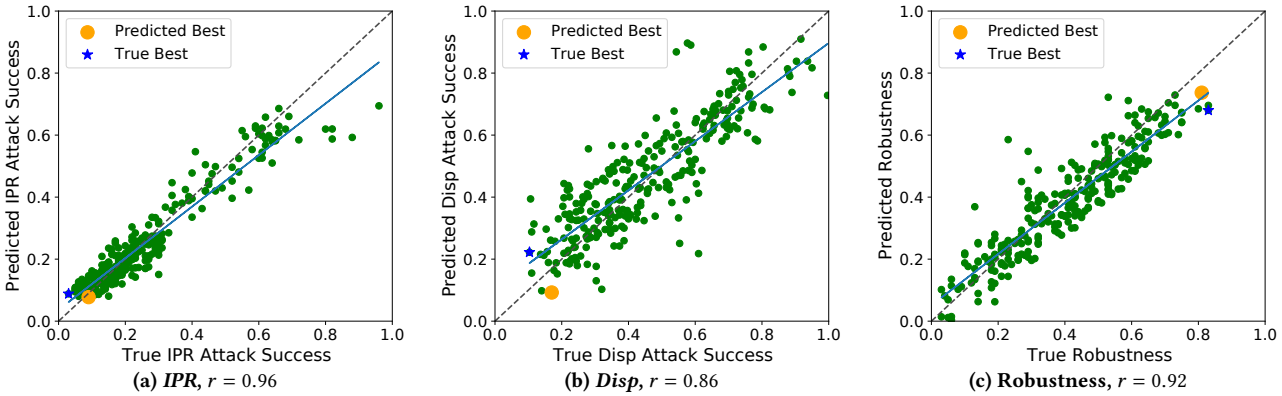
As with scenario 1, the robustness proxies chosen to predict *IPR* and *Disp* success and model robustness vary, as shown in Table 6. As in Sec. 5.2.2, some version of *Kreuk* attacks was chosen for every predictor.

Fig. 5 shows prediction results after training our linear regressor. We calculate that using robustness proxies to predict the order in which to evaluate models avoids attacking 98.9% of the models in the pool to find the most robust possible model. Also, as can be seen in Fig. 3, scenario 2 finds more robust models faster than scenario 1, likely because the admissibility criterion filters out obviously bad models that would otherwise be included in the training data.

Scenario 3: adaptive training. In the previous two scenarios, robustness proxies were used to determine the order of evaluating models. In scenario 3, we use robustness proxies to also adapt how we create the pool of malware detectors (originally described in Sec. 5.2.1) to save time there, as well.

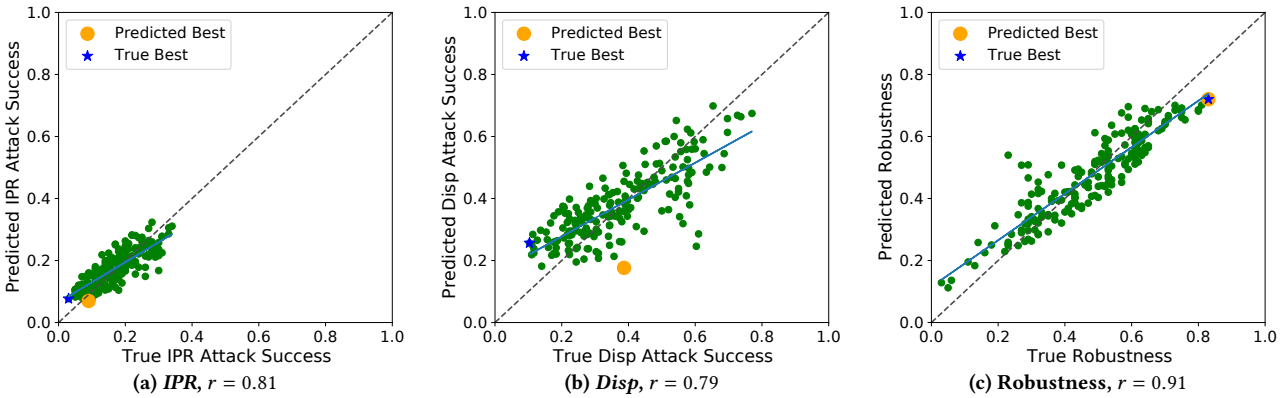
For this scenario, we combine the approaches of the previous two scenarios: First, we select a subset of hyperparameter values with which to train models. These resulting models are then used to calculate robustness proxies. Because this subset makes up 27 out of the 125 total hyperparameter configurations (from choosing three of the possible hyperparameter values for three different hyperparameters, as done in scenario 1, so $3^3 = 27$) we train for scenarios 1 and 2, this pool creation costs $\frac{27}{125} = 22\%$ of the initial malware detector pool creation cost of scenarios 1 and 2 (but the same amount of attack time). Then, we train a predictor from this training data and use it as an additional admissibility criterion to guide the training of the models corresponding to the remaining 98 hyperparameter configurations (configurations first described in scenario 1).

Specifically, we guide the training as follows. First, we train models using the 98 remaining hyperparameter configurations to only one epoch (rather than 3 epochs); we then decide whether to continue training them only if (1) they meet the admissibility



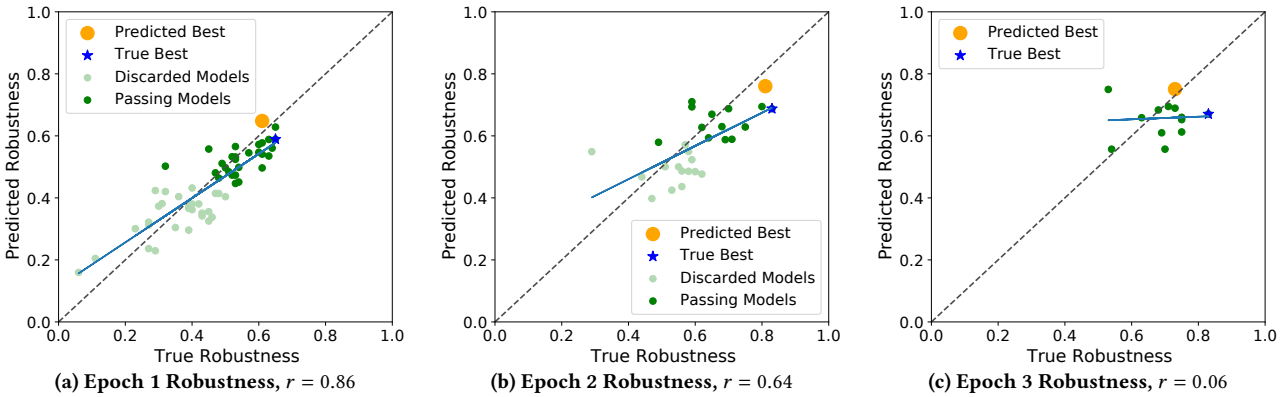
(a) *IPR*, $r = 0.96$ (b) *Disp*, $r = 0.86$ (c) **Robustness**, $r = 0.92$

Figure 4: Scenario 1: Robustness proxy predictors can predict the attack success of *IPR* and *Disp* attacks and robustness of a model when trained with evaluations of fewer models trained from a subset of hyperparameter values (Sec. 5.2.2). r is the Pearson correlation coefficient.



(a) *IPR*, $r = 0.81$ (b) *Disp*, $r = 0.79$ (c) **Robustness**, $r = 0.91$

Figure 5: Scenario 2: Robustness proxy linear regressors can predict the attack success of *IPR* and *Disp* attacks and robustness of a model when trained with evaluations of fewer models trained from a randomly selected subset of admissible models (Sec. 5.2.2). When these predictions are combined with *Kreuk* attack success, overall robustness of a model can be more accurately predicted.



(a) **Epoch 1 Robustness**, $r = 0.86$ (b) **Epoch 2 Robustness**, $r = 0.64$ (c) **Epoch 3 Robustness**, $r = 0.06$

Figure 6: Scenario 3: Robustness proxy linear regressors can be used to filter out a large portion of models that should not be trained further (Sec. 5.2.2). This progression of plots shows the increasing true robustness of models that are chosen to be trained further. Faded out green dots represent models discarded before training the next epoch. r is the correlation coefficient.

criterion from scenario 2 (at least 80% TPR at 0.1% FPR as described in Sec. 5.2.2), and (2) the robustness proxy predictor predicts the model is in the top 50% of the models by robustness. In other words, we stop training the less promising half of the candidate models (as well as any that did not meet the scenario 2 criterion) after one epoch. We repeat the filtering process after the second epoch, again discarding at least 50% of the models. Finally, we use the predictor to order models trained for 3 epochs by their predicted robustness.

Fig. 6 visualizes this process. Fig. 6a shows the first step of predicting the robustness of models after one epoch of training. The faded green dots represent the models/configurations that were not selected to be trained further based on the predictor's robustness estimation. 42 configurations were not selected for further training because they did not meet scenario 2's admissibility requirements (TPR > 80% at 0.1% FPR). 28 of the original 98 models/configurations survived the first round.

Fig. 6b shows the predicted and measured robustness of the remaining models after they are trained for another epoch. Of these, 12 models survived the next filtering step. Finally, Fig. 6c shows the final prediction of the surviving models' robustness after three epochs of training. The blue star, representing the model *empirically measured* (using real attacks) to be most robust in each stage, easily makes the cut for further training, as shown in Fig. 6a and Fig. 6b.

Similar to the previous scenarios, scenario 3 shows that substantial effort can be saved by using robustness proxies to help identify the most robust model. Moreover, this scenario shows that we can save time not only when evaluating trained models but also by training *fewer* models to begin with. Specifically, we trained 27 configurations to three epochs (for training data), 70 models to one epoch (as they were discarded after the first stage (Fig. 6a)), 16 to two epochs (discarded after the second stage (Fig. 6b)), and the remaining 12 configurations to three epochs. This amounts to a total of $27\times3+70\times1+16\times2+12\times3 = 219$ epochs of training, saving ~40% of the training time (375 epochs) compared to scenarios 1 and 2, even when accounting for the 27 initial models' training time.

Unlike with scenarios 1 and 2, the same trained predictor was used for each of the plots in Fig. 6. The robustness proxies chosen to predict robustness included `grad_l2_norm`, `grad_nord`, `grad_mean`, `IG_sord`, `kreuk001`, `kreuk005`, `thresh`, and `TPR`.

Scenarios 1–3 show that robustness proxies can be used to avoid attacking a substantial portion of the model pool when searching for the most robust model. The next section compares these predicted-to-be-robust models to the most robust possible model found in Sec. 4, demonstrating that we can reduce the number of models needed to be attacked and/or trained while retaining the ability to find robust models.

Performance of GreedyBlock models predicted to be most robust. Table 4 shows the robustness of the most robust model you would find if you only attacked the top 10% of models predicted to be most robust in each scenario. As shown in Fig. 3, the *GreedyBlock*-trained models found in these scenarios match the robustness of the best *GreedyBlock* model found in Sec. 4. In other words, this means that in each scenario, we almost always find the best possible model by only attacking the top 10% of models predicted to be most robust. Additionally, Table 4 also quantifies how many fewer epochs of *GreedyBlock*-training and how many fewer models need to be attacked in each scenario to achieve these results.

Search Method	Robustness Found ↑	Compute Required: Training (# epochs) ↓	Compute Required: Attacking (# models) ↓
Best possible	**0.83**	375	375
Scenario 1	**0.83**	375	110
Scenario 2	**0.83**	375	69
Scenario 3	**0.83**	219	50

Table 4: Robustness and compute requirements of finding the best *GreedyBlock* model in entire pool (as done in Sec. 4) vs the best model found in top 10% predicted robust models in scenarios 1, 2, and 3 (almost always finds the best model). Arrows show if lower/higher is better.

6 Discussion

This section discusses some benefits and limitations of our work.

6.1 Faster Training and Evaluation Benefits

Our results show that this work's methods of training and evaluating robust malware detectors can save a significant amount of computation. While this can be useful in speeding up a robust model, there are other benefits to faster training and evaluation. For example, reduced computation requirements lowers monetary cost, power consumption, and carbon emissions [6]. Also, faster retraining increases the practicality of staying accurate on the latest binaries by keeping up with data drift, an issue identified in previous work [10].

Additionally, the models described in prior work require a scaled-out infrastructure to generate enough adversarial examples, which is costly [27]. In fact, ML models are growing more expensive to train across multiple domains in general [6]. In contrast, our training method can be completed on a single server with a CPU. This significant reduction in hardware requirements and thus cost democratizes the ability to train robust raw-binary malware detectors for use in companies, schools, and cases where there is a lack of resources to train on a cluster of servers.

6.2 Limitations

While we put considerable effort into making our methods and results strong and useful to the scientific community and cybersecurity practitioners, we acknowledge that there are limitations to our work and findings.

While *VTFeed* is a dataset of binaries used in previous works [20, 27, 28] constructed from a direct feed of real-world binaries being analyzed [11], it is still a single dataset, which brings into question our results' generalizability to other datasets. However, the results reported in previous work [28], derived primarily using the *VTFeed* dataset, were nearly the same as results reported in a previous version of the work that used a different compiled binary dataset [37]. This implies these techniques and trends persist across different compiled binary datasets. Moreover, some of the base techniques

we used for *GreedyBlock*-training and the robustness proxies, such as IG [42], N-Ord, S-Ord [47], selecting the most important places in an input sample to modify [49], and selecting the most evasive byte values to use in an attack [26], were all developed and tested on non-*VTFeed* datasets and in different domains [42, 47, 49]. Therefore, we expect these techniques to generalize to other datasets.

Furthermore, robustness proxies are not tied to any specific attack types. Their weighting is adjusted *during predictor training* to best predict the success of the specific attacks that are executed during this process. In our experiments in Sec. 5.2.2, robustness proxies are predicting robustness against a suite of attacks (*IPR*, *Disp*, and *Kreuk*).

We also acknowledge there could be other robustness proxies yet to be discovered that are more predictive than those we have proposed. Additionally, there could be other methods of using robustness proxies to better select the most robust models or guide the training process (e.g., pair-wise hypothesis tests). Nevertheless, we believe these would be complementary advances used in conjunction with the ones proposed in this paper.

7 Conclusion

This paper has presented new methods for training and evaluating raw-binary malware detectors robust to evasion attacks. We first showed that our best *GreedyBlock*-trained model is more robust to attacks, more accurate, and trained faster than the state of the art adversarially trained models (Sec. 4.2). Then, in two separate scenarios, we showed that robustness proxies can predict the robustness of a model, allowing us to avoid attacking over 90% of unseen models when searching for the most robust model in a pool, a marked improvement over attacking all unseen models. Finally, in a third scenario, we showed that these robustness predictions can also guide the training process, saving even more computation without sacrificing the final models' robustness (Sec. 5.2). We expect that these methods will help make robust raw-binary malware detectors more accessible to the computer security community.

Acknowledgments

This work was supported in part by U.S. Army Research Office under MURI grant W911NF2110317; by National Science Foundation awards 2338301 and 2338302; by Intel with a Rising Star Faculty Award; by a Maof prize for outstanding young scientists; by the Ministry of Innovation, Science & Technology, Israel under grant 0603870071; by the United States-Israel Binational Science Foundation under grant 2023641; and by the Defence Science and Technology Agency, Singapore.

References

[1] D. W. Aha and R. L. Bankert. A comparative evaluation of sequential feature selection algorithms. In *Proc. AISTATS*, 1995.
[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*, 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] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proc. FAccT*, 2021.
[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] Y. Chen, Z. Ding, and D. Wagner. Continuous learning for android malware detection. In *Proc. USENIX Security*, 2023.
[11] Chronicle. Virustotal. https://www.virustotal.com/, 2004–. Accessed 6/17/2019.
[12] A. Datta, M. Fredrikson, K. Leino, K. Lu, S. Sen, and Z. Wang. Machine learning explainability and robustness: Connected at the hip. In *Proc. KDD*, 2021.
[13] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. In *Proc. ITASEC*, 2019.
[14] B. G. Doan, S. Yang, P. Montague, O. De Vel, T. Abraham, S. Camtepe, S. S. Kanhere, E. Abbasnejad, and D. C. Ranasinghe. Feature-space bayesian adversarial learning improved malware detector robustness. In *Proc. AAAI*, 2023.
[15] 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.
[16] M. Fernández-Delgado, M. Sirsat, E. Cernadas, S. Alawadi, S. Barro, and M. Febrero-Bande. An extensive experimental survey of regression methods. *Neural Networks*, 111:11–34, 2019.
[17] M. Galovič, B. Bosanský, and V. Lisý. Improving robustness of malware classifiers using adversarial strings generated from perturbed latent representations. In *Proc. NeurIPSW*, 2021.
[18] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *Proc. ICLR*, 2015.
[19] J. Heo, S. Joo, and T. Moon. Fooling neural network interpretations via adversarial model manipulation. In *Proc. NeurIPS*, 2019.
[20] Z. Huang, N. G. Marchant, K. Lucas, L. Bauer, O. Ohrimenko, and B. I. P. Rubinstein. Rs-del: Edit distance robustness certificates for sequence classifiers via randomized deletion. In *Proc. NeurIPS*, 2023.
[21] I. Incer, M. Theodorides, S. Afroz, and D. Wagner. Adversarially robust malware detection using monotonic classification. In *Proc. IWSPA*, 2018.
[22] 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.
[23] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 2006.
[24] H. Koo and M. Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proc. AsiaCCS*, 2016.
[25] 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.
[26] 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.
[27] K. Lucas, S. Pai, W. Lin, L. Bauer, M. K. Reiter, and M. Sharif. Adversarial training for raw-binary malware classifiers. In *Proc. USENIX Security*, 2023.
[28] 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.
[29] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *Proc. ICLR*, 2018.
[30] 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.
[31] 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.
[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
[33] 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.
[34] 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.
[35] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo. Data mining methods for detection of new malicious executables. In *Proc. IEEE S&P*, 2001.
[36] 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.
[37] M. Sharif, K. Lucas, L. Bauer, M. K. Reiter, and S. Shintre. Optimization-guided binary diversification to mislead neural networks for malware detection. *arXiv preprint*, 2019.
[38] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Proc. ICLRW*, 2014.

[39] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014.
[40] O. Suciu, S. E. Coull, and J. Johns. Exploring adversarial examples in malware detection. In *Proc. AAAIW*, 2018.
[41] R. Sun, M. Xue, G. Tyson, T. Dong, S. Li, S. Wang, H. Zhu, S. Camtepe, and S. Nepal. Mate! are you really aware? an explainability-guided testing framework for robustness of malware detectors. In *Proc. ESEC/FSE*, 2023.
[42] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. *Proc. ICML*, 2017.
[43] 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.
[44] 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.
[45] Z. Wang. *On the Feature Alignment of Deep Vision Models: Explainability and Robustness Connected At Hip.* PhD thesis, Carnegie Mellon University, 2023.
[46] Z. Wang, M. Fredrikson, and A. Datta. Robust models are more interpretable because attributions look normal. In *Proc. ICML*, 2021.
[47] Z. Wang, P. Mardziel, A. Datta, and M. Fredrikson. Interpreting interpretations: Organizing attribution methods by criteria. In *Proc. CVPRW*, 2020.
[48] E. Wong, L. Rice, and J. Z. Kolter. Fast is better than free: Revisiting adversarial training. In *Proc. ICLR*, 2020.
[49] 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.
[50] X. Zhang, N. Wang, H. Shen, S. Ji, X. Luo, and T. Wang. Interpretable deep learning under fire. In *Proc. USENIX Security*, 2020.

A Performance on *AvastNet*

We conduct smaller-scale experiments to verify that our methods work on another malware detector architecture, *AvastNet* [25].

A.1 *GreedyBlock* Performance on *AvastNet*

To verify *GreedyBlock*-training worked on other malware detector architectures, we also trained a smaller pool of models using the *AvastNet* architecture [25], where *GreedyBlock*-training also outperformed adversarial training.

As shown in Table 7, using *GreedyBlock*-training on *AvastNet* [25] as the base model can produce malware detectors with higher robustness and natural accuracy than adversarially trained models that used *IPR* and *Disp* attacks reported in prior work [27]. Specifically, *GreedyBlock*-training, applied in the same way as discussed earlier in this section produced a model with a robustness of 0.71 and a natural accuracy of 0.96 TPR at 0.1% FPR, while the most robust adversarially trained model had a robustness of 0.01 (due to being vulnerable to *Kreuk* attacks) and a natural accuracy of 0.91 TPR at 0.1% FPR.

A.2 Robustness Proxies on *AvastNet*

We also evaluated the performance of the robustness proxies on the *AvastNet* [25] architecture via a smaller-scale experiment. The results are as strong as for *MalConv*–the *AvastNet* model predicted as the most robust by proxies was also the most robust against *IPR* and *Disp* attacks. We trained with *GreedyBlock* and *IPR*- and *Disp*-attacked 101 *AvastNet* detectors, and tested robustness proxies with scenario 2 (i.e., admissibility criteria removes low-performing models, predictor trained on 20% subset, robustness predicted for the rest–Sec. 5.2.2). As shown in Fig. 9, robustness proxies' predictions achieved a correlation coefficient of 0.83 with the observed robustness, and the model predicted as most robust was also identified as such by attacks against all models.

Robustness Proxy	*IPR*	*Disp*	Robustness
grad_linf_norm	✓	✓	✓
grad_nord		✓	✓
grad_sord		✓	✓
grad_mean	✓		✓
IG_l2_norm	✓		
IG_linf_norm	✓		✓
IG_nord	✓	✓	
IG_sord	✓	✓	✓
kreuk001	✓	✓	✓
kreuk003	✓	✓	✓
kreuk005		✓	✓
kreukmean	✓	✓	
thresh	✓	✓	✓
TPR	✓	✓	✓

Table 5: Robustness proxies selected to predict *IPR* success, *Disp* success, and robustness in Sec. 5.2.2.

Robustness Proxy	*IPR*	*Disp*	Robustness
grad_l2_norm	✓		
grad_linf_norm			✓
grad_nord		✓	✓
grad_sord		✓	✓
grad_mean			✓
IG_l2_norm	✓		✓
IG_linf_norm	✓	✓	✓
IG_nord		✓	✓
IG_sord		✓	✓
IG_mean		✓	
kreuk001	✓	✓	✓
kreuk003		✓	
kreuk005		✓	✓
kreukmean		✓	
thresh	✓		
TPR	✓	✓	

Table 6: Robustness proxies selected to predict *IPR* success, *Disp* success, and robustness in Sec. 5.2.2.

B Robustness Proxies Selected

This section outlines which robustness proxies were selected to predict *IPR* attack success, *Disp* attack success, and robustness in scenarios 1 and 2 described in Sec. 5.2. Scenario 1's selected proxies are shown in Table 5, while scenario 2's are shown in Table 6.

C Few-byte Attacks

Because *GreedyBlock*-training augmented binaries by adding blocks of multiple evasive bytes, we ensured that this did not inadvertently make the models more vulnerable to attacks that only modify a few bytes. To do this, we attacked both the original model and best *GreedyBlock* model with binaries in which the most important 1 to 5 bytes (according to Integrated Gradients [42]) were changed

Training Approach	Attack Success Rate (ASR)							Robustness ↑	TPR ↑	Time (s) ↓
	IPR ↓	*Disp*-0.01 ↓	-0.03 ↓	-0.05 ↓	*Kreuk*-0.01 ↓	-0.03 ↓	-0.05 ↓			
Original	0.47	0.89	0.96	0.98	0.98	1.00	1.00	0.00	0.95	–
IPR-training	0.09	0.82	0.88	0.94	0.97	0.98	1.0	0.00	0.95	13920K
Disp-training	0.26	0.12	0.10	0.12	0.87	0.96	0.99	0.01	0.91	6060K
Best *GreedyBlock*	**0.08**	**0.05**	**0.09**	**0.08**	**0.07**	**0.10**	**0.12**	**0.88**	**0.96**	**3213K**

Table 7: Comparison of prior work's trained models on *AvastNet* [25, 27] vs. the best *GreedyBlock*-trained models in attack success, robustness, and training time. Column title arrows show if lower or higher is better.

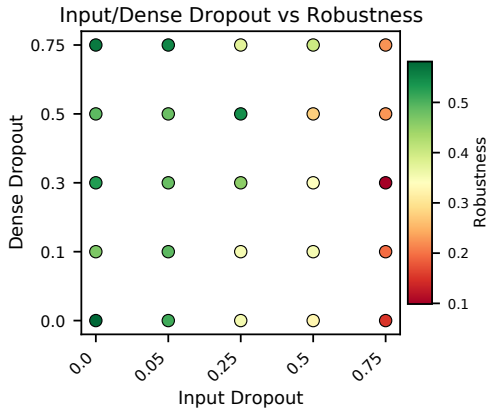


Figure 8: This plot shows the relationship between dropout rate in *GreedyBlock* training and the resultant robustness.

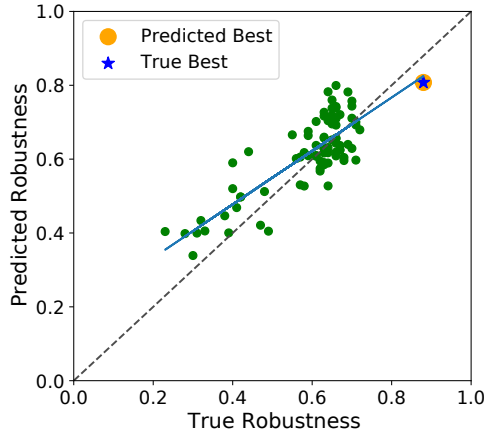


Figure 9: *AvastNet* model robustness prediction using scenario 2: robustness proxies can predict model robustness when trained with evaluations of fewer randomly selected admissible models (Sec. 5.2.2); $r = 0.83$.

to the most evasive byte values (to the targeted detector). Of 214 such attacks, 1 evaded the original model, and none evaded the GreedyBlock-trained model. Based on these results, we conclude that *GreedyBlock*-training does not make the model more vulnerable to few-byte attacks.

D *GreedyBlock* Parameter Analysis

To identify whether some parameter values tended to produce more robust models, we plotted the robustness of our models trained with different *GreedyBlock* parameters in Fig. 7 and Fig. 8. Overall, the most notable trends are that models trained with a *Disp* budget of 0.0 have a low robustness (as expected as this is equivalent to no *GreedyBlock*-training), and that large values of input dropout tend to result in lower robustness.

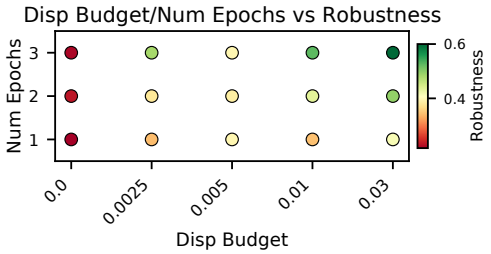


Figure 7: This plot shows the relationship between *Disp* budget and number of epochs in *GreedyBlock* training and the resultant robustness.