On the Robustness of Malware Detectors to Adversarial Samples

Muhammad Salman, Benjamin Zi Hao Zhao, Hassan Jameel Asghar, Muhammad Ikram, Sidharth Kaushik, Mohamed Ali Kaafar

Macquarie University, Sydney, Australia {muhammad.salman, ben_zi.zhao, hassan.asghar, muhammad.ikram, sidharth.kaushik, dali.kaafar}@mq.edu.au

Abstract. Adversarial examples add imperceptible alterations to inputs with the objective to induce misclassification in machine learning models. They have been demonstrated to pose significant challenges in domains like image classification, with results showing that an adversarially perturbed image to evade detection against one classifier is most likely transferable to other classifiers. Adversarial examples have also been studied in malware analysis. Unlike images, program binaries cannot be arbitrarily perturbed without rendering them non-functional. Due to the difficulty of crafting adversarial program binaries, there is no consensus on the transferability of adversarially perturbed programs to different detectors. In this work, we explore the robustness of malware detectors against adversarially perturbed malware. We investigate the transferability of adversarial attacks developed against one detector, against other machine learning-based malware detectors with different feature space, and code similarity techniques, specifically, locality sensitive hashing-based detectors. Our analysis reveals that adversarial program binaries crafted for one detector are generally less effective against others. We also evaluate an ensemble of detectors and show that they can potentially mitigate the impact of adversarial program binaries. Finally, we demonstrate that substantial program changes made to evade detection may result in the transformation technique being identified, implying that the adversary must make minimal changes to the program binary.

1 Introduction

Adversarial examples are inputs with slight, imperceptible modifications that induce misclassification in machine learning models. The efficacy of this attack has been well documented in the domain of image classification where perturbed images that are otherwise human-recognizable are misclassified by victim machine learning models [1,2]. A series of recent works have also highlighted this vulnerability in machine learning-based malware detection [3,4,5,6,7]. A key difference between the two domains is that while slight changes to an image, e.g., by injecting white noise, still deems it recognizable (by a human), arbitrary changes to a program can render it dysfunctional. One major consequence of this difference in the study of adversarial examples is that while in the domain

of images, adversarial example attacks are largely transferable [5], meaning adversarial examples crafted for a target machine learning model can be used to fool a different machine learning model; Some success in transferring malware has been demonstrated [7], however, this is conditional on similarities between the model's feature space. However, when the model's feature space varies, this is largely false in the malware detection domain [6].

This lack of transferability has profound implications for adversarial example attacks in malware. For instance, real-world systems rarely rely on a single malware detector. Consider, for instance, the VirusTotal service.[1] Given a program binary or its cryptographic hash, it returns a detailed report on the malicious nature of the program. Included in this report are its cryptographic hash (e.g., SHA-256), locality sensitive hash (LSH) digests (e.g., `ssdeep` [8]), as well as detection results from various security vendors (scanners) some of whom employ machine learning-based detectors [9]. Thus, a malware program adversarially perturbed to evade a particular target machine learning model will not evade detection against an ensemble of detectors. To understand why an adversarial example attack on a machine learning model may not be transferable to a machine learning-based or non-machine learning-based detector consider an LSH algorithm for malware detection. Roughly, such an algorithm divides its input into blocks, computes the hash of each block and concatenates the results to create the final digest. Thus, if a certain block is unchanged, the hash of that block will remain the same. One of the techniques to create adversarial examples is to append code to the end of the program [10]. Since this is only added to the end of the program, the LSH-based detection scheme will show the same digest for the blocks of the program before the appended part, and hence the program will still be detected as malware.

A naive way to solve this issue, from the attacker's perspective, would be to completely obfuscate the program, using packers for example. However, the presence of obfuscation may itself result in the program being flagged as potential malware [11, §5]. Indeed, packing with popular packers increases the likelihood of being detected [9, §4.4]. Also, obfuscation may also induce large changes. Too many changes are not ideal as they would create new signatures that could be used to identify the technique used to perturb a program [12, §4]. Therefore, in the spirit of adversarial examples in the image domain, it may make sense to only consider the attack potent if the amount of changes made to the code is minimal.

In this paper, we empirically investigate these assertions. More concretely, our contributions are as follows:

– We empirically demonstrate that an adversarial example attack targeted on one detector is not in general transferable to another when the feature space is different. To demonstrate this, we take a recently proposed framework called malware makeover [9] and use it to transform program binaries to evade detection against a deep neural network (DNN) based model trained

[1] See https://www.virustotal.com/

on raw bytes (MalConv [13]), as is done in [9]. We then test these transformed binaries against the original model, i.e., MalConv, an `ssdeep` detector (a commonly used LSH algorithm), and a random forest (RF) based detector trained on the EMBER dataset [14] with features (known as EMBER feature) via LIEF [15], a set of features automatically extracted from program binaries. Our results show that the Malconv targeted transformed binaries are poor in evading detection against the other detectors. This is rather surprising, since as we show in Section 2.3, evading `ssdeep` detection is not difficult. We repeat the same procedure by taking these other detectors as the target for transforming programs and show that results stay the same.

– We then investigate whether an ensemble of detectors is capable of mitigating the impact of adversarially transformed binaries, as the above conclusion may imply, using different decision rules: minority, majority, and consensus. We include all three detectors in the ensemble and report the impact on the true/false positive rates of the original binaries, and the impact on these rates if the binaries are transformed taking one of these detectors as the target detector to create adversarial samples. Our results indicate that a majority voting strategy is capable of substantially mitigating this adversarial attack since its success rate is significantly lower on non-target detectors. We show that this conclusion is also backed by results from the VirusTotal service.
– We demonstrate that substantial program changes create a new signature. This has been raised as a possible issue with transformation techniques that change the program too much [12, §4]. We confirm this by training a model to distinguish between original and modified binaries using the malware makeover technique [9]. This implies that an adversary aiming to evade an ensemble of detectors would need to make significant changes, leading to detection as heavily obfuscated.

2 Preliminaries

2.1 Definitions

We assume that we have a set $D = B \cup M$ of benign B and malware M programs and their labels. Each program $x \in D$ has the label 0 if $x \in B$, else it has the label 1. The two subsets B and M of D are disjoint.

True and False Positives. Given a malware detector A, we define its type-I (false positive) and type-II (false negative) error rates as:

$$\text{FPR}_A = \frac{|A(x) = 1 : x \in B|}{|B|}, \ \text{FNR}_A = \frac{|A(x) = 0 : x \in M|}{|M|}$$

Upon applying an evasive technique $\mathcal{E}$ on the programs in D, we define the same types of errors as:

$$\text{FPR}_A^{\mathcal{E}} = \frac{|A(x') = 1 : x' \leftarrow \mathcal{E}(x), x \in B|}{|B|}, \text{FNR}_A^{\mathcal{E}} = \frac{|A(x') = 0 : x' \leftarrow \mathcal{E}(x), x \in M|}{|M|}$$

The changes in FPR and FNR measure the evasiveness of the technique. Instead of FNR, we shall report the true positive rate (TPR), i.e., TPR $= 1 -$ FNR.

Program Similarity. We take the definition of binary code similarity from [16]. That is, two binaries are similar if compiled from two similar source codes. We are also interested in the similarity of the original program x and its evasive counterpart $x' \leftarrow \mathcal{E}(x)$. This similarity is captured by a distance metric μ_0. We shall use the normalized Levenstein distance (NLD) as this metric [17].

Machine Learning vs Hash-based Detection. For machine learning-based detection, we assume a binary classifier f, which takes a program binary x as input and outputs 1 or 0, indicating malware or benign, respectively. In contrast, for hash-based detection, we assume a signature-based scheme. One such family of hash functions is locality-sensitive hashing (LSH). The hash-based detection scheme is endowed with the triplet (H, μ, θ), respectively, a hash function, a metric on the hashes, and a threshold. The metric μ on the hashes takes as input two hashes $h' = H(x')$ and $h'' = H(x'')$ and outputs $\mu(h', h'')$. Given a hash h and a set of signatures S, we define $\text{dist}_\mu(h, S) = \min_{h' \in S} \mu(h, h')$. The hash-based scheme works as follows:

1. During training, for each $x \in D$, i.e., all programs in the dataset, $H(x)$ is calculated and stored it in the database S. These are called *signatures*.
2. Given a program x (unknown if malware/benign), it calculates its hash $H(x)$.
3. Compute $d = \text{dist}_\mu(h, S)$. If $d \leq \theta$, output 1(malware), else output 0(benign).

In the one extreme, h could be a cryptographic hash function, in which case we can set $\theta = 0$, and the program outputs 1 if the hash is identical to some signature in the database. Note that the distance function in the hash domain may not be the same metric used to calculate the similarity of the original programs. Depending on the situation, multiple programs may pass the test of Step 3. In this case, we only consider the detector as successful if the only program passing test of Step 3 when given $h = H(\mathcal{E}(x))$ is x, i.e., the original program.

Threat Model. The adversary aims to modify a program so it is misclassified by the target detector while retaining functionality. As discussed in the introduction, the transformed program must remain similar to the original in terms of edit distance, measured by the NLD metric μ_0 above. Given this, the goal of the adversary is as follows: Given the set $D = B \cup M$ of benign and malware programs, a detector A with false positive and true positive rates of FPR_A and TPR_A, respectively, a distance metric μ_0 and a distance threshold θ_0, construct an evasive technique $\mathcal{E}$ such that (1) For all $x \in D$, $\mu_0(x, x') \leq \theta_0$, where $x' \leftarrow \mathcal{E}(x)$. (2) $\text{FPR}_A^{\mathcal{E}} > \text{FPR}_A$ and $\text{TPR}_A^{\mathcal{E}} < \text{TPR}_A$.

2.2 Malware Makeover and Extension

We first describe the algorithm behind the malware makeover attack from [9].

White-Box Attack. The white-box malware makeover attack targets a machine learning detector that uses gradient descent for minimizing its loss, e.g., a neural network. The white-box attack transforms a binary $\mathbf{b}$ to produce a new binary $\mathbf{b}'$ by introducing changes that increase the loss of the detector. These changes are guided by the gradient of the input features, aiming to misclassify

the binary $\mathbf{b}'$. The algorithm stops when the loss becomes high enough for the detector to misclassify the binary $\mathbf{b}'$.

Black-Box Attack. The black-box attacks involve manipulating the input binary $\mathbf{b}$ to decrease the probability of the correct label without computing the gradient. By rearranging the loss function, the attacker aims to produce a new binary $\mathbf{b}'$ by finding perturbations that lead to a decrease in the probability of the correct label, ultimately causing misclassification. Further details on the white-box and black-box attacks from [9] are given in Appendix A.

Transformations: Malware Makeover utilizes two transformation families initially designed to secure program binaries and prevent code-reuse attacks. The first introduces in-place randomization (IPR) [18], preserving binary functionality while obscuring its structure from attackers. Lucas et al. [9] enhanced the IPR approach to generate a wider range of functionally equivalent binaries by allowing iterative transformations, conservatively applying changes, and expanding supported instructions and conventions. A simple example of an IPR transformation (substituting equivalent instruction) is [9]: `add ebx 0x10 --> sub ebx, -0x10`.

The second family of transformations, derived from Displacement (Disp) [19], relocates code segments to a new section, replacing them with a jump instruction and updating relative addresses to preserve functionality. Lucas et al. [9] extended Disp by allowing the displacement of any consecutive instructions and replacing them with semantic no-operations (nops) that do not affect memory or registers [20]. To keep the binary size within limits (e.g., 1% over the original), displacement is constrained by a byte budget. The number of functions is divided by this budget, moving that many bytes per function. We build on the transformation framework and codebase from Malware Makeover [21], including its *private* version–acquired direct correspondence with the authors of [9]. Our extensions to this codebase are detailed in Section 3.3.

Excluding Headers: We exclude the header in the training and classification of binaries for our malware detection model, inspired by the malware makeover approach [9], which also removes headers. This is due to the finding in [22] that deep neural networks (DNNs) often rely on header characteristics for classification, rather than on meaningful features from the data and text sections where malware typically resides. This allows adversaries to evade detection with minimal header modifications, highlighting the need for a model that focuses on substantive program features.

2.3 Locality Sensitive Hashing and Adversarial Example Attack

Locality sensitive hash (LSH) functions like `ssdeep` operate differently from cryptographic hashes by applying a hash on disjoint input chunks, maintaining similar digests unless many chunks change. Given a string s, the `ssdeep` algorithm uses a rolling window of a specified length (in terms of bytes). At each byte sequence covered by the rolling window, it computes a rolling hash. If the rolling hash of the current byte sequence (determined by the window) is equal to $-1 \pmod{b}$, it triggers the algorithm to compute the cryptographic hash of the current chunk, i.e., all byte sequences until now. The 6 least significant bits

of this hash are retained. These bits are converted into a Base64 representation, resulting in the Base64 hash character of the `ssdeep` hash of the current chunk. The window then moves to the next byte sequence until the next "trigger point" is reached, and the next Base64 hash character is appended to the previous hash character. A total of 64 chunks are used in the default setting. The modulus b is referred to as the block size and satisfies the approximate relation: $b \approx \frac{n}{S}$, where n is the file size in bytes, and $S = 64$ denotes the number of chunks [23,24]. If the number of chunks is fewer than $S/2 - 1 = 31$, the algorithm sets the block size to $b/2$, and the process is repeated. Below is an example of the `ssdeep` hash of a string of length 500:

```
12:+xK3x+44q49oMeCsavrOyp9YQRHqHsaQCvuvhn+QyVyjfA5:j3X4q4DeCsaviypeQRH4LQFcQyj
```

The last substring starting from the '`:`' is the `ssdeep` hash with the block size $12 \times 2 = 24$. Since the number of Base64 characters is less than 31, the algorithm sets the block size to 12 (the integer at the start of the hash) and computes the hash shown between the two `:`'s. In Appendix B, we examine an attack on `ssdeep` documented in [24], where the input is natural language text and involves altering one byte in each chunk.

2.4 EMBER Features

For one of our detectors, we use the EMBER feature set, which is a set of engineered features extracted from software binaries commonly found in benchmark evaluation datasets first introduced by EMBER [14] and reinforced by SOREL [25], containing 900K and 20M software binaries respectively. The EMBER features are structured into nine distinct feature categories, encapsulating both parsed heuristics and format-agnostic statistical measures. To ensure that the features are not extracted from the header of the binary file (see Section 2.2), we only extract features from the data and text sections. Thus, we only retain ByteHistogram, ByteEntropy, StringExtractor, and SectionInfo features, excluding GeneralFileInfo, HeaderFileInfo, ImportsInfo, ExportsInfo, and DataDirectories. Interested readers are referred to [14] for a detailed description of each extracted feature.

3 Experimental Setup

In this section, we describe the evaluation datasets, followed by a description of the detectors used in our experiments. We visualize our experimental setup with regard to dataset preparation, choice of malware detectors, and the creation of transformed binaries in Figure 1.

3.1 Evaluation Dataset

Our evaluation is based on a sample from the SOREL-20M dataset [25], a dataset of 20M Windows PE files. This dataset contains hashes of both benign and malicious files, along with their EMBER (LIEF) feature vectors [14]. Each sample is accompanied by metadata, and malicious behavior labels based on details supplied by detection vendors at the point of collection. The is the set of samples on which transformations are to be applied to the original binaries. In our

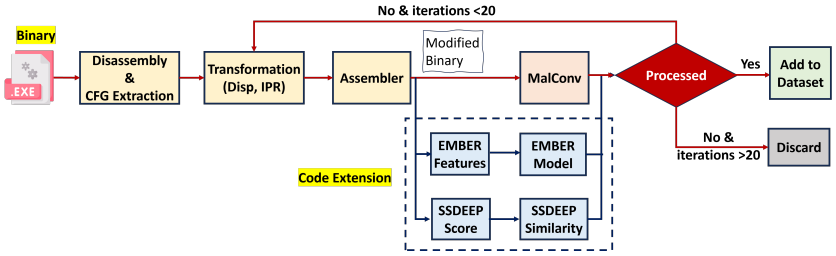


Fig. 1: Pipeline for functionality preserving transformation of binaries using Malware Makeover.

evaluation, we randomly chose 3,362 malware binaries for the test set from the SOREL-20M repository of malware samples. For copyright reasons, SOREL does not distribute benign binaries. As we discuss next, we need to obtain the raw binary to be able to perform transformations. For this reason, the benign hashes obtained from SOREL were queried through the VirusTotal service, and if the sample existed, it was downloaded for evaluation. This resulted in 1,564 test benign binaries.

3.2 Malware Detectors

MalConv. The MalConv detector from [13] is our canonical detection model. This is a neural network architecture designed for malware detection using raw byte sequences of executable files. It is based on a convolutional neural network (CNN) architecture that processes raw byte sequences of executable files and consists of layers including raw byte embedding, 1D convolutions, temporal max-pooling, fully connected layers, and a softmax output layer. The model is designed to consider both local and global contexts while examining an entire file, allowing it to capture important features for malware detection. The model is trained on a dataset of 400,000 executable files split evenly between benign and malicious classes.

EMBER-feature based RF (EMBER-RF). We implemented a Random Forest model utilizing the default parameters and trained on the EMBER feature set. The training set for this model consists of 200K balanced malware and Benign samples, subsampled from the SOREL dataset [25]. These training samples are disjoint from the aforementioned evaluation set (c.f. Section 3.1). The resulting EMBER-RF model achieved an accuracy of 97.09% when trained and tested on 90% and 10% of the dataset respectively.

`ssdeep`. This is an LSH used widely for signature-based malware detection. An overview of how the algorithm works is shown in Section 2.3. We take the `ssdeep` hash of malicious and benign binaries in the test dataset to serve as the local database of the detector, against which any new (transformed) binaries are checked for malicious or benign behavior. We use the Python implementation [8] of `ssdeep`, which provides a similarity value we shall use to `compute` a function that gives a similarity score between 0 and 100. This is used as the metric μ defined in Section 2.1.

Ensemble. We also consider an ensemble detector taking all three detectors into account. The parameter $1 \leq m \leq 3$ determines the decision rule. All decisions are made on the positive label, i.e., malware. The *minority rule*, $m = 1$, means that the program is labeled as malware if at least one detector classifies it as such. Otherwise, it is classified as benign. The *majority rule*, $m = 2$, requires at least two detectors classifying the program as malware. The *consensus rule*, $m = 3$, requires unanimous classification of malware. The choice of m has an obvious impact on the TPR and FPR rates, as we shall experimentally evaluate.

3.3 Creating Transformed Binaries

The transformations from malware makeover require the list of functions contained in a program binary. Given a program binary, we therefore (a) disassemble it through IDA [26] to obtain its control flow graph (CFG), and (b) from the CFG, extract functions for the malware makeover algorithm. Once the functions have been extracted we can run the malware makeover algorithm with a target detector. Figure 1 depicts our pipeline for creating adversarial binaries.

For each malware detector, discussed in Section 3.2, we formulated a targeted adversarial attack to alter the classification of all binary samples within the evaluation dataset, as depicted in Figure 1. A white-box attack strategy was applied to MalConv, leveraging its vulnerability to techniques effective against models based on gradient descent, as discussed in Section 2.2. This is the default attack as used by the authors of malware makeover [9]. In contrast, black-box attacks were employed against EMBER-RF and `ssdeep`, utilizing differing transformation techniques based on their susceptibility.

For the malware makeover attack on EMBER-RF, we imported the EMBER-RF model into the malware makeover program and executed it. However, to assess whether the modification to the binary is successful, we need to extract EMBER features after each alteration to a binary and evaluate them using the EMBER-RF model. We incorporated the EMBER feature extraction script from [27,14] into the malware makeover framework to automate this. However, the integration of the EMBER feature extraction script raised several incompatibility issues, since the EMBER repository's code is in Python 3 and the malware makeover is based on Python 2.7. To address this incompatibility, we initially created a socket to send over the modified binary to the EMBER feature script and get a feature vector. However, this process introduced greater latency. We therefore decided to convert the EMBER feature extraction code to Python 2.7 to ensure compatibility with the malware makeover architecture and make the transformation process faster.

We also modified the malware makeover's codebase to integrate `ssdeep`, i.e., computing `ssdeep` hashes and computing the similarity between the hashes of the original and modified binaries after each alteration until a threshold is reached. Instead of using a modified attack on `ssdeep` as outlined in Section 2.3, we used malware makeover's black box algorithm (with the similarity difference used as a proxy for confidence value difference), so that all three attacks can be evaluated under the same framework.

Table 1: Performance of malware detectors on the original binaries.

Detector	Performance Metrics		
	Acc	**TPR**	**FPR**
MalConv	84.34%	80.11%	8.18%
EMBER-RF	81.17%	77.07%	11.57%
SSDeep	100.00%	100.00%	0.00%

4 Transferability of Adversarial Examples

4.1 Performance of Detectors on Original Dataset.

Table 1 shows the performance of the three detectors on the original binaries in our test dataset (evaluation dataset from Section 3.1). Alongside the TPR and FPR, we show the accuracy (Acc), which will be used throughout this paper. The performance of all three detectors is very good, with `ssdeep` showing a 100% TPR indicating that all binaries were significantly different from one another to cause this hash-based detector to raise any false positives.

Table 1 contains only 4,329 out of the 4,926 binaries in our test dataset. This is because while running malware makeover with the MalConv detector as the target detector, we could only run 4,329 out of the 4,926 binaries, due to the hard binary size restriction of 2MB in malware makeover, and IDA failing to extract CFGs. Likewise, for various other reasons, malware makeover also caused errors on several program binaries with the other two detectors as the target.

Table 2 shows the percentage of malware and benign binaries that were successfully processed by malware makeover given a particular malware detector. We consider a binary as successfully processed if it either evades the target model or malware makeover runs out of transformations to try within a time limit of 3 hours or a maximum number of iterations of 20. The binaries that were not successfully processed are discarded from the pool of binaries for evaluation for that target model. They may or may not have successfully evaded the target model had we given malware makeover more time or more iterations. For EMBER-RF and `ssdeep`, the percentage of successfully processed binaries is even lower.

The table also shows the transformation types that were successful in evading a target model. Specifically, *Disp* transformation was applied to both MalConv and EMBER-RF, while the *IPR* transformation was chosen for its efficacy in circumventing `ssdeep`. Our findings revealed that the *Disp* transformation, while effective against MalConv and EMBER-RF, did not compromise the integrity of `ssdeep`, highlighting the unique resilience of each detection system to different adversarial manipulations. This is most likely due to the lack of functions on which *Disp* transformation can be applied even while using higher budgets.

4.2 Transferability of MalConv Adversarial Examples

We first use MalConv as the target detector and use malware makeover's white-box attack to create adversarial samples of the original binaries. Table 3 (MalConv TTB) shows the resulting performance of the MalConv transformed binaries on all three malware detectors. Not surprisingly, the attack is successful on

Table 2: Percentage of binaries successfully processed by malware makeover and attack parameters used across malware detectors.

Target	Attack Parameters				Processed # (%)	
	Type	Transformation	# of Iterations	Budget	Malware	Benign
Total	-	-	-	-	3362 (100%)	1564 (100%)
MalConv	whitebox	Disp	20	0.05	2765 (82.24%)	1564 (100%)
EMBER-RF	blackbox	Disp	20	0.05	1148 (34.15%)	212 (13.56%)
SSDeep	blackbox	IPR	20	-	1889 (56.19%)	644 (41.18%)

Table 3: Performance evaluation of malware detectors at targeted transform binaries (TTB) for **MalConv**, **EMBER-RF**, and **SSDEEP**.

Detector	MalConv TTB			EMBER-RF TTB			SSDEEP TTB		
	Acc	TPR	FPR	Acc	TPR	FPR	Acc	TPR	FPR
Original Binaries									
MalConv	84.34%	80.11%	8.18%	84.41%	84.32%	15.09%	84.64%	80.68%	3.73%
EMBER-RF	81.17%	77.07%	11.57%	59.12%	60.10%	46.23%	86.06%	84.75%	10.09%
SSDeep	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%
Transformed Binaries									
MalConv	31.00%	38.23%	81.78%	84.41%	84.32%	15.09%	85.08%	81.00%	2.95%
EMBER-RF	77.85%	73.82%	15.03%	40.00%	46.34%	94.34%	76.94%	74.80%	16.77%
SSDeep	94.48%	97.07%	10.10%	95.00%	95.82%	9.43%	13.26%	16.52%	96.27%

MalConv with a significant decrease in TPR, dropping from 80% to 38%, alongside a significant increase in FPR, from 8.18% to 81.78%. However, the same transformed binaries have relatively little impact on the other two detectors. Note that according to our definition of evading detection 2.1, this shows that the binaries do evade detection against the other two detectors as well since the FPR has increased together with a decreased TPR. However, relatively speaking the decrease is less prominent. This solidifies our viewpoint that if the target system employs a variety of detectors, adversarial binaries constructed for only one such detector are likely to still be detected by others, and hence these transformed binaries are not highly transferable.

For `ssdeep` we need to determine a similarity (distance) threshold beyond which the hash of a binary is considered not to be similar to a hash in the database. Figure 2 shows the rate of successfully detecting a transformed binary in the database through their `ssdeep` hashes across different similarity thresholds. The rate is relatively unchanged for any threshold less than 0.4. We chose a threshold of 0.25 for `ssdeep`, which is used for the remainder of this paper.

4.3 Transferability of EMBER-RF Adversarial Examples

To test that the non-transferability of the adversarial attack is not just the result of a particular detector, i.e., MalConv, we repeated the above experiment but this time using EMBER-RF as the target model. We use the black-box version of the malware makeover algorithm since there are no gradients involved in this classifier. Note that the features are now completely changed: raw bytes in MalConv versus EMBER features in EMBER-RF. From Table 2, we have a reduced set of 1148 malware and 212 binaries for EMBER-RF as the target model, since the rest of the binaries were not successfully processed by malware

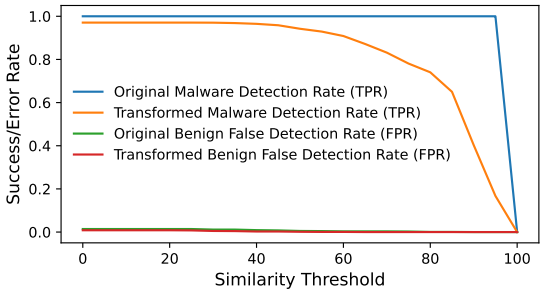


Fig. 2: `ssdeep` analysis of original binaries compared with Malconv targeted transformed binaries.

makeover. We therefore recompute the TPR and FPR of all three detectors on the original binaries from this reduced set shown in Table 3 (EMBER-RF TTB). For this reduced set EMBER-RF has a fairly high FPR (46.23%). However, this is further increased and TPR further decreased in the transformed binaries. Thus the changes do successfully evade detection. Once again, these modifications did not significantly affect the performance of the other detectors; in particular, Malconv's metrics did not change. Thus, these results show that adversarial examples that are specially designed to evade EMBER-RF model are not entirely transferable to the other detection methods in our study.

4.4 Transferability of `ssdeep` Adversarial Examples

Following the evaluation of malware detectors on MalConv and Ember-RF adversarial examples, we first establish baseline performance by only selecting those original samples that were successfully processed by the malware makeover algorithm for `ssdeep`: 1889 malware and 644 benign, from Table 2. Table 3 (SSDEEP TTB) presents the performance of the three malware detectors against `ssdeep` targeted transformed binaries. There is a sharp rise in FPR from 0% to 96.27% and a substantial drop in TPR from 100% to 16.52% for `ssdeep`. Once again, the impact on EMBER-RF is minimal, while MalConv's metrics showed improvement. These results indicate that adversarial examples crafted to circumvent `ssdeep` detection capabilities fail to affect the efficacy of other malware detectors significantly.

5 Performance of the Ensemble Detector

Our results from the previous section indicate that a possible mitigation technique against adversarially transformed binaries is to use an ensemble of detectors. In this section we this ensemble approach, incorporating all three detectors—MalConv, EMBER-RF, and `ssdeep`—to determine the efficacy of an ensemble in detecting transformed binaries. This analysis involves selecting a subset of adversarial examples that have been successfully processed by the malware makeover algorithm for all three detectors. For the purpose of establishing baseline performance, a corresponding subset of original binaries is also compiled.

Table 4: Performance evaluation of individual malware detectors on the common test set. TTB stands for targeted transformed binaries.

Detector	Original Binaries			MalConv TTB			EMBER-RF TTB			SSDeep TTB		
	Acc	TPR	FPR	Acc	TPR	FPR	Acc	TPR	FPR	Acc	TPR	FPR
MalConv	78.64%	75.74%	3.96%	37.91%	39.77%	73.27%	89.39%	89.27%	9.90%	78.93%	76.24%	4.95%
EMBER-RF	71.43%	72.77%	36.63%	56.15%	61.88%	78.22%	53.04%	61.06%	95.05%	65.49%	66.34%	39.60%
SSDeep	100.00%	100.00%	0.00%	95.62%	97.19%	13.86%	95.90%	97.52%	13.86%	8.91%	10.40%	100.00%

Table 5: Performance evaluation of Minority Rule, Majority Rule, and Consensus Rule ensembles on original and transformed binaries. TTB stands for targeted transformed binaries.

Binaries	Minority Rule			Majority Rule			Consensus Rule		
	Acc	TPR	FPR	Acc	TPR	FPR	Acc	TPR	FPR
Original	94.34%	100.00%	39.60%	95.05%	94.39%	0.99%	60.68%	54.13%	0.00%
MalConv TTB	85.29%	98.68%	95.05%	70.72%	76.40%	63.37%	33.66%	23.76%	6.93%
EMBER-RF TTB	86.14%	99.83%	96.04%	92.50%	94.72%	20.79%	59.69%	53.30%	1.98%
SSDeep TTB	92.93%	99.50%	46.53%	54.74%	54.29%	42.57%	55.45%	48.18%	0.99%

The performance of each detector, providing individual verdicts/classification on this common subset of original and transformed binaries, is presented in Table 4. A key thing to note here is that unfortunately, the performance of EMBER-RF on the original binaries in this common subset is lower than its performance on the complete list of original binaries (see Table 1). This discrepancy has been factored in our conclusions in the following.

Minority Rule. We start with the minority rule ($m = 1$) as defined in Section 3.2. Table 5 shows the performance evaluation of the minority rule ensemble on both original and transformed common subsets. The ensemble resulted in a perfect TPR of 100% on original binaries, indicating that all malicious binaries were correctly identified. However, the 39.60% FPR suggests that a significant portion of benign binaries were incorrectly classified as malicious. This is mainly due to EMBER-RF's poor FPR on the original binaries. On the transformed binaries, while the TPR remains high (>98%) for all adversarial sets, there is a sharp increase in FPR to 95.05% and 96.04% in the case of MalConv and EMBER-RF adversarial set showing that the ensemble incorrectly classifies a large number of benign binaries as malicious. The minority rule however performs comparatively better on the `ssdeep` adversarial set, with a high TPR of 99.50% and a slight increase in FPR to 46.53% when compared to the original set, indicating a better balance between identifying malicious and benign binaries compared to the other transformed sets. The high FPR of the minority rule ensemble is not surprising, as only a single detector can influence the decision on a program being malware, which includes the target model as well.

Majority Rule. From Table 5, it can be seen that the majority rule ensemble achieves a high TPR of 94.39% with an extremely low FPR of 0.99%, indicating excellent detection of actual malicious binaries and rarely misclassifying benign files as malicious. However, there's a noticeable decline in TPR to 76.40% and a considerable rise in FPR to 63.37%, suggesting that the ensemble is less effective at detecting both malicious and benign binaries when dealing with MalConv-

evasive binaries. This again is mostly due to EMBER-RF's poor performance on these transformed binaries (see Table 4). This explains why the ensemble achieves a high TPR (>92%) and a reasonably low FPR (<21%) when EMBER-RF is the target model, as the majority now includes the other two detectors, MalConv and `ssdeep` which show good TPR and FPR. The performance of the majority rule is also low when `ssdeep` is the target, once again owing mostly to EMBER-RF's poor FPR on the original samples. From these observations, the majority rule ensemble is effective in still producing a high TPR and low FPR as long as the majority of the detectors are not evaded by a transformation technique.

Consensus Rule. The consensus rule ensemble requires all detectors to agree that a sample is malicious before classifying it as such. This strategy tends to yield a lower TPR but a much lower FPR, aiming for precision in identifying malware. The results for the consensus rule are presented in Table 5. On the original binaries, the FPR is at 0%, indicating no benign files are misclassified as malicious. The TPR is moderate at 54.13%, indicating that only over half of the actual malware are detected. The TPR significantly drops to 23.76%, showing that the ensemble misses the majority of malware transformed to evade MalConv; however, there is a slight decrease in TPR to 53.30% and 48.18% (when compared to the original binaries) in the Ember-RF and SSDeep adversarial examples, suggesting that these transformations slightly affect the consensus rule malicious detection capability. On the other hand, the FPR increases modestly to 6.93% for the MalConv adversarial set and slightly increases to 1.98% and 0.99% for the Ember-RF and SSDeep adversarial sets, respectively, which is still relatively low, suggesting that few benign files are misclassified as malicious.

Keeping this in mind, the consensus rule ensemble appears to err on the side of caution, opting for high specificity (low FPR) at the expense of sensitivity (TPR). This means it is highly reliable at confirming benign files but at the risk of failing to detect a significant amount of actual malware, particularly when dealing with adversarially transformed malware targeting MalConv.

As we have shown detector ensembles have the capacity to detect adversarial malware targeted against specific detectors, we draw attention to the renowned tool of VirusTotal. VirusTotal is an ensemble of the leading anti-virus detection tools in current use, against which we gather further evidence to support the robustness of ensemble techniques against adversarial examples, we provide a full description and results in Appendix C.

6 Are Highly Transformed Binaries Detectable?

We return to our discussion in the introduction on the necessity of keeping perturbations to a program binary at a minimum. Recall that unlike the image domain, an adversarially transformed program binary need not look similar from a static code analysis point of view to its original binary, as long as it maintains functionality. However, it has been pointed out in the literature that too many changes to a program, e.g., via obfuscation, would make it easy to detect a transformed binary as it would create new signatures [12, §4]. We first analyze how much of the binaries has been transformed against each target detector,

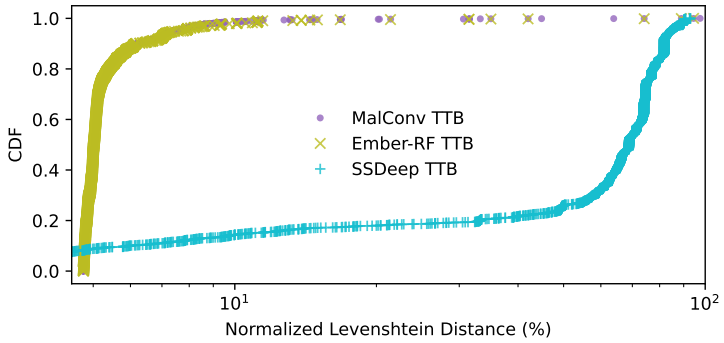


Fig. 3: Cumulative Distribution Function (CDF) vs. Normalized Levenshtein Distance (NLD) for All Three Adversarial Transformations

and then use a machine learning algorithm to distinguish between original and transformed binaries.

6.1 Amount of Adversarial Perturbation

Recall that one of the goals of the attacker is to keep the amount of transformations to a minimum. To check how much perturbation is applied to a binary, we use the normalized Levenshtein distance (NLD) on the original and transformed binaries for each of the test sets used in Table 3. We show the cumulative distribution function (CDF) of the NLD between the original and transformed binaries in Figure 3. The majority of the binaries transformed with MalConv and EMBER-RF as targets, are within 10% NLD of their original counterparts, indicating little changes to the codes of these binaries. On the other hand, the binaries transformed for `ssdeep` evasion are substantially changed. Note that this is also due to the different transformation techniques used for this detector: IPR vs Disp. From the adversarial point-of-view, too many changes are not ideal as we discuss in the next section.

6.2 Detecting Highly Transformed Binaries

To evaluate whether highly transformed binaries are more easily detectable we trained two machine learning models to classify original and transformed binaries, i.e., transformed via malware makeover. The first classifier is a random forest (RF) model while the second is the LightGBM model (briefly discussed in § 3.2), which is a tree-based lightweight gradient boosting framework built on an ensemble of weak learners (typically decision trees) to create a strong predictive model. The two models were trained on a data set split into 80% for training and 20% for testing, which included both original and transformed binaries. For both these models, we used raw bytes as the features. The results in Section 6.1 show that when the target model is MalConv or EMBER-RF the binaries are only slightly transformed in terms of the normalized Levenshtein distance (NLD). On the other hand, the binaries are heavily transformed when the target is `ssdeep` again concerning NLD. Thus, we would expect the models to show better performance on the latter.

Table 6: Comparative performance of the **Random Forest** and **LightGBM** models on detecting adversarially transformed binaries with MalConv, Ember-RF, and SSDeep as targets. TTB stands for targeted transformed binaries.

Target	**Random Forest**			**LightGBM**		
	Acc	**TPR**	**FPR**	**Acc**	**TPR**	**FPR**
MalConv TTB	63.11%	61.37%	35.07%	99.65%	99.77%	0.47%
EMBER-RF TTB	63.05%	58.18%	31.97%	98.53%	99.27%	2.23%
SSDeep TTB	71.45%	70.36%	27.51%	70.81%	71.35%	29.75%

Table 7: Comparative performance of the **LightGBM** model on highly modified (top 20) and least modified (bottom 20) `ssdeep` targeted transformed binaries

Binaries	**Performance Metrics**				
	Acc	**FS**	**TPR**	**FPR**	**CM**$\left(\begin{smallmatrix} TN & FP \\ FN & TP \end{smallmatrix}\right)$
SSDeep-Top-20	97.5%	97.56%	100%	5%	$\begin{smallmatrix} 19 & 1 \\ 0 & 20 \end{smallmatrix}$
SSDeep-Bottom-20	52.5%	53.66%	55%	50%	$\begin{smallmatrix} 10 & 10 \\ 9 & 11 \end{smallmatrix}$

Our results summarized in Table 6 are in contrast to this impression. The performance for MalConv and EMBER-RF targeted transformed binaries is comparable with TPRs of 61.37% and 58.18% and FPRs of 35.07% and 31.97%, respectively, indicating moderate success. But the RF model shows improved performance on binaries transformed via `ssdeep`: 70.36% TPR and 27.51% FPR. However, the LightGBM model shows an almost perfect TPR of more than 99% and a relatively low FPR of 0-2%, while showing similar performance to the RF model on `ssdeep` transformed binaries. This discrepancy may be mostly due to the difference between the types of transformation used in the three detectors: MalConv and EMBER-RF use the Disp transformations whereas `ssdeep` uses the IPR transformations.

We hypothesize that the high classification performance of detecting Disp transformations is the inclusion of semantic `nop` operations. These semantic `nops` may present as unusual, out-of-place values of high entropy which are leveraged as an indicator for transformation by the classifiers of this section. On the other hand, `ssdeep`'s IPR operations are more subtle, thereby blending into the original binaries, however, with too many replacements, as seen in the binaries of large NLD, detection becomes possible again.

Therefore to test whether high transformations lead to better detection, we trained the LightGBM model on `ssdeep` original and transformed binaries only, as we focus on only one type of transformation. We then test the model's performance on the top 20 and bottom 20 transformed binaries in terms of NLD. Our results in Table 7 show that the model almost perfectly predicts highly transformed binaries, and shows no better than random guesses on binaries that are least transformed. This suggests that highly transformed binaries may be easier to detect as malware, as one would expect benign binaries to not be using malware makeover or other transformation techniques so heavily. This justifies the adversarial objective of making minimum perturbations to the original program.

7 Related Work

Our work investigates the robustness of malware detectors against adversarial software binaries when transferred to detectors of different feature sets and model architectures.

Adversarial malware observes roots from adversarial examples, where subtle changes induce misclassifications in machine learning models [1]. Deep learning architectures, increasingly used in malware detection, are particularly vulnerable to such inputs [28,29]. Core to producing such examples, we have shown in Section 6.1, the importance of limiting perturbation size in malware, as with larger transformations, comes increased detection by SSDeep.

Recent research has demonstrated the transferability of adversarial malware under various conditions. Demontis et al. [30] show that adversarial malware can transfer when the complexity of the target model is high, and the gradients of the models are aligned. A more recent study reveals [7] that transferability exists across a diverse range of models, including Neural Networks, SVM, Logistic Regression, and Random Forest, provided the feature space remains consistent. Our study explores the transferability of adversarial malware when the feature space is significantly different. We have shown that adversarial malware does not easily transfer between foundationally different detectors. Further, with these differing models, existing defences like adversarial training that would enhance model robustness against adversarial malware [13,31,32]. Not all detector architectures lend themselves to be augmented with additional training information.

For example, our use of `ssdeep` is representative of similarity-based detection, and not an inherently deep process, thereby complementing traditional adversarial malware creation. We do note the development of deep-learning structures within hashing-based approaches, notably, NeuralHash. NeuralHash combines both deep learning and hashing techniques [33], unfortunately, the deep structures may potentially harbour vulnerability to adversarial attacks.

8 Conclusion and Future Work

We have demonstrated that malware binaries adversarially transformed to evade a single detector often fail against multiple detectors, especially if they employ diverse architectures. Our study involved detectors based on raw binary bytes, feature extraction (EMBER), and locality-sensitive hashing. To mitigate the impact of adversarial transformations, employing an ensemble of detectors with majority rule is a straightforward approach. While this ensemble can still be evaded by sophisticated adversaries, it increases the perturbation to the binary, leading to identifiable signatures for transformation techniques. As observed, highly transformed binaries are easily detectable. An interesting area for future work is to develop an adversarial transformation algorithm that can defeat a majority of detectors in an ensemble while keeping changes to a minimum.

Acknowledgements

This work was partially supported by the Australian Defence Science and Technology (DST) Group under the Next Generation Technology Fund (NGTF).

References

1. N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE S&P*, 2017.
2. W. E. Zhang, Q. Z. Sheng, A. Alhazmi, and C. Li, "Adversarial attacks on deep-learning models in natural language processing: A survey," *ACM TIST*, 2020.
3. G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *IEEE ICASSP*, 2013.
4. K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," *arXiv preprint arXiv:1606.04435*, 2016.
5. F. Tramèr, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "The space of transferable adversarial examples," *arXiv preprint arXiv:1704.03453*, 2017.
6. O. Suciu, S. E. Coull, and J. Johns, "Exploring adversarial examples in malware detection," in *IEEE SPW*, 2019.
7. Y. Hu, N. Wang, Y. Chen, W. Lou, and Y. T. Hou, "Transferability of adversarial examples in machine learning-based malware detection," in *CNS*, 2022.
8. DinoTools, "ssdeep 3.4." https://pypi.org/project/ssdeep/, Mar 2024.
9. K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, "Malware makeover: breaking ml-based static analysis by modifying executable bytes," in *AsiaCCS*, 2021.
10. M. Ebrahimi, N. Zhang, J. Hu, M. T. Raza, and H. Chen, "Binary black-box evasion attacks against deep learning-based static malware detectors with adversarial byte-level language model," in *RSEML*, 2021.
11. F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ml attacks in the problem space," in *IEEE S&P*, 2020.
12. H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, "Learning to evade static pe machine learning malware models via reinforcement learning," *arXiv preprint arXiv:1801.08917*, 2018.
13. E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," in *Workshops at AAAI*, 2018.
14. H. S. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models," *ArXiv e-prints*, 2018.
15. R. Thomas, "Lief - library to instrument executable formats." https://lief.quarkslab.com/, Apr 2017.
16. A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *USENIX Security*, 2022.
17. L. Yujian and L. Bo, "A normalized levenshtein distance metric," *TPAML*, 2007.
18. V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *IEEE S&P*, pp. 601–615, IEEE, 2012.
19. H. Koo and M. Polychronakis, "Juggling the gadgets: Binary-level code randomization using instruction displacement," in *AsiaCCS*, 2016.
20. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *IEEE S&P*, 2005.
21. keenlooks, "Malware Makeover." https://github.com/pwwl/enhanced-binary-diversification, date = 2024-03-13.
22. L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," *arXiv preprint arXiv:1901.03583*, 2019.

23. J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital investigation*, vol. 3, pp. 91–97, 2006.
24. H. Baier and F. Breitinger, "Security aspects of piecewise hashing in computer forensics," in *ISIMIF*, 2011.
25. R. Harang and E. M. Rudd, "Sorel-20m: A large scale benchmark dataset for malicious pe detection," 2020.
26. Hex Rays, "Ida." https://hex-rays.com/, date = 2024-03-07.
27. Elastic Malware Benchmark, "Ember." https://github.com/elastic/ember/blob/master/ember/features.py, date = 2024-03-07.
28. A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *AISS*, 2018.
29. X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *TNNLS*, 2019.
30. A. Demontis, M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru, and F. Roli, "Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks," in *28th USENIX security symposium (USENIX security 19)*, pp. 321–338, 2019.
31. Y. Zhang, H. Li, Y. Zheng, S. Yao, and J. Jiang, "Enhanced dnns for malware classification with gan-based adversarial training," *JCVMT*, 2021.
32. K. Lucas, S. Pai, W. Lin, L. Bauer, M. K. Reiter, and M. Sharif, "Adversarial training for {Raw-Binary} malware classifiers," in *USENIX Security*, 2023.
33. L. Struppek, D. Hintersdorf, D. Neider, and K. Kersting, "Learning to break deep perceptual hashing: The use case neuralhash," in *ACM FAT*, 2022.

Appendix

A Malware Makeover

White-Box Attack. The white-box malware makeover attack targets a machine learning detector that uses gradient descent to minimize its loss, for example, a neural network. Let $\mathbf{b}$ denote a raw binary. We assume an embedding function E that takes a binary $\mathbf{b}$ and outputs a feature vector $\mathbf{x}$. The white-box malware makeover algorithm takes a function $f \in \mathbf{b}$, and transforms it to produce a new binary $\mathbf{b}'$. Let $E(\mathbf{b}') = \mathbf{x}'$. These transformations will be introduced shortly. The transformed binary can be represented as $\mathbf{x}' = \mathbf{x} + \boldsymbol{\delta}$. If the resulting vector increases the loss of the detector then the transformation is retained, otherwise it is rejected. We can view the *displacement vector* as $\boldsymbol{\delta} = \alpha g(\mathbf{x})$. Here $\alpha > 0$ is a small scalar, and g is the gradient w.r.t. input $\mathbf{x}$. Let $\ell(\mathbf{x})$ denote the loss on input $\mathbf{x}$. Then using Taylor's approximation:

$$\begin{aligned}\ell(\mathbf{x} + \boldsymbol{\delta}) &\approx \ell(\mathbf{x}) + \langle g(\mathbf{x}), \boldsymbol{\delta} \rangle = \ell(\mathbf{x}) + \langle g(\mathbf{x}), \alpha g(\mathbf{x}) \rangle \\ &= \ell(\mathbf{x}) + \alpha \langle g(\mathbf{x}), g(\mathbf{x}) \rangle = \ell(\mathbf{x}) + \alpha \|g(\mathbf{x})\|_2^2 > \ell(\mathbf{x}).\end{aligned}$$

Thus, after obtaining $\mathbf{x}'$, we check if the dot product $\langle g(\mathbf{x}), \boldsymbol{\delta} \rangle > 0$, where $\boldsymbol{\delta} = \mathbf{x}' - \mathbf{x}$. If this is true, then the loss has increased. In other words, we are traveling in the direction of the gradient to increase loss. At some point, the loss becomes high enough for the detector to misclassify, at which point the algorithm stops.

Black-Box Attack. Let us assume that we have the cross-entropy loss. Then if K denotes the number of classes and c is the index of the correct label (one-hot encoded), we have:

$$\ell(\mathbf{x}) = -\sum_{i=1}^{K} y_i \ln p_i(\mathbf{x}) = -\ln p_c(\mathbf{x}).$$

Rearranging we get: $e^{-\ell(\mathbf{x})} = p_c(\mathbf{x})$.

Thus, we can instead check if $p_c(\mathbf{x}') < p_c(\mathbf{x})$. If that is the case then necessarily $\ell(\mathbf{x}') = \ell(\mathbf{x} + \boldsymbol{\delta}) > \ell(\mathbf{x})$. The advantage is that this makes it a black-box attack rather than a white-box attack since we do not need to compute the gradient. The above should work for any loss function since: $\ell(\mathbf{x}) \propto 1/p_c(\mathbf{x})$, Hence this attack can be applied to models other than neural networks (which usually employ gradient descent). This black-box attack is also mentioned in [9]. The main point where the white-box attack from [9] deviates from its black-box counterpart is where we can set arbitrary integer values to *semantic nops*. To use an example from [9], suppose the integer value of the semantic nop corresponds to the ith byte in the binary $\mathbf{b}$. We replace the ith byte of $\mathbf{b}$ by $b \in \{0, 1, \ldots, 255\}$, such that the dot product $\langle g(E(\mathbf{b})), \boldsymbol{\delta} \rangle$ is maximized, where $\boldsymbol{\delta} = E(\mathbf{b}') - E(\mathbf{b})$ and

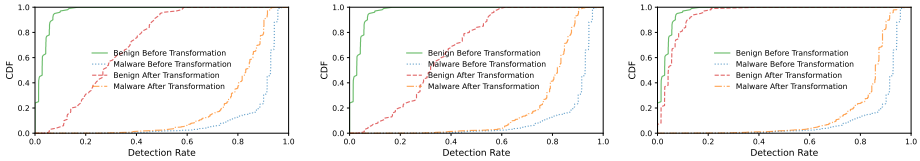
$$\mathbf{b} = (b_1, b_2, \ldots, b_i, \ldots, b_n)$$
$$\mathbf{b}' = (b_1, b_2, \ldots, b', \ldots, b_n)$$

for a binary of length n bytes. However, we could do the same attack in the black-box setting by replacing the ith byte of $\mathbf{b}$ by every possible byte value $b \in \{0, 1, \ldots, 255\}$ and choosing the resulting binary $\mathbf{b}'$ that results in the least probability $p_c(E(\mathbf{b}'))$. Of course the drawback here is that we need to do inference a total of 256 times for each such semantic nop (instead of obtaining this information through the gradient in the white-box version of the attack).

B Adversarial Example Attack on `ssdeep`:

One of the attacks on `ssdeep` reported in [24] when the input is natural language text is to change one byte in each chunk. To ensure that trigger points are not impacted, the last 7 bytes in each chunk are left untouched (corresponding to the window size). In fact, by utilizing the fact that the `ssdeep` similarity checking algorithm only considers a match if there is one common substring in the two hashes of length 7 [24], even less number of changes need to be made. A simpler version of this attack, that makes changes to a character in every substring of size $b \approx n/64$, seems to work nearly 100 percent of the time. Thus, we need not even identify chunks. This can be explained as follows. The rolling hash identifies the end of a chunk if the current byte sequence evaluates to $-1 \pmod b$. If the rolling hash is pseudorandomly distributed, then the probability that the current sequence of bytes evaluates to this is $1/b$. The probability that a chunk has not been identified in the first $7b$ rolling windows is given by: $(1 - 1/b)^{7b} \approx e^{-7} = 0.00090359397765$.

This is very low, and hence the probability that the simpler attack will work is high. Implementing this attack in the malware domain requires changes in each chunk of a given binary file. To maintain functionality, these changes should be meaningful. One such example could be using code substitution. For instance,



(a) **MalConv** transformation. (b) **Ember-RF** transformation. (c) **SSDeep** transformation.

Fig. 4: CDFs of VirusTotal detection rates for benign and malware *before* and *after* transformation.

among the different transformation types mentioned in the work from Lucas et al. [9] to evade a target machine learning malware detector is replacing instructions with equivalent ones, which preserve length. For example, `sub eax,4` can be replaced by `add eax,-4`. This is an example of an IPR transformation. If each block contains such an instruction, we can replace them with equivalent ones and be able to evade detection via `ssdeep`.

C Results on VirusTotal

A major motivation for us to assess the performance of an ensemble of detectors to mitigate adversarial examples is the use of reports from multiple security vendors (scanners) by VirusTotal. This provides a buffer against misclassifications as long as there is consensus amongst multiple scanners. To check the detection rate of the transformed binaries by VirusTotal, we took the 101 benign and 606 malware binaries (total of 707 binaries) common amongst the three target detectors (Table 4). We then use the VirusTotal API with each original binary and its transformed variant to obtain labels returned by the VirusTotal scanners. The total number of scanners varies across binaries, with a minimum of 51 and a maximum of 72 scanners, averaging 70.78% per binary. From the labels thus returned, we define the *detection rate* as being the fraction of scanners labelling the given binary as malware.

Figures 4 (a), (b), and (c) depict the results for malware binaries when targeted by MalConv, EMBER-RF, and `ssdeep` detectors, respectively. In all cases, the detection rate of transformed malware binaries notably decreases. Conversely, the detection rate increases for transformed benign binaries, especially for MalConv and EMBER-RF, suggesting that the transformations deceive scanners into identifying benign binaries as malware. Binaries targeted with `ssdeep` as the detector show a lower likelihood of misclassification compared to the other two detectors. Employing the majority rule (detection rate of 0.5) indicates minimal misclassification of malware binaries by the scanners. This suggests that using an ensemble of detectors with majority rule could be an effective measure against adversarial attacks on malware binaries, as discussed in Section 5. Moreover, attempts by adversaries to evade multiple detectors would likely increase perturbation in the original binary, resulting in its detection as malware, as discussed in Section 6.2.