COPYCAT: Practical Adversarial Attacks on Visualization-Based Malware Detection

Aminollah Khormali
University of Central Florida
aminkhormali@Knights.ucf.edu

Ahmed Abusnaina
University of Central Florida
ahmed.abusnaina@knights.ucf.edu

Songqing Chen
George Mason University
sqchen@gmu.edu

DaeHun Nyang
INHA University
nyang@inha.ac.kr

Aziz Mohaisen
University of Central Florida
mohaisen@ucf.edu

ABSTRACT

Despite many attempts, the state-of-the-art of adversarial machine learning on malware detection systems generally yield unexecutable samples. In this work, we set out to examine the robustness of visualization-based malware detection system against adversarial examples (AEs) that not only are able to fool the model, but also maintain the executability of the original input. As such, we first investigate the application of existing off-the-shelf adversarial attack approaches on malware detection systems through which we found that those approaches do not necessarily maintain the functionality of the original inputs. Therefore, we proposed an approach to generate adversarial examples, COPYCAT, which is specifically designed for malware detection systems considering two main goals; achieving a high misclassification rate and maintaining the executability and functionality of the original input. We designed two main configurations for COPYCAT, namely AE padding and sample injection. While the first configuration results in untargeted misclassification attacks, the sample injection configuration is able to force the model to generate a targeted output, which is highly desirable in the malware attribution setting. We evaluate the performance of COPYCAT through an extensive set of experiments on two malware datasets, and report that we were able to generate adversarial samples that are misclassified at a rate of 98.9% and 96.5% with Windows and IoT binary datasets, respectively, outperforming the misclassification rates in the literature. Most importantly, we report that those AEs were executable unlike AEs generated by off-the-shelf approaches. Our transferability study demonstrates that the generated AEs through our proposed method can be generalized to other models.

KEYWORDS

Malware Detection; Visualization; Adversarial Examples; Deep Learning

1 INTRODUCTION

As the threat of malicious software (malware) is growing rapidly, malware detection is becoming a higher priority within the computer security research community and software security industry alike. To address this threat, there has been a wide variety of research works presenting various techniques for detection, using signatures [1], static analysis [2], and dynamic analysis [3]. Although these methods are powerful in analyzing malware, they have multiple shortcomings, such as their limited scalability [4].

The static analysis based approach to malware detection is ineffective, since it is prone to obfuscation. Dynamic analysis, while addressing obfuscation by observing actual behavior of malware at the execution time, is often resources-intensive, and subject to dynamic analysis evasion (e.g., sandbox detection), making it insufficient alone. Those shortcomings and the need for an automation technique that extrapolates insight into software from features of behavior have opened a new avenue for malware analysis through visualization, often incorporated into automatic malware detection techniques that often use machine and deep learning [5–8].

Visualization has been leveraged over both shallow and deep learning algorithms, and in many security applications, including computer forensics [9], network monitoring [10], and software security [6–8]. In malware analysis of visualization techniques, hard-to-analyze code information is transformed into an image, where features of different executable files are visualized. Compared to conventional methods, including those mentioned above, the detection process using visualization is simpler and faster (analogous to static analysis), due to the viability of deep learning approaches in recognizing patterns, even over obfuscated code [11]. Moreover, the utilization of visualization techniques along with deep learning models has been shown to improve the performance of malware detection in multiple ways: reduced time and memory consumption, improved robustness and detection accuracy, and being non-intrusive [6, 12–14]. To this end, several studies have incorporated deep learning-based networks for detecting malicious executable files from benign ones using visualization techniques [15, 16].

Although applications of deep learning models are actively explored in a wide range of applications, *e.g.*, health-care, industry, and cyber-security [17, 18], deep learning itself has been shown to be vulnerable to Adversarial Examples (AEs) [19, 20]. AEs, which are carefully crafted inputs, are created by adding a small perturbation to the original input of a machine learning algorithm in order to produce the adversaries' desirable outputs, such as misclassification [21, 22]. Since the crafted samples are generated by applying limited changes on the original inputs, the crafted samples are very similar to the original ones, and are not necessarily outside of the training data manifold. Algorithms crafting adversarial samples are designed to minimize the perturbation, thus making it hard to distinguish adversarial samples from legitimate ones.

Shortcomings of the Prior Work. There has been several studies, some of which are concurrent to this work, which have several shortcomings which we outline here and address in the subsequent sections. Most importantly, while generating adversarial samples, it

is important to ensure executability of the resulting samples (code), *a common shortcoming of the prior work.* Liu *et al.* [23] presented adversarial attacks on machine learning-based malware visualization detection methods, although they considered the entire sample as a candidate for perturbation, corrupting the adversarial sample's code. Of note in this direction of generating adversarial samples while not corrupting the sample is the seminal work of Grosse *et al.* [13], who investigated adversarial attacks on neural network based android malware detection models. In their work, perturbations were added into malware's none functional parts. Their approach achieved a misclassification rate of 63% over all malware samples, which is—although promising—quite low. Moreover, although their method maintains the functionality of the code, it does not necessarily maintain its executability, since changing the application manifest will affect the permissions, software and hardware components access, intents, etc., which are necessary for the malicious applications to execute the code by the correct API calls.

Although adversarial learning has been an active research area, and outside of the aforementioned works, there is very little research done on understanding the impact of adversarial attacks on deep learning-based malware detection [13, 24, 25], particularly those utilizing visualization. In parallel to our work, the authors of [24, 25] suggested injection of a sequence of bytes for AE generation. Our work differs in multiple points. First, our work tackles the visualization-based malware detection approach, which has been proven to be faster and more efficient as a malware detection method [13, 14]. Second, we investigate the targeted misclassification attacks, in addition to untargeted misclassification which was the focus of the work in [24, 25]. Third, we explore and confirm the generalization of the proposed method by evaluating it using two malware datasets, Windows and IoT binaries, and transferability to other algorithms. Finally, we explore the performance of a defense method, adversarial training, on the detection of the generated adversarial malware examples.

Contributions. Motivated by the aforementioned issues, our main goal is generating *adversarial malware samples that (1) fool the classifier and (2) execute as intended.* First, we investigate the executability of the adversarial examples generated by leveraging adversarial attack methods into deep learning-based malware visualization detection systems. In particular, we explore multiple well-established adversarial attack methods with the visualization-based approach to generate adversarial samples with least L_2-distance from original input for both attacks and defenses. We find that although the generated adversarial examples can successfully fool the classifier, none of them is still executable. Therefore, we further propose COPYCAT, which operates by padding binaries of a certain class into the binaries of another label, to maintain the executability of the original input, while forcing the model to act as desired by the adversary, with targeted and untargeted misclassification. Specifically, the contributions are:

- We investigate the application of five generic adversarial methods for attacking visualization-based deep learning models for the malware detection. We find that although these methods outperform state-of-the-art in their evasion rate, the generated AEs

do not necessarily maintain the executability of the files. We conclude that those adversarial learning algorithms are impractical candidates for understanding the robustness of such detectors.
- Given the shortcomings of those off-the-shelf adversarial algorithms for generating practical malware samples that evade detectors, we propose a new approach, called COPYCAT, which is specifically designed for generating executable samples, in order to maximize the attack success rate while maintaining the functionality and executability of the original files.
- We conduct comprehensive experiments to evaluate the performance of COPYCAT for generating adversarial executable samples under both targeted and untargeted attacks settings. In particular, we have designed and evaluated various configurations of COPYCAT using two malware datasets, for Windows and IoT, demonstrating the effectiveness of the proposed approach in generating executable AEs. COPYCAT is shown to have a misclassification rate of 98.9% and 96.5% on Windows and IoT malware datasets, respectively, and all of the AEs are executable.

Organization. In section 2 we provide a brief background. We describe COPYCAT in section 3, and follow it with performance evaluation and discussion in section 4. We review the related work in section 5. Finally, limitations and future work are discussed in section 6, followed by the conclusion in section 7.

2 BACKGROUND AND PRELIMINARIES

We incorporate malware visualization from the computer-vision community, and the deep learning algorithms and adversarial attacks from the artificial intelligence community into a single process for generating adversarial malware samples. In this section, we describe some background information and basic models. First, we need to transform malware binaries into images from which features are extracted, as described in section 2.2. In section 2.3, we describe the high-level idea of the adversarial machine learning and its importance in the domain of deep learning-based malware detection systems. We note that we will provide more technical details of AEs generation methods in section 3.1. Finally, we discuss potential defenses against AEs in section 2.4.

2.1 Deep learning-based Malware Detection

In COPYCAT, and as a baseline, we design a deep learning-based classifier based on the Convolutional Neural Network (CNN) architecture. CNNs are a type of deep, feed-forward artificial neural networks, widely used for image classification tasks [26, 27]. Deep learning models offer several advantages compared to traditional machine learning algorithms. For example, they extract and learn representative features automatically, which in turn means that they require minimal pre-processing effort compared to traditional machine learning methods where features have to be extracted through feature-engineering algorithms using domain knowledge [28].

We designed two CNN models, one for each dataset, with three consecutive convolutional layers with ReLu activation, followed by a fully connected layer outputting softmax function values for each class. We tested different values for the number of convolutional layers, batch size, and epochs to improve the classification accuracy of the model over test samples of the two malware datasets. For the Windows-based malware binaries we achieved 99.12% accuracy rate

after 50-epoch training and batch size of 150. Similarly, we trained another CNN model with the same architecture for IoT malware binaries and achieved 99.67% accuracy rate on test samples after 50-epoch training and a batch size of 150. More detailed information about convolutional neural networks can be found in [26].

2.2 Visualization

The different sections of code containing specific information about the software (malware) can be converted and organized as a two-dimensional array of unsigned integers, which is then transformed into a grayscale image. Recently, various visualization methods [6, 12, 29] have been proposed to analyze malware using this idea. The main goal of malware visualization techniques is conversion of hard-to-identify code information into an image, and acceleration of malware detection process [30]. The visualization process starts reading the malware samples as 8-bit unsigned integer vector, and categorizing it into set of two-dimensional array. The values in the array are transformed into a range of 0 and 255 that represents a grayscale image. The pixels corresponding to 0 result in black pixels while 255 result in white pixels. It is worth noting that the images' width is determined and fixed based on the characteristics of the dataset, while the height of the image can change based on the size of the malware sample. The output of the visualization step would be images with the same width and probably different heights. However, before applying the learning algorithm, we need re-scale the images to a fixed width and height.

Illustration. A high-level flowchart of the malware visualization process is illustrated in Figure 1. In this process, each malware binary is first converted into an 8-bit vector and then transformed into a grayscale image with a fixed width of w_1 (vector to matrix conversion). The height of the image depends on the original malware size. However, the re-scaled images of all binaries are all of the same width and height, where w_1 and w_2 are equal while h_1 and h_2 are not necessarily equal.

We note that each binary file consists of various sections, such as .text, .rdata, .data, .rsrc [31]. The .text section contains the executable code, the .data section includes both uninitialized code and initialized data, while the .rsrc section contains all resources of the module. All sections are combined to construct the image corresponding to the binary, and four different samples from our benign and malware samples are visually illustrated in Figure 2, where 2(a) and 2(b) show benign and malicious Windows binaries, while 2(c) and 2(d) show benign and malicious IoT binaries, respectively. As seen in Figure 2 different fragments of benign and malware samples represent unique image textures, which are leveraged in the deep learning model to detect malicious binaries [32].

2.3 Attacks in Deep Learning (Threat Model)

The main goal of the attacks against deep learning is to modify an input sample x such that it is misclassified by the model f, where it has been shown that an attacker is able to craft AEs, e.g., x', by applying a small perturbation to the original input such that the model is forced to produce an output of the attacker's choice [33–35]. Adversarial learning attacks on deep learning models can be categorized from multiple point of view, and based on the adversary's goals and capabilities [36], as highlighted in the following.
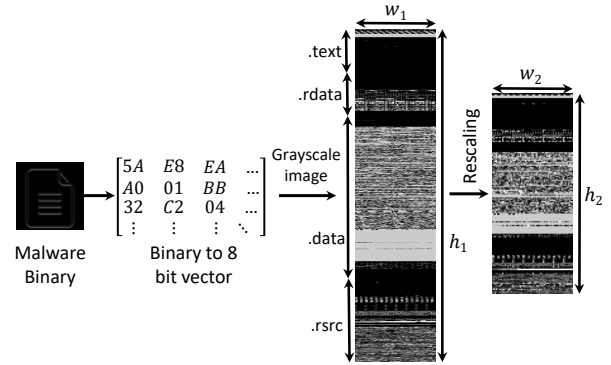

Figure 1: High-level flowchart of the visualization process. Note $w_1 = w_2$, while not necessarily the case for h_1 and h_2.

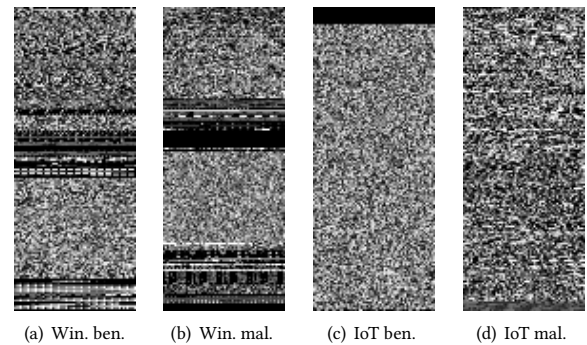(a) Win. ben.   (b) Win. mal.   (c) IoT ben.   (d) IoT mal.

Figure 2: Sample visualization of benign and malicious binaries. Here, 2(a) and 2(b) show benign and malicious Windows binaries, respectively. While, benign and malicious IoT binaries are shown in 2(c) and 2(d), respectively.

2.3.1 Goals. The goal of the adversary in deep learning is to force the model into incorrect results. The main goal of the adversary can be represented based on the nature of the incorrectness.

- **Confidence reduction.** The goal is to reduce the classifier's confidence, by presenting ambiguity in the predictions of the model. This attack does not necessarily result in incorrect outputs.
- **Untargeted misclassification.** Producing adversarial examples, e.g., x', that force the model output $f(.)$ to be any class other than the original one; this is, $f(x') \neq f(x)$.
- **Targeted misclassification.** Crafting adversarial examples, e.g., x', that force $f(.)$ to be any specific class $f(x') = t$. Note that this type of attack is more complex than the confidence reduction and the untargeted misclassification attacks.

2.3.2 Capabilities. Considering attacks conducted at the test time, the adversarial attacks can be categorized based on the information and capabilities made available to the adversary into white-box attacks and black-box attacks.

- **Model architecture and training data.** The adversary has full knowledge and access to the training data x and the deep learning model f used for classification, including the training functions

and algorithms, the number and type of layers, neurons' activation functions, link weights and bias, loss, etc.

- **Model architecture** In this work we assume that the adversary has full knowledge about the model architecture and associated activation functions, the number of layers, link weights, bias, etc.
- **Training data** In this work we assume that the adversary builds a new deep learning model based on his collected dataset that is representative of the original dataset in both characteristics and distribution. Note that the adversary has no information regarding the original model and its properties.
- **Oracle** The adversary has an oracle access to the model and no prior knowledge about the architecture and its internals. Moreover, we assume that the adversary can learn about the relationship between supplied inputs and the corresponding output of the model through rate limited/unlimited queries.
- **Sample** Here the adversary is able to collect pairs of inputs and outputs associated with an unknown deep learning model. However, the adversary cannot change the input value to investigate its impact on the model output. Therefore, it is only practical when large number of samples are available.

2.3.3 Our Threat Model. In this work we designed two sets of configurations for COPYCAT depending on the utilized approach. Namely, while the AE padding approach assumes that the adversary has full knowledge of the topology of the model, the link weights, etc., the sample injection configuration is a black-box attack. Moreover, we assume that the adversary's goal is to conduct both targeted and untargeted attacks. Based on that, part of this study falls into the standard white-box attack model, while the other part falls into black-box attack model.

2.4 Robustness

Both shallow and deep learning models play a critical role in detecting malicious applications and analyzing sequential streams of data. Therefore, several defenses have been proposed to increase the robustness of such models against adversarial learning attacks [37]. The ultimate goal for defensive techniques is making the decision based on meaningful features, and more immune to changes in features that are less important [37]. Adversarial training is the most common defensive technique, and is implemented by training the model over the original data and the generated adversarial examples from different attack methods. This defense technique ensures better accuracy against such attacks, but while lowering the original accuracy due to the change in classification paradigm [37, 38]. Some extensions to adversarial training were implemented to improve the classification accuracy, including the virtual adversarial training using smoothing and LDS gradient, by Miyato *et al.* [39], and ensemble adversarial training, which improves the quality of adversary training sample, by Tramer *et al.* [40]. Transfer learning techniques, such as distillation, were used by Papernot *et al.* [21], to improve the robustness of the aforementioned approach.

3 METHODS AND TECHNIQUES

There are several studies that incorporate deep learning-based classifiers for detecting malicious binaries from benign ones. In addition, it has been shown that deep learning models are not immune to adversarial attacks, and can be easily fooled. Although the concept

of adversarial machine learning is an active research area, there are very few research works that have investigated the impact of adversarial attacks on the deep learning-based malware detection systems [23]. Therefore, the key goals of our study is to investigate the impact of adversarial attacks on the performance of deep learning-based malware detector, as well as generating adversarial malware samples that are executable. In particular, we aim to generate adversarial samples that can fool the deep learning model such that malware binaries are classified as benign, and vice versa.

To do so, we design three different configurations: an adversarial attack configuration, a defense configuration, and an executability configuration. The first configuration utilizes several existing attack methods—previously unexplored for this problem domain—to generate adversarial binary samples. The second configuration explores potential defense strategies to increase the robustness of deep learning-based malware detectors against adversarial examples. In the executability configuration, we focus on generating adversarial samples that are not only successful in fooling the classifier, but also are executable. In the following, we will describe step-by-step aspect of these configurations.

3.1 Generic Adversarial Attacks

To investigate off-the-shelf adversarial attacks on malware detection, we utilized five generic algorithms capable of generating AEs. The primary goal of this configuration is to generate AEs that successfully fool the traditional visualization-based malware detection designed using deep learning networks. This attack configuration consists of two modules, the traditional visualization-based malware detection system (① in Figure 3) and adversarial example generator module (② in Figure 3), reviewed in the following.

3.1.1 Visualization-based Malware Detection System. This module consists of a visualization and deep learning sub-modules. The visualization sub-module transforms the binary code of each (potential) malware sample into a grayscale image, while the deep learning sub-module, which is based on a convolutional neural network with multiple layers, is designed and trained on a training set of grayscale images and evaluated using the test images for tuning. Note that this model builds the baseline model for malware detection, which will be mainly used for adversarial binary generation in the subsequent modules.

3.1.2 Adversarial Example Generator. The main responsibility of this module is to create AEs using off-the-shelf adversarial machine learning algorithms. The adversarial sample generator module, shown in Figure 3 takes the trained model and test set of grayscale images as inputs and generates adversarial binaries using one of five different algorithms: the Fast Gradient Sign Method (FGSM), due to Goodfellow *et al.* [41], Carlini and Wagner method (C&W), due to Carlini and Wagner [33], DeepFool, due to Moosavi-Dezfooli *et al.* [19], Momentum Iterative Method (MIM), due to Dong *et al.* [42], and Projected Gradient Descent (PGD), due to Madry *et al.* [38]. We briefly describe each attack method in Appendix A.

3.1.3 Shortcomings. This attack configuration assumes the availability of an intermediary state, rather than the malware binary, where binaries are already transformed into grayscale images (visualized) accessible by the adversary, and the adversary tries to force
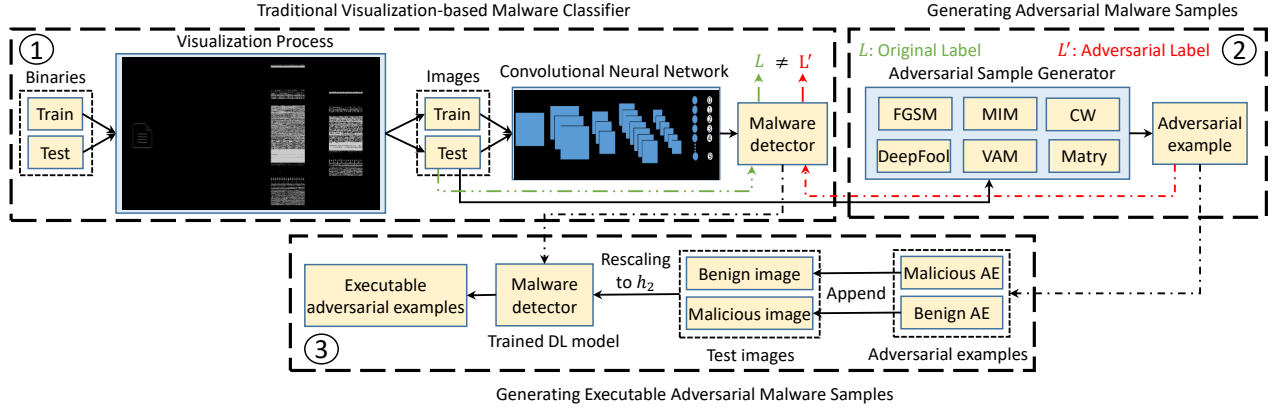
Figure 3: General structure of the visualization-based generic adversarial attacks. As shown, it consists of three modules. Module 1 is for the visualization-based malware detector, module 2 is for generating adversarial samples based on generic adversarial learning algorithms. Note that the AEs generated at module 2 are not necessarily executable. Thus, to maintain the executability of the crafted AE, the generated AEs from 2 are being appended to the original test images in the third module.

the trained DL model into misclassification. While this constitutes a major shortcoming of this configuration, it is not the only one: another shortcoming is that the generated AEs in this configuration are not guaranteed to be executable since the applied perturbation into the grayscale image may result in corrupted binary code.

In conclusion, this configuration is well-suited to cases where the detection design is based on the visualization approach alone, where malware binary code are readily transformed into grayscale images. This configuration, however, will not be effective with end-to-end detection (i.e., where the detector starts with the binary, does the transformation via visualization, then learning and detection).

3.2 Defense Configuration

The goal of the defense is to improve the robustness of the malware detection system against AEs. Although there are multiple defensive algorithms against AEs, such as defensive distillation [21] and adversarial training [41, 43, 44], none of them is completely effective. Nonetheless, in this study we utilized adversarial training to improve the robustness of our malware detection system, as it is the most successful adversarially robust model so far [37]. The basic idea of adversarial training is to inject AEs into the training procedure to increase the robustness of the model [37], where the goal is to solve the following adversarial empirical risk minimization:

$$\min_{\theta} \; E_{(x, y) \sim \widetilde{D}} \left[max_{\delta \epsilon S} \; L(x + \delta, y; \theta) \right],$$

where x is the input image, y is the model output, θ is the model parameters, and δ is a small perturbation.

The steps taken to defend against the adversarial samples are shown in algorithm 1. The algorithm takes $Dataset_{org}$, $Labels$, and CNN_{tr} as inputs, generates adversarial samples against CNN_{tr} using the previous five attacking methods and train a new method over the combination of the original dataset and generated adversary examples. Each sample in the original dataset will be represented with additional five adversary samples each from different attack. This will increase the robustness of the model as the decision will be related to the core features that preserve the information,

rather than being affected by the change of features that is not fundamentally meaningful [37], which may lead to overfitting.

3.3 COPYCAT

In a nutshell, generating adversarial samples while affecting the executability of the application is counter-intuitive, and is an easy way to infer whether a piece of software is malicious or benign (by attempting to execute it). Moreover, altering a malware sample while not paying attention to modifications in the malicious piece of code (by, for example, overriding it), while may fool the classifier, goes against the whole point of adversarial example generation from a practical standpoint. Therefore, we propose two approaches to generate samples that 1) fool the classifier, and 2) maintain the executability of the original binary (malicious or benign).

The key idea in this work is inspired by the work of Cha *et al.*'s [45], where binaries of an application specified by a certain architecture are padded into an application from another architecture to change the behavior and application logic depending on the system used for running it. As such, we consider the machine learning model as an additional system on which a sample needs to operate, along with the actual system on which the binary will run, and customize our sample to operate successfully on both as intended. Note that, and unlike the work of Cha *et al.*'s [45], our program sizes do not need to be identical in size: we have the advantage of constructing a sample of a minimal size only to fool the classifier, while not altering the actual sample for a guaranteed executability.

Executable Adversarial Examples. COPYCAT contributes two methods for executable adversarial samples generation:

- **AE Padding.** Our first method is called "AE Padding", in which COPYCAT generates an AE x' for an original sample x using one of the five previous attack methods. Then, we convert x' to an adversarial image with the same dimensions of the original sample image, and append the adversarial image to the end of the original sample image. Appending pixels at the end is equivalent to appending the code into a non-reachable code area, which will not affect the executability of the program but will change

Algorithm 1: Sample algorithm for adversarial training as a defence against adversarial attacks. Here, the model (M_o) trained over the original dataset (D_o) and the corresponding labels (L_o) are inputs of the function ($AdvTraining$) and adversarially robust model (M_a) is its output.

---

1 Function **AdvTraining** (D_o, L_o, M_o);
  Input : D_o, L_o, M_o
  Output: M_A
2 $D_t \leftarrow D_o$
3 $L_t \leftarrow L$
4 **for** *attack IN adversary_attacks* **do**
5     $AEs \leftarrow attack.generate(M_o)$
6     $D_t \leftarrow D_t \cup AEs$
7     $L_t \leftarrow L_t \cup L_o$
8 **end**
9 $M_A \leftarrow$ New Model
10 $M_A.train(D_t , L_t)$
11 **return** M_A

---

the output pattern of the visualization, and likely leading to a misclassification (which needs to be confirmed). The general architecture of this configuration of COPYCAT is shown in Figure 3, where the third module in ③ is responsible for appending AEs, generated in ②, to the end of the original sample image. Note that the appended images were not necessarily executable, whereas the output of the third module is executable.

- **Sample Injection.** The second method is called "Sample Injection", in which we inject the binaries of a sample from the targeted class into an unreachable area of the binaries (code) of the attacked sample, usually at the end of the code. This process will ensure the executability of the original binary while changing the pattern of the visualization, leading to a model misclassification.

Concurrent Work. Two other concurrent studies [24, 25] proposed similar approaches to ours. However, unlike those concurrent studies which produced untargeted misclassification attacks, COPYCAT produces both targeted and untargetted misclassification. In other word, COPYCAT can force the model to label the original input as any other desired class. This is a particularly important objective in malware family classification applications, since classification, for example, is used for attribution, designing the correct remedies, and risk management procedures, etc. Moreover, although a minor contribution, we explored and confirmed that our approach generalized equally well to IoT malware. Moreover, we show that COPYCAT is able to achieve an untargeted misclassification rate of 98.90%, compared to the untargetted classification results of 60% in the concurrent published work [24]. Finally, COPYCAT was able to achieve a targeted misclassification rate of 73.50% and 96.5% for Windows and IoT samples, respectively, which are promising. We use a CNN model with a general architecture that can accommodate multiple types of malware, unlike MalConv [24].

4 EVALUATION AND DISCUSSION

We evaluate the performance of COPYCAT for generating executable AEs through comprehensive experiments. To do so, we

Table 1: Family-level distribution of Windows malware.

Malware Family	Number of Samples	
	#	%[1]
Ramnit	1,534	13.14%
Lollipop	2,470	21.15%
Kelihos ver3	2,942	25.19%
Vundo	451	3.86%
Simda	41	0.35%
Tracur	685	5.87%
Kelihos ver1	386	3.30%
Obfuscator.ACY	1,158	9.91%
Gatak	1,011	8.65%
Benign Sample	1,000	8.56%
Total	11,678	100%

assume that the adversary aims to conduct both targeted and untargeted attacks, with white-box access to the model. We evaluate the performance of COPYCAT against two set of datasets, namely Windows and IoT binaries. In the following, we review the datasets, the experimental setup, the results and finding, and a discussion.

4.1 Dataset

We used two datasets: Microsoft Windows malware binaries and Internet of Things malware binaries. We used those two types of malware for the following reason. First, while Windows malware is popular, IoT malware is on the rise, and deserves consideration in evaluations. Second, because machine learning algorithms outcomes depend on the used dataset, having multiple datasets would be better than one. Finally, we note that no prior work used such emerging IoT malware dataset, although our work can be easily applied to other malicious binaries, such as Android applications as well. To this end, in the following we review the used datasets.

- **Windows Malware Binaries.** To carry out our experiment on representative Windows malware samples and facilitate reproducibility of results, we used open source malware dataset, called BIG 2015. BIG 2015 is introduced in the Kaggle Microsoft Malware Classification Challenge [32]. The dataset consists of 10,678 labeled malware samples classified into nine different malware families, and 1,000 benign samples randomly selected from valid executables of Windows 10 binaries. Table 1 shows the distribution of the samples across the nine malware families.
- **IoT Malware Binaries.** We select IoT malware for their prevalence, as they are very popular for the increasing use of IoT devices, their flexibility (Linux applications) and portability (developed for multiple platforms). For our IoT binaries, we obtained a set of 2,899 randomly selected IoT malware samples in binary format from the IoTPOT project [46]. In addition, we collected a set of 207 benign IoT binaries, for a total of 3,106 IoT binaries.

Table 2 shows the distribution of the malware samples across families, the variety in malware IoT samples helps with creating more robust models exposed to more patterns and behaviors which increases the decision confidence of the classifier. Table 3 shows

Table 2: Distribution of IoT malware samples across families (labels assigned by antivirus scanners). %[1] refers to the representation of families among malware samples.

Arch	Number of Malware	
	#	%[1]
Gafgyt	2,609	89.99%
Mirai	185	6.38%
Tsunami	67	2.32%
Singleton	32	1.10%
Hajime	6	0.21%
Total	2,899	100%

Table 3: IoT dataset: malware and benign binaries.

Label	Number of Samples	
	#	%[1]
Benign	207	6.66%
Malware	2899	93.34%
Total	3,106	100%

the labels used in the training process of the DL model, summing up the number of benign and malicious samples outlined above.

4.2 Experimental Setup

For a transparent evaluation, in the following we describe the experimental setup we use for generating the adversarial examples.

- **Algorithms implementation.** All attack methods are implemented using Cleverhans [47], a python library with state-of-the-art implementations for various adversarial methods.
- **Parameters.** Each attack method has different parameters. We conducted several experiments with different values for such parameters to achieve a high misclassification rate. For example, DeepFool has two parameters to be tuned, the overshooting value and the number of iterations, which were tuned to 100 and 0.05, respectively, to achieve 96.43% accuracy with the Windows malware model. The parameters of each attack method are reported in Table 4 (for both datasets), while the obtained results for other attack methods are reported in Table 5 and Table 6 for the Windows and IoT datasets, respectively.
- **Evaluation system.** All experiments are conducted using Python 3.6 running over Ubuntu 16.04 and using a system of i5-8500 CPU operating at 3.00GHz, with 32GB DDR4 RAM, 512GB SSD, and NVIDIA GTX980 Ti graphical processor unit (GPU).

4.3 Evaluation Metrics

We use three evaluation metrics: the mislcassification rate (MR), perturbation magnitude (PM), and running time (RT). MR is used as an indication of the success rate for fooling the detection system. When the predicted label $\mathrm{argmax}_k P(y_k|\bar{x}_i)$ for the adversarial sample $\bar{x}_i$ is not same as the correct class label y_i of the original sample

Table 4: Parameters used for each adversarial attack methods. Here ϵ refers to the distortion parameter, L.R. refers to learning rate, # Iter. refers to the number of iterations, and O. refers to overshooting parameter in DeepFool method. The sign (–) means the parameter is not applicable to the method.

Method	ϵ	# Iter.	L.R.	O.
FGSM	0.3	–	–	–
C&W	–	100	0.1	–
DeepFool	–	100	–	0.05
PGD	0.3	250	–	–
MIM	0.3	250	–	–

x_i, this results in misclassification, and the rate is calculated as:

$$\frac{1}{n} \sum_i I(\mathrm{argmax}_k P(y_k|\bar{x}_i) \neq y_i).$$

We use L_p, which measures the PM by $p-norm$ distance as:

$$\|\delta\|_p = \left(\sum_{i=1}^{n} \|\bar{x}_i - x_i\|^p \right)^{\frac{1}{p}}$$

For $p-norm$, studying the L_0, L_2 and L_∞ is very common [33]. In this study, we measure the count of changes in the adversarial example compared to the original sample and L_2 distance of the adversarial example and the original one. Note that these two metrics indicate magnitude of the perturbations. Finally, we measure the running time (RT) required to generate adversarial examples (in seconds) as a metric to compare the complexity of various attacks.

4.4 Results and Discussion

In order to provide better perspective of our findings, by following the main key contributions highlighted in our methodology, this section is broken down into attack configuration (section 4.4.1), defense configuration (section 4.4.2), and executable configuration (section 4.4.3). For each of these configurations, we discuss the results for both of the Microsoft Windows and IoT malware datasets.

4.4.1 Generic Attacks Configuration. Our main goal is to generate adversarial samples that are able to fool the DL-based malware detector. Note that this configuration is based on applying the perturbations on the images generated from malware binaries, thus there is no guarantee regarding the executability of the crafted AEs.

Windows. We generate adversarial examples using five generic adversarial attack methods mentioned in section 3.1. Table 5 shows the results for the different attack methods in terms of misclassification rate, perturbation magnitude, and running time. As shown in Table 5, PGD achieved the highest misclassification rate of 99.63%, while perturbing 89.14% of the pixels. However, PGD is expensive in term of its running time. On the other hand, FGSM and C&W are the top two methods with the lowest running times, while reaching reasonable misclassification rate of 99.08% and 99.45%, respectively.

One interesting observation from Table 5 is that all of these generic adversarial attack algorithms change a huge portion of the original image's pixels (e.g., more than 71%), which in turn correspond to parts of the original binaries. Thus, it can be easily

Table 5: Generating adversarial examples for Windows binaries using generic adversarial attack methods.

Method	MR (%)	Pixels		L_2 Dist.	RT (s)
		#	%		
FGSM	99.08	8,594	83.92	25.86	**112.10**
PGD	**99.63**	9,128	89.14	23.11	1,339.60
C&W	99.45	**7,316**	**71.44**	4.09	2,071
MIM	99.54	8,732	85.27	25.08	6,875
DeepFool	99.45	7,750	75.68	**3.23**	15,545.70

Table 6: Generating adversarial examples for IoT binaries using different adversarial attack methods

| Method	MR (%)	Pixels		L_2 Dist.	RT (s)
		#	%		
PGD	**100**	9,452	92.30	23.36	3,979.80
FGSM	93.19	9,558	93.33	26.98	**51**
C&W	**100**	**8,932**	**87.22**	**2.44**	975.30
MIM	**100**	9,296	90.78	24.77	3,926.50
DeepFool	**100**	9,258	90.41	3.68	1,798.10

expected that the applied perturbation on the grayscale images may result in a corrupted binary that will not be executable.

IoT. The same set of attacks and evaluation metrics as in section 4.4.1 are applied on the IoTPOT dataset. As such, the results with the IoTPOT dataset are shown in Table 6. As shown, we can observe that all adversarial attack methods achieved a misclassification rate of 100%, except for the FGSM method. Here the C&W method is shown to outperform other methods considering the L_2 distance between the generated AEs and original inputs, although with a larger number of changed pixels. In particular, we notice that the number of changed pixels varies from 87.22% in C&W to 93.33% in the FGSM approach. The huge number of changed pixels in the generic adversarial approaches may make the crafted adversarial binaries unexecutable, which is not desirable in effective evasion techniques for software (that is supposed to be executed).

Executability. We note *all* adversarial examples resulting from both malware datasets, using all methods above were unexecutable.

4.4.2 Defense Configuration. Defense configuration aims to improve the robustness of the DL-based malware detection systems against various generic adversarial attack methods.

Windows. We implement the adversarial training method to increase the robustness of the DL-based malware detector against generic adversarial attack methods. To do so, we trained a model using the original inputs and the corresponding adversarial examples, which were generated using the aforementioned generic algorithms. The misclassification rates before and after the adversarial training for all adversarial attack methods are shown in Table 7. As it can be seen from Table 7, training over adversarial examples improves the robustness of the neural network against adversarial learning attacks. For instance, the evasion rate has dropped significantly from 99.08% into 8.68% (which is more than 90% improvement on the robustness of the model) for FGSM in Windows binaries after adversarial training. We observe similar pattern for other adversarial attack approaches.
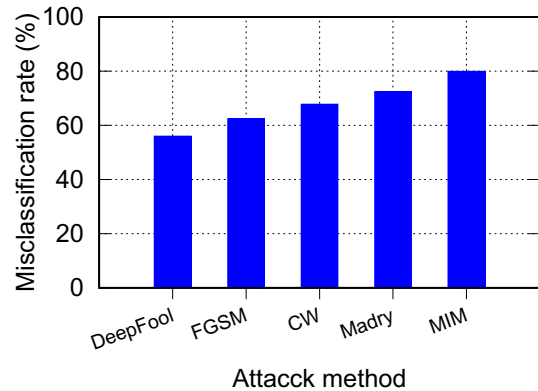
Figure 4: Misclassification rate of benign executable files due to appended malicious adversarial samples. We found that the misclassification rate is highly correlated with the # of distorted pixels and the L_2 distance.

Table 7: Misclassification rate on Windows and IoT before and after applying adversarial training. * indicates to the misclassification rate after applying adversarial training.

Method	Misclassification (%)		Misclassification* (%)	
	Windows	IoT	Windows	IoT
FGSM	99.08	93.19	8.68	0.80
C&W	99.45	100	26.12	15.76
DeepFool	99.45	100	83.24	97.43
PGD	99.63	100	27.80	5.47
MIM	99.54	100	77.76	4.98

IoT. We implement the same defense for the IoT malware detector by training a model on the IoT dataset and generating the corresponding adversarial examples. The misclassification rate before and after adversarial training for all adversarial attack methods are shown in Table 7. As it can be seen from Table 7, training over adversarial examples improves the robustness of the neural network against the adversarial learning attacks. For instance, the robustness of the DL-based IoT malware detection system has improved more than 99% for the FGSM approach in IoT binaries after adversarial training. The same pattern is observed for other methods.

4.4.3 COPYCAT. The results of our proposed approach, COPYCAT, for generating executable adversarial binaries are reported as followed. **Windows.** In this paper, we introduced AE padding and sample injection. Figure 4 shows the results of the AE padding method, by appending the adversary example, generated using generic adversarial atatck methods, to the binaries of the original executable sample. As it can be seen, we achieved the highest misclassification of 79.9% using MIM as our adversarial example generator. Table 8 and Table 13 show the results of the sample injection method, by injecting benign sample binaries at the end of malware sample binaries (M2B), and vice versa (B2M)—when we inject a benign sample into a malware sample we are in fact converting a malware sample into a benign sample (M2B). In the first case, we achieved an overall misclassification rate of 98.9%, while the misclassification rate of M2B was 73.5% by injecting a

benign sample with 4.4MB size. In the second case, we achieve a misclassification rate of B2M (regardless of the class) of 98.6% with a malware sample of size 11.5MB.

IoT. We show results for the sample injection method, which consists of injecting benign binaries into malware and vice versa. Table 9 shows the results of injecting benign binaries into malware samples, by using a benign sample of size 11.5MB, we achieve a misclassification of 96.5%. Whereas Table 10 shows the results of injecting malware binaries into benign samples, achieving a misclassification accuracy of 95.65% using a malware of size 2.4MB.

Executability. *All* adversarial examples that resulted from both malware datasets, using the two methods above, were executable.

4.5 Generalization with Transferability

To investigate the generalization of the generated adversarial binaries using COPYCAT, several experiments are conducted. To do so, we train another DL model with the following network structure: we use three consecutive fully connected layers of size 64 connected to the input vector, followed by a dropout with a probability of 0.5. Similarly, the output of the dropout function is fully connected with another softmax dense layer. Throughout our experiments we found that the injection of malware samples can force this Deep Neural Network (DNN)-based model to misclassification, with a similar performance to that of the CNN-based malware detection system. The results of the benign-to-malware misclassification attack are shown in Table 11, where it is concluded that the generated adversarial binaries are general, and can be used to possibly fool other models with different structures. For instance, the sample injection configuration of COPYCAT is able to force the DNN-based malware detection system to misclassify benign samples to malicious with a varying misclassification rate from 78.21% to 100% for samples injected with size of 2.2MB and 11.5MB, respectively. In addition, we found that the injection of a benign sample into a malicious sample can result in an evasion rate of up to 89.28% in a DNN-based malware detection system, as the detailed results are shown in Table 15.

We also observed a similar set of results for the IoT dataset, as shown in Table 12 and Table 14 for benign and malicious sample injection, respectively. For example, we were able to achieve a misclassification rate of 80% and 97.58% by injecting a benign and malicious file, respectively, in DNN-based IoT malware detection.

4.6 Size Trade-off

Injection method's outcomes (misclassification) depend on the size of the injected sample, due to the behavior of NN in general. Neural networks are well known for recognizing patterns, and increasing the size of the injected sample will increase the portion of sample's pattern against the overall layout. In general, a larger injected sample results in a higher effect on misclassification.

The attacks discussed in section 4.4.1 do not guarantee the executablity of the generated sample, as each pixel in the visualization corresponds to some of the data in the original sample. Changing one bit (or pixel) may corrupt the sample, leading to unexecutable adversarial example. Our proposed methods preserve the functionality and executability by padding data into unreachable point of the application, which is usually after the exit code. These methods

will not affect the application in any way except by increasing its size, which we consider as a trade-off cost of executability.

5 RELATED WORK

Machine and deep learning algorithms for malware detection have been actively pursued. For example, malware visualization is a technique that transforms binaries into images, which are then used to build machine/deep learning-based malware detection systems [29, 30]. Cui *et al.* [15] introduced a malware detection method using deep learning, and by transferring the malicious code into grayscale images. They achieved an accuracy rate of 94.5% on the Vision Research Lab dataset [48], claiming better accuracy rate and execution time compared to static [2] and dynamic feature analysis [4]. Similarly, Ni *et al.* [49] proposed a malware detection system, called MCSC, that is build over 10805 grayscale images consisting of nine different malware families. Su *et al.* [14] presented a lightweight IoT malware classifier that works based on visualization and CNN model, and achieves a detection rate of 94% over goodware and DDoS malware classification tasks [50]. Fu *et al.* [16] presented a fine-grained malware detector, through RF, KNN, and SVM models, that was trained over colored images generated from malware binaries. Their method was able to achieve a malware detection accuracy rate of 97.47%, and family classification accuracy rate of 96.85% using random forest model. Moreover, Fan *et al.* [51] presented a Metagraph2vec based malware detection system. Hou *et al.* [52] investigated android malware detection system based on structured heterogeneous information network.

Arp *et al.* [53] presented a light-weight method, DREBIN, for detecting malicious android applications based on machine learning models trained over features extracted from static analysis. McDermott *et al.* [54] introduced a Mirai detector based on deep learning, which could distinguish Mirai botnet traffic from normal IoT IP camera traffic with 99% accuracy. Pajouh *et al.* [55] explored the potential of using recurrent neural networks to detect ARM-based IoT malware, achieving detection accuracy of 98.18% using LSTM structures to analyze OpCode sequences. Furthermore, Wang *et al.* [56] proposed an adversary resistant neural network with an application to malware detection based on feature nullification.

Although addressing a different problem, and using a different approach, COPYCAT is inspired in part by Cha *et al.*'s [45], where they propose the idea of platform-independent programs. Moreover, the approach in this work is in line with although concurrent to the work of Kolosnjaji *et al.* [24] and kreuk *et al.* [25]. However, COPYCAT is different in multiple aspects, as highlighted earlier; *e.g.* performing both untargeted and targeted misclassification, tackling the visualization-based malware detection approach, providing higher evasion rate, and targeting both windows and IoT malware. We claim that COPYCAT is able to acheive high attack success rate while generating executable adversarial binaries, unlike the work of Liu *et al.* [23], where the main focus is maximizing the transferability regardless of the executability of the generated binaries.

6 LIMITATION AND FUTURE WORK

We have explored two methods to generate adversarial examples that fool the visualization-based malware detection system, while

Aminollah Khormali, Ahmed Abusnaina, Songqing Chen, DaeHun Nyang, and Aziz Mohaisen

Table 8: Size vs. MR of B2M (Windows)

Malware size	MR(%)
0.82 MB	82.1
1.10 MB	88.5
2.20 MB	96.4
11.50 MB	**98.6**

Table 9: Size vs. MR of M2B (IoT).

Benign size	MR (%)
1.0 MB	67.5
1.9 MB	76.7
3.6 MB	95.5
11.5 MB	**96.5**

Table 10: Size vs. MR of B2M (IoT).

Malware size	MR (%)
256 KB	56.50
553 KB	92.27
1.1 MB	95.17
2.4 MB	**95.65**

Table 11: Transferability: Size vs. MR of B2M (Win).

Malware size	MR(%)
0.82 MB	90.09
1.10 MB	94.06
2.20 MB	78.21
11.50 MB	**100**

Table 12: Transferability of M2B (IoT).

Benign size	MR (%)
1.0 MB	56.00
1.9 MB	70.50
3.6 MB	36.75
11.5 MB	**80.00**

Table 13: Size vs. MR of M2B and overall (Windows).

Benign Size	Misclassification (%)	
	Overall	Benign
1.2 MB	76.2	56.1
1.4 MB	90.2	37.8
3.5 MB	89.3	63.0
4.4 MB	**98.9**	**73.5**

Table 14: Transferability: Size vs. MR of B2M (IoT).

Malware size	MR (%)
256 KB	85.02
553 KB	82.60
1.1 MB	95.17
2.4 MB	**97.58**

Table 15: Transferability: Size vs. MR of M2B and overall (Windows).

Benign Size	Transferability (%)	
	Overall	Benign
1.2 MB	37.5	23.7
1.4 MB	72.4	18.1
3.5 MB	66.0	42.3
4.4 MB	**89.3**	**34.4**

output of the deep model while guaranteeing the functionality and executability of the generated sample. By designing three configurations; an adversarial attack configuration, a defense configuration and an executability configuration, we have evaluated the performance of the proposed attacks through extensive experiments. The executable configuration is capable of generating adversarial examples that successfully force the deep learning-based model to a misclassification rate of 98.9% and 96.5% on Windows and IoT malware datasets, respectively, while preserving executability and maintaining the same functionality of the original malware.

Authors contribution. AK and AA contributed equally; names are ordered by seniority. AK validated the generic attacks and COPYCAT. AA developed all scripts, obtained all results, and helped analyzing them. AA drew the figures. SC and DN helped advising and providing feedback on initial ideas and helped revising the work. AA and AK wrote initial draft that was incorporated in this manuscript, written by AM. AM developed the idea of COPYCAT.

preserving executability and functionality of the binaries. However, both of the proposed methods require appending data into the original sample. Generalizing this over multi-class classifiers, requires padding of at least one AE of each target class to ensure targeted misclassification. For example, to support a 10-class classifier would require multiplying the size of the original sample by approximately 10 times. This, in turn, constitutes the main shortcoming of the AE padding method. In addition, our results indicate that the performance of the sample injection method positively correlates with the size of the injected sample. Furthermore, the adversarial training method depends on the injected AEs similar to the ones used by the adversary, which may not be the case in practice. It is computationally expensive and almost impossible to find adversarial inputs by all attack models, which results in that the adversarial training is not-adaptive to unknown attacks.

7 CONCLUSION

Recent research shows that deep learning is vulnerable to adversarial examples, whereby an adversary can manipulate the input of a deep learning model in such an unnoticeable way by adding minimal perturbation to the input to generate a desired output, *i.e.,* misclassification. In this work, we have studied several generic adversarial attacking methods and incorporated them into deep learning malware detection, showing a great success in fooling classifiers. To allow misclassification while maintaining executability, a shortcoming of state-of-the-art, we proposed AE padding and sample injection, two new adversarial methods to manipulate the

REFERENCES

[1] P. Faruki, V. Laxmi, A. Bharmal, M. S. Gaur, and V. Ganmoor, "Androsimilar: Robust signature for detecting variants of android malware," *Journal of Information Security and Applications*, vol. 22, pp. 66–80, 2015.

[2] H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 479174, 2015.

[3] A. Bulazel and B. Yener, "A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web," in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium.* ACM, 2017, p. 2.

[4] A. Al-Dujaili, A. Huang, E. Hemberg, and U. O'Reilly, "Adversarial deep learning for robust detection of binary encoded malware," in *Proceedings of the IEEE Security and Privacy Workshops, SP Workshops*, 2018, pp. 76–82.

[5] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th international symposium on visualization for cyber security*, 2011, p. 4.

[6] K. Han, J. H. Lim, B. Kang, and E. G. Im, "Malware analysis using visualized images and entropy graphs," *International Journal of Information Security*, vol. 14, no. 1, pp. 1–14, 2015.

[7] B. C. M. Cappers, P. N. Meessen, S. Etalle, and J. J. van Wijk, "Eventpad: Rapid malware analysis and reverse engineering using visual analytics," in *IEEE Symposium on Visualization for Cyber Security, VizSec 2018, Berlin, Germany, October 22, 2018*, 2018, pp. 1–8.

[8] M. Angelini, G. Blasilli, P. Borrello, E. Coppa, D. C. D'Elia, S. Ferracci, S. Lenti, and G. Santucci, "Ropmate: Visually assisting the creation of rop-based exploits," in *IEEE Symposium on Visualization for Cyber Security, VizSec 2018, Berlin, Germany, October 22, 2018*, 2018, pp. 1–8.

[9] S. Chen, S. Chen, N. V. Andrienko, G. L. Andrienko, P. H. Nguyen, C. Turkay, O. Thonnard, and X. Yuan, "User behavior map: Visual exploration for cyber security session data," in *IEEE Symposium on Visualization for Cyber Security, VizSec 2018, Berlin, Germany, October 22, 2018*, 2018, pp. 1–4.

[10] R. R. Gómez, Y. Nadji, and M. Antonakakis, "Towards designing effective visualizations for dns-based network threat analysis," in *2017 IEEE Symposium on*

Visualization for Cyber Security, VizSec 2017, Phoenix, AZ, USA, October 2, 2017, 2017, pp. 1–8.

[11] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang, "Large-scale and language-oblivious code authorship identification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018,* 2018, pp. 101–114.

[12] A. Makandar and A. Patrot, "Malware class recognition using image processing techniques," in *Proceedings of the 2017 International Conference on Data Management, Analytics and Innovation (ICDMAI),* 2017, pp. 76–80.

[13] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, "Adversarial examples for malware detection," in *Proceedings of the 22nd European Symposium on Research Computer Security - ESORICS, Part II,* 2017, pp. 62–79.

[14] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, "Lightweight classification of iot malware based on image recognition," in *Proceedings of the 2018 IEEE Annual Computer Software and Applications Conference, COMPSAC,* 2018, pp. 664–669.

[15] Z. Cui, F. Xue, X. Cai, Y. Cao, G. Wang, and J. Chen, "Detection of malicious code variants based on deep learning," *IEEE Trans. Industrial Informatics,* vol. 14, no. 7, pp. 3187–3196, 2018.

[16] J. Fu, J. Xue, Y. Wang, Z. Liu, and C. Shan, "Malware visualization for fine-grained classification," *IEEE Access,* vol. 6, pp. 14 510–14 523, 2018.

[17] B. Alipanahi, A. Delong, M. T. Weirauch, and B. J. Frey, "Predicting the sequence specificities of dna-and rna-binding proteins by deep learning," *Nature biotechnology,* vol. 33, no. 8, p. 831, 2015.

[18] A. Mohaisen, O. Alrawi, and M. Mohaisen, "AMAL: high-fidelity, behavior-based automated malware analysis and classification," *Computers & Security,* vol. 52, pp. 251–266, 2015.

[19] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: A simple and accurate method to fool deep neural networks," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR,* 2016, pp. 2574–2582.

[20] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen, "Adversarial learning attacks on graph-based IoT malware detection systems," in *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems, ICDCS,* 2019.

[21] N. Papernot, P. D. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *Proceedings of the IEEE Symposium on Security and Privacy, SP Workshops,* 2016, pp. 582–597.

[22] A. Khormali, A. Abusnaina, D. Nyang, M. Yuksel, and A. Mohaisen, "Examining the robustness of learning-based ddos detection in software defined networks," in *Proceedings of the IEEE conference on dependable and secure computing, IDSC,* 2019.

[23] X. Liu, J. Zhang, Y. Lin, and H. Li, "Atmpa: Attacking machine learning-based malware visualization detection methods via adversarial examples," 2019.

[24] 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 *Proceedings of the 26th European Signal Processing Conference, EUSIPCO,* 2018, pp. 533–537.

[25] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Deceiving end-to-end deep learning malware detectors using adversarial examples," in *Proceedings of the NeurIPS Workshop on Security in Machine Learning,* 2018.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of Advances in neural information processing systems,* 2012, pp. 1097–1105.

[27] D. C. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *IEEE Conference on Computer Vision and Pattern Recognition, Providence,* 2012, pp. 3642–3649.

[28] "Deep learning and machine learning differences: Recent views in an ongoing debate," Mar 2017. [Online]. Available: http://www.dataversity.net/deep-learning-machine-learning-differences-recent-views-ongoing-debate/

[29] J. Zhang, Z. Qin, H. Yin, L. Ou, and Y. Hu, "IRMD: malware variant detection using opcode image recognition," in *Proceedings of the 22nd IEEE International Conference on Parallel and Distributed Systems, ICPADS,* 2016, pp. 1175–1180.

[30] D. Lee, I. S. Song, K. J. Kim, and J.-h. Jeong, "A study on malicious codes pattern analysis using visualization," in *Proceedings of the International Conference on Information Science and Applications (ICISA),* 2011, pp. 1–5.

[31] G. Conti, S. Bratus, A. Shubina, A. Lichtenberg, R. Ragsdale, R. Perez-Alemany, B. Sangster, and M. Supan, "A visual study of primitive binary fragment types," *White Paper, Black Hat USA,* 2010.

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

[33] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy, SP 2017,* 2017, pp. 39–57.

[34] B. Wang, Y. Yao, B. Viswanath, H. Zheng, and B. Y. Zhao, "With great training comes great vulnerability: practical attacks against transfer learning," in *Proceedings of the 27th {USENIX} Security Symposium ({USENIX} Security 18),* 2018, pp.

1281–1297.

[35] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the ACM on Asia Conference on Computer and Communications Security, AsiaCCS,* 2017, pp. 506–519.

[36] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *IEEE European Symposium on Security and Privacy, EuroS&P,* 2016, pp. 372–387.

[37] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, "There is no free lunch in adversarial robustness (but there are unexpected benefits)," *arXiv preprint arXiv:1805.12152,* 2018.

[38] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations,* 2018.

[39] T. Miyato, S.-i. Maeda, M. Koyama, K. Nakae, and S. Ishii, "Distributional smoothing with virtual adversarial training," in *Proceedings of the 2016 International Conference on Learning Representations.,* 2016.

[40] F. Tramèr, A. Kurakin, N. Papernot, D. Boneh, and P. D. McDaniel, "Ensemble adversarial training: Attacks and defenses," in *Proceedings of the 2018 International Conference on Learning Representations.,* 2018.

[41] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *Proceedings of the 2015 International Conference on Learning Representations.,* 2015.

[42] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, "Boosting adversarial attacks with momentum," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* 2018.

[43] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *Proceedings of the 2014 International Conference on Learning Representations.,* 2014.

[44] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security,* 2017, pp. 506–519.

[45] S. K. Cha, B. Pak, D. Brumley, and R. J. Lipton, "Platform-independent programs," in *Proceedings of the 17th ACM conference on Computer and communications security,* 2010, pp. 547–558.

[46] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "Iotpot: A novel honeypot for revealing current iot threats," *JIP,* vol. 24, no. 3, pp. 522–533, 2016.

[47] N. Papernot, N. Carlini, I. Goodfellow, R. Feinman, F. Faghri, A. Matyasko, K. Hambardzumyan, Y.-L. Juang, A. Kurakin, R. Sheatsley *et al.,* "cleverhans v2. 0.0: an adversarial machine learning library," *arXiv preprint arXiv:1610.00768,* 2016.

[48] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th international symposium on visualization for cyber security,* 2011, p. 4.

[49] S. Ni, Q. Qian, and R. Zhang, "Malware identification using visualization images and deep learning," *Computers & Security,* vol. 77, pp. 871–885, 2018.

[50] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "Iotpot: analysing the rise of iot compromises," *EMU,* vol. 9, p. 1, 2015.

[51] Y. Fan, S. Hou, Y. Zhang, Y. Ye, and M. Abdulhayoglu, "Gotcha - sly malware!: Scorpion A metagraph2vec based malware detection system," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018,* 2018, pp. 253–262.

[52] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Hindroid: An intelligent android malware detection system based on structured heterogeneous information network," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 2017, pp. 1507–1515.

[53] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: effective and explainable detection of android malware in your pocket," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS,* 2014.

[54] C. D. McDermott, F. Majdani, and A. Petrovski, "Botnet detection in the internet of things using deep learning approaches," in *Proceedings of the International joint conference on neural networks 2018 (IJCNN 2018).* ACM, 2018.

[55] H. HaddadPajouh, A. Dehghantanha, R. Khayami, and K.-K. R. Choo, "A deep recurrent neural network based approach for internet of things malware threat hunting," *Future Generation Computer Systems,* vol. 85, pp. 88–96, 2018.

[56] Q. Wang, W. Guo, K. Zhang, A. G. O. II, X. Xing, X. Liu, and C. L. Giles, "Adversary resistant deep neural networks with an application to malware detection," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 2017, pp. 1145–1153.

A APPENDIX

This section provides further information regarding adversarial learning algorithms utilized in this study and their categorization based on adversary's goal and knowledge.

A.1 Adversarial Learning Algorithms

In this study, we utilized five generic adversarial attack methods to generate visualized AEs. A brief description of these algorithms are provided in the following, and we refer more interested readers to the original works.

A.1.1 Fast Sign Gradient Sign Method (FGSM). FGSM is a fast method designed to generate adversarial examples without guaranteeing similarity of the generated examples, when compared to the original samples [41]. FGSM works based on a one-step gradient update, which can be expressed using:

$$\delta = \epsilon \cdot sign\left(\nabla_x J_\theta\left(x, l\right)\right), \tag{1}$$

where δ is the perturbation parameter, ϵ is a small scalar value that limits the distortion of the image and sets the magnitude of the perturbation. Moreover, $sign(.)$ denotes the sign function, $J(.)$ is the cost function, where x is the input image and l is the label associated with it, and ∇ computes the gradient of the cost function J around the current value of x. Finally, the output adversarial image is by:

$$x' = x + \delta, \tag{2}$$

where x' is the adversarial example corresponding to image x. Controlling the distortion parameter ϵ will affect the L_2 norm distance between the original and the adversarial examples. A higher ϵ value corresponds to a larger L_2 distance. For visualization, x' values are clipped to fit into [0-255] channel scale [14].

A.1.2 Carlini & Wagner (C&W) Method. Carlini and Wagner [33] introduced three gradient based attacks by optimizing the penalty and distance metrics on L_∞, L_2, and L_0 norms in the process of generating adversarial examples, as follows:

$$min\, ||\delta||_p^2 : \; g\left(x + \delta\right) = y' \;\&\; x + \delta \in X, \tag{3}$$

where δ is the perturbation parameter, $g(.)$ is the objective function based on the hinge loss, y' is the targeted class, and x is the input image. This method ensures that the added perturbation will be small and only as required to misclassify to class y'. A small modification can be made to launch non-targeted attacks by computing:

$$min\, ||\delta||_p^2 : \; g\left(x + \delta\right) \neq y \;\&\; x + \delta \in X. \tag{4}$$

In this work, we focus on the L_2-based C&W attack to generate adversarial examples. The L_2 distance represents the required changes into the binaries to generate the adversarial example, where a lower L_2 distance corresponds to a smaller perturbation in the binary to generate the example. The perturbation δ is defined by:

$$\delta = \frac{1}{2}\left(tanh\left(w\right) + 1\right) - x, \tag{5}$$

where $tanh(.)$ is the hyperbolic tangent function, and w is an auxiliary variable optimized by:

$$\min_w ||\frac{1}{2}\left(tanh\left(w\right) + 1\right)||_2 + c \cdot g\left(\frac{1}{2}\left(tanh\left(w\right) + 1\right)\right), \tag{6}$$

where c is a constant. The C&W method minimizes the L_p norm distance between the generated adversary example and the original image to increase the similarity and harden the detection process.

A.1.3 DeepFool Method. DeepFool is originally designed to be untargeted iterative attack based on the L_2 distance metric [19]. It views the neural network as a linear structure, where each class is separated with a hyperplane. To generate an adversarial example, DeepFool increases the distance per iteration between the input and the class associated with that input. The class is defined by:

$$\hat{k}(x) = arg_k\, max\, f_k(x), \tag{7}$$

where $f_k(x)$ is the output of the objective function corresponding to class k, which represents the class with the highest probability. Assuming that $f(x)$ is an affine classifier represented by:

$$f(x) = W^T x + b, \tag{8}$$

then the perturbation needed to misclassify input x is described by:

$$arg_r\, min\, ||r||_2 \quad s.t.\, \exists k : w_k^T\left(x_0 + r\right) + b_k \geq w_{\hat{k}(x_0)}^T\left(x_0 + r\right) + b_{\hat{k}(x_0)}, \tag{9}$$

where w_k is mapped to the kth column of W. To generalize this method for multi-class non-linear structure, we find the next closest class to the input x using the following:

$$\hat{l}(x_0) = arg\, \min_{k \neq \hat{k}(x_0)} \frac{|f_k\left(x_0\right) - f_{\hat{k}(x_0)}\left(x_0\right)|}{||w_k - w_{\hat{k}(x_0)}||_2}. \tag{10}$$

By finding the minimum perturbation needed to misclassify the model using (11), we generate the AE using $x' = x + r_*$.

$$r_*\left(x_0\right) = \frac{|f_{\hat{l}(x_0)}\left(x_0\right) - f_{\hat{k}(x_0)}\left(x_0\right)|}{||w_{\hat{l}(x_0)} - w_{\hat{k}(x_0)}||_2^2}\left(w_{\hat{l}(x_0)} - w_{\hat{k}(x_0)}\right). \tag{11}$$

A.1.4 Momentum Iterative Method. Momentum iterative method (MIM) is based on applying momentum gradient over basic FGSM to generate adversarial examples [42], the main goal is to maintain efficiency against black box models. The main object for this method is to find x' that misclassify the model that satisfy the equation (12) where J is the loss function, and ϵ is small scalar that control the maximum distortion allowed.

$$arg\, \max_{x'}\, J\left(x', y\right), \quad s.t.\, ||x' - x||_\infty \leq \epsilon \tag{12}$$

The momentum gradient (M_g) then calculated using (13).

$$M_{g_{t+1}} = \mu M_{g_t} + \frac{\nabla_x J_\theta\left(x'_t, l\right)}{||\nabla_x J_\theta\left(x'_t, l\right)||} \tag{13}$$

where μ is the decay factor. Initially, x'_0 is the original input, and M_{g_0} is 0. Each iteration, x' will be updated according to this equation:

$$x'_{t+1} = x'_t + \epsilon \cdot sign\left(M_{g_t} + 1\right) \tag{14}$$

after n number of iterations, the x'_{t+1} will be returned as the adversarial example for input x.

A.1.5 PGD Method. Madry *et al.* introduced an iterative projected gradient descent (PGD) attack [38]. The aim of PGD is to generate an adversarial sample under a minimized empirical risk with a trade of high performance cost. The original empirical risk minimization (*ERM*) for the model defined by:

$$\mathbb{E}_{(x,y)\sim D}[L(x,y,\theta)], \tag{15}$$

where L is the loss function. By modifying the definition of *ERM* and allowing the adversary to perturb the input, S is described in:

$$\min_{\theta} \rho(\theta) : \rho(\theta) = \mathbb{E}_{(x,y)\sim D}[\max_{\delta \in S} L(x+\delta,y,\theta)], \tag{16}$$

where δ is the perturbation, and $\rho(\theta)$ is the objective function to minimize. As an iterative method, x'_{t+1} is updated each iteration according to the previous x'_t and the generated perturbation, after n iterations, x'_{t+1} will be returned as the adversarial sample. Our goal with PGD is to minimize the risk while adding perturbation in order to generate the adversarial example.