Enhancing Deep Neural Networks Against Adversarial Malware Examples

Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu

Abstract—Machine learning based malware detection is known to be vulnerable to adversarial evasion attacks. The state-of-the-art is that there are no effective countermeasures against these attacks. Inspired by the AICS'2019 Challenge organized by the MIT Lincoln Lab, we systematize a number of principles for enhancing the robustness of neural networks against adversarial malware evasion attacks. Some of these principles have been scattered in the literature, but others are proposed in this paper for the first time. Under the guidance of these principles, we propose a framework for defending against adversarial malware evasion attacks. We validated the framework using the Drebin dataset of Android malware. We applied the defense framework to the AICS'2019 Challenge and won, without knowing how the organizers generated the adversarial examples. However, we see a ~22% difference between the accuracy in the experiment with the Drebin dataset (for binary classification) and the accuracy in the experiment with respect to the AICS'2019 Challenge (for multiclass classification). We attribute this gap to a fundamental barrier that without knowing the attacker's *manipulation set*, the defender cannot do effective Adversarial Training.

Index Terms—Adversarial Machine Learning, Deep Neural Networks, Malware Classification, Adversarial Malware Detection.

1 Introduction

Malware remains a big threat to cyber security despite communities' tremendous countermeasure efforts. For example, Symantec [2] reports seeing 355,419,881 new malware variants in year 2015, 357,019,453 in year 2016, and 669,974,865 in year 2017. Worse yet, there is an increasing number of malware variants that attempted to undermine anti-virus tools and indeed evaded many malware detection systems [3].

In order to cope with the increasingly severe situation, we have to resort to machine learning techniques for automating the detection of malware in the wild [4]. However, machine learning based techniques are vulnerable to adversarial evasion attacks, by which an adaptive attacker perturbs or manipulates malware examples into adversarial examples that would be detected as benign rather than malicious (see, for example, [5], [6], [7], [8]). The state-of-the-art is that there are many attacks, but the problem of effective defense is largely open. This is indeed the context in which the AICS'2019 Malware Classification Challenge is proposed. In a broader context, adversarial malware examples are a particular kind of attacks against adversarial machine learning. Although adversarial machine learning has received much attention in application domains such as image processing (see, e.g., [9], [10], [11]), the problem of adversarial malware examples are much less investigated [6], [12].

The AICS'2019 Challenge mentioned above is essentially about whether we can defend adversarial examples that wages evasion attacks in the dark, meaning:

- We (i.e., any team participating in the Challenge as the defender) are given a training set in the form of anonymized feature representation by the Challenge organizer (i.e., we do not even know what the feature names are), as well as the corresponding ground-truth labels. We are informed by the Challenge organizer that the training data contains *no* adversarial malware examples.
- We are given a test set (again, in anonymized feature representation) and are told that the test set contains both adversarial malware examples and non-adversarial examples. We do not know what attacks are used by the Challenge organizers to generate those adversarial examples. We do not know which examples in the test set are adversarial examples. This means that we neither know which are adversarial examples, nor the attacks that are used to generate them, nor the manipulation set.
- Our task is to classify the test set into the correct types of malware examples, including both adversarial examples and non-adversarial examples.

The setting of the Challenge is realistic because in the real world defenders are often in the dark because they do not know what methods are used by the attackers to generate adversarial examples, which are adversarial malware examples, what are the attackers' manipulation sets. The importance of the problem in defending against adversarial malware examples and the realistic scenario of the Challenge motivated the present study.

1.1 Our Contributions

In this paper, we make the following contributions. First, we propose, to the best of our knowledge, the first systematic framework that aims to enhance the robustness

A preliminary version of the paper was presented at AICS'2019, which does not published formal proceedings [1].

- *D. Li and Q. Li are with School of Computer Science and Engineering, Nanjing University of Science and Technology.*
- *Y. Ye is with Department of Computer and Data Sciences, Case Western Reserve University.*
- *S. Xu is with Department of Computer Science, University of Texas at San Antonio. E-mail: shouhuai.xu@utsa.edu*

of malware classifiers against adversarial evasion attacks. The framework is designed under the guidance of a set of principles, some of which are known but scattered in the literature (e.g., using an ensemble of classifiers), but others are explicitly proposed for the first time, such as the following. We propose using the capability of the optimal white-box attack to bound the capability of any ℓ_p $(p \geq 1)$ norm based gray-box attack from above, and propose using semantics-preserving representation learning for malware classification and detection.

Second, we empirically validated the effectiveness of the framework against 11 grey-box attacks and 10 white-box attacks (i.e., 21 attacks in total). The 11 grey-box attacks include the Random attack, the Mimicry attack [13], the Fast Gradient Sign Method (FGSM) attack [10], the Grosse attack [12], the Bit Gradient Ascent (BGA) attack [6], the Bit Gradient Ascent (BCA) attack [6], four variants of the Projected Gradient Descent (PGD) attack [14], and the Elastic-net Attacks against DNNs (EAD) attack [15]. The 10 white-box attacks leverage the victim models directly and the attack algorithms are the same as the latter 10 ones mentioned above. Among these attacks, the four variants of the PGD attack and the EAD attack (i.e., five in total) are used to be investigated in other application settings and are adapted to the adversarial malware detection domain for the first time in the present paper. In these experiments, adversarial malware examples are generated by manipulating regular malware examples while preserving their malicious functionalities. Our findings include:

- Two defenses, namely adversarial regularization and Denoising Auto-Encoder (DAE), can be used even if there are no adversarial malware evasion attacks. This is important because in practice the defender might not know whether the attacker will be waging adversarial malware evasion attacks or not, meaning that defense against adversarial examples should have no side-effect. That is, the defender can always use these two defenses without worrying about any significant side-effects.
- The popular defense of Adversarial Training is effective against grey-box adversarial malware evasion attacks, but not effective against white-box adversarial malware evasion attacks. This is because a white-box attacker knows how the defender trained its defense model and can generate adversarial examples that are far from the ones in the defender's training set. This suggests that it is much harder to defend against white-box attacks.

Third, we apply the framework to the AICS'2019 adversarial malware classification challenge organized by the MIT Lincoln Lab. According to the Challenge organizers, there were "over 300 participants attempted to download and classify the malware data set" [16] and we won the Challenge by achieving a 73.60% Harmonic mean score (which is the metric the organizer chose to use before making the data available); i.e., we achieve the *highest* score (i.e., classification accuracy) among all of the participating teams.

Fourth, after announcing we won the Challenge, the organizer made the test data with ground-truth labels publicly available at http://www-personal.umich.edu/~arunesh/AICS2019/challenge.html. In order to understand why we only achieved a 73.60% Harmonic mean score, we leveraged the test data ground-truth labels to conduct a further study. Using a retrospective analysis of the AICS'2019 Challenge, we find that Adversarial Training based models (leveraging adversarial examples generated by ourselves) tend to overfit the perturbations in the training set, this may be caused by the imbalance in the training set (because we do not observe this phenomenon in the experiment based on the balanced Drebin dataset) and/or caused by the fact that the training set is small.

Fifth, we observe that the framework is highly effective against adversarial malware evasion attacks on the Drebin dataset (which gives binary ground-truth labels), but only achieves a 73.60% in the Harmonic Mean score against the AICS'2019 Challenge (with or without knowing the ground truth of the test set). From this large discrepancy we draw the following insights. (i) Without knowing *any* information about the attack (which is the case of the AICS'2019 Challenge), adversarial training and other existing defenses (including our framework) have limited capability against adversarial evasion attacks. This is "information" barrier (i.e., knowing nothing about the attack) may be the norm in cyber defense against adversarial malware evasion attacks, highlighting the importance of making effort to collect information about the attacker and attacks. That is, "knowing the enemy" is important, as can be shown using the following extreme scenario: If the defender knows how the attacker manipulates non-adversarial examples to generate adversarial examples, then the defender can simply "undo" the manipulation, which degenerates the problem to malware detection in the absence of adversarial examples, leading to much higher detection accuracy. It is worth mentioning that "knowing the enemy" is indeed the first principle we propose when we took the Challenge and *before* we draw the insight mentioned above, as shown in the previous version of the present paper [1]. (ii) The "noise" in the adversarial examples that are incorporated into the training set may mislead the defender's learning/optimization algorithms, causing ineffective classifiers.

Last but not the least, we made our the code of our models publicly available at https://github.com/deqangss/aics2019_challenge_adv_mal_defense.

1.2 Related Work

Since the present paper focuses on defense against adversarial malware classification, we review existing studies in this topic by emphasizing two complementary approaches: *input prepossessing* and *adversarial training*.

Input Prepossessing. Input prepossessing transforms the input to a different representation with the aim to reduce the degree of perturbation to the original input. For example, Random Feature Nullification (RFN) randomly nullifies features in the training and test phases [17]; hash transformation leverages a locality-preservation property to reduce the degree of perturbation to the original input [18]; DroidEye [19] quantizes binary feature representation via count featurization.

Our framework uses binarization to reduce the degree of perturbation, which is inspired by the idea of feature

squeezing in the context of image processing [20]. This effectively reduces the perturbation space because there are now only two kinds of perturbations: flipping from '1' to '0' or flipping '0' to '1'. This means that we effectively consider the number of perturbations but not the 'scale' of perturbation (i.e., reducing the sensitivity of classifiers to small degrees of perturbation).

Adversarial Training. Adversarial training augments the training data with adversarial examples to improve the robustness of classifiers. This idea has been independently proposed in different application settings, including [10], [21] and [22], [23]. In particular, it has been proposed to consider adversarial training with the optimal attack, which in a sense corresponds to the worst-case scenario and therefore could lead to classifiers that are robust against the non-optimal attacks [6]. The challenge is of course to find the optimal attack. In our framework, we use this approach to regularize our model and seek the optimal attack via the gradient descent method.

1.3 Paper Outline

The rest of the paper is organized as follows. Section 2 presents the adversarial evasion attacks, including five attacks that are adapted to the adversarial malware detection domain for the first time. Section 3 describes our defense framework. Section 4 validates our defense framework with a real-world dataset. Section 5 presents the results when applying the framework to the AICS'2019 Challenge *without* knowing anything about the attack. Section 6 presents our further study after winning the AICS'2019 Challenge and being given the ground-truth labels of the test data of the Challenge.

2 Adversarial Malware Evasion Attacks

2.1 Basic Idea

Consider a non-adversarial malware example $z \in \mathcal{Z}$, where $\mathcal{Z}$ is the set of all possible malware examples or *example space*. Consider feature representation of z, denoted by $\mathbf{x} \in \mathcal{X}$, where $\mathcal{X}$ is the *feature space*. The feature representation $\mathbf{x}$ can be obtained via some *feature extraction* methods. A classifier $f : \mathcal{X} \to \mathcal{Y}$ takes $\mathbf{x}$ as input and outputs its label $y \in \mathcal{Y}$, where $\mathcal{Y}$ is the *label space*.

Consider the feature space $\mathcal{X}$, the adversarial evasion attack attempts to manipulate or perturb $\mathbf{x}$ into an adversarial version, denoted by $\mathbf{x}'$ such that

$$f(\mathbf{x}') \neq f(\mathbf{x}), \tag{1}$$

$$\text{s.t.} \quad \mathbf{x}' \in \mathcal{M}(\mathbf{x}, \mathcal{S}_{\mathbf{x}}) \tag{2}$$

$$||\mathbf{x}' - \mathbf{x}|| \leq \epsilon \tag{3}$$

where $\mathcal{M}(\mathbf{x}, \mathcal{S}_{\mathbf{x}})$ is the set of representation vectors derived from the non-adversarial feature representation $\mathbf{x}$ and a *manipulation set* $\mathcal{S}_{\mathbf{x}}$ (i.e. the set of manipulations that can preserve the malicious functionality of malware examples), $||\cdot||$ is a metric of interest (e.g., ℓ_p norm for some $p \in \{1, 2, \infty\}$), and ϵ is the upper bound on the perturbation operation. Let $\mathcal{M} = \{\mathcal{M}(\mathbf{x}, \mathcal{S}_{\mathbf{x}}) | \mathbf{x} \in \mathcal{X}\}$, namely the adversarial malware example feature representation set. Note that Eq.(3) says that perturbations may be bounded from above by a given ϵ in the norm. The *perturbation vector* is denoted by $\delta_{\mathbf{x}} = \mathbf{x}' - \mathbf{x}$. Since the manipulation is conducted in the feature space, the attacker needs to map $\mathbf{x}'$ back into the example space $\mathcal{Z}$ in order to obtain an executable adversarial malware example $z' \in \mathcal{Z}$. This is a requirement that distinguishes adversarial malware detection from adversarial machine learning in other application domains.

In the present paper, we focus on the classifier f that is learned as a neural network model, $\mathbf{F} : \mathcal{X} \to \mathbb{R}^o$, which outputs (*softmax*) the probability mass function over o classes (e.g., $o = 2$ means the classifier's output means whether a file z is *benign* or *malicious*). Let $L : \mathbb{R}^{|o|} \times \mathcal{Y} \mapsto \mathbb{R}$ be an appropriate loss function (e.g., cross-entropy). Since it is difficult to formulate the constraint in Eq.(1), researchers have proposed considering relaxation in two scenarios:

- In the case of *non-targeted* attacks, the objective is to maximize the cost of classifying the $\mathbf{x}'$ as y, namely

$$\max_{\mathbf{x}' \in \mathcal{M}} L(\mathbf{F}(\mathbf{x}'), y). \tag{4}$$

- In the case of *targeted* attacks, the objective is to minimize $L(\mathbf{F}(\mathbf{x}'), y_t)$ subject to $\mathbf{x}' \in \mathcal{M}$, namely

$$\min_{\mathbf{x}' \in \mathcal{M}} L(\mathbf{F}(\mathbf{x}'), y_t), \tag{5}$$

  where y_t ($y_t \neq y$) is a target label of $\mathbf{x}'$.

2.2 Threat Model

The threat model against malware classifiers and detectors can be specified by *what the attacker knows*, *what the attacker can do*, and *how the attacker wages the attack*.

2.2.1 What the attacker knows

There are three kinds of models from this perspective. A *black-box* attacker knows nothing about classifier f except what is implied by f's responses to the attacker's queries. A *white-box* attacker knows all kinds of information about f, including its learning algorithms, model parameters, defenses strategies, etc. A *grey-box* attacker knows an amount of information about f that resides in between the preceding two extremes. For example, the attacker may know the training set or feature definitions.

2.2.2 What the attacker can do

In evasion attacks, the attacker only can manipulate the test data, and more specifically malware examples, while obeying some constraints. One constrain is to preserve the malicious functionality of malware. Although the attacker can manipulate a malware example by inserting and deleting operators [24], [25], a simplifying assumption is to consider insertion only (e.g., flipping a feature value from '0' to '1' [5], [6], [12], [17], [26], [27], [28]. The other constraint is to maintain the relation between features. Using the ACIS'2019 malware classification challenge as an example, we note that n-gram (uni-gram, bi-gram, and tri-gram) features reflect sequences of Windows system API calls. This means that when the attacker inserts an API call into a malware example, several features related to this API call will need to be changed according to the definition of n-gram features.

2.2.3 How the attacker wages the attack

Researchers generate adversarial malware examples using various machine learning techniques such as genetic algorithms, reinforcement learning, generative networks, feed-forward neural networks, decision trees, and Support Vector Machine (SVM) [5], [22], [25], [26], [29], [30], [31]. In order to generate adversarial malware examples effectively and efficiently, attacks often leverage the gradients with respect to inputs of neural network [10], [32]. We consider a broad range of attack algorithms, some of which were introduced in the context of malware detection but the others were introduced in the context of image classification and then adapted to the context of malware detection.

Random Attack. We introduce this attack as a baseline attack in the adversarial malware detection domain. In this attack, the attacker randomly modifies a feature at each iteration until a predefined number steps or no more features to manipulate any more (if applicable). Note that the attack is algorithm-agnostic.

Mimicry Attack. This attack was introduced in [5], [13], [26], [33] for studying adversarial malware detection. In this attack, the attacker perturbs or manipulates a malware example such that the resulting adversarial version mimics a chosen benign example as much as possible. In order to reduce the degree of perturbations, the attacker may select the benign example to be close to the malware example that is to be manipulated to become an adversarial malware example.

FGSM Attack. This attack was introduced in the context of image classification [10] and then adapted to adversarial malware detection [6], [19]. FGSM perturbs a feature vector $\mathbf{x}$ in the direction of the ℓ_∞ norm of the gradients of the loss function with respect to the input, namely:

$$\mathbf{x}' = \mathrm{Proj}_{\mathcal{M}}\left(\mathbf{x} + \varepsilon \cdot \mathrm{sign}(\nabla_{\mathbf{x}} L(\mathbf{F}(\mathbf{x}), y))\right),$$

where $\varepsilon > 0$ is a scalar known as *step size*, $\nabla_{\mathbf{x}}$ indicates the derivative of the loss function $L(\mathbf{F}(\mathbf{x}), y)$ with respect to $\mathbf{x}$, and $\mathrm{sign}(\cdot)$ is the element-wise sign function that returns 0 when the input is $x = 0$ and returns 1 (-1) when the input $x > 0$ $(x < 0)$, and $\mathrm{Proj}_{\mathcal{M}}(\cdot)$ projects an input into $\mathcal{M}$.

Grosse Attack. This attack was introduced by Grosse et al. [12] in the context of malware detection. The attack considers sensitive features, namely the features have large positive gradients as far as the softmax output is concerned. The attack is to manipulate the absence of a feature (e.g., not making a certain API call) into the presence of the feature (i.e., making the API call), while preserving their malicious functionalities. These sensitive features can be identified by leveraging the gradients of the *softmax* output of a malware example with respect to the input.

BGA Attack and **BCA Attack**. In the context of malware detection, Al-Dujaili et al. [6] proposed two separate methods, dubbed BGA and BCA, for solving Eq. (4) when manipulating malware examples in binary encoded feature representation. Both attack methods iterate multiple steps. In each iteration, BGA increases the feature value from '0' to '1' if corresponding partial derivative of the loss function with respect to the input is greater than or equal to the gradient's ℓ_2 norm divided by $1/\sqrt{dim}$, where dim is the input dimension. In contrast, BCA flips '0' to '1' for a component at the iteration corresponding to the maximum gradient of the loss function with respect to the input. In order to preserve malware examples' malicious functionalities, both attack methods insert API calls into malware examples in the manipulation process.

PGD Attack. PGD [14] was introduced as a method for solving Eq. (4) in the context of image classification. This method is an iterative procedure to find perturbations via

$$\delta_{\mathbf{x}}^{i+1} = \mathrm{Proj}\left(\delta_{\mathbf{x}}^{i} + \alpha \cdot \nabla_{\delta_{\mathbf{x}}} L(\mathbf{F}(\mathbf{x} + \delta_{\mathbf{x}}^{i}), y)\right), \tag{6}$$

where $\alpha > 0$ is the step size, $\nabla_{\delta_{\mathbf{x}}}$ indicates the derivative of the loss function $L(\mathbf{F}(\mathbf{x} + \delta_{\mathbf{x}}^{i}), y)$ with respect to $\delta_{\mathbf{x}}$, and Proj projects perturbations into a feasible domain (e.g., ℓ_∞ norm ball). In practice, the gradients may be very small, causing a performance concern. This motivated researchers to "normalize" the gradients in a direction of interest, such as the ℓ_∞ norm, ℓ_2 norm, or ℓ_1 norm [21]. For each norm, the steepest direction is leveraged to calculate perturbations [34]. For example, the ℓ_∞ norm case is a fine-grained FGSM attack that performs FGSM in multiple steps with step size α $(\alpha < \varepsilon)$ in the context of FGSM. For a sparse attack (in the ℓ_1 norm), the steepest direction corresponds to the coordinate with the maximum absolute gradient value.

We propose adapting all of the three kinds of attacks, namely PGD constrained by a norm ball of ℓ_1, ℓ_2 or ℓ_∞, into the context of adversarial malware detection, these three variant attacks are respectively dubbed PGD-ℓ_1, PGD-ℓ_2 and PGD-ℓ_∞. In reality, the attacker does not have to project perturbations into the ℓ_p norm ball, which suggests us to use Adam [35] to solve Eq. (4) when waging an attack, leading to a new variant of the attack, dubbed PGD-Adam. We further propose letting $\mathrm{Proj}(\cdot)$ project perturbed examples into the domain $[0, 1]^{dim}$ where dim is the input dimension, and propose mapping perturbed examples into $\mathcal{M}$ via a rounding operator after the last iteration.

EAD. This attack was introduced in the context of image classification [15], attempting to generate an adversarial example by making small manipulations to non-adversarial examples. Given $(\mathbf{x}, y)$, instead of using Eq. (4) to guide the attack, the attack instead uses the following loss function that incorporates the elastic-net regularization, namely

$$\min_{\delta_{\mathbf{x}}} c \cdot g(\mathbf{x} + \delta_{\mathbf{x}}, y) + \beta ||\delta_{\mathbf{x}}||_1 + ||\delta_{\mathbf{x}}||_2^2, \tag{7}$$

where c is penalty factor which is identified during the optimization process,

$$g(\mathbf{x} + \delta_{\mathbf{x}}, y) = \max\{[\mathbf{Z}(\mathbf{x} + \delta_{\mathbf{x}})]_y - \max_{j \neq y} [\mathbf{Z}(\mathbf{x} + \delta_{\mathbf{x}})]_j, -\kappa\},$$

$\mathbf{Z}$ is the layer preceding the *softmax* one, and κ is a hyper-parameter.

We propose adapting the EAD attack to the context of adversarial malware detection, while using the ℓ_1 decision rule [15].

3 Adversarial Malware Defense Framework

Rather than purely presenting the defense framework, we also present the principles that we used to guide our design of the defense framework. This is important because these principles can serve as a stating point towards building a comprehensive set of principles that can guide effective defense in the future.

3.1 Guiding Principles

These principles are geared towards neural network classifiers, which are chosen as our focus because deep learning techniques are increasingly employed in malware defense, but their vulnerability to adversarial evasion attacks has yet to be tackled [36].

3.1.1 Principle 1: Knowing the enemy

This principle says that we should strive to extract useful information about the data as much as we can. This kind of information will offer insights into designing countermeasures. For example, we can ask questions of the following kinds:

- Is the training set imbalanced? If the training set is not balanced, various methods need to be considered for alleviating the imbalance issue. For example, the widely-used oversampling is to expand the examples of the "minority" classes via random and repetitive sampling [37].
- Are there sufficiently many examples? This issue is important especially when there are a large number of features and when neural networks are considered. One widely-used method is data augmentation, which generates new examples by making slight modifications on original examples [38].
- Are there simple indicators of adversarial examples? If there are simple indicators of adversarial examples, we can possibly design tailored classifiers for them.

3.1.2 Principle 2: Bridging the gap between countermeasures against grey-box attacks and countermeasures against white-box attacks

In grey-box attacks, the attacker knows some information about the feature set and therefore can train a surrogate classifier $\hat{f} : \mathcal{X} \to \mathcal{Y}$ from a training set (where the realization of $\hat{f}$ is a neural network $\hat{\mathbf{F}}$), leveraging the transferability from $\hat{f}$ to f to generate adversarial examples. Consider an input $\mathbf{x}$ for which a grey-box attacker generates perturbations using

$$\hat{\delta}_{\mathbf{x}} = \underset{||\delta_{\mathbf{x}}|| \leq \epsilon}{\arg\max}\, L(\hat{\mathbf{F}}(\mathbf{x} + \delta_{\mathbf{x}}), y),$$

the change to the loss of f incurred by $\hat{\delta}_{\mathbf{x}}$ is

$$\begin{aligned} |\Delta L| &= \left| L(\mathbf{F}(\mathbf{x} + \hat{\delta}_{\mathbf{x}}), y) - L(\mathbf{F}(\mathbf{x}), y) \right| \\ &= \left| \int_0^{\hat{\delta}_{\mathbf{x}}} \nabla L(\mathbf{F}(\mathbf{x} + \delta), y) d\delta \right| \\ &= \left| \int_0^1 \nabla L(\mathbf{F}(\mathbf{x} + t\hat{\delta}_{\mathbf{x}}), y)^\top \hat{\delta}_{\mathbf{x}} dt \right| \\ &\leq \epsilon \sup_{||\delta|| \leq \epsilon} \left\| \nabla L(\mathbf{F}(\mathbf{x} + \delta), y) \right\|_* , \end{aligned}$$

where "$||\cdot||_*$" means the dual norm of $||\cdot||$ and ∇ is the operator for computing partial derivatives of the loss function with respect to input of neural network $\mathbf{F}$. The preceding observation indicates that corresponding to the same (and potentially large) perturbation upper bound ϵ, the loss incurred by grey-box attacks is upper bounded by the loss incurred by white-box attacks. This suggests us to focus on the robustness of classifier f against the optimal white-box attack because it accommodates the worst-case scenario. It is worth mentioning that our observation, which applies to an arbitrary perturbation upper bound ϵ, enhances an earlier insight that holds for a small perturbation upper bound ϵ [39].

3.1.3 Principle 3: Using ensemble of classifiers rather than a single one (i.e., not putting all eggs in one basket)

This is suggested by the observation that no single classifier may be effective against all kinds of evasions. Worse yet, we may not know the kinds of evasions the attacker uses to generate adversarial examples. Therefore, we propose using *ensemble* learning to enhance the robustness of neural networks based malware classifiers.

An ensemble can be constructed by many methods (e.g., bagging, boosting, or stacking multiple classifiers). The generalization error of an ensemble decreases significantly with the ensemble size when the base classifiers are effective and mutual independent [40]. For example, *random subspace* [41] is seemingly particularly suitable for formulating malware classifier ensembles because the dimension of malware feature vectors is often very high, which indicates a high vulnerability of malware classifiers to adversarial examples [13], [42].

Since the output of a neural network (with *softmax*) is the probability mass function over the classes in question, the final prediction result is produced according to these probabilities. Formally, an ensemble $f_{en} : \mathcal{X} \to \mathcal{Y}$ contains a set of neural network classifiers $\{f_i\}_{i=1}^{l}$, namely $f_{en} = \{f_i : \mathcal{X} \to \mathcal{Y};\ (1 \leq i \leq l)\}$. Given a test example $\mathbf{x}$, each classifier f_i defines a conditional probability on predicting y. We treat the base classifier equally, as suggested by the study [43], and the voting method is

$$f_{en}(\mathbf{x}) = \frac{1}{l} \sum_{i=1}^{l} f_i(\mathbf{x}). \tag{8}$$

3.1.4 Principle 4: Using input transformation to reduce the degree of perturbation caused by evasion manipulation

In the context of malware classification, input transformation techniques, such as adversarial feature selection [44] and random feature nullification [17], can reduce the degree of perturbation in adversarial examples so as to improve the robustness of classifiers. In typical applications, the defender does not know what kinds of evasion manipulations are used by the attacker to generate adversarial examples, including the number of features that are manipulated by the attacker. Therefore, we propose considering a spectrum of evasion manipulations, from manipulating a few features (measured by, for example, the ℓ_1 norm) to manipulating a large number of features (but the magnitude of these manipulations may be small and therefore measured for example by the ℓ_∞ norm). Moreover, we may give higher weights

to the transformation techniques that can simultaneously reduce the degrees of perturbations in terms of the ℓ_∞ norm, ℓ_0 norm, or ℓ_2 norm. This suggests us to propose using the *binarization* technique [20]: When the feature value of the ith feature, denoted by x_i, is smaller than a threshold Θ_i, we binarize x_i to 0; otherwise, we binarize x_i to 1. This input transformation reduces the perturbation space to two kinds: flipping '0' to '1' and flipping '1' to '0'.

3.1.5 Principle 5: Using adversarial training to "inject" immunity into classifiers

Adversarial training (also known as proactive training [22]) incorporates some adversarial examples into the training set. Various kinds of *heuristic* training strategies have been proposed (see, e.g., [9], [10], [12], [22], [45]. However, these strategies typically deal with some specific evasion methods and therefore are not known to effective against other evasion methods. Inspired by *robust optimization*, Madry et al. [14] proposed solving the saddle point problem so as to improve the robustness of neural networks against a wide range of adversarial examples. This strategy has been adapted to the context of malware detection in [6]. The preceding discussion suggests us to train neural networks that can accommodate the optimal attack in $\mathcal{M}(\mathbf{x}, \mathcal{S}_\mathbf{x})$, namely

$$\min_\theta \mathbb{E}_{(\mathbf{x},y)\in\mathcal{X}\times\mathcal{Y}} \left[L(\mathbf{F}(\mathbf{x}), y) + \max_{\mathbf{x}'\in\mathcal{M}} L(\mathbf{F}(\mathbf{x}'), y) \right], \quad (9)$$

where θ denotes the trainable parameters of neural network $\mathbf{F}$ (as the realization of f_i). Intuitively, the min-max optimization penalizes perturbations in the manipulation set because the loss incurred by the optimal attack is minimized [6], [10], [14].

In the term of small perturbations, the min-max optimization regularizes DNNs. Putting another way, if we assume the perturbations are small and bounded from above in the ℓ_p norm, the inner maximization can be approximated using the first-order Taylor expansion at point $\mathbf{x}$, namely:

$$\max_{||\delta_\mathbf{x}||\leq\epsilon} L(\mathbf{F}(\mathbf{x}+\delta_\mathbf{x}), y) \approx \max_{||\delta_\mathbf{x}||\leq\epsilon} [L(\mathbf{F}(\mathbf{x}), y) + \nabla L^\top \delta_\mathbf{x}].$$

The preceding optimization problem can be solved to obtain the optimal solution $L(\mathbf{F}(\mathbf{x}), y) + \epsilon||\nabla L||_*$ as shown in [46]. We observe that the regularization penalizes the dual-norm gradients of the loss function with respect to the input. Given the empirical finding of Drucker and Le Cun [47] that penalizing input gradients of DNN improves classification accuracy, the min-max optimization can be seen as a regularization technique. This motivates us to propose performing the min-max optimization, which does not require the defender to know the manipulation set.

The key challenge is to search for the optimal attack against DNNs, which use non-linear activation functions that exhibit local minima. Exact solution can be obtained by using, for example, the Mixed Integer Linear Program (MILP), which however is not efficient. Nevertheless, Madry et al. [14] empirically showed that PGD can solve the inner maximum problem effectively. In the context of adversarial malware example, we (i.e., the defender) should project perturbations to accommodate the set $\mathcal{M}$ because an adversarial example must preserve the malicious functionality of the non-adversarial malware example in question, rather than projecting perturbations into a norm-based ϵ ball; this is unlike the setting of image classification (e.g., $\epsilon = 0.3$ on ℓ_∞ norm [14]). Therefore, we propose optimizing via Adam, which is a gradient descent based method, because it converges faster than the standard approach (see Eq. (6)). In order to cope with the problem of local minima, we propose running the inner maximizer several times, each with a random initial point near the training data, and then selecting the point that maximizes the cross-entropy loss.

In the process of solving the inner optimization, it is infeasible to generate perturbations following the gradient descent direction because of the discrete domain $\mathcal{M}$. Therefore, we propose searching for a small neighborhood of $\tilde{\delta}_\mathbf{x}$ to find a well-aligned vector $\delta_\mathbf{x}$ to maximize $\tilde{\delta}_\mathbf{x}^\top \cdot \delta_\mathbf{x}$ under a norm constraint on the distance between $\tilde{\delta}_\mathbf{x}$ and $\delta_\mathbf{x}$. However, this may lead to an exponential computational complexity. We therefore propose treating the feature representation values as continuous on a set (e.g., $[0, 1]$) and apply Adam to the inner maximization with a projection to "clip" a perturbed vector into $[0, 1]^{dim}$. Finally, we leverage a rounding operator to map generated adversarial malware examples to $\mathcal{M}$. It is known that this strategy works well in many cases [5], [6], [44].

3.1.6 Principle 6: Using semantics-preserving representations

Adversarial malware examples must assure that a manipulated example is preserving the malicious functionality of the original malware example. This suggests us to strive to learn neural network models that are sensitive to malware semantics, but not the perturbations because the latter must preserve the malicious functionality of the original malware. Specifically, we propose using *denoising autoencoder* to learn semantics-preserving representations because they can make neural network less sensitive to perturbations. A denoising autoencoder $ae = d \circ e$ unifies two components: an encoder $e : \mathcal{X} \to \mathcal{H}$ that maps an input $M(\mathbf{x})$ to a latent representation $\mathbf{r} \in \mathcal{H}$ and a decoder $d : \mathcal{H} \to \mathcal{X}$ that reconstructs $\mathbf{x}$ from $\mathbf{r}$, where the $\mathcal{H}$ is the latent representation space and M refers to some operations applied to $\mathbf{x}$ (e.g., adding Gaussian noises to $\mathbf{x}$). Vincent et al. [48] showed that the lower bound of the *mutual information* between $\mathbf{x}$ and $\mathbf{r}$ is maximized when the reconstruction error is minimized. In the case of Gaussian noise $\eta \sim \mathcal{N}(0, \sigma^2)$ and reconstruction loss

$$\mathbb{E}_{\eta\sim N(0,\sigma^2)} \left\| ae(\mathbf{x}+\eta) - \mathbf{x} \right\|_2^2, \quad (10)$$

Alain and Bengio [49] showed that the optimal $ae^*(\mathbf{x})$ is

$$ae^*(\mathbf{x}) = \frac{\mathbb{E}_\eta \left[p(\mathbf{x}-\eta)(\mathbf{x}-\eta) \right]}{\mathbb{E}_\eta \left[p(\mathbf{x}-\eta) \right]}, \quad (11)$$

where $p(\cdot)$ is the probability density function. Eq.(11) says that representations of a well-trained denoising autoencoder are insensitive to $\mathbf{x}$ because of the weighted average from the neighbourhood of $\mathbf{x}$, which is reminiscent of the *attention* mechanism [50].

Figure 1 depicts how the learned encoder is leveraged for classification, where the encoder structure is the same as described in [48]. Two examples of noise are:

- *Salt-and-pepper noise* η: A fraction of the elements of original example $\mathbf{x}$ are randomly selected, and

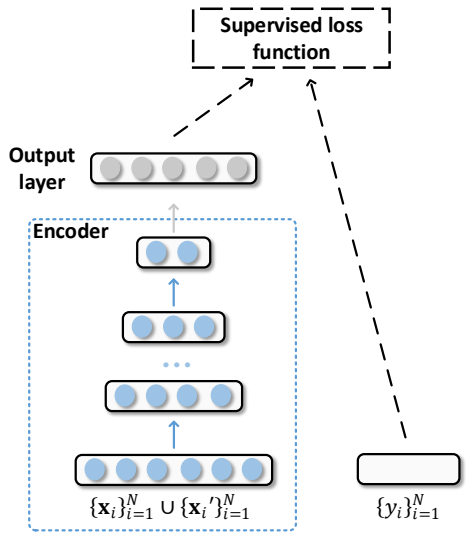


Figure 1: Illustration of neural network for classification. Blue dashed box contains the learned encoder, which and the output layer comprises the learned neural network classifier. The model parameters are tuned to minimize the supervised loss function.

then set their values as their respective minimum or maximum (i.e., effectively flipping '1' to '0' or flipping '0' to '1').

- *Adversarial perturbation* $\delta_{\mathbf{x}}$: A perturbation $\delta_{\mathbf{x}}$ is added to $\mathbf{x}$ such that classifier f or base classifier f_i misclassifies adversarial example $\mathbf{x}' = \mathbf{x} + \delta_{\mathbf{x}}$.

Note that when an input transformation technique mentioned above is used together with a denoising autoencoder, the former should be applied first and the latter is applied to the transformed input.

Given a training example $\mathbf{x}$ over the feature space $\mathcal{X}$, the risk of a denoising autoencoder is

$$\min_{\tilde{\theta},\xi} \mathbb{E}_{\mathbf{x}\in\mathcal{X}} \left[L_{ae}(\mathbf{x}, ae(\mathbf{x}+\eta)) + L_{ae}(\mathbf{x}, ae(\mathbf{x}')) \right], \qquad (12)$$

where $L_{ae} : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ calculates the mean-square error, the learnable parameters $\tilde{\theta}$ and ξ respectively belongs to the encoder and decoder.

3.2 Turning Principles into A Framework

The principles discussed above guide us to propose a framework for adversarial malware classification and detection, which is highlighted in Figure 2 and elaborated below. Specifically, we need to examine whether or not the input has some issues (e.g., imbalanced data) that need to be coped with via an appropriate prepossessing (according to Principle 1). We propose using an ensemble f_{en} of classifiers $\{f_i\}_{i=1}^l$ (according to Principle 3), which are trained from random subspace of the original feature space. Each classifier f_i is hardened by three countermeasures: input transformation via binarization (according to Principle 4); adversarial training models on the optimal attacks using gradient descent technique (green arrows in Figure 2, according to Principle 2 and Principle 5); semantics-preservation is achieved via an encoder and a decoder (according to Principle 6). In order to attain adversarial training and at the same time semantics-preservation, we learn classifier f_i via block coordinate descent to optimize different components of the model.

Algorithm 1: Training classifier f_i

Input: Training set (X, Y), training epoch N_{epoch}, mini-batch size N, the number of repeat times K, and an iteration step T.

1 Cope with issues like imbalanced input;
2 Select a ratio Λ of sub-features to the feature set;
3 Transform input X to $\overline{X}$ via binarization;
4 **for** $epoch = 1$ to N_{epoch} **do**
5 Sample a mini-batch $\{\mathbf{x}_i, y_i\}_{i=1}^N$ from the $(\overline{X}, Y)$;
6 **for** $r = 1$ to K **do**
7 Apply slight salt-and-pepper noises to $\{\mathbf{x}_i\}_{i=1}^N$;
8 Calculate perturbation $\{\delta_{\mathbf{x}_i}^r\}_{i=1}^N$ using Adam with T steps and rounding $\{\mathbf{x}_i + \delta_{\mathbf{x}_i}^r\}_{i=1}^N$ into $\mathcal{M}$, respectively;
9 **end**
10 Select $\mathbf{x}_i'$ from $\left\{\mathrm{Proj}_{\mathcal{M}}(\mathbf{x}_i + \delta_{\mathbf{x}_i}^r)\right\}_{r=1}^K$ for $\mathbf{x}_i$ $(i = 1, \cdots, N)$ to maximize the cross-entropy loss;
11 Calculate the reconstruction loss via Eq.(12);
12 Backpropagate the loss and update the denoising autoencoder parameters;
13 Calculate the adversarial training loss via Eq.(9);
14 Backpropagate the loss and update classifier parameters;
15 **end**

Putting the pieces together, we obtain Algorithm 1 for training individual classifiers. The training procedure consists of the following steps. (i) Given a training set (X, Y), we randomly select a ratio Λ of sub-features to the feature set, and then transform X into $\overline{X}$ via the binarization technique discussed above. (ii) We sample a mini-batch $\{\mathbf{x}_i, y_i\}_{i=1}^N$ from $(\overline{X}, Y)$, and calculate the perturbations $\delta_{\mathbf{x}_i}$ and adversarial examples $\mathbf{x}_i'$ for $\mathbf{x}_i \in \{\mathbf{x}_i\}_{i=1}^N$ according to Lines 5-10 in Algorithm 1. (iii) We pass the $\{\mathbf{x}_i'\}_{i=1}^N$ through the denoising autoencoder to compute the reconstruction loss with respect to the target $\{\mathbf{x}_i\}_{i=1}^N$ via Eq.(12), and update the parameters of the denoising autoencoder. (iv) We pass both the $\{\mathbf{x}_i + \delta_{\mathbf{x}_i}\}_{i=1}^N$ and $\{\mathbf{x}_i\}_{i=1}^N$ through the neural networks to compute the classification error with respect to the ground-truth label $\{y_i\}_{i=1}^N$ via Eq.(9), and update the parameters of the classifier via backpropagation. Note that Steps (ii)-(iv) are performed in a loop. The output of the training algorithm is a neural network classifier.

4 Validating the Framework via the Drebin Dataset

We validate the effectiveness of the framework using the Drebin dataset [51] of Android malware, while considering 11 grey-box attacks and 10 white-box attacks.

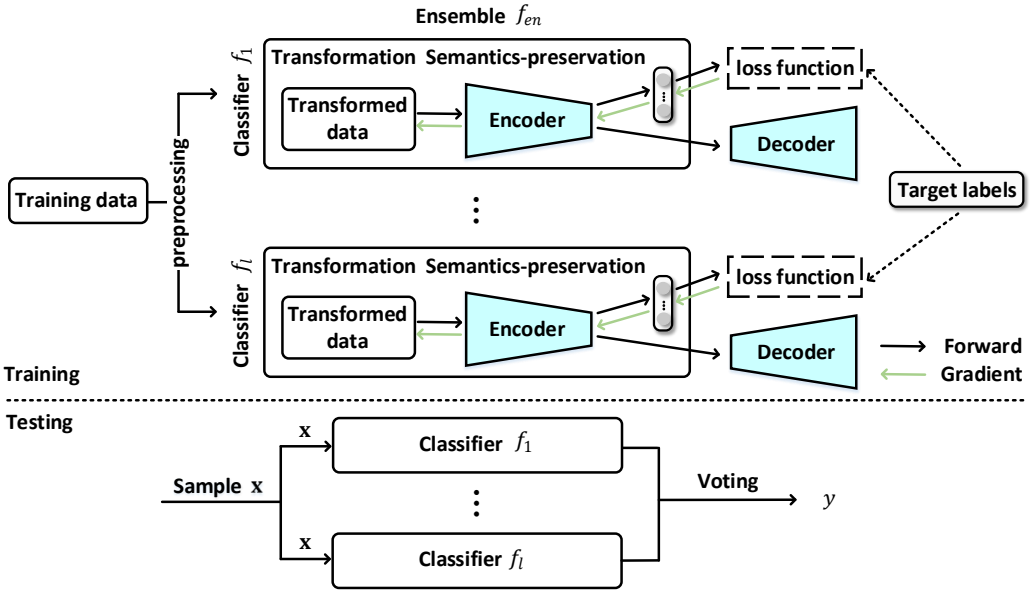


Figure 2: Overview of the proposed malware classification framework. In the training phase, an ensemble of l neural network classifiers are trained, with each classifier hardened by three countermeasures (i.e., input transformation, semantics-preserving, and adversarial training on the transformed data). In the test phase, the label of an example is determined according to the result of voting by the l classifiers.

4.1 Data Pre-Processing

Dataset. The Drebin dataset [51] contains 5,615 malicious Android packages (APKs). The dataset also provides feature values of 123,453 benign examples, together with their SHA256 values but not the examples themselves. We downloaded these benign applications corresponding to the given SHA256 values from the APK markets (e.g., Google Play, AppChina, etc.), and collected 54,829 APKs in total. We sent all of these examples (i.e., malicious and benign alike) to the VirusTotal service [52]. Surprisingly, 12,496 benign APKs were detected as malicious (rather than benign) by at least one scanners, and most of them were detected as *Adware* or *Trojan*; this suggests that the original Drebin training set has been contaminated by the *poisoning attack*. We thus removed these 12,496 benign examples from the original benign dataset, leaving 42,333 benign APKs. An example is treated as malicious if there are at least two scanners say it is malicious, and is treated as benign if no scanners detect it. The resulting dataset contains 5,615 malicious APKs and 42,333 benign APKs, and 47,948 examples in total. We split the dataset into three disjoint sets for training (60%), validation (20%), and test (20%), respectively.

Feature Extraction. APK is an archive file which mainly contains *AndroidManifest.xml*, *classes.dex*, and others (e.g., *res*, *assets*). The *manifest* file describes an APK's information, such as the name, version, announcement, library files used by the application. The source code is compiled to build the *.dex* file which is understandable by the Dalvik Virtual Machine (DVM).

Following prior adversarial learning studies [12], [13], [51], we use the Drebin features, which consist of 8 subsets of features, including 4 subsets of features extracted from *AndroidManifest.xml* (denoted by S_1, S_2, S_3, S_4, respectively) and 4 subsets of features extracted from the disassembled dexcode (denoted by S_5, S_6, S_7, S_8, respectively). More specifically, (**i**) S_1 contains the features that are related to the access of an APK to the hardware of a smartphone (e.g., camera, touchscreen, or GPS module); (**ii**) S_2 contains the features that are related to the permissions requested by the APK in question; (**iii**) S_3 contains the features that are related to the application components (e.g., *activities*, *service*, *receivers*, etc.); (**iv**) S_4 contains the features that are related to the APK's communications with the operating system; (**v**) S_5 contains the features that are related to the critical system API calls, which cannot run without appropriate permissions or the *root* privilege; (**vi**) S_6 contains the features that correspond to the used permissions; (**vii**) S_7 contains the features that are related to the API calls that can access sensitive data or resources on a smartphone; (**viii**) S_8 contains the features that are related to IP addresses, hostnames and URLs found in the disassembled code.

In order to extract features of the applications, we utilize the Androgurad 3.3.5, which is a reverse engineering toolkit for APK analysis [53]. Note that 141 APKs cannot be analyzed. Moreover, a feature selection is conducted to remove those low-frequency features for the sake of computational efficiency. As a result, we keep 10,000 features with high frequencies. The APK is mapped on the feature space as a binary feature vector, where '1' ('0') corresponding to a feature represent the presence (absence) of the feature in an APK.

4.2 Training Classifiers

Classifiers. In order to validate the defense framework, we use and compare five classifiers for the defender: (**i**) the basic DNN with no effort made to defend against adversarial examples; (**ii**) hardened DNN incorporating adversarial training with known manipulation set (dubbed Adversarial Training), which manifests Principle 2 (i.e., norm-based

grey-box attacks can be bounded by the worst-case white-box attack) and Principle 5 (i.e., min-max adversarial training); (**iii**) hardened DNN incorporating adversarial training for regularization (dubbed Adversarial Regularization) because the defender may not know anything about the manipulation set, which is true in the case of the AICS'2019 challenge; (**iv**) DAE-based classifier incorporating Principle 6 (i.e., semantics-preserving representations); (**v**) classifier hardened by both Adversarial Training and DAE (dubbed AT+DAE); (**v**) ensemble of AT+DAE classifiers in the random subspace (manifesting Principle 3). Note that we do not consider Principle 1 (i.e., knowing your enemy) because we will elaborate attacks in the next subsection. Since the feature representation is binary, we cannot consider Principle 2 (i.e., binarization) either.

Hyper-parameters Setting. We use DNNs with two fully-connected hidden layers (each layer having neurons 160) and the ReLU activation function. All classifiers are optimized by using Adam with epochs 150, mini-batch size 128, and learning rate 0.001. For Adversarial Training, the inner maximization is optimized by using Adam with learning rate 0.02 and iteration steps $T = 100$ to search adversarial examples as many as possible. For Adversarial Regularization, we set the learning rate as 0.01 for Adam and conduct preliminary experiments to tune the maximum iteration steps T. Finally, the T is set as 60 in Adversarial Regularization. We use an ensemble of 5 base classifiers. Our preliminary experiments suggest us to learn base classifiers from the entire training set and the entire feature set. Unless with special mentioning, all classifiers that require to solve the inner maximization are trained without random starting points so as to ease the analysis (i.e., $K = 0$).

4.3 Attack Experiments and Classification Results

We consider a threat model that is specified by whether the attacker wages grey-box or white-box attacks, and constraints on the attacker's manipulation operation.

Grey-box vs. White-box Attacks. We consider two attack scenarios. (**i**) *Grey-box attacks*: In this setting, we simulate the attack scenario of the AICS'2019 Challenge organizers. That is, the attacker knows the dataset, feature set, but not the defender's learning algorithm. The attacker generates adversarial examples from a surrogate classifier. We consider a surrogate model of two fully-connected hidden layers (200 neurons each layer) and train the model on the training set using the Adam optimizer with batch size 128, epochs 30, learning rate 0.001. (**ii**) *White-box attacks*: In this setting, the attacker knows everything about the malware detector. Therefore, the adversarial examples are directly generated from the corresponding malware detector.

Manipulations Constraints. Given an APK, we consider both *incremental* and *decremental* manipulations. The incremental manipulation allows the attacker to insert some objects (e.g., *activity*) into an APK example to avoid detection. The decremental manipulation allows the attacker to hide some objects (e.g., *activity*) to evade detection. In any case, the adversarial example should preserve the malicious functionality of the malware from which the adversarial example is derived.

When the attacker uses incremental manipulations, the attacker can insert some manifest features (e.g., request extra permissions and hardware, state additional *services*, *Intent-filter*, etc.). However, some elements are hard to insert, such as *content-provider*, because the absence of URI will corrupt an APK example. With respect to the *.dex* file, a dead function or class (which is never called) containing specified system APIs can be injected without destroying the APK example. The similar means can be performed for the *string* injection (e.g., IP address), as well.

When the attacker uses decremental manipulations, the APK's information in the *xml* files can be changed (e.g., package name). However, it is impossible to remove *activity* entirely because an *activity* may represent a class implemented in the *.dex* code. Nevertheless, we can rename an *activity* and change its relevant information (e.g., *activity label*), while noting that the related components in the *.dex* should be modified accordingly. The other components (e.g., *service*, *provider* and *receiver*) also can be modified in the similar fashion, and the resource files (e.g., images, icons) can be manipulated as well. In term of *dexcode*, the method names and class names that are defined by developers could be modified, too. Note that the corresponding statement, instantiation, reference, and announcements should be changed accordingly. Moreover, user-specified *strings* can be obfuscated using encryption and the cipher-text will be decrypted at running time. Further, the attacker can hide *public* and *static* system APIs using Java reflection and encryption together. This is shown by the example in List 1. All of the modifications mentioned above only obfuscate an APK without changing its functionalities.

```
public void hideAPI() throws Exception{
// hide 'println'
String e_str = "ExMLXEZUDw";
// get 'println'
String p_str = decryptStr(e_str);
Class c = java.lang.System.class;
Field f = c.getField("out");
Method m = f.getType().
getMethod(p_str,String.class);
m.invoke(f.get(null), "hello world!");
return void
}
```

Listing 1: Java code to hide the API method "println".

One challenge is that the attacker needs to perform fine-grained manipulations on compiled files automatically at scale, while preserving the functionalities of malware samples. This important because a small change in a malware example can render the file unexectuable. Since Android APIs have upgraded multiple times during the past 5 years, the attacker has to inject compatible APIs into an APK example when manipulating a malware example. The preservation of functionalities may be estimated by using a dynamic malware analysis tool, (e.g., VirusTotal or Sandbox).

Mapping Manipulations to Feature Space. The aforementioned manipulations modify static Android features, such as API calls, OpCode, call graphs, and so forth. We observe that two kinds of perturbations can be applied to the Drebin feature space as follows:

Table 1: Overview of manipulations on feature space, where ✓(✗) indicates that the feature addition or removal operation can (cannot) be performed on features in the corresponding subset.

Feature sets		Addition	Removal
manifest	S_1 Hardware	✓	✗
	S_2 Requested permissions	✓	✗
	S_3 Application components	✓	✓
	S_4 Intents	✓	✗
dexcode	S_5 Restricted API calls	✓	✓
	S_6 Used permission	✗	✗
	S_7 Suspicious API calls	✓	✓
	S_8 Network addresses	✓	✓

- **Feature addition.** The attacker can increase the feature values (e.g., flipping '0' to '1') of an appropriate objects, such as components (e.g., *activity*), system APIs, and IP address.
- **Feature removal.** The attacker can flip '1' to '0' by removing or hiding objects (e.g., *activity*, APIs.)

Table 1 summarize our manipulations on the Drebin feature space. We observe that neither addition nor removal operation can be applied to S_6 because this subset of features depends on S_5 and S_7, meaning that modifications on S_5 or S_7 will cause changes to features in S_6.

Evasion Attacks Setting. We randomly select 800 malware examples from the test set for waging evasion attacks. For this purpose, we use the attack algorithms described in Section 2.2. In the settings of Random, Grosse, BGA, BCA, and ℓ_1-PGD attacks, we iterate these algorithms until reaching a predefined maximum number of steps, while noting that Grosse, BGA, and BGA attacks leverage feature addition only. For waging the Mimicry attack, we use 10 benign examples to guide the perturbation of a single malware example, leading to 10 adversarial examples; then, we select from these 10 the adversarial example that causes the miss-classification with the smallest perturbation. For other attacks, we round the perturbations to accommodate the discrete domain $\mathcal{M}$. In order to prevent the perturbations from being suppressed by rounding, we run attack algorithms with sufficient iterations and a proper step size as follows. We set $\epsilon = 1.0$ for the FGSM attack. In ℓ_∞ norm and Adam based PGD attacks, the step size is $\alpha = 0.01$ with iterative times 100. The ℓ_2 norm PGD attack is performed for 100 iteration steps with step size 1.0. For the EAD attack under the grey-box attack, we set $\beta = 0.1$ and $\kappa = 64.0$ using the ℓ_1 decision rule [15] for improving the transferability of adversarial malware examples. In the white-box attack setting, we tune the κ for each model. All attacks perform these manipulations to preserve the non-adversarial malware examples' malicious functionality.

4.4 Experimental Results

The Case of No Adversarial Attacks. Table 2 summarizes the classification results on the test set, which are measured with the standard metrics of False Negative Rate (FNR), False Positive Rate (FPR), and classification Accuracy (i.e., the percentage of the test examples that are classified correctly) [54]. We observe that when compared with the Basic DNN, Adversarial Training achieves a lower FNR (0.438% lower) but a higher FPR (1.457% higher). A similar result is exhibited by DAE, AT+DAE and Random Subspace. This can be explained as follows: by injecting adversarial malware examples into the training set, the learning process makes the model search for malware examples in a bigger space, explaining the drop in FNR and increase in FPR and therefore a slightly drop ($\leq 1.74\%$) in the classification accuracy. Adversarial Regularization achieves a comparable classification accuracy as Basic DNN, but the highest FNR among the classifiers we considered. This may be caused by the fact that when DNN is regularized in an adversarial fashion, no information about the manipulation set is known. In summary, we draw:

Insight 1. *In the absence of adversarial attacks, Adversarial Training and DAE can detect more malware examples than the Basic DNN (because of their smaller FNR), at the price of a small side-effect in the FPR and therefore the classification accuracy.*

Table 2: Effectiveness of the defense framework when there are no adversarial attacks.

Defense	FNR (%)	FPR (%)	Accuracy (%)
Basic DNN	3.684	0.320	99.28
Adversarial Training	3.246	1.777	98.05
Adversarial Regularization	4.737	0.190	99.27
DAE	3.246	0.450	99.22
AT+DAE	3.246	1.694	98.12
Random Subspace	2.456	2.464	97.54

The Case of Grey-box Attacks. Table 3 reports the classification results of the defense framework against grey-box attacks. We make the following observations. First, Basic DNN cannot defend against evasion attacks and is completely ruined by attacks that include Mimicry, FGSM, Grosse, BGA, BCA, PGD-ℓ_1, PGD-ℓ_∞ and EAD. Second, Adversarial Training significantly enhances the robustness of DNN, achieving a 85.63% accuracy against the Mimicry attack and a 100% accuracy against other 6 attacks (i.e., BGA, BCA and 4 variants of PGD). Third, Adversarial Regularization, without seeing any adversarial examples, can defend against FGSM, PGD-ℓ_∞, PGD-ℓ_2 and PGD-Adam attacks, but are not effective against attacks such as EAD, Grosse, BCA, and PGD-ℓ_1. A similar phenomenon is observed for DAE. Nevertheless, when using Adversarial Training and DAE together, namely AT+DAE, the defense achieves the highest robustness against evasion attacks than using Adversarial Training and DAE individually, except for the Mimicry attack and the FGSM attack. Fourth, the Random Subspace defense consists of five AT+DAE classifiers and achieves the highest among the considered defenses against the attacks investigated. In summary, we draw:

Insight 2. *Under grey-box attacks, Adversarial Training is an effective defense against evasion attacks; DAE offers some defense capability that may not be offered by Adversarial Training; using an ensemble of five AT+DAE classifiers is more effective than using a single AT+DAE classifier against evasion attacks; without knowing the attacker's manipulation set, Adversarial Regularization enhances the robustness of Basic DNN but cannot defend attacks such as Grosse.*

The Case of White-box Attacks. Table 4 reports the classification results against white-box attacks. We make the fol-

Table 3: Effectiveness of the defense framework against grey-box adversarial malware evasion attacks.

Attack	Accuracy (%)					
	Basic DNN	Adversarial Training (AT)	Adversarial Regularization	DAE	AT+DAE	Random Subspace
No Attack	96.63	97.00	95.63	96.88	96.50	**97.75**
Random Attack	100.0	100.0	100.0	100.0	100.0	**100.0**
Mimicry 10x	35.25	85.63	34.88	52.63	85.13	**89.88**
FGSM [10]	4.00	97.50	95.88	96.88	96.75	**98.00**
Grosse [12]	1.13	97.00	11.75	65.13	97.63	**99.38**
BGA [6]	0.25	100.0	71.13	100.0	100.0	**100.0**
BCA [6]	0.25	100.0	49.50	58.00	100.0	**100.0**
PGD-ℓ_1	0.25	100.0	43.88	53.88	100.0	**100.0**
PGD-ℓ_2	58.63	100.0	99.75	100.0	100.0	**100.0**
PGD-ℓ_∞	0.25	100.0	100.0	100.0	100.0	**100.0**
PGD-Adam	52.50	100.0	100.0	100.0	100.0	**100.0**
EAD [15]	5.50	92.88	20.25	72.00	95.13	**98.38**

Table 4: Effectiveness of the defense framework against white-box adversarial malware evasion attacks.

Attack	Accuracy (%)					
	Basic DNN	Adversarial Training (AT)	Adversarial Regularization	DAE	AT+DAE	Random Subspace
Mimicry 10x	11.63	68.25	14.88	40.88	69.13	**79.75**
FGSM [10]	0.00	97.00	95.00	96.88	96.50	**97.75**
Grosse [12]	0.00	60.75	16.63	35.50	81.13	**91.75**
BGA [6]	0.00	97.00	91.50	74.00	96.50	**97.50**
BCA [6]	0.00	61.13	16.63	35.38	81.50	**91.75**
PGD-ℓ_1	0.00	69.50	21.88	51.00	81.25	**88.50**
PGD-ℓ_2	3.00	**93.63**	82.13	89.75	91.13	91.63
PGD-ℓ_∞	0.00	**90.38**	89.75	35.38	85.50	73.63
PGD-Adam	1.13	**95.13**	89.63	88.25	92.88	90.00
EAD [15]	0.00	**88.38**	24.50	2.50	82.13	60.50

lowing observations. First, all attacks can almost completely evade Basic DNN, but the Mimicry attack is, relatively speaking, less effective because this attack leverages less information about the classifiers than what the other attacks do.

Second, Adversarial Training is effective against the FGSM attack, the BGA attack and the PGD-Adam attack, but not effective against the Grosse attack, the BCA attack, and the PGD-ℓ_1 attack because these attacks manipulate a few features when generating adversarial examples and these manipulations are unlikely perceived by Adversarial Training (owing to the fact that Adversarial Training penalizes adversarial spaces searched by Adam). Although the EAD attack also perturbs malware example using small perturbations, some of the perturbations are suppressed by the projection to accommodate the discrete domain, explaining why Adversarial Training can defend against EAD at an accuracy of 88.38%.

Third, as expected, Adversarial Regularization is less effective than Adversarial Training. Adversarial Regularization achieves a 91.50% accuracy against the white-box BGA attack, in contrast to the 71.13% accuracy against the grey-box BGA attack. This is counter-intuitive but can be attributed to the fact that Adversarial Regularization can render some gradient-based methods, such as BGA, useless, which is a phenomenon known as *gradient-masking* [55], [56], [57].

Fourth, AT+DAE achieves considerable robustness against those attacks, with at least a 81.13% accuracy except for the Mimicry attack, which can defeat the AT+DAE defense because Mimicry can make adversarial malware examples similar to benign ones [13].

Fifty, the ensemble of Random Subspace defense achieves the highest accuracy against the Mimicry attack, the Grosse attack and the BCA attack than the other defenses, with about 10% higher accuracy when compared with the AT+DAE defense. However, the ensemble of Random Subspace achieves lower accuracy than AT+DAE against the PGD-ℓ_2 attack, the PGD-ℓ_∞ attack, the PGD-Adam attack, and the EAD attack. This may be caused by the fact that the classifier AT+DAE cannot effectively mitigate these attacks. In summary, we draw:

Insight 3. *Adversarial Training cannot effectively defend against white-box attacks that were not considered in the training phase; the ensemble of Random Subspace defense is effective against several white-box attacks but not others. That is, no defenses can defeat all kinds of white-box attacks.*

5 Applying the Framework to the AICS'2019 Challenge without Knowing Anything about the Attacks

The challenge is in the context of adversarial malware classification (more precisely, *multiclass classification*), namely constructing evasion-resistant, machine learning based malware classifiers. The dataset, including both the training set and the test set, consists of feature vectors extracted from Windows malware examples, each of which belongs to one of the following five classes: *Virus*, *Worm*, *Trojan*, *Packed malware*, and *AdWare*. For each example, the features are collected by the challenge organizer via dynamic analysis, including the Windows API calls and further processed unigram, bigram, and trigram API calls. The feature names (e.g., API calls) and the class labels are "obfuscated" by the challenge organizer as integers, while noting the obfuscation preserves the mapping between the features and the integers representation of them. For example, three API calls are

represented by three unique integers, say 101, 102, and 103; then, a trigram API call "101;102;103" means a sequence of API calls 101, 102, and 103. In total there are 106,428 features.

The test set consists of adversarial examples and non-adversarial examples (i.e., unperturbed malware examples). Adversarial examples are generated by a variety of perturbation methods, which are not known to the participating teams. However, the ground-truth labels of the test examples are not given to the participating teams. This means that the participating teams cannot calculate the accuracy of their detectors by themselves. Instead, they need to submit their classification results (i.e., labels on the examples in the test set) to the challenge organizer, who will calculate the classification score of each participating team. The Challenge organizer decided to use the Macro F1 score as the classification accuracy metric. The Macro F1 score is the unweighted mean of the F1 score [58] for each class of objects in question (i.e., type of malware in this case). The final score is the Harmonic mean upon the two Macro F1 scores, namely the one for the adversarial examples in the test data and the other for the non-adversarial examples in the test data. Given these two numbers, say a_1 and a_2, their their harmonic mean $\frac{2a_1a_2}{a_1+a_2}$.

5.1 Basic Analysis of the AICS'2019 Challenge

As discussed in Principle 1 of the framework, our basic analysis aims to identify some basic characteristics that should be taken into consideration when adapting Algorithm 1 to this specific case study.

Is the training set imbalanced? The training set consists of 12,536 instances, and the test set consists of 3,133 instances. The training set contains 8,678 instances in class '0', 1,883 instances in class '1', 771 instances in class '2', 692 instances in class '3', and 512 instances in class '4'. We can calculate the maximum ratio between the number of instances in different classes is 16.95, indicating that the training set is highly imbalanced.

In order to cope with the imbalance in the training set, we use the *Oversampling* method to replicate randomly selected feature vectors from a class with a small number of feature vectors. The replication process ends until the number of feature vectors is comparable to that of the largest class (i.e., the class with the largest number of feature vectors), where "comparable" is measured by a predefined ratio. In order to see the effect of this ratio, we use a 5-fold cross validation on the training set to investigate the impact of this ratio. The classifier consists of neural networks with two fully-connected layers (each layer having 160 neurons with the ReLU activation function), which are optimized via Adam with epochs 50, mini-batch size 128, learning rate 0.001. The model is selected when achieving the best Macro F1 score on the validation set.

Table 5 shows that the Macro F1 score decreases as the oversampling ratio of minority classes increases. In order to make each mini-batch of training data contain examples from all classes, which would be critical in muticlass classification, our experience suggests us to select the 30% ratio.

Are there sufficiently many examples? Machine learning, especially deep learning, models need to be trained with a "large" number of examples, where "large" is relative to the number of features. In the challenge dataset, the training set contains 12,536 examples while noting that there are 106,428 features, which may lead to overfitting. To cope with this, we note that adversarial training is a data augmentation method [10], which can be leveraged to regularize the resulting classifiers.

Table 5: Accuracy (%) and Macro F1 score (%) are reported with a 95% confidence interval with respect to the ratio parameter (%), where '—' means learning a classifier using the original training set.

Ratio (%)	Accuracy (%)	Macro F1 (%)
—	93.20±1.04	85.52±1.12
30	92.86±0.75	85.47±1.04
40	92.38±1.00	84.87±1.07
50	92.21±0.60	84.87±1.00
60	92.48±1.12	84.62±1.01

Are there simple indicators of adversarial examples? In the first test set published by the challenge organizer, we see negative values for some features. These negative values would indicate that they are adversarial examples. In the revised test set provided by the challenge organizer, there are no negative feature values, meaning that there are no simple ways to tell whether an example is adversarial or not. In spite of this, we can speculate on the count of perturbed features by comparing the number of nonzero entries corresponding to feature vectors in the training set and feature vectors in the test set.

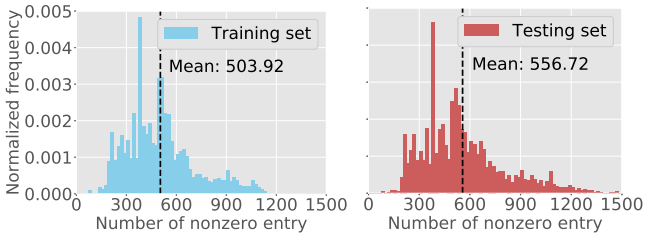


Figure 3: Histogram of the normalized frequency of the number of nonzero entries of feature vectors in the training set vs. test set. The dashed line represents the mean value.

Figure 3 shows the normalized frequency of the number of nonzero entries of feature vectors in the training set vs. test set. We observe that their normalized frequencies are similar except that some test examples have more nonzero entries (> 1200). Their mean values are much smaller than the input dimension $(106,428)$, suggesting that the average number of perturbed features may be small.

5.2 Classification Results: Challenge Winner

We train 10 neural network classifiers to formulate an ensemble, including 4 classifiers using the input transformation, adversarial training, and semantics-preservation techniques discussed in the framework, and the other 6 classifiers using the input transformation and adversarial training techniques because examples are perturbed without preserving their malicious functionality in the training. Since we do not have access to the malware examples, we cannot tell whether a feature perturbation preserves the

malware functionality or not. For adversarial training, we have to perform its variant Adversarial Regularization. The inner maximization performed by using gradient descent with respect to the transformed input iterates $T = 55$ times via the Adam optimizer [35] with learning rate 0.01. We leverage the random start points and $K = 5$. The ratio for random subspace method is set as $\Lambda = 0.5$. Each base classifier has two fully-connected hidden layers (each layer having neurons 160), uses the ELU activation function, and is optimized by Adam with mini-batch size 128 and learning rate 0.001. The ensemble achieves a Macro F1 score of 88.30% upon non-attack dataset, a 63.0% Macro F1 score under attacks, and a Harmonic mean on both Macro F1 scores of 73.60%. This is the highest Harmonic Mean score among the participating teams. Although this score is not ideal, this may be inherent to the fact that we as the defender do not know any information about the attack. This leads to:

Insight 4. *The information "barrier" that the defender does not know the attacker's manipulation set is a fundamental one because the attacker may use adversarial malware examples that are far away from what the defender would use to train its defense model.*

This insight suggests that we may treat 70%-80% as a reasonably high classification accuracy when nothing about the attack is known, rather than a 90%-100% accuracy that may be achieved in the absence of adversarial malware evasion attacks.

6 Applying the Framework to the AICS'2019 Challenge after Knowing the Ground Truth of the Test Set

After the Challenge organizer announced that we won the Challenge, the ground-truth labels of the test set are released so that we can conduct further study. We stress that we still do not know the attacks that were used by the Challenge organizer to generate the adversarial examples.

6.1 Training Classifiers

Classifier. We consider and compare five classifiers: (**i**) the Basic DNN without incorporating any defense; (**ii**) hardened DNN incorporating the binarization technique [20] (dubbed Binarization); (**iii**) hardened DNN incorporating adversarial regularization (dubbed Adversarial Regularization); (**iv**) hardened DNN incorporating both Binarization and Adversarial Regularization (denoted as Binarization+AR); (**v**) an ensemble of Binarization+AR classifiers.

Hyper-parameter Settings. All of the DNNs we use have two fully-connected hidden layers (each layer having 160 neurons), use the ReLU activation function, and are optimized by Adam with epochs 30, mini-batch size 128, and learning rate 0.001. For Adversarial Regularization, we preform the inner maximization via Adam (with learning rate 0.01). Our preliminary experiments suggest us to set iterations $T = 60$. The starting point is chosen from $K = 5$ initialized points with salt-and-pepper noises, which have a noise ratio η^r chose uniformly at random from 0 to $\eta^r_{max} = 10\%$. This means at most 10% of the features can be changed by salt-and-pepper noises. For ensembles, we train 5 Binarization+AR classifiers, each of which is learned from a 80% data randomly selected from the training set, with a $\Lambda = 0.5$ fraction of features. We augment the training set for the last three classifiers as described in Section 5.1.

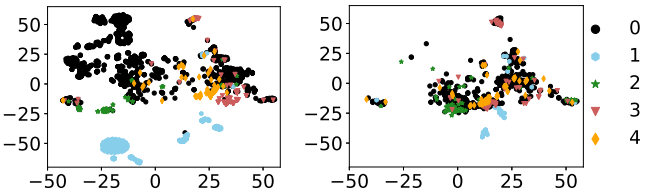
6.2 Classification Results

Table 6 reports the results with and without adversarial attacks. We make the following observations. First, Adversarial Regularization significantly improves the Macro F1 score against the attacks when compared with the Basic DNN (a 23.93% higher Macro F1 score). The Macro F1 score of Adversarial Regularization in the absence of adversarial attacks drops slightly when compared with the Basic DNN ($\approx$ 1%). Second, by comparing Binarization (row 2) and the Basic DNN, Binarization can improve the robustness of DNN against adversarial attacks a little bit (a 0.47% increase in the Macro F1 score). Third, the Random Subspace defense achieves a higher classification accuracy than Binarization+AR, in the presence or absence of adversarial attacks.

Hyper-parameters Sensitivity. In Adversarial Regularization, η^r_{max} is crucial and is set manually. Intuitively, a greater η^r_{max} lets the defense perceive a larger space, but inhibiting the convergence of training. In addition, we want to know whether the oversampling is useful or not for Adversarial Regularization. We thus conduct a group of experiments to justify these settings. Table 7 shows the experimental results. We observe that the Macro F1 score in the presence of adversarial evasion attacks increase with the increase of η^r_{max} from 0% to 10%. Meanwhile, Accuracy and Macro F1 score do not decrease in the absence of adversarial evasion attacks, and actually slightly increase at $\eta^r_{max} = 1\%$. Furthermore, when the oversampling technique is leveraged at $\eta^r_{max} = 10\%$ (the last row), both Accuracy and Macro F1 score in the absence of adversarial evasion attacks decrease slightly ($< 1\%$). Nevertheless, the Macro F1 score in the presence of adversarial evasion attacks increases from 56.22% to 58.93%. This leads us to draw:

Insight 5. *Oversampling is not necessary when there are no adversarial evasion attacks, but improves the effectiveness of Adversarial Regularization against adversarial evasion attacks.*

6.3 Retrospective Analysis of the AICS'2019 Challenge



(a) No attacks (b) Attacks

Figure 4: Visualization of learned representations of Binarization+AR with no attacks (a) and attacks (b), respectively.

We now analyze the impact of imbalanced training set. In the Challenge, the majority and minority classes are treated equally important because of the use of the Macro F1 score as the classification accuracy metric. Owing to the

Table 6: Classifiers Accuracy (%) and Macro F1 score (%) with no attacks vs. using grey-box adversarial evasion attacks respectively, and the Harmonic mean (%) of the two Macro F1 scores.

Classifiers	No attacks (%)		Attacks (%)		Harmonic mean (%)
	Accuracy	Macro F1	Accuracy	Macro F1	
Basic DNN	**96.24**	**88.91**	63.46	35.00	50.23
Binarization	95.80	87.99	63.79	35.47	50.56
Adversarial Regularization (AR)	95.66	87.98	72.02	58.93	70.58
Binarization+AR	95.62	87.87	75.22	59.87	71.22
Random subspace	95.93	88.58	**76.02**	**62.95**	**73.60**

Table 7: Accuracy (%) and Macro F1 score (%) of Adversarial Regularization in the absence of adversarial evasion attacks vs. the presence of adversarial evasion attacks, with respect to the maximum salt-and-pepper noise ratio η^r_{max}, where $*$ means that a classifier is learned using oversampling.

Noise Ratio (%)	No attacks (%)		Attacks (%)	
	Accuracy	Macro F1	Accuracy	Macro F1
$\eta^r_{max} = 0$	96.11	88.43	69.68	49.87
$\eta^r_{max} = 0.1$	95.93	88.10	74.00	50.52
$\eta^r_{max} = 1$	**96.24**	**89.14**	73.11	55.98
$\eta^r_{max} = 10$	96.19	88.46	**77.11**	56.22
$\eta^r_{max} = 20$	96.06	88.14	75.11	51.23
$\eta^r_{max} = 10^*$	95.66	87.98	72.02	**58.93**

use of the cross-entropy loss function, DNN-based classifiers treat each example independently and equally important. Therefore, there is a gap between the training and the test (because the training data is imbalanced). This suggests that Adversarial Regularization may suffer from this issue because more adversarial perturbations can be generated for the majority classes than from the minority classes. In order to explore this problem, we visualize the learned representations (output of the last hidden layer) of Binarization+AR. The 160 dimensions of the representation are reduced to 2 dimensions by using t-SNE [59].

Figure 4 presents the visualization of the result. We observe that the majority class "0" overwhelms the representation space. More specifically, Figure 4a shows that the classifier can separate most of data clearly when there are no adversarial evasion attacks; Figure 4b shows that the representation of class "0" tends to mix with the representation of the other classes when there are adversarial evasion attacks. This indicates that the classifier would make mistakes when dealing with the intersection regions. This leads to:

Insight 6. *Data imbalance makes it harder to defend against adversarial evasion attacks.*

On the other hand, Adversarial Regularization overfits the perturbations searched by the inner maximizer unexpectedly. Figure 5 demonstrates this phenomenon. we observe that, instead of dropping consistently, the cross-entropy loss induced by the adversarial evasion attacks increases significantly after about 10 epochs. Meanwhile, the cross-entropy loss on the test set with no adversarial evasion attacks changes slightly, until the number of epochs approaches 100. This means that the DNN will memorize the perturbations. Therefore, new defense techniques are needed in order to achieve a much higher accuracy against the Challenge instances. This suggests:

Insight 7. *Without knowing the manipulation set, unsupervised learning may play an important role because unsupervised defenses are devised without using label information about the perturbed examples.*

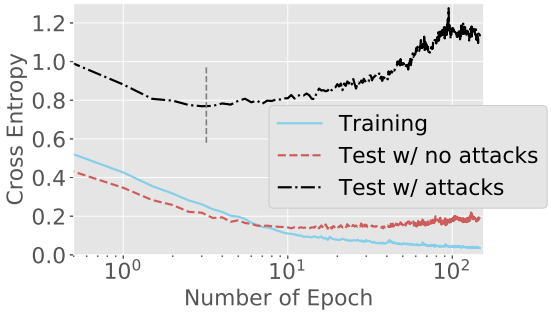


Figure 5: Cross entropy loss of the classifier hardened by Adversarial Regularization over the training set, the test set with no adversarial evasion attacks, and the test set with adversarial evasion attacks, with respect to the number of epochs.

7 Conclusion

We have systematized six principles for enhancing the robustness of neural network classifiers against adversarial evasion attacks in the setting of malware classification. These principles guided us to design a framework, which leads to a concrete training algorithm. We validated our framework using a real-world dataset and applied it to the AICS'2019 Challenge. We drew a number of insights that are useful for real-world defenders. We hope this paper will inspire more research into this important problem. Future research problems are abundant, such as: extending and refining the principles against evasion attacks, seeking principles to enhance robustness of adversarial malware detection against poisoning attacks, designing more robust techniques against adversarial malware examples.

References

[1] D. Li, Q. Li, Y. Ye, and S. Xu, "Enhancing robustness of deep neural networks against adversarial malware samples: Principles, framework, and aics'2019 challenge," *CoRR*, vol. abs/1812.08108, 2018. [Online]. Available: http://arxiv.org/abs/1812.08108

[2] Symantec. (2018) Symantec @ONLINE. [Online]. Available: https://www.symantec.com/security-center/threat-report

[3] CISCO. (2018) Cisio @ONLINE. [Online]. Available: https://www.cisco.com

[4] Y. Ye, T. Li, D. A. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Comput. Surv.*, vol. 50, no. 3, pp. 41:1–41:40, 2017.

[5] I. C. B. Biggio and D. M. et al., "Evasion attacks against machine learning at test time," in *Machine Learning and Knowledge Discovery in Databases: European Conference*. Springer, 01 2013, pp. 387–402.

[6] A. Al-Dujaili, A. Huang, E. Hemberg, and U.-M. OReilly, "Adversarial deep learning for robust detection of binary encoded malware," in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 76–82.

[7] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Make evasion harder: An intelligent android malware detection system," in *Proceedings of the Twenty-Seventh IJCAI*, 2018, pp. 5279–5283.

[8] L. Chen, Y. Ye, and T. Bourlai, "Adversarial machine learning in malware detection: Arms race between evasion attack and defense," in *EISIC'2017*, 2017, pp. 99–106.

[9] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples (2014)," *arXiv preprint arXiv:1412.6572*.

[11] A. C. Serban and E. Poll, "Adversarial examples-a complete characterisation of the phenomenon," *arXiv preprint arXiv:1810.01185*, 2018.

[12] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 62–79.

[13] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli, "Yes, machine learning can be more secure! a case study on android malware detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 4, pp. 711–724, July 2019.

[14] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[15] P.-Y. Chen, Y. Sharma, H. Zhang, J. Yi, and C.-J. Hsieh, "Ead: elastic-net attacks to deep neural networks via adversarial examples," in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[16] M. Nazir. (2019) Utsa wins global cyber security challenge @ONLINE. [Online]. Available: https://www.eurekalert.org/pub_releases/2019-01/uota-uwg011819.php

[17] Q. Wang, W. Guo, K. Zhang, and et al., "Adversary resistant deep neural networks with an application to malware detection," in *Proceedings of the 23rd KDD*. ACM, 2017, pp. 1145–1153.

[18] D. Li, R. Baral, T. Li, H. Wang, Q. Li, and S. Xu, "Hashtran-dnn: A framework for enhancing robustness of deep neural networks against adversarial malware samples," *arXiv preprint arXiv:1809.06498*, 2018.

[19] L. Chen, S. Hou, Y. Ye, and S. Xu, "Droideye: Fortifying security of learning-based classifier against adversarial android malware attacks," in *FOSINT-SI'2018*, 2018, pp. 253–262.

[20] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *arXiv preprint:1704.01155*, 2017.

[21] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," *arXiv preprint arXiv:1611.01236*, 2016.

[22] L. Xu, Z. Zhan, S. Xu, and K. Ye, "An evasion and counter-evasion study in malicious websites detection," in *CNS, 2014 IEEE Conference on*. IEEE, 2014, pp. 265–273.

[23] ——, "Cross-layer detection of malicious websites," in *Third ACM Conference on Data and Application Security and Privacy (CODASPY'13)*, 2013, pp. 141–152.

[24] H. Dang, Y. Huang, and E.-C. Chang, "Evading classifiers by morphing in the dark," in *CCS*. ACM, 2017, pp. 119–133.

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

[26] P. L. Nedim rndic, "Practical evasion of a learning-based classifier: A case study," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 197–211.

[27] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici, "Generic black-box end-to-end attack against rnns and other calls based malware classifiers," *arXiv preprint*, 2017.

[28] L. Chen, S. Hou, and Y. Ye, "Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks," in *ACSAC*. USA: ACM, 2017, pp. 362–372.

[29] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers: A case study on pdf malware classifiers," in *NDSS*, January 2016.

[30] W. Hu and Y. Tan, "Generating adversarial malware examples for black-box attacks based on gan," 02 2017.

[31] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 38th IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 39–57.

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

[33] O. Suciu, R. Marginean, Y. Kaya, H. D. III, and T. Dumitras, "When does machine learning FAIL? generalized transferability for evasion and poisoning attacks," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1299–1316. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/suciu

[34] Z. Kolter and A. Madry. (2019, May) Adversarial robustness - theory and practice. [Online]. Available: https://adversarial-ml-tutorial.org/

[35] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[36] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware detection by eating a whole exe," *arXiv preprint arXiv:1710.09435*, 2017.

[37] M. Buda, A. Maki, and M. A. Mazurowski, "A systematic study of the class imbalance problem in convolutional neural networks," *Neural Networks*, vol. 106, pp. 249–259, 2018.

[38] J. Lemley, S. Bazrafkan, and P. Corcoran, "Smart augmentation learning an optimal data augmentation strategy." *IEEE Access*, vol. 5, pp. 5858–5869, 2017.

[39] A. Demontis, M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru, and F. Roli, "On the intriguing connections of regularization, input gradients and transferability of evasion and poisoning attacks," *arXiv preprint arXiv:1809.02861*, 2018.

[40] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American statistical association*, vol. 58, no. 301, pp. 13–30, 1963.

[41] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Transactions on PAMI*, vol. 20, no. 8, pp. 832–844, 1998.

[42] C.-J. Simon-Gabriel, Y. Ollivier, B. Schölkopf, L. Bottou, and D. Lopez-Paz, "Adversarial vulnerability of neural networks increases with input dimension," *arXiv preprint arXiv:1802.01421*, 2018.

[43] B. Biggio, G. Fumera, and F. Roli, "Multiple classifier systems for robust classifier design in adversarial environments," *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 27–41, 2010.

[44] F. Zhang, P. P. Chan, B. Biggio, D. S. Yeung, and F. Roli, "Adversarial feature selection against evasion attacks," *IEEE transactions on cybernetics*, vol. 46, no. 3, pp. 766–777, 2016.

[45] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533*, 2016.

[46] C. Lyu, K. Huang, and H.-N. Liang, "A unified gradient regularization family for adversarial examples," in *Proceedings of the 2015 IEEE International Conference on Data Mining (ICDM)*, ser. ICDM '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 301–309. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2015.84

[47] H. Drucker and Y. Le Cun, "Improving generalization performance using double backpropagation," *IEEE Transactions on Neural Networks*, vol. 3, no. 6, pp. 991–997, 1992.

[48] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of machine learning research*, vol. 11, no. Dec, pp. 3371–3408, 2010.

[49] G. Alain and Y. Bengio, "What regularized auto-encoders learn from the data-generating distribution," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3563–3593, 2014.

[50] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1412–1421.

[51] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[52] (2018, May) Virustotal. [Online]. Available: https://www.virustotal.com
[53] A. Desnos. (2019) Androguard @ONLINE. [Online]. Available: https://github.com/androguard/androguard
[54] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, "A survey on systems security metrics," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 1–35, Dec. 2016.
[55] A. Athalye, N. Carlini, and D. A. Wagner, "Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples," *CoRR*, vol. abs/1802.00420, 2018.
[56] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," *arXiv preprint arXiv:1705.07204*, 2017.
[57] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against deep learning systems using adversarial examples," *arXiv preprint*, 2016.
[58] Y. Sasaki *et al.*, "The truth of the f-measure," *Teach Tutor mater*, vol. 1, no. 5, pp. 1–5, 2007.
[59] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.