Adversarial Examples for Malware Detection

Kathrin Grosse[1], Nicolas Papernot[2], Praveen Manoharan[1], Michael Backes[1], and Patrick McDaniel[2]

[1]CISPA, Saarland University
Saarland Informatics Campus

[2]Pennsylvania State University
School of Electrical Engineering and CS

Abstract. Machine learning models are known to lack robustness against inputs crafted by an adversary. Such adversarial examples can, for instance, be derived from regular inputs by introducing minor—yet carefully selected—perturbations.
In this work, we expand on existing adversarial example crafting algorithms to construct a highly-effective attack that uses adversarial examples against malware detection models.To this end, we identify and overcome key challenges that prevent existing algorithms from being applied against malware detection: our approach operates in discrete and often binary input domains, whereas previous work operated only in continuous and differentiable domains. In addition, our technique guarantees the malware functionality of the adversarially manipulated program. In our evaluation, we train a neural network for malware detection on the DREBIN data set and achieve classification performance matching state-of-the-art from the literature. Using the augmented adversarial crafting algorithm we then manage to mislead this classifier for 63% of all malware samples. We also present a detailed evaluation of defensive mechanisms previously introduced in the computer vision contexts, including distillation and adversarial training, which show promising results.

1 Introduction

Starting with the use of naive Bayes classifiers for spam detection [1], machine learning has been increasingly applied to solve core security problems. For instance, anomaly detection creates a model of expected behavior in order to detect network intrusions or other instances of malicious activities [35]. Classification with machine learning is also applied to automate the detection of unwanted software like malware [29], or to automate source code analysis [33].

This includes Deep neural networks (DNNs) in security-critical applications, such as malware detection [6,31]. While the benefits applying DNNs are undisputed, previous work has also shown that, as is the case for many machine learning models, they lack robustness to adversarially crafted inputs known as *adversarial examples*. These inputs are derived from legitimate inputs by adding carefully chosen perturbations that force models to output erroneous predictions [36,9,25]. .

To evaluate the applicability of adversarial examples to a core security problem, we chose the settings of malware detection. In contrast to the task of image

classification, the span of acceptable perturbations is greatly reduced: the model input is now a set of features taking discrete values. Thus, acceptable perturbations must correspond exactly to one of these discrete values. Furthermore, the similarity criteria defined by human perception is replaced by the more challenging requirement that perturbations do not jeopardize the software's malware functionality pursued by the adversary.

In this paper, we show that android malware detection that uses neural networks, with performance comparable to the state-of-the-art, is easy to deceive with adversarial examples. Furthermore, we find that hardening the model to increase its robustness to these attacks is a very difficult task. Our attack approach elaborates on an adversarial example crafting algorithm previously introduced in [25]. Our approach thus generalizes to any malware detection system using a differentiable classification function.

Contributions. We expand the method originally proposed by Papernot et al. [25] to attack Android malware detection. We adapt it to handle binary features while at the same time preserving the Apps malicious functionality.

Applying the attack, *we are able to mislead our best performing malware detector (on the DREBIN dataset [2]) at rates higher than* 63 *%.*

As a second contribution, we investigate potential defense mechanisms for hardening malware detection models trained using DNNs.

We consider defensive distillation [27] and adversarial training [36,9]. The findings of our experimental evaluation of the aforementioned mechanisms is twofold. Applying defensive distillation reduces the rates at which adversarial examples are misclassified, but the improvement observed is often negligible. In comparison, training the model intentionally with adversarially crafted malware applications improves its robustness, as long as the perturbation introduced during adversarial training is carefully chosen.

2 Background

In this section, we explain the general concepts used in this paper. We first give a short introduction to malware detection. Afterwards, we move to the machine learning algorithm we apply, neural networks. Subsequently, we discuss adversarial machine learning with a focus on neural networks. We end the section by briefly reviewing defenses that have been proposed so far.

2.1 Malware Detection

Due to the increasing amount of published programs and applications, malware detection has become application of machine learning. The quality of detection depends then however heavily on the provided features. The literature generally differentiates two types of such features: static and dynamic features. Static features can directly be collected from the application's code and include, for example, n-gram frequencies in the code, opcode usage or control flow graph properties. Dynamic features, the nowadays more popular category, samples features

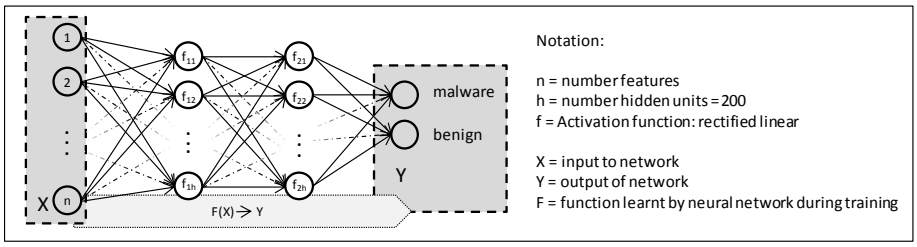

Fig. 1. The structure of deep feed-forward neural network as used in our setting.

from the application during runtime, observing general behavior, access and communication patterns.

As an example of an approach combining static and dynamic analysis we mention Marvin[20], which extracts features from an application while running it in an analysis sandbox and observing data flow, network behavior and other operations. This approach reaches an accuracy of 98.24% of malicious applications with less than 0.04% false positives.

In malware detection, not only accuracy, but also the false positive and false negative rates matter – classifying malware as benign might lead to a loss of trust by the users, whereas false negatives might lead to great financial loss for companies whose benign applications got classified as malware.

2.2 Neural Networks

We will now have detailed look at neural networks and introduce the required notation and definitions. Neural networks consist of elementary computing units—named *neurons*—organized in interconnected *layers*. Each neuron applies an *activation function* to its input to produce an output. Figure 1 illustrates the general structure of the network used throughout this paper and also introduces the notation used here.

Starting with the model input, each network layer produces an output used as input by the next layer. Networks with a single intermediate—*hidden*—layer are qualified as *shallow neural networks* whereas models with multiple hidden layers are *deep neural networks*. Using multiple hidden layers is interpreted as hierarchically extracting representations from the input [8], eventually producing a representation relevant to solve the machine learning task and output a prediction.

A neural network model $\mathbf{F}$ can be formalized as the composition of multi-dimensional and parametrized functions f_i each corresponding to a layer of the network architecture—and a representation of the input:

$$\mathbf{F} : \boldsymbol{x} \mapsto f_n(...f_2(f_1(\boldsymbol{x}, \theta_1), \theta_2)..., \theta_n) \tag{1}$$

where each vector θ_i parametrizes layer i of the network $\mathbf{F}$ and includes weights for the links connecting layer i to layer $i-1$. The set of model parameters $\theta = \{\theta_i\}$

is learned during training. For instance, in supervised settings, parameter values are fixed by computing prediction errors $f(x) - \boldsymbol{y}$ on a collection of known input-output pairs $(\boldsymbol{x}, \boldsymbol{y})$.

2.3 Adversarial Machine Learning

DNNs, like numerous machine learning models, have been shown to be vulnerable to adversarial manipulations of their inputs [36]. Adversarial goals thereby vary from simple *misclassification* of the input in a class different from the legitimate source class to *source-target misclassification* where samples from any source class are to be misclassified in a chosen *target class*. The space of adversaries was formalized for multi-class deep learning classifiers in a taxonomy [25]. Adversaries can also be taxonomized by the knowledge of the targeted model they must possess to perform their attacks.

Crafting an adversarial example $\boldsymbol{x}^*$—misclassified by model $\mathbf{F}$—from a legitimate sample $\boldsymbol{x}$ can be formalized as the following problem [36]:

$$\boldsymbol{x}^* = \boldsymbol{x} + \delta_{\boldsymbol{x}} = \boldsymbol{x} + \min \|\boldsymbol{z}\| \text{ s.t. } \mathbf{F}(\boldsymbol{x} + \boldsymbol{z}) \neq \mathbf{F}(\boldsymbol{x}) \tag{2}$$

where $\delta_{\boldsymbol{x}}$ is the minimal perturbation $\boldsymbol{z}$ yielding misclassification, according to a norm $\|\cdot\|$ appropriate for the input domain.

Due to the non-linearity and non-convexity of models learned by DNNs, a closed form solution to this problem is hard to find. Thus, algorithms were proposed to select perturbations approximatively minimizing the optimization problem stated in Equation 2. The *fast gradient sign method* introduced by Goodfellow et al. [9] linearizes the model's cost function around the input to be perturbed and selects a perturbation by differentiating this cost function with respect to the input itself and not the network parameters like is traditionally the case during training. The *forward derivative* based approach introduced by Papernot et al. [25] evaluates the model's output sensitivity to each input component using its Jacobian matrix. From this, we derive a saliency map ranking the individual features by their influence for a particular class.

All previous attack are white-box attacks, since they require access to the differentiable model. Additionally, black-box attacks leveraging both of the previous approaches to target unknown remotely hosted DNNs was proposed in [24]. The attack first approximates the targeted model by querying it for output labels to train a substitute model, which is then used to craft adversarial examples also misclassified by the originally targeted model.

Several approaches have also been presented in the literature to harden classifiers against such crafted inputs. Goodfellow et al. [9] employed an explicit training with adversarial examples. Papernot et al. [27] proposed distillation as another potential defense, of which a simpler alternative—label smoothing—was investigated by Warde-Farley et al. [38]. Since both, adversarial training and distillation, have only been investigated in the image classification setting, we will evaluate their performance for malware detection in Section 5.

3 Methodology

This section describes the approach to adversarial crafting for malware detection. We start by describing the data and how we train and configure the DNNs. Thereafter, we describe in detail how we craft adversarial examples, and detail how the perturbation search during adversarial example crafting needs to be adapted to our settings of malware detection.

3.1 Application Model

In the following, we describe the representation of applications we use as input to our malware detector. In this work, we focus on statically determined features of applications. As a feature, we understand some property that the statically evaluated code of the application exhibits. This includes whether the application uses a specific system call or not, as well as a usage of specific hardware components or access to the Internet.

A natural way to represent such features is using *binary indicator vectors*: Given features $1, \ldots, M$, we represent an application using the binary vector $\mathbf{X} \in \{0,1\}^M$, where X_i indicate whether the application exhibits feature i, i.e. $\mathbf{X}_i = 1$, or not, i.e. $\mathbf{X}_i = 0$. Due to the varied nature of applications that are available, M will typically be very large and sparse: each single application only exhibits very few features relatively to the entire feature set. This leads to very sparse feature vectors, and overall, a very sparsely populated space of applications in which we try to successfully separate malicious from benign applications.

3.2 Training the Malware Classifier

In this section, we describe how we train a malware detector using DNNs.

While Dahl et al. [6] use a neural network to classify malware, their approach uses random projections and dynamic data. Since perturbing dynamically gathered features is a lot more challenging than modifying static features, we consider the simpler, static case in this work and leave the dynamic case for future work. Also Saxe et al.[31] proposed a well functioning detection system based on a neural network, which is, to the best of our knowledge, not publicly accessible.

We will thus train out own neural network malware detection system. This also enables us to consider a worst case attacker having full knowledge about model and training data.

Since the binary indicator vector $\mathbf{X}$ we use to represent an application does not possess any particular structural properties or interdependencies, like for example images, we apply a regular, feed-forward neural network as described in Section 2 to solve our malware classification task.

We use a rectifier as the activation function for each hidden neuron in our network. As output, we employ a softmax layer for normalization of the output probabilities. the output is thus computed as

$$\mathbf{F}_i(\mathbf{X}) = \frac{e^{x_i}}{e^{x_0} + e^{x_1}} \ , \ x_i = \sum_{j=1}^{m_n} w_{j,i} \cdot x_j + b_{j,i} \tag{3}$$

Algorithm 1 Crafting adversarial examples for Malware Detection

Input: $\mathbf{x}$, $\mathbf{y}$, $\mathbf{F}$, $\mathbf{k}$, $\mathbf{I}$

$\mathbf{x}^* \leftarrow \mathbf{x}$
$\Gamma = \{1 \dots |\mathbf{x}|\}$
while $\arg\max_j \mathbf{F_j}(\mathbf{x}^*) \neq \mathbf{y}$ and $||\delta_\mathbf{X}|| < \mathbf{k}$ **do**
    Compute forward derivative $\nabla \mathbf{F}(\mathbf{x}^*)$
    $i_{max} = \arg\max_{j \in \Gamma \cap \mathbf{I}, X_j = 0} \frac{\partial \mathbf{F}_y(\mathbf{X})}{\partial \mathbf{X}_j}$
    if $i_{max} \leq 0$ **then**
        return Failure
    end if
    $\mathbf{x}^*_{i_{max}} = 1$
    $\delta_\mathbf{x} \leftarrow \mathbf{x}^* - \mathbf{x}$
end while
return $\mathbf{x}^*$

To train our network, we use standard gradient descent and standard dropout.

3.3 Crafting Adversarial Malware Examples

We next describe the algorithm that we use to craft adversarial examples against the malware detector we trained in the previous section. The goal of adversarial example crafting in malware detection is to mislead the detection system, causing the output of the classifier for a particular application to change according to the attacker's goal.

More formally, we start with $X \in \{0,1\}^m$, a binary indicator vector that indicates which features are present in an application. Given X, the classifier $\mathbf{F}$ returns a two dimensional vector $\mathbf{F}(\mathbf{X}) = [\mathbf{F}_0(\mathbf{X}), \mathbf{F}_1(\mathbf{X})]$ with $\mathbf{F}_0(\mathbf{X}) + \mathbf{F}_1(\mathbf{X}) = 1$ that encodes the classifiers belief that $\mathbf{X}$ is either benign ($\mathbf{F}_0(\mathbf{X})$) or malicious ($\mathbf{F}_1(\mathbf{X})$). We take as the classification result y the option that has the higher probability, i.e. $y = \arg\max_i \mathbf{F}_i(\mathbf{X})$. The goal of adversarial example crafting now is to find a small perturbation δ such that the classification results y' of $\mathbf{F}(X + \delta)$ is different from the original results, i.e. $y' \neq y$. We denote y' as our *target class* in the adversarial example crafting process.

Our goal is to have a malicious application classified as benign, i.e. given a malicious input $\mathbf{X}$, the classification results $y' = 0$. Note that our approach naturally extends to the symmetric case of misclassifying a benign application.

We adopt the adversarial example crafting algorithm based on the Jacobian matrix

$$\mathbf{J_F} = \frac{\partial \mathbf{F}(\mathbf{X})}{\partial \mathbf{X}} = \left[\frac{\partial \mathbf{F}_i(\mathbf{X})}{\partial \mathbf{X}_j} \right]_{i \in 0,1, j \in [1,m]}$$

of the neural network $\mathbf{F}$ put forward by Papernot et al. [25]. Despite it originally being defined for images, we show that a careful adaptation to a different domain is possible. Note, in particular, that this approach is not restricted to the specific DNN we described in the previous section, but to any differentiable classification function F.

To craft an adversarial example, we take mainly two steps. In the first, we compute the gradient of $\mathbf{F}$ with respect to $\mathbf{X}$ to estimate the direction in which a perturbation in $\mathbf{X}$ would change $\mathbf{F}$'s output. In the second step, we choose a perturbation δ of $\mathbf{X}$ with maximal positive gradient into our target class y'. For malware misclassification, this means that we choose the index $i = \mathsf{arg\,max}_{j \in [1,m], \mathbf{X}_j = 0} \mathbf{F}_0(\mathbf{X}_j)$ that maximizes the change into our target class 0 by changing $\mathbf{X}_i$. We repeat this process until either a) we reached the limit for maximum amount of allowed changes or b) we successfully cause a misclassification. A pseudo-code implementation of the algorithm is given in Algorithm 1.

Ideally, we keep the change small to make sure that we do not cause a negative change of $\mathbf{F}$ due to intermediate changes of the gradient. For computer vision, this is not an issue since the values of pixels are continuous and can be changed by as arbitrarily small perturbations as permitted by the encoding of the image. In the malware detection case, however, we do not have continuous data, but rather discrete input values: since $\mathbf{X} \in 0, 1^m$ is a binary indicator vector, our only option is to increase one component in $\mathbf{X}$ by exactly 1 to retain a valid input to $\mathbf{F}$. This motivates the changes to the original algorithm in [25].

Note finally that we only consider positive changes for positions j at which $\mathbf{X}_j = 0$, which correspond to adding features the application represented by $\mathbf{X}$ (since $\mathbf{X}$ is a binary indicator vector). We discuss this choice in the next subsection.

3.4 Restrictions on adversarial examples

To make sure that modifications caused by the above algorithms do not change the application too much, we bound the maximum distortion δ applied to the original sample. As in the computer vision case, we only allow distortions δ with $\|\delta\| \leq k$. We differ, however, in the norm that we apply: in computer vision, the L_∞ norm is often used to bound the maximum change. In our case, each modification to an entry will always change its value by exactly 1, and we thus use the L_1 norm to bound the overall number of features modified. We further bound the number of features to $k = 20$ (see Appendix B for details).

While the main goal of adversarial example crafting is to achieve misclassification, for malware detection, this cannot happen at the cost of the application's functionality: feature changes determined by Algorithm 1 can cause the application in question to lose its malware functionality in parts or completely. Additionally, interdependencies between features can cause a single line of code that is added to a malware sample to change several features at the same time. We discuss this issue more in detail in Appendix A.

To maintain the functionality of the adversarial example, we restrict the adversarial crafting algorithm as follows: first, we will only change features that result in a single line of code that needs to be added to the real application. Second, we only modify *manifest* features which relate to the `AndroidManifest.xml` file contained in any Android application. Together, both of these restrictions ensure that the original functionality of the application is preserved. Note that this approach only makes the crafting adversarial examples harder: instead of

Classifier/MR	Accuracy	FNR	FPR	MR	Dist.
Sayfullina et al. [32]	91%	0.1	17.9	–	–
Arp et al. [2]	93.9%	1	6.1	–	–
Zhu et al. [39]	98.7%	7.5	1	–	–
ours, 0.3	98.35%	9.73	1.29	63.08	14.52
ours, 0.4	96.6%	8.13	3.19	64.01	14.84
ours, 0.5	95.93%	6.37	3.96	69.35	13.47

Table 1. Performance of the classifiers. Given are used malware ratio (MWR), accuracy, false negative rate (FNR) and false positive rate (FPR). The misclassification rates (MR) and required average distortion (Dist.,in number of added features) with a threshold of 20 modifications are given as well. The last five approaches use the DREBIN data set.

using features that have a high impact on misclassification, we skip those that are not manifest features.

4 Experimental Evaluation

We evaluate the training of the neural network based malware detector and adversarial example-induced misclassification of inputs on it. Through our evaluation, we want to validate the following two hypotheses.

First, that the neural network based malware classifier achieves performance comparable to state-of-the-art malware classifiers (on static features) presented in the literature.

Second, the adversarial example crafting algorithm discussed in Section 3.3 allows us to successfully mislead the neural network we trained. As a measure of success, we consider the misclassification rate achieved by this algorithm. The misclassification rate is defined as the percentage of malware samples that are classified as benign after being altered, but are correctly classified before.

We base our evaluations on the DREBIN data set, originally introduced by Arp et al. [2]: DREBIN contains 129.013 android applications, of which 123,453 are benign and 5,560 are malicious. There are 8 feature classes, containing 545,333 static features, each of which is represented by a binary value that indicates whether the feature is present in an application or not. This directly translates to the binary indicator vector $\mathbf{X} \in \{0,1\}^M$ to represent applications, with $M = 545,333$. A more detailed breakdown of the DREBIN data set can be found in Appendix B.

4.1 DNN Model

We train numerous neural network architecture variants, according to the training procedure described in Section 3. Since the DREBIN data set has a fairly unbalanced ratio between malware and benign applications, we experiment with different ratios of malware in each training batch to compare the achieved performance values. The number of training iterations is then set in such a way that

all malware samples are at least used once. We evaluate the classification performance of each of these networks using accuracy, false negative and false positive rates as performance measures. We decided to pick an architecture consisting of two hidden layers each consisting of 200 neurons and provide more details about the performance of other architecture is a longer version of this paper. In Table 1 the accuracy as well as positive and negative false negative rates are displayed.

In comparison, Arp et al. [2] achieve a 6.1% false negative rate at a 1% false positive rate. Sayfullina et al. [32] even achieve a 0.1% false negative rate, however at the cost of 17.9% false positives. Saxe & Berlin [31] report 95.2% accuracy given 0.1 false positive rate, where the false negative rate is not reported. Zhu et al. [39], finally, applied feature selection and decision trees and achieved 1% false positives and 7.5 false negatives. As we can see, our networks are close to this trade-offs and can thus be considered comparable to state-of-the-art.

4.2 Adversarial Malware Crafting

Next, we apply the adversarial example crafting algorithm described in Section 3 and observe how often the adversarial inputs are able to successfully mislead our neural network based classifiers. As mentioned previously, we quantify the performance of our algorithm through the achieved misclassification rate, which measures the amount of previously correctly classified malware that is misclassified after the adversarial example crafting. In addition, we also measure the average number of modifications required to achieve misclassification to assess which architecture provided a harder time being mislead. As discussed above, we allow at most 20 modification to any of the malware applications.

The performance results are listed in Table 1. As we can see, we achieve misclassification rates from roughly 63% up to 69%. We can observe that the malware ratio used in the training batches is correlated to the misclassification rate: a higher malware ratio generally results in a lower misclassification rate.

While the set of frequently modified features across all malware samples differ slightly, we can observe trends for frequently modified features across all networks. For the networks of all malware ratios, the most frequently modified features are permissions, which are modified in roughly 30-45% of the cases. Intents and activities come in at second place, modified in 10-20% of the cases.

More specifically, for the network with ratio 0.3, the feature `intent.category.DEFAULT` was added to 86.4% of the malware samples. In the networks with the other malware ratios, the most modified feature was `permission.MODIFY_AUDIO_SETTINGS` (82.7% for malware ratio 0.4 and 87% for malware ratio 0.5).

Other features that are modified frequently are for example `activity.SplashScreen`, `android.appwidget.provider` or the GPS feature. And while for all networks the `service_receiver` feature was added to many malware samples, other are specific to the networks: for malware ratio 0.3 it is the `BootReceiver`, for 0.4 the `AlarmReceiver` and for 0.5 the `Monitor`.

Overall, of all features that we decided to modify (i.e. the features in the manifest), only 0.0004%, or 89, are used to mislead the classifier. Of this very

feature	total (0.3)	total (0.4)	total (0.5)
activity	16 (3)	14 (5)	14 (2)
feature	10 (1)	10 (3)	9 (3)
intent	18 (7)	19 (5)	15 (5)
permission	44 (11)	38 (10)	29 (10)
provider	2 (1)	2(1)	2 (1)
service_receiver	8 (1)	6 (1)	8 (1)
$\sum$	99 (25)	90 (26)	78 (23)

Table 2. Feature classes from the manifest and how they were used to provoke misclassification. Values in brakets denote number of features used in $> 1,000$ Apps.

small set of features, roughly a quarter occurs in more than 1,000 adversarially crafted examples. A more detailed breakdown can be found in Table 2.

Since our algorithm is able to successfully mislead most networks for a large majority of malware samples, we validate the hypothesis that our adversarial example crafting algorithm for malware can be used to mislead neural network based malware detection systems.

5 Defenses

In this section, we investigate the applicability of two defense mechanisms previously introduced—defensive distillation (Papernot et al. [27]) and adversarial training (Szegedy et al. [36])—in the setting of malware classification. We also investigated feature selection as a defense, but leave the description of the approach a longer version of this paper, since it did not yield conclusive results.

To measure the effectiveness of defensive mechanisms against adversarial examples, we monitor the misclassification rates. The misclassification rate is defined as the percentage of malware samples that are misclassified after the application of the adversarial example crafting algorithm, but were correctly classified before. We simply compare these rates of the original network and the network where the mechanism was applied.

5.1 Distillation

We will investigate now a defense introduced in the context of a computer vision application, distillation, and investigate its applicability in binary, discrete cases such as malware detection. We first introduce the concept of distillation as used by Papernot et al. [27]. Afterwards, we present our evaluation.

While distillation was originally proposed by Hinton et al. [12] as a way to transfer knowledge from large neural networks to a smaller ones, Papernot at al. [27] recently proposed using it as a defensive mechanism against adversarial example crafting. They motivate this through its capability to improve the second network's generalization performance (i.e. classification performance on test samples) and the smoothing effect on the decision boundary.

The idea is, in a nutshell, to use an already existing classifier $\mathbf{F}(\mathbf{X})$ that produces probability distribution over the classes $\mathcal{Y}$. This output is used, as labels, to train a second model F'. Since the new label contain more information about the data $\mathbf{X}$ than the simple class labels, the network will perform similar or better than the original network F. In the original idea, the second trained network is smaller than the first one, whereas in Papernot et al.'s approach, both networks are of the same size.

An important detail in the distillation process is the slight modification of the final softmax layer (cf. Equation 3) in the original network $\mathbf{F}$: instead of the regular softmax normalization, we use

$$\mathbf{F}_i(X) = \left(\frac{e^{z_i(x)/T}}{\sum_{l=1}^{|\mathcal{Y}|} e^{z_l(x)/T}} \right), \tag{4}$$

where T is a distillation parameter called *temperature*. For $T = 1$, we obtain the regular softmax normalization commonly used in training. If T is large, the output probabilities approach a more uniform distribution, whereas for small T, the output of $\mathbf{F}$ will become more extreme. To achieve a good distillation result, we use the output of the original network $\mathbf{F}$ produced at a high temperature T and use this output to train the new network $\mathbf{F}'$.

The overall procedure for hardening our classifier against adversarial examples can thus be summarized in the following three steps.

1. Given the original classifier $\mathbf{F}$ and the samples $\mathcal{X}$, construct a new training data set $D = \{(\mathbf{X}, \mathbf{F}(X)) \mid \mathbf{X} \in \mathcal{X}\}$ that is labeled with $\mathbf{F}$'s output at high temperature.
2. Construct a new neural network $\mathbf{F}'$ with the same architecture as $\mathbf{F}$.
3. Train $\mathbf{F}'$ on D.

Note that both step two and step three are performed under the same high temperature T to achieve a good distillation performance.

Evaluation We now apply the above procedure on our originally trained classifiers and examine the impact of distillation as a defensive mechanism against adversarial examples in the domain of malware detection. Figure 2 shows the effects of distillation on misclassification compared to the original models. We use a rather low temperature of 10, since we observe a strong decrease of accuracy when distilling on higher temperatures. In general we observe a strong increase of the false negative rate, and a slight increase in the false positive rate. For ratio 0.5, it raises from 4 to 6.4, whereas it is equivalent for 0.3. The accuracy varies in between 93-95%.

We further observe that the misclassification rate drops significantly, in some cases to 38.5% for ratio 0.4. The difference in the average number of perturbed features, however, is rather small. The number of perturbed features is 14 for ratio 0.3 to 16 for the other two.

Using distillation, we can strengthen the neural network against adversarial examples. However, the misclassification rates are still around 40%. Additionally,

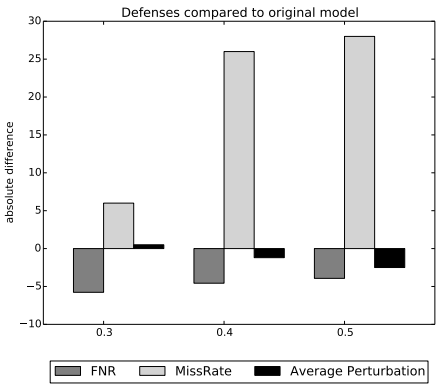

Fig. 2. False negative rates, misclassification rates and average required distortions after applying distillation, original networks are the baseline. For FNR and misclassification rate, higher is better. Average distortion should be negative.

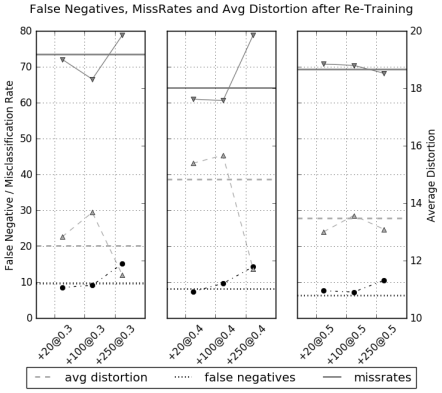

Fig. 3. Misclassification rates, false negative rates and average required distortion achieved on adversarially trained networks. Regular network's performance is given as baseline, indicated by horizontal lines.

we pay this robustness with a less good classifier. The effect is further not as strong as on computer vision data. Papernot et al. [27] reported rates around 5% after distillation for images. We further observed that higher temperature (> 25), as used in computer vision settings, strongly harms accuracy.

5.2 Adversarial Training

We now apply adversarial training and investigate its influence on the robustness on the resulting classifier. As before, we first introduce the technique of adversarial training and then report the results we observed.

Adversarial training means to additionally train our classifier with adversarially crafted samples. This method was originally proposed by Szegedy at al. [36] and involves the following steps:

1. Train the classifier $\mathbf{F}$ on original data set $D = B \cup M$, where B is the set of benign, and M the set of malicious applications
2. Craft adversarial examples A for $\mathbf{F}$ using the forward gradient method described in Section 3.3
3. Iterate additional training epochs on $\mathbf{F}$ with the adversarial examples from the last step as additional, malicious samples.

By applying adversarial training, we aim to improve the model's generalization, i.e. predictions for samples outside of our training set. Good generalization generally makes a classifier less sensitive to small perturbations, and therefore also more resilient to adversarial examples.

Evaluation We now present the results when applzing adversarial training to our networks. Using $n_1 = 20$, $n_2 = 100$ and $n_3 = 250$ additional adversarial examples, we continued their training. We combined the adversarial examples to create training batches by mixing them with benign samples at each network's malware ratio. We then trained the network for one more epoch on one training batch and re-evaluated their susceptibility against adversarial examples.

Figure 3 illustrates the performance (false negative rate) of the adversarially trained networks and the misclassification rate Algorithm 1 achieved on them (in misclassification rate and average required distortion). We grouped networks by their malware ratio during training.

For the network trained with malware ratio 0.3 and 0.4, we observe a reduction of the misclassification rate, and an increase of the required average distortion for n_1 and n_2 additional training samples. For instance, we achieve a misclassification rate of 67% for the network trained with 100 additional samples at 0.3 malware ratio, from 73% for the original network. A further increase of the adversarial training samples used for adversarial training, however, causes the misclassification rate to increase again to 79% for both malware ratios.

For the networks trained with malware ratio 0.5, the misclassification rate only decreases if we use 250 adversarial training samples. Here, we reach 68% misclassification rate, down from 69% for the original network. For fewer amount of adversarial examples for adversarial training, the misclassification rate remains very similar to the original case. It seems that the network trained with 0.5 malware ratio is fitting very close to the malware samples it was trained on, and therefore requires more adversarial examples to generalize and improve its robustness against adversarial example crafting.

Overall, we can conclude that simple adversarial training does improve the neural network's robustness against adversarial examples. The number of adversarial examples required to improve the robustness depend heavily on the training parameters we chose for training the original networks. However, choosing too many may also further degrade the network's robustness against adversarial examples. This is likely explained by the fact that when too many adversarial examples are used for training, the neural network then overfits to the particular perturbation style used to craft these adversarial examples.

5.3 Summary and Discussion of Evaluation

We evaluated two potential defensive mechanisms, adversarial retraining and distillation.

Adversarial training achieved consistent reduction of misclassification rates across different models. The amount of adversarial training samples has a significant impact on this reduction. Iteratively applying adversarial training to a network may further improve the network's robustness. Unfortunately, this defense is only effective against the perturbation styles that are fed to the model during training.

Distillation does have a positive effect, but does not perform as well as in the computer vision setting. It remains unclear whether this is due to the binary nature or the unbalanced classes of the data. This is left as future work.

Finally, we note that these defenses are non-adaptive: an adversary may exploit knowledge of the defense deployed to evade it.

.

6 Related Work

The following discussion of related work complements the references included in Section 2. The security of machine learning is an active research area [26]. Barreno et al. [3] give a broad overview of attacks against machine learning systems. Previous work showed that adversarial examples can be constructed for different algorithms and also generalize between machine learning techniques in many cases [9,24,36,4,21,30].

Many more defenses have been developed than used here. We will thus focus on introducing the main ideas relevant to neural networks and malware. There are many more variants of adversarial training [15,11,23], all slightly differing in their objectives from the original version introduced by Goodfellow et al. [9].

Other approaches include blocking the gradient flow [37], changing the activation function [17], or directly classifying adversarial examples as out of distribution [7,22,13,10]. Finally, also the application of statistics has been investigated [19,28]. An exhaustive study of defenses in a single article is, due to the variety and number of approaches, not feasible. We thus focused on the two most promising approaches.

Related to adversarial examples for malware, Hu and Tan [14] propose another approach to generate examples which is however based on generative adversarial networks.

Further Biggio et al. [4] propose a method that is based on gradient descent. They evaluate their adversarial examples similar to Šrndić and Laskov [18], who show the viability of adversarially crafted inputs against a PDF malware detection system based on random forests. Their adversarial example crafting algorithm, however, focuses on features in the *semantic gap* between the specific classifier they study and PDF renderers, i.e. this gap includes features that are only considered by the classifier, but not by the renderer. While this allows them to generate unobservable adversarial perturbations, their approach does not generalize to arbitrary classifiers.

In contrast, our approach considers all editable features and identifies those that need to be perturbed in order to achieve misclassification. Our technique is applicable to any differentiable machine learning classifier. While this still requires the identification of suitable application perturbations that correspond to feature perturbations, as we discussed in Section 3, this is mostly an orthogonal problem that needs to be solved independently.

7 Conclusion and Future Work

In this paper, we investigated the viability of adversarial example crafting against neural networks in a domain different from computer vision and relevant to core security problematics. On the DREBIN data set, we achieved misclassification rates of up to 69% against models that achieve classification performance comparable to state-of-the-art models from the literature. Further, our adversarial examples have no impact on the malware's functionality. Threat vectors like adversarial examples need to be taken into account by defenders.

As a second contribution, we examined two potential defensive mechanisms for hardening our neural networks against adversarial examples. Our evaluations of these mechanisms showed the following: first, distillation does improve misclassification rates, but does not decrease them as strongly as observed in computer vision settings. Secondly, adversarial training achieves consistent reduction of misclassification rates across architectures.

Acknowledgments

Nicolas Papernot is supported by a Google PhD Fellowship in Security. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 610150. This work was further partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0344). This research was also sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies,either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes not with standing any copyright notation here on.

References

1. I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, G. Paliouras, and C. D. Spyropoulos. An evaluation of naive bayesian anti-spam filtering. *arXiv preprint cs/0006013*, 2000.
2. D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of NDSS*, 2014.
3. M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010.
4. B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Srndic, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *ECML PKDD 2013, Proceedings, Part III*, pages 387–402, 2013.

5. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.
6. G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *Proceedings of the 2013 IEEE ICASSP*, pages 3422–3426, 2013.
7. Z. Gong, W. Wang, and W.-S. Ku. Adversarial and Clean Data Are Not Twins. *ArXiv e-prints*, Apr. 2017.
8. I. Goodfellow, Y. Bengio, and A. Courville. Deep learning. Book in preparation for MIT Press, 2016.
9. I. J. Goodfellow et al. Explaining and harnessing adversarial examples. In *Proceedings of ICLR*, 2015.
10. K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel. On the (Statistical) Detection of Adversarial Examples. *ArXiv e-prints*, Feb. 2017.
11. S. Gu and L. Rigazio. Towards deep neural network architectures robust to adversarial examples. *CoRR*, abs/1412.5068, 2014.
12. G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. *ArXiv e-prints*, 2015.
13. H. Hosseini, Y. Chen, S. Kannan, B. Zhang, and R. Poovendran. Blocking Transferability of Adversarial Examples in Black-Box Learning Systems. *ArXiv e-prints*, Mar. 2017.
14. W. Hu and Y. Tan. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. *ArXiv e-prints*, Feb. 2017.
15. A. G. O. II, C. L. Giles, and D. Kifer. Unifying adversarial training algorithms with flexible deep data gradient regularization. *CoRR*, abs/1601.07213, 2016.
16. A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.
17. D. Krotov and J. J. Hopfield. Dense Associative Memory is Robust to Adversarial Inputs. *ArXiv e-prints*, Jan. 2017.
18. P. Laskov et al. Practical evasion of a learning-based classifier: A case study. In *Proceedings of the 36th IEEE S&P*, pages 197–211, 2014.
19. X. Li and F. Li. Adversarial examples detection in deep networks with convolutional filter statistics. *CoRR*, abs/1612.07767, 2016.
20. M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Proceedings of the 39th Annual International Computers, Software and Applications Conference (COMPSAC)*, 2015.
21. Y. Liu, X. Chen, C. Liu, and D. Song. Delving into transferable adversarial examples and black-box attacks. *CoRR*, abs/1611.02770, 2016.
22. J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff. On detecting adversarial perturbations. *CoRR*, abs/1702.04267, 2017.
23. T. Miyato, A. M. Dai, and I. J. Goodfellow. Virtual adversarial training for semi-supervised text classification. *CoRR*, abs/1605.07725, 2016.
24. N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, and al. Practical black-box attacks against deep learning systems using adversarial examples. *arXiv preprint arXiv:1602.02697*, 2016.

25. N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of IEEE EuroS&P*, 2016.
26. N. Papernot, P. McDaniel, A. Sinha, and M. Wellman. Towards the science of security and privacy in machine learning. *arXiv preprint arXiv:1611.03814*, 2016.
27. N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a defense to adversarial perturbations against deep neural networks. *Proceedings of IEEE S&P*, 2015.
28. S. S. A. B. G. Reuben Feinman, Ryan R. Curtin. Detecting adversarial samples from artifacts. *CoRR*, abs/1703.00410, 2017.
29. K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.
30. A. Rozsa, M. Günther, and T. E. Boult. Are Accuracy and Robustness Correlated? *ArXiv e-prints*, Oct. 2016.
31. J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *10th International Conference on Malicious and Unwanted Software, MALWARE*, pages 11–20, 2015.
32. L. Sayfullina, E. Eirola, D. Komashinsky, P. Palumbo, Y. Miché, A. Lendasse, and J. Karhunen. Efficient detection of zero-day android malware using normalized bernoulli naive bayes. In *Proceedings of IEEE TrustCom*, pages 198–205, 2015.
33. A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *CIS*, pages 329–333. IEEE, 2010.
34. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
35. R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE S&P*, pages 305–316. IEEE, 2010.
36. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *Proceedings of ICLR*. Computational and Biological Learning Society, 2014.
37. Q. Wang, W. Guo, A. G. O. II, X. Xing, L. Lin, C. L. Giles, X. Liu, P. Liu, and G. Xiong. Using non-invertible data transformations to build adversary-resistant deep neural networks. *CoRR*, abs/1610.01934, 2016.
38. D. Warde-Farley and I. Goodfellow. Adversarial perturbations of deep neural networks. In T. Hazan, G. Papandreou, and D. Tarlow, editors, *Advanced Structured Prediction*. 2016.
39. Z. Zhu and T. Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of ACM SIGSAC*, pages 767–778, 2016.

A Generating Suitable Perturbations

Finding suitable perturbations for arbitrary features with interdependencies boils down to an optimization problem where gradient information determined by the adversarial crafting algorithm is used as a fitness function to rate per-

ID	Name	Manifest	Code	#
S_1	Hardware Components	✓		4513
S_2	Permissions	✓		3812
S_3	Components	✓		218951
S_4	Intents	✓		6379
S_5	Restr. API Calls		✓	733
S_6	Used Permissions		✓	70
S_7	Susp. API Calls		✓	315
S_8	Network Addresses		✓	310447

Fig. 4. Feature Types, where they are collected and their cardinality.

	benign	malicious
1st Q.	23	35
Mean	48	62
3rd Q.	61	83
max	9661	666

Fig. 5. Some basic statistics on the number of features per app in the DREBIN data set. Q. denotes Quantiles.

turbations (by weighing each feature affected by a perturbation with the feature's gradient). This was, in fact, essentially the intuition behind the adversarial saliency maps introduced by [25] to find adversarial image perturbations. Finding a suitable perturbation thus boils down to finding the perturbation with maximal fitness. To this end, it is necessary to identify the set of all possible perturbations that can be performed without altering an applications behavior (as by the restrictions formulated above). This issue is, in general, highly dependent on the specific application domain and orthogonal to the general adversarial crafting problem we examine in this paper. We consider this, however, a very fruitful direction for future work.

B Data Set

In this Appendix, we provide some more details about the DREBIN data set, originally introduced by Arp et al. [2].

The 8 feature classes in DREBIN cover various aspects of android applications, including: A) Permissions and hardware component access requested by each application (e.g. for `CAMERA` or `INTERNET` access). B) Restricted and suspicious (i.e. accessing sensitive data, e.g. `getDeviceID()`) API-calls made by the applications. C) application components such activities, service, content provider and broadcast receivers used by each applications, and D) intents used by applications to communicate with other applications. Table 4 lists each feature class and its cardinality.

In Figure 5 we give average and quantile statistics on the amount of features exhibited by the applications in DREBIN. Given these numbers, we decide to set our distortion bound $k = 20$ – assuming we are modifying an application of average size, it still remains within the two main quartiles when adding at most 20 features.