Bachelor thesis
Computing Science

Radboud University

Effectiveness of Common Defense Mechanisms Against Adversarial Inputs for DNN based Android Malware Detectors

Author:
Hendrik Willing
s4735439
H.Willing@student.ru.nl

First supervisor:
Dr. Veelasha Moonsamy
email@veelasha.org

Second assessor:
prof. dr. Martha Larson
M.Larson@cs.ru.nl

June 24, 2019

Abstract

Tremendous popularity of Android devices makes Android malware detection an important issue that needs to be resolved. Machine learning algorithms are often used to detect malicious applications (*malwares*). Many of these machine learning models have shown to be vulnerable to *adversarial inputs*: inputs intentionally crafted to cause a machine learning model to misclassify. This vulnerability is a general problem in various state-of-the-art machine learning algorithms.

This work focuses on a malware detector based on a deep neural network (DNN). We evaluate different defensive measures, including feature reduction and ensemble learning, whether they can effectively reduce the sensitivity of deep neural network based malware detectors against adversarial inputs. This study shows that especially ensemble learning is able to reduce the success of adversarial samples. An ensemble of a deep neural network and K-Nearest Neighbor (KNN) model was used to improve the detection rate of adversarial inputs from 54% to almost 90% for the studied models. However, the evaluation also shows that these methods can limit the harm only to a certain extent.

Contents

Chapter 1

Introduction

Over the last years, smartphones and tablets have become extremely popular. One of the most popular operating systems on those devices is Android [2]. Due to its popularity, Android is an interesting target for attacks. Attackers offer their malicious software in the form of small applications (apps). According to current analysis in Google's annual *Android Security and Privacy Year in Review* [3] the percentage of potentially harmful app installs amounts to 0.04% for the official *Google Play* store and 0.92% for Apps outside of the Google Play store. Even though this seems to be a very low percentage, with over 2 million available apps in the Google Play store [4], security is a major aspect that needs to be considered. G Data identified more than four million malwares in 2018; more than ever before [5][6]. For users, these apps masquerade as normal and useful. However, in the background they perform malicious actions to gain access to the device or to harm their users. To protect users' privacy and to be able to detect malicious apps, machine learning has become widely established. Different machine learning algorithms and models are used for classification to determine whether an app is benign or malicious. Nevertheless, recent research has shown that many machine learning algorithms lack robustness against *adversarial inputs*. These are inputs formed by applying some minor changes to earlier classified data such that the model gives an incorrect answer - often with high confidence.

In this bachelor's thesis, we investigate the following question:

> Can a machine learning model for Android malware detection be made more robust against adversarial inputs?

Therefore, we take a look at deep neural networks, one type of machine learning models. We analyze whether different pre-processing techniques can improve the performance of the classifier against adversarial inputs. With this thesis, we would like to answer the questions of whether common defense mechanisms are able to counteract adversarial inputs. Further, we want to

identify which features and which properties of apps are especially important to differentiate between malware and cleanware. This makes it possible to additionally determine how those mechanisms can be used to improve both, the accuracy of a classifier and the robustness against adversarial inputs. After each of these techniques (*defense mechanisms*) is tested against adversarial inputs, the results are compared and analyzed to determine why a certain combination of features and model performed that way. Finally, we make suggestions on which method is able to improve a malware classifier and to increase its robustness against adversarial inputs.

The thesis will mainly base upon research from Grosse et al. [1]. In their research, they showed that adversarial samples are not only a threat for image classification but that they can also be applied in the domain of malware detection. We will rebuild their model and use its performance as a basis, to determine the efficiency of each defense mechanism.

Chapter two will give some background information about basic terms. In chapter three, we will discuss the previous research done in this field. Chapter four deals with the set up of the thesis. The chapter will give an overview of the methodology and provide all the technical details of the experiments. Chapter five discusses the results and findings of the experiments and chapter six will conclude this thesis and give suggestions for future research.

The source code of this thesis is available at `https://github.com/hendrikw257/defense-adversarial-android-malware`

Chapter 2

Preliminaries

2.1 Android App Structure

Android is an open-source operating system that was initially released in 2008. It runs on top of a modified Linux kernel and runs Java-written applications, called apps.

Android apps are bundled in the form of an APK file. When unpacked, an app consists of several different files and folders. One file that is mandatory for each app is called *AndroidManifest.xml*. It contains metadata of the app and describes its essential information, such as permissions. To access the system, each app must request permissions. A user needs to accept all permissions the app requests to be able to install it. Once installed, apps can interact with each other and the system through application programming interface (API) calls.

Furthermore, Android apps consist of various components [7]: activities, services, broadcast receivers, and content providers. An activity is the representation of a single screen that handles interactions between users and apps. Services are components that run in the background of the operating system to perform long-running operations while a different application is running in the foreground. Broadcast receivers respond to broadcast messages from other applications or the system. They allow an app to respond to broadcast announcements outside of a regular user flow. A content provider manages a shared set of app data and stores them in the file system. It also supplies data from one app to another on request.

2.2 Machine Learning

Machine learning algorithms classify given data based on different properties they identify in that data. These algorithms allow us to better understand all kinds of data and make predictions about it. Therefore, machine learning algorithms make use of statistics and mathematics to analyze given data and

give us new insights into that data. Furthermore, machine learning models search for interesting patterns in data, which can help to gain a better understanding of the relations between specific properties of it. In the context of security, machine learning models support us in detecting threats and malicious behavior in software. One fundamental difference between machine learning algorithms and other kinds of algorithms is that traditional algorithms define what a computer needs to do. Machine learning algorithms, in contrast, learn how to solve a problem while they get trained on representative example data. Instead of simply deciding based on preconfigured rules, a machine learning malware detector learns to determine whether an application is benign or malicious by learning from given examples of cleanware and malware. The model, therefore, extracts different features from each of the given examples to figure out dependencies and to be able to make accurate decisions.

2.2.1 Neural Networks

A *neural network* is one type of machine learning algorithm. It is a network of small units, called *neurons*. Each neuron itself is a small, simple function. Neurons output a function based on their input data and some parameters, which are optimized during training. In a neural network, the neurons are arranged in a graph that is organized in a number of *layers* linked to each other.

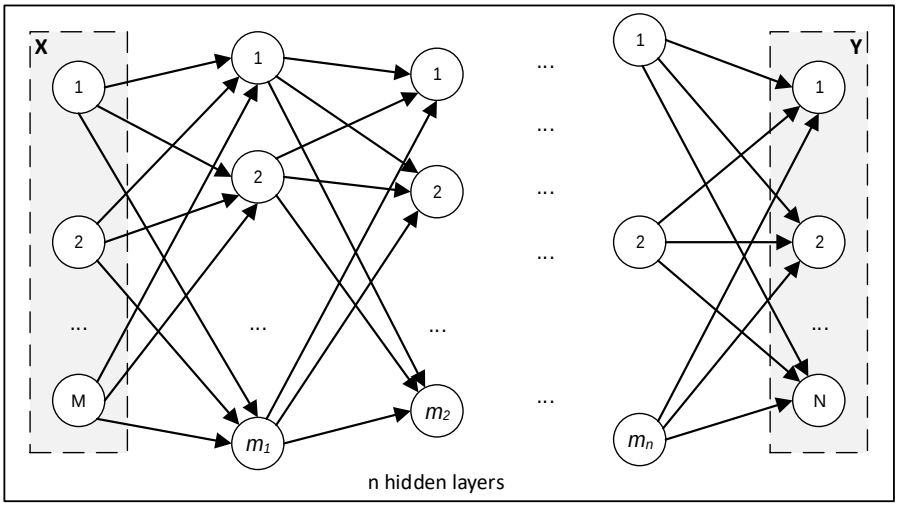
In this thesis, we will use a *deep feed-forward neural network* (Figure 2.1), where *deep* means that the network consists of multiple layers, called *hidden layers*. Feed-forward neural networks consist of multiple layers in that each neuron is connected to all neurons in the next layer, but connections never go backward.

2.2.2 Adversarial Samples

Adversarial samples are crafted by adding small perturbations to inputs to force a targeted model to misclassify the sample. Often these perturbations are so small that they cannot be detected by humans.

In this research, we will use the *forward derivative* based approach, which was initially introduced by Papernot et al. [8] and slightly adjusted by Grosse et al. [1]. This approach evaluates the model's output sensitivity to each input component using its Jacobian matrix, the *forward derivative*. After that, a perturbation is selected which is based on the derivative. For a successful application of this method, full knowledge of the model's architecture and parameters is needed. This might not be the case for real-life attacks since often attackers do not have full access to the architecture of a model. An attack like this is called *black-box* attack: an attacker is only able to observe labels given by an unknown deep neural network (black-box) in response to

chosen inputs. Papernot et al. [9] proposed such an attack. They use the output labels of a remote deep neural network to create a local substitute model. Adversarial samples are then crafted on the local substitute model. Since the methods are basically the same for both attacks, we will skip the step of creating a substitute model and evaluate each method on a model, which we have full access to.



Notation
F: function learned by neural network during training
X: input of neural network
Y: output of neural network
M: input dimension (number of neurons on input layer)
N: output dimension (number of neurons on output layer)
n: number of hidden layers in neural network
f: activation function of a neuron
H_k: output vector of layer k neuron

Indices
k: index for layers (between 0 and n + 1)
i: index for Input X component (between 0 and N)
j: index for output Y component (between 0 and M)
p: index for neurons (between 0 and m_k for any layer k)

Figure 2.1: The structure of deep feed-forward neural network with notations used in this thesis, via Papernot et al. [8]

Chapter 3

Related Work

The process of analysis in malware detectors can be divided into two main groups: static analysis and dynamic analysis [10]. In static analysis, the features are extracted from an application without executing. The analysis is done based on the extracted features. One disadvantage of this technique is that an application might hide some of its functions by making use of encryption or dynamically downloaded code from an external server. These methods cannot be detected by static analysis. In order to detect those cases and to defend against them, dynamic analysis is used. During dynamic analysis, the malware runs in an isolated environment - a *sandbox* - in that the application is monitored. This allows a malware detector to analyze the dynamic behavior of an application. Dynamic analysis thereby reveals what an application really does when it attacks a real device.

Next to these methods, malware detectors use signatures or file hashes to detect malicious software. They generate a unique fingerprint for known malware and compare them to new samples to determine whether an application is benign or malicious. This method has one significant disadvantage, namely each malicious sample must be known before it can be classified. It offers no protection against new and unknown threats, the so-called *zero-day vulnerabilities*. In the fast-evolving sector of cyber security, however, this ability is crucial to make a good model and be able to provide proper protection. That is one of the reasons for the growing importance of machine learning models in that area.

3.1 Adversarial Samples

In research, published in 2014, Szegedy et al. [11] made a concerning discovery: several machine learning models are vulnerable to *adversarial samples*. This means that machine learning models consistently misclassify inputs, which are built by intentionally adding small perturbations to existing samples from a dataset. The perturbations can be really small, such that they

are not detectable by humans. These small changes can, however, be sufficient to cause a model to output the wrong result with high confidence.

Goodfellow et al. [12] argue that the linear nature of neural networks is the primary cause of the vulnerability to adversarial perturbation. They proposed a simple and fast method to craft adversarial methods that are consistently misclassified by machine learning models with high confidence. Furthermore, they show that the problem of adversarial samples can be generalized across architectures and training sets. Papernot et al. [8] built on that approach and introduced different algorithms to craft adversarial samples. They focus especially on deep neural networks (DNNs). To craft adversarial samples, the algorithms need a precise understanding of the mapping between inputs and outputs of DNNs. The algorithms then reliably produce samples correctly classified by humans but misclassified by a DNN while only modifying a small number of input features per sample.

These two papers focused mainly on the domain of image classification. To show that this is a general problem of machine learning models and that it is not limited to a specific domain, Grosse et al. [1] transferred the results to the domain of Android malware classification. They showed that the same methods and algorithms can be used to slightly manipulate the input of a malware classifier such that it will be misclassified, without breaking the functionality of the application. They additionally evaluate different potential defense mechanisms against adversarial crafted samples. To perform the experiments, we will make use of the model they describe in their research. Their model serves as a basis with that we can compare the results of our experiments. We will also use some of their results and evaluate additional defense mechanisms. Moreover, we will make use of their algorithm to craft adversarial samples to perform the evaluation.

The techniques described in the research above do require full knowledge about the configuration of the model. As Papernot and Goodfellow et al. showed [13], full knowledge is not necessary to be able to successfully craft adversarial samples. An attacker may create and train a local substitute model, which is then used to craft adversarial samples. After that, the attacker can transfer these samples to a victim model that an attacker has only limited information about. They show that adversarial samples crafted for one model might also deceive other models. It demonstrates that machine learning models are in general vulnerable to adversarial samples regardless of their structure.

3.2 Android Malware Detection

One of the biggest benefits of machine learning models is their ability to generalize. It allows malware detectors that are based on machine learning algorithms to adapt properly to new, previously unseen attacks. This is

one of the reasons why machine learning models are increasingly applied in security-related tasks over the last years.

To deal with the variability of attacks and to understand how to improve models against different attacks, Demontis et al. [14] model different attackers with different skills and capabilities. With their work, they show that machine learning can be used to improve security if one takes adversaries into account and proactively anticipates attackers.

Saracino et al. [15] developed an Android app for Android malware detection. Their app evaluates the behavior of an application on different levels to determine if it is malicious or benign. Additionally, they propose a taxonomy of malware into different classes and describe common patterns across the classes.

In this thesis, we will evaluate different defense mechanisms.

The first method that we will evaluate is the reduction of features to represent the applications. Similar approaches were followed by Xiong et al. [16] and Grosse et al. [1]. They reduce the feature space by selecting features based on how they are represented in the dataset. By formulating different restrictions, they reduced the size of the feature vectors and removed some of the features from the initial sample set. Only features that complied with their restrictions were kept in the dataset.

A different approach was followed by Moonsamy et al. [17]. They searched for patterns in the permissions of Android applications by investigating the differences of permission patterns between benign and malicious apps. This distinction can be important to determine whether an app is benign or malicious.

To evaluate the first defense method, we will make use of a combination of these two approaches. First, we will reduce the number of features by applying different manually selected restrictions. After that, we will search for the most important patterns in the remaining features. The findings of the analysis will then be used to further reduce the amount of features.

Another method that we will evaluate is an ensemble of the DNN with another machine learning algorithm. This means that we train multiple models instead of a single model and combine the predictions from each of these models. This can result in predictions that are better than any single model. Abouelnaga et al. [18] followed a similar approach in their work. They trained various machine learning algorithms and evaluated the performance of different combinations of them. They showed that it is possible to improve the accuracy of a model and how fine-tuning the parameters can further improve the model.

Xiong et al. [16] propose two different methods to combine clustering and classification. The resulting model does not only outperform the single model accuracy but they also improved the ability to detect new and

previously unknown malware families.

We will use the findings of these papers to build our own ensemble. Next to the optimal selection of parameters, we will also incorporate the vulnerability of the second model to adversarial samples, which were initially crafted to mislead a DNN [13].

Chapter 4

Research

Methodology

This section describes the setup and process of the experiments. For this thesis, we will make use of the machine learning model and configuration described in [1]. To evaluate the model's performance, the DREBIN [19][20] dataset is used.

The next sections describe the configuration of the model and the dataset. We further inspect the process of training the model and the algorithm to craft adversarial samples. After that, we implement and evaluate different defense mechanisms.

4.1 Reproduction of Results from Grosse et al. [1]

4.1.1 Classifier

To be able to perform the experiments, we start by implementing the machine learning model, a deep neural network, as described in [1]. The model serves as a fundament from which we can expand further and evaluate different defense methods on. Moreover, we will use it to compare the efficiency of each of the applied methods. As shown in the paper, the deep neural network achieves performances comparable to other current malware classifiers. The model consists of various layers - between one and four - and different amounts of neurons in each layer. Each layer contains between 10 and 300 neurons. Between the layers, a dropout of 50% was chosen. This means that the output of 50% of all neurons in each layer is ignored and set to 0. It helps the model to avert overfitting. After the final hidden layer, they use a *softmax* layer to normalize the output of the network. Moreover, they use *rectified non-linearity* as activation function for each hidden neuron in the network:

$$\forall p \in [1, m_k] : f_{k,p}(x) = \max(0, x)$$

The activation function describes a calculation each neuron does on its inputs to decide whether it should be "fired" (activated) or not. As model input, static features of applications are used. These are information about an application, which can be extracted without executing the application but rather information that can be extracted from, e.g. the *AndroidManifest.xml*, such as permissions.

4.1.2 DREBIN Dataset

The DREBIN dataset contains 5,560 malwares and 123,453 cleanwares, which, in total, contain 545,333 different features. The dataset provides a text file for each app containing all features of that application. First of all, we have to check whether the dataset is the same compared to the research from Grosse et al. [1] to make sure that our model is trained on the same data. For that, we build a table as in [1] (table 4.1) and make use of a set of *Python* scripts. Initially, we extract the amount of features from the dataset and compare them to the amount of features in the given paper. We do this by iterating through all the files and reading it line by line. Each line contains the feature class and the feature. For every feature class, a Python list is created in that the features are stored. After reading all the files, we count the number of unique features in each of the lists.

During comparison, we noticed that most of the categories match (see table 4.1). However, we see that the amounts of *network addresses* do not match. Overall, we identified 41 network addresses more than in the given paper.[1] After summing up all the individual features we get a total of 545,333 features, which is the same as in the underlying paper. The different amount of *network addresses* therefore does not affect our model. Overall, we train and evaluate our model with the same total amount of features.

ID	Category	Amount ([1])	Amount
S_1	Hardware Components	4,513	4,513
S_2	Permissions	3,812	3,812
S_3	Components	218,951	218,951
S_4	Intents	6,379	6,379
S_5	Restr. API Calls	733	733
S_6	Used Permissions	70	70
S_7	Susp. API Calls	315	315
S_8	Network Addresses	310,447	310,488

Table 4.1: **Amount of features distinguished by category**

[1] This mismatch was also communicated to the author of the paper [1]. We were, however, not able to resolve this difference.

The categories above actually contain different types of features. Table 4.2 shows the distribution of features over the different categories.

Category-ID	**Type**	**Amount**
S_1	provider	$4,513$
S_2	permission	$3,812$
S_3	service_receiver	$33,222$
S_3	activity	$185,729$
S_4	intent	$6,379$
S_5	call	733
S_6	real_permission	70
S_7	api_call	315
S_8	url	$310,488$
Unassigned	feature	72

Table 4.2: **Amount of features distinguished by type**

The DREBIN dataset distinguishes feature types as in table 4.2 above. Each feature type is, again, assigned to one of the categories from table 4.1.

We can see that feature types *url* and *activity* contain, by far, the most features. URLs describe every single network address an app tries to connect with and they belong to category S_8. Activities describe a single screen of an Android app. Since an app often consists of various screens, the number of activities can be very large. As for URLs, there are no restrictions for the name of activities. The name of it can be freely chosen by the publisher of the app. This explains the high amount of features, compared to other types.

Category S_3 contains the components of an application. An Android app consists of four different components: *activities, services, content providers,* and *broadcast receivers*. This means that the category combines the two feature types *service_receiver* and *activity* from the dataset.

One feature type that was not assigned to any category is *feature*. It mainly describes features of category S_1, like *'android.hardware.location'* or *'android.hardware.sensor.compass'*. Features of this type were not assigned to any category in [1], which is the reason that we also do not assign them.

However, we include them in the final feature vector as we will see in the following section.

4.1.3 Preparing the Data for Training

We represent each app in the form of a *binary feature vector*, which we will then feed into the model. These vectors contain all features of an app. They are defined as $\mathbf{X} \in \{0,1\}^M$, where X_i indicates whether the feature i is part of the application ($\mathbf{X}_i = 1$) or not ($\mathbf{X}_i = 0$). Since there are many different functionalities, M can become very large. As we saw in the previous section (4.1.2), the DREBIN dataset contains $545,333$ different features. This means that for our model $M = 545,333$.

We, again, use a Python script to create the feature vectors. Each vector $\mathbf{X}$ is represented by a *NumPy*[2] array of size M. For each feature of that application, the value at the corresponding position in the array is set to 1, $\mathbf{X}[i] = 1$. The values at all the remaining indexes are set to 0.

4.1.4 Training the Malware Classifier

To build a classifier, we make use of the Python library *TensorFlow*[3]. The feature vectors from the previous section are used to represent the apps. To determine the best classifier, the authors of [1] compared several classifiers with different configurations (table 4.3). Every configuration gets the same binary feature vectors as input. The models differ in the number of hidden layers and the amount of neurons per layer. To train the model, we craft batches of size $1,000$. Each batch contains a pre-defined ratio of malwares as defined in table 4.3. First, we randomly select the amount of malware samples according to the ratios. After that, we fill up the batch with randomly selected benign apps. For training, we split each batch into mini-batches of size 100 and train the model using 10 epochs per iteration. To evaluate the performance of each of the configurations, we determine accuracy, false negative and false positive rates as measurements. Based on these measures, we compare the performance of our model with the base model.

4.1.5 Evaluating the Malware Classifier

The batches for the evaluation of the model are created in the same way as for the training. First, we randomly select the number of malwares according to the malware ratio. Next, we fill up the batch with the appropriate amount of cleanwares. Each model configuration is tested with 5 different batches. After that, we compute the average of all resulting measures of the validation

[2] `https://www.numpy.org/`
[3] `https://www.tensorflow.org`

batches. The results of the evaluation of the base classifier are depicted in table 4.3.

Configuration	**MWR**	**Accuracy ([1])**	**FNR ([1])**	**FPR ([1])**	**Accuracy (our model)**	**FNR (our model)**	**FPR (our model)**
[200]	0.4	97.83	8.06	1.86	**95.99**	**4.57**	**3.63**
[200]	0.5	95.85	5.41	4.06	**96.35**	**4.37**	**2.93**
[10, 10]	0.3	97.59	16.37	1.74	**96.53**	**6.87**	**2.00**
[10, 10]	0.4	94.85	9.68	4.90	**96.72**	**4.37**	**2.60**
[10, 10]	0.5	94.75	7.34	5.11	**95.73**	**5.37**	**3.20**
[10, 200]	0.3	97.53	11.21	2.04	**95.93**	**9.80**	**1.63**
[10, 200]	0.4	96.14	8.67	3.6	**96.63**	**6.43**	**1.33**
[10, 200]	0.5	94.26	5.72	5.71	**95.93**	**6.03**	**2.10**
[200, 10]	0.3	95.63	15.25	3.86	**96.93**	**7.73**	**1.10**
[200, 10]	0.4	93.95	10.81	5.82	**96.59**	**4.07**	**3.00**
[200, 10]	0.5	92.97	8.96	6.92	**96.88**	**4.23**	**2.00**
[50, 50]	0.3	96.57	12.57	2.98	**96.47**	**6.17**	**2.37**
[50, 50]	0.4	96.79	13.08	2.73	**96.60**	**4.43**	**2.77**
[50, 50]	0.5	93.82	6.76	6.11	**96.23**	**4.13**	**3.43**
[50, 200]	0.3	97.58	17.30	1.71	**96.71**	**8.87**	**0.90**
[50, 200]	0.4	97.35	10.14	2.29	**96.53**	**3.90**	**3.20**
[50, 200]	0.5	95.65	6.01	4.25	**96.03**	**4.27**	**3.67**
[200, 50]	0.3	96.89	6.37	2.94	**96.06**	**7.87**	**2.27**
[200, 50]	0.4	95.87	5.36	4.06	**96.11**	**5.53**	**2.20**
[200, 50]	0.5	93.93	4.55	6.12	**95.63**	**7.70**	**1.03**
[100, 200]	0.4	97.43	8.35	2.27	**96.58**	**5.70**	**1.90**
[200, 100]	0.4	97.32	9.23	2.35	**95.82**	**6.00**	**2.97**
[200, 100]	0.5	96.35	6.66	3.48	**96.39**	**2.80**	**4.37**
[200, 200]	0.1	98.92	17.18	0.32	**97.55**	**16.67**	**0.90**
[200, 200]	0.2	98.38	8.74	1.29	**96.41**	**14.17**	**0.97**
[200, 200]	0.3	98.35	9.73	1.29	**96.45**	**8.40**	**1.50**
[200, 200]	0.4	96.6	8.13	3.19	**96.91**	**5.27**	**1.63**
[200, 200]	0.5	95.93	6.37	3.96	**96.60**	**3.53**	**3.27**
[200, 300]	0.3	98.35	9.59	1.25	**95.94**	**10.37**	**1.37**
[200, 300]	0.4	97.62	8.74	2.05	**95.47**	**4.00**	**4.90**
[300, 200]	0.2	98.13	9.34	1.5	**95.57**	**5.87**	**4.07**
[300, 200]	0.4	97.29	8.06	2.43	**96.39**	**6.20**	**1.90**
[200, 200, 200]	0.1	98.91	17.84	0.31	**97.53**	**18.87**	**0.63**
[200, 200, 200]	0.4	97.69	10.34	1.91	**95.95**	**4.03**	**4.07**
[200, 200, 200, 200]	0.4	97.42	13.08	1.07	**96.52**	**4.80**	**2.63**
[200, 200, 200, 200]	0.5	97.5	12.37	2.01	**96.27**	**4.91**	**2.53**

Table 4.3: **Comparison of the performance of the different configurations of our malware detector with the model from the literature.** Given are used malware ratio (MWR), accuracy, false negative rate (FNR) and false positive rate (FPR)

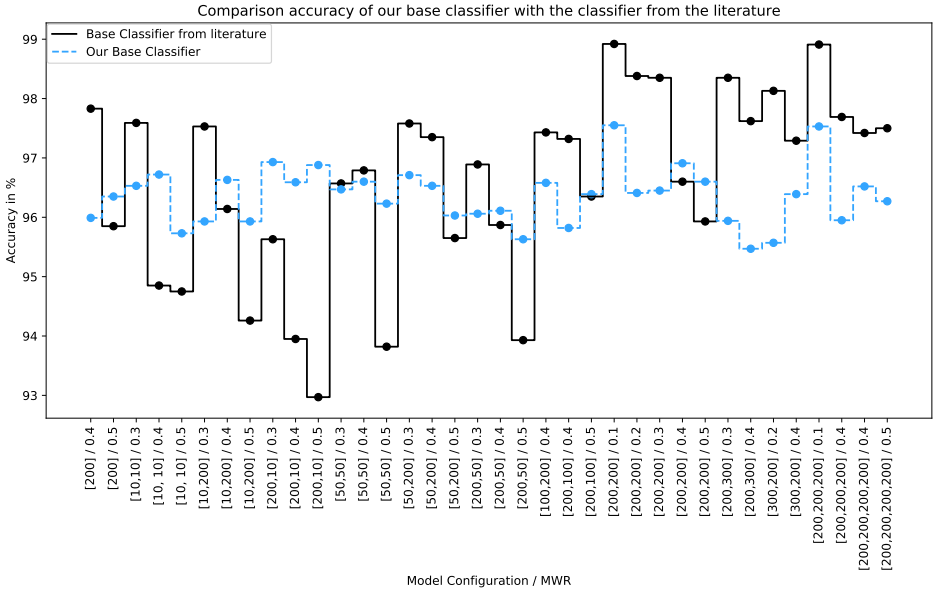


Figure 4.1: Comparison of the performance of the base classifier from the literature [1] and our base classifier.

When we compare the results of our model with the one from the literature, we can see that our model is able to achieve similar outputs. The accuracy of the two models is consistently within a range of, at most, $\pm 2.6\%$ for all configurations. Also, the false positive rate and false negative rate behave the same for most of the configurations. The small discrepancies between the values are mainly a consequence of the fact that the source code of the classifier from the literature was not accessible. As a result, the model was rebuilt based on its description in the given paper. Since both models work with the same dataset the generalizations, and therefore the achieved performance approximately should be the same. In reality, however, slight differences in the methodology of training and composition of training sets and validation sets lead to minor differences in the values of the model's hyperparameters. As a consequence, the performance of both models slightly differ. These differences are also depicted in false positive rates and false negative rates. Even though the differences are rather large for various configurations, the values generally evolve proportionally evenly across the configurations. The main reason for those differences is the composition of datasets, which are randomly assembled. Overall, our model achieves performances comparable to the network from Grosse et al. [1].

4.1.6 Crafting Adversarial Samples

The goal of adversarial samples is to entice a machine learning model to misclassify a certain input. In the domain of malware detection, this means that an attacker wants to change the input of a malware detection system such that the classification result changes according to the desired result of the attacker. Usually, this means that an attacker would like to slightly adjust the application by adding or removing features such that it will not be detected by a malware detector. Since, in our case, we use binary feature vectors to represent an application, it is enough to adjust the value of the feature vector at the corresponding index.

Grosse et al. [1], introduced an algorithm (Algorithm 1) that can be used to craft adversarial inputs for our DNN. They adopt a crafting algorithm, which was introduced by Papernot et al. [8] to craft adversarial samples in the domain of image classification. It makes use of the *forward derivative* approach. This approach evaluates the model's output to each input component by computing its Jacobian matrix. Adversarial samples are crafted in mainly two steps. In the first step, they compute the gradient of $\mathbf{F}$ with respect to the input $\mathbf{X}$ to estimate the direction in which a perturbation in $\mathbf{X}$ would change $\mathbf{F}$'s output. This is called the forward derivative and it is essentially the Jacobian of the function corresponding to what the neural network learned during training. Secondly, they choose a perturbation δ of $\mathbf{X}$ with maximal positive gradient into the target class. In other words, they choose the index i_{max} that causes the maximum change into the target class by changing $\mathbf{X}_{i_{max}}$, the value of $\mathbf{X}$ at index i_{max}.

Algorithm 1: Crafting adversarial samples

input: x, y, F, k

1 $\mathbf{x}^* \leftarrow \mathbf{x}$
2 $\Gamma = \{1 \dots |\mathbf{x}|\}$
3 **while** $\arg\max_j \mathbf{F}_j(\mathbf{x}^*) \neq \mathbf{y}$ and $\| \delta_{\mathbf{x}} \| < \mathbf{k}$ **do**
4 Compute forward derivative $\nabla \mathbf{F}(\mathbf{x}^*)$
5 $i_{max} = \arg\max_{j \in \Gamma \cap \mathbf{I}, X_j = 0} \frac{\partial \mathbf{F}_y(\mathbf{x}^*)}{\partial \mathbf{X}_j}$
6 **if** $i_{max} \leq 0$ **then**
7 **return** Failure
8 $\mathbf{x}^*_{i_{max}} = 1$
9 $\delta_{\mathbf{x}} \leftarrow \mathbf{x}^* - \mathbf{x}$
10 **return** $\mathbf{x}^*$

Describing the process of adversarial sample crafting and the algorithm more formally, we start with a binary feature vector, $\mathbf{X} \in \{0,1\}^M$, which represents the features of an application. Given $\mathbf{X}$, classifier $\mathbf{F}$ returns a two dimensional vector $\mathbf{F}(\mathbf{X}) = [\mathbf{F}_0(\mathbf{X}), \mathbf{F}_1(\mathbf{X})]$, for which holds that $\mathbf{F}_0(\mathbf{X}) + \mathbf{F}_1(\mathbf{X}) = 1$. This vector contains the output of the classifier. It describes the assessment with the probabilities to which extent $\mathbf{X}$ is benign

($\mathbf{F}_0(\mathbf{X})$) or malicious ($\mathbf{F}_1(\mathbf{X})$). The final classification result y is the option with the higher probability, so $y = \arg\max(\mathbf{F}(\mathbf{X}))$. The goal is to find a small perturbation δ such that the result of the classification changes: $y' = \arg\max(\mathbf{F}(\mathbf{X} + \delta))$ and $y' \neq y$. To achieve this, the algorithm iteratively changes the value of $\mathbf{X}$ that causes the maximum change into the direction of the classification result towards the target class, $y' = 0$.

Ideally, the changes should be kept small to make sure that they do not negatively affect intermediate perturbations. However, for malware classification, there are some additional challenges compared to adversarial sample crafting in the domain of image classification. The values of pixels are continuous, which means that we can change them in very small steps. For malware detection, however, there are only discrete values 0 and 1. One of the restrictions of the algorithm is that it only adds features to $\mathbf{X}$. It does not remove any. This is to make sure that the application still functions after the changes and that the functionality of the application is not disturbed. In other words, the algorithm only changes $\mathbf{X}_i$ from 0 to 1 and not vice versa.

The performance of our model is depicted in table 4.4 and figure 4.2. As for the performance of the original model on regular data in section 4.1.5, we notice differences in achieved misclassification rates between our model and that from the literature. Again, this is mainly based on the fact that training and validation sets are randomly selected.

Configuration	MWR	MR ([1])	Distortion ([1])	MR (our model)	Distortion (our model)
[200]	0.4	81.89	11.52	**41.59**	**6.85**
[200]	0.5	79.37	11.92	**51.22**	**6.33**
[10, 10]	0.3	69.62	13.15	**27.80**	**10.30**
[10, 10]	0.4	55.88	16.12	**34.83**	**10.87**
[10, 10]	0.5	84.05	11.48	**37.50**	**14.17**
[10, 200]	0.3	75.47	12.89	**27.47**	**11.22**
[10, 200]	0.4	55.70	14.84	**37.03**	**9.26**
[10, 200]	0.5	57.19	14.96	**47.33**	**10.20**
[200, 10]	0.3	50.07	14.96	**33.27**	**10.39**
[200, 10]	0.4	35.31	17.79	**39.30**	**8.31**
[200, 10]	0.5	36.62	17.49	**48.13**	**9.08**
[50, 50]	0.3	61.71	15.37	**30.70**	**8.21**
[50, 50]	0.4	60.02	14.70	**39.70**	**8.58**
[50, 50]	0.5	40.97	17.64	**50.23**	**7.94**
[50, 200]	0.3	79.25	11.61	**30.50**	**5.84**
[50, 200]	0.4	69.44	13.95	**38.20**	**10.03**
[50, 200]	0.5	64.66	15.16	**45.83**	**10.43**
[200, 50]	0.3	66.55	14.99	**31.03**	**8.37**
[200, 50]	0.4	58.31	15.76	**40.53**	**7.47**
[200, 50]	0.5	62.34	14.54	**49.93**	**6.61**
[100, 200]	0.4	74.93	12.87	**39.87**	**7.77**
[200, 100]	0.4	71.42	13.12	**76.18**	**40.97**
[200, 100]	0.5	73.02	12.98	**44.50**	**11.39**
[200, 200]	0.1	78.28	10.99	**10.87**	**4.42**
[200, 200]	0.2	63.49	13.43	**20.73**	**4.99**
[200, 200]	0.3	63.08	14.52	**30.63**	**6.57**
[200, 200]	0.4	64.01	14.84	**40.47**	**6.27**
[200, 200]	0.5	69.35	13.47	**48.60**	**8.28**
[200, 300]	0.3	70.99	13.24	**30.70**	**5.27**
[200, 300]	0.4	61.91	14.19	**42.13**	**8.60**
[300, 200]	0.2	69.96	13.62	**22.23**	**7.35**
[300, 200]	0.4	63.51	14.01	**40.77**	**6.59**
[200, 200, 200]	0.1	75.41	10.50	**10.60**	**7.08**
[200, 200, 200]	0.4	71.31	13.08	**41.17**	**8.78**
[200, 200, 200, 200]	0.4	62.66	14.64	**39.00**	**7.93**

Table 4.4: **Comparison of the performance of our model with the model from the literature on adversarial input.** Given are the different configurations of our malware detector, malware ratio, misclassification rates (MR), and required average distortion (in number of added features)

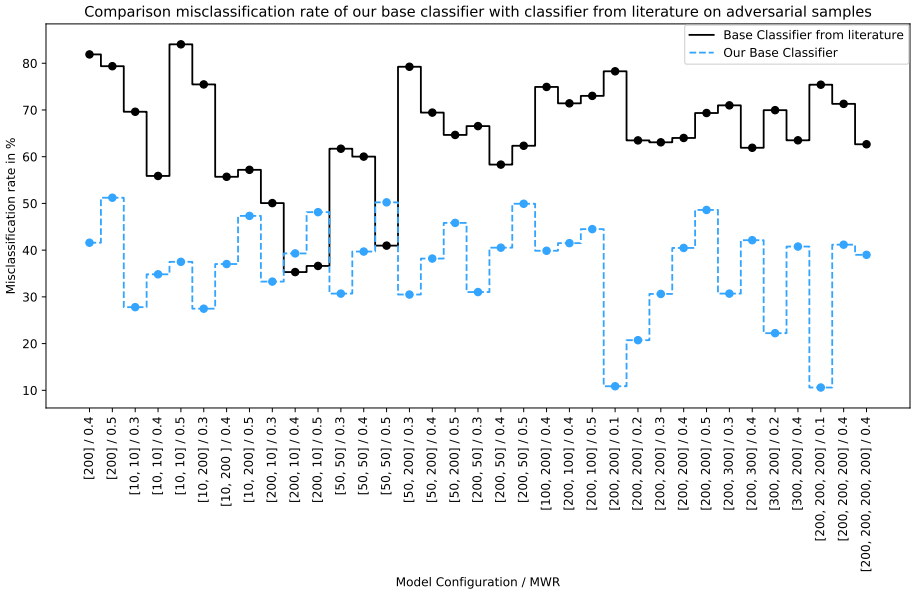


Figure 4.2: Comparison of the performance of the base classifier from the literature and our own base classifier on adversarial crafted samples.

4.2 Defenses

In this section, we evaluate different potential defense mechanisms to figure out whether they can help to support neural network based malware detectors against adversarial samples. We measure the effectiveness of each method by comparing the classification results with the results of the conventional model from section 4.1.

First, we look at feature reduction: we reduce the number of features to decrease the size of the feature vectors. On the one hand, we limit the number of adjustable features for an attacker. On the other hand, we hope to make the neural network more sensitive to changes in the input. We desire to make it more difficult for attackers to find proper features for changes, necessary to craft adversarial samples.

After that, we consider an ensemble of different machine learning models. Research by Papernot et al. [13] suggests that different machine learning models are not generally vulnerable to the same adversarial samples. By combining different machine learning models, we hope to get models that individually may have weaknesses against adversarial inputs but do cancel out these weaknesses when combined.

Next, we combine the methods of feature reduction and ensemble learning. Instead of using the entire feature space, we evaluate our ensemble of models on the reduced feature space.

Finally, we inspect two methods already proposed in the literature:

distillation [21][1] and re-training on adversarial samples [11][1].

4.2.1 Feature Reduction

The amount of features in the DREBIN dataset leads to feature vectors with vast dimensionality. This leads to high computational costs. Furthermore, it means that the feature set contains many features with low support. Support describes the fraction by which an itemset is represented in the dataset. In other words, the percentage to which the feature occurs concerning other features in the dataset. As a result, these features are not important for classification. Another drawback of such large feature vectors is that it is comparatively easy for an attacker to find possible features that cause a significant change in the model's prediction. This makes it easier for an attacker to craft adversarial samples.

In the first step, we eliminate some of the feature classes. The first category of features we remove is the category of *network addresses*. Attackers can change URLs very easily and without much effort. Another aspect is that most of the addresses in the dataset are unique. From $310,488$ addresses in total, only $57,392$ appear in at least three apps. Next to the class of URLs, we remove the features of the type *activity*. In Android Apps, activities are only the representation of a single screen of an App that acts as a user interface. The name of an activity can be freely chosen. Almost every application contains an application called '*Main*' but other than that, there are no further restrictions or requirements. This gives us almost the same picture as for network addresses: the majority of the names are unique to single apps. These categories contain a large number of features while having only little effect on malware detection, which means that we can remove them.

The whole dataset contains $545,333$ features in total. After removing these types, the dataset still contains $49,116$ features. On the resulting dataset, we make a feature analysis to extract the most important features and to determine how to reduce the feature space even more. During the analysis, we noticed that the dataset contains many features that do occur in only a few apps and some features that have very high support (see table 4.5). Those features have only little effect on the results of malware detection. Therefore, we additionally remove features that occur in 5 applications or less. Furthermore, we remove all features with a support $\geq 90\%$ in both classes, malwares and cleanwares. For the DREBIN dataset, this means that the three features *'feature::android.hardware.touchscreen'*, *'intent::android.intent.action.MAIN'*, and *'intent::android.intent.category.LAUNCHER'* are removed. All these features have a support of more than 93% in both target classes.

cleanwares		malwares	
feature	**support**	**feature**	**support**
feature::android.hardware.-touchscreen	99.78%	feature::android.hardware.-touchscreen	99.35%
intent::android.intent.action.-MAIN	97.48%	intent::android.intent.action.-MAIN	96.24%
intent::android.intent.category.-LAUNCHER	95.99%	permission::android.permission.-INTERNET	95.74%
call::getSystemService	84.68%	intent::android.intent.category.-LAUNCHER	93.96%
real_permission::android.-permission.INTERNET	83.78%	call::getSystemService	93.26%
permission::android.permission.-INTERNET	83.42%	real_permission::android.-permission.INTERNET	89.78%
call::getPackageInfo	57.19%	permission::android.permission.-READ_PHONE_STATE	88.69%
call::printStackTrace	53.43%	real_permission::-android.permission.-READ_PHONE_STATE	75.29%
permission::android.permission.-ACCESS_NETWORK_STATE	51.69%	call::getDeviceId	67.64%

Table 4.5: **Most frequent features of cleanware and malware**

After the removal of all these features as explained above, we were able to reduce the size of each feature vector $\mathbf{X}$ to $4,109$.

In total, the DREBIN dataset contains malwares from 179 different families. Arp et al. [19] extracted the five most common features in the top 20 families (table 4.6)

Category	Malware Family																			
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**
S_1	✓			✓				✓					✓							
S_2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S_3			✓	✓	✓					✓		✓		✓			✓			
S_4		✓				✓	✓						✓			✓				
S_5	✓				✓															
S_6																				✓
S_7	✓	✓		✓		✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	
S_8			✓					✓	✓	✓	✓				✓	✓		✓	✓	✓

Table 4.6: **Contribution of the most used features of the top** 20 **malware families.**

Table 4.6 shows that category S_2 (*requested permissions*) - are represented in all malware families. The reason for that is rather simple: every app needs to define the permission it wants to use in its *AndroidManifest.xml*. Features from category S_7 are used second-most by malwares of the top 20 malware families. These findings may indicate that this category can be very important for Android malware detection. None of the most frequent features we identified (table 4.5) originates from this category.

Evaluation

In order to evaluate the effectiveness of feature reduction, we first need to adjust our model such that it is able to handle the new input dimension. As we will see, the performance of the model on regular data is comparable with the performance of the base classifier from section 4.1.5. Nevertheless, the performance of the new model continuously is slightly lower than the performance of our base model. Therefore, feature reduction massively reduces the computational overhead resulting in a reduction of running time.

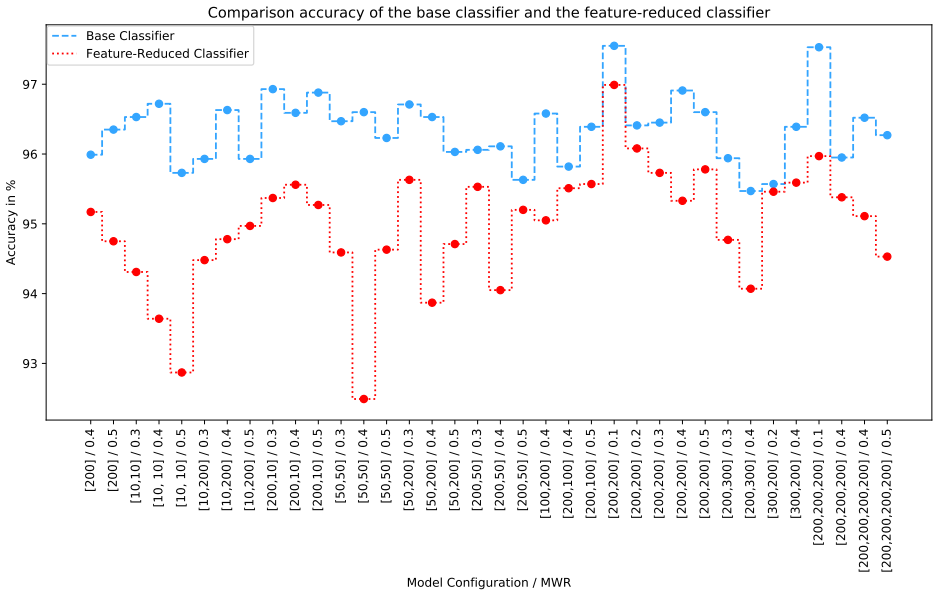


Figure 4.3: Comparison of the performance of the base classifier, trained on the entire feature space, and a classifier, trained on the reduced feature space.

In comparison with the performance of the base model, we notice that both models achieve approximately the same results, even though the performance of the feature-reduced model is consistently below the performance of the original model (figure 4.3). This means that by reducing the amount

of features, we do not lose too much information necessary for malware detection. The model is still able to detect malware pretty well. Only for a few configurations, like the configuration with two layers of ten neurons that were trained on datasets with malware ratio 0.5, we notice a massive drop in achieved accuracy compared to the original model. On average, the original model is 1.3% better than the feature-reduced model.

The performance on adversarial samples, in contrast, demonstrates that a reduced feature space causes a higher misclassification (figure 4.4), while at the same time reducing the amount of minimum average amount of changes (figure 4.5). For attackers, this has the consequence that crafting an adversarial sample becomes easier when the amount of features reduces. These observations are similar to the findings of Grosse et al. The feature reduction methods they describe in their paper were also not able to strengthen a malware classifier on the input of adversarial samples.

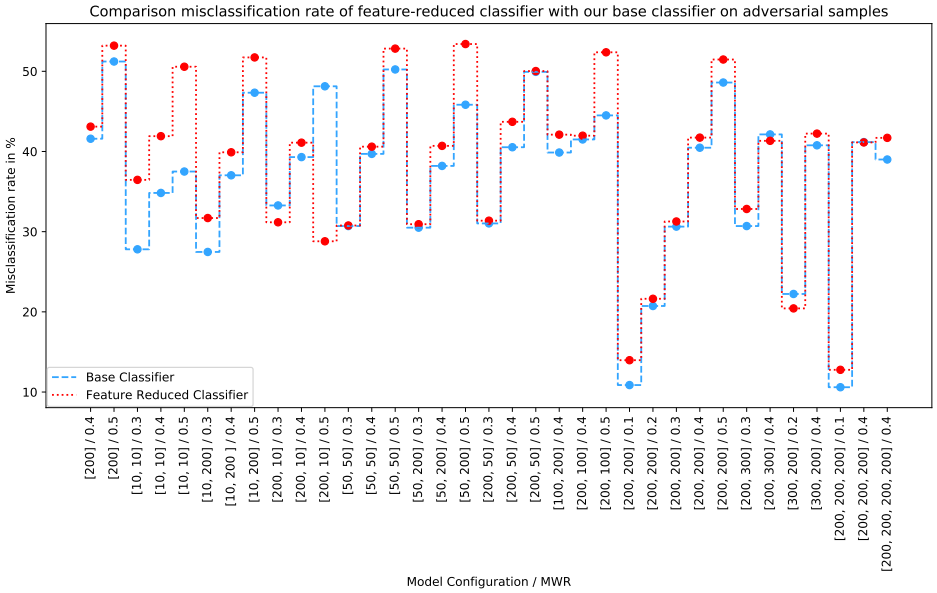


Figure 4.4: Comparison of the performance of the base classifier, trained on the entire feature space, and a classifier, trained on the reduced feature space on adversarial samples.

As the graph (figure 4.5) shows, the model that was trained on a reduced feature space consistently has a higher misclassification rate. Compared to the base model, which achieves a misclassification rate between 10.6% and 51.22%, the misclassification rate of the feature-reduced model ranges from 12.77% to 53.4%. On average, each configuration of the model trained on a reduced feature space performs about 2% worse according to its misclassification rate.

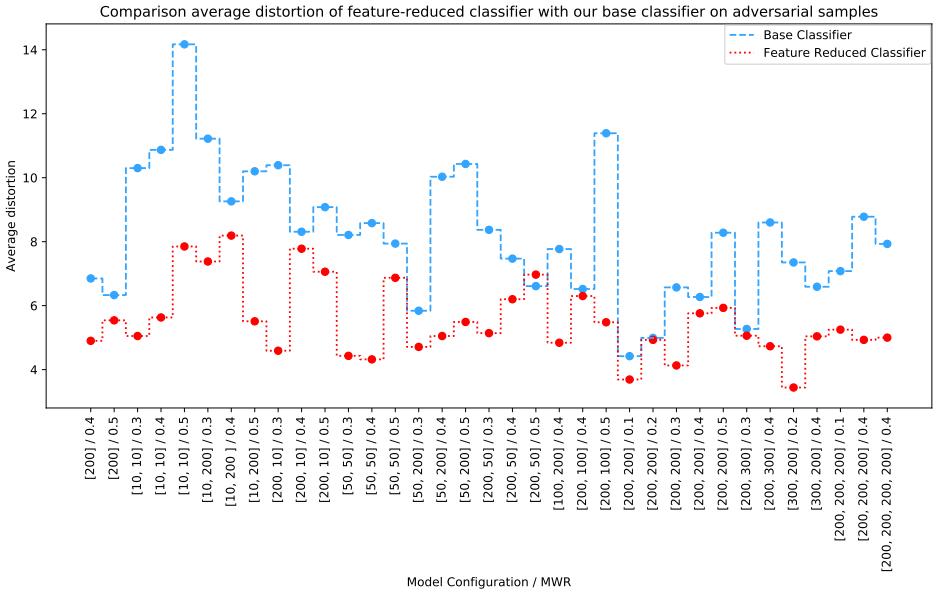


Figure 4.5: Comparison of the average distortion of the base classifier, trained on the entire feature space, and a classifier, trained on the reduced feature space on adversarial samples.

The minimum amount of features that need to be changed to successfully craft an adversarial sample decreased to 3.44, compared to 4.42 for our base model. On average, the distortion decreased by about 2 (2.66). In other words, for each configuration, we need to change about 2 features less to adjust a malware to cause our malware detector to classify it as benign.

The reduced amount of features negatively affects the detection rate of adversarial samples. Instead of making it more difficult for an attacker to find suitable features to change, fewer features make it easier for an attacker to find features and craft adversarial samples. This may be related to the fact that each feature, in the reduced feature space, has a higher weight. The value of every individual feature has a greater impact on the final prediction of the model. One change in an iteration of the crafting algorithm 1 causes a greater shift in the direction of the target class.

All in all, we can infer that feature reduction does not help to increase the robustness of a neural network against adversarial input. It even weakens our model on both regular inputs and especially on adversarial inputs. Therefore, feature reduction is not a suitable method to strengthen a neural network based malware classifier against adversarial crafted samples.

4.2.2 Ensemble of Models

Next, we take a look at a combination of two different machine learning models. We combine the deep neural network from section 4.1 with a K-Nearest Neighbors (KNN) model. As earlier research [13] suggests, KNN is one of the machine learning models that is least vulnerable to adversarial samples, which were initially crafted to mislead deep neural networks. We, therefore, hope not only to improve the overall performance of the malware detector but more importantly, we hope to reduce the model's misclassification rate on adversarial samples. For the implementation of the K-Nearest Neighbors algorithm, we use the Python machine learning library *scikit-learn*[4].

K-Nearest Neighbors is a supervised machine learning algorithm used for classification. It scans through all past experiences and looks up the k closest data points. In other words, it looks for the k points nearest to the new point to predict the class of it. The performance of a KNN model depends on the value of K. To determine the best possible value for K, we compare the results of the model on datasets with malware ratios ranging from 0.1 to 0.5 and with k's up to 20. The results of that analysis are depicted in figure 4.6.

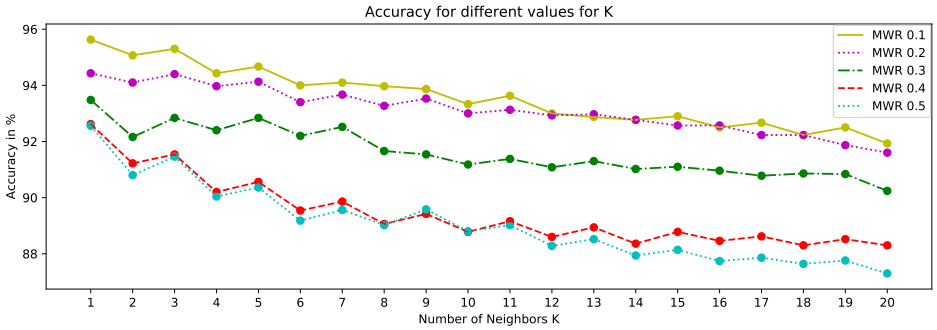


Figure 4.6: Validation accuracy of the KNN model with values for K ranging from 1 to 20 on a dataset containing different ratios of malwares.

As we can see, for values ranging between $k = 3$ and $k = 7$, the three graphs of datasets with malware ratio 0.1, 0.2, and 0.3 are very similar. Even though the models with lower malware ratio almost consistently achieve better performances. Between $k = 1$ and $k = 7$ the accuracy on datasets with malware ratio 0.3 almost remains constant. The graphs for datasets with malware ratio 0.4 and 0.5 are also pretty similar. Both graphs level out at about 92.5% and slowly decline after that. For the datasets with malware ratios of 0.4 and 0.5, the accuracy already deteriorates from $k = 3$. Overall, we see that the lower the malware ratio in a dataset, the better the accuracy of the

[4] `https://scikit-learn.org/stable/`

KNN model. On datasets with lower malware ratio the model constantly achieves better performance for all tested values of K out of the tested ratios. The optimal value for our model seems to be 3. Especially at the two graphs for datasets with higher malware ratio we see that the accuracy at $k = 4$ drops by about 1% compared to the accuracy at $k = 3$. After that, the accuracy further declines. We will, therefore, train our KNN with $k = 3$. This means the model decides, whether a sample is benign or malicious, based on the three nearest data points to that sample. As the DNN, given an input feature vector, the model **KNN** returns a two dimensional vector $\mathbf{KNN(X)} = [\mathbf{KNN}_0(\mathbf{X}), \mathbf{KNN}_1(\mathbf{X})]$, where $\mathbf{KNN}_0(\mathbf{X}) + \mathbf{KNN}_1(\mathbf{X}) = 1$. The vector describes the probabilities the KNN model beliefs that $\mathbf{X}$ is benign ($\mathbf{KNN}_0(\mathbf{X})$) or malicious ($\mathbf{KNN}_1(\mathbf{X})$).

In this thesis, we will use the model averaging approach. Model averaging is an approach where each model of an ensemble contributes to the final prediction. Typically, each ensemble member contributes an equal amount to the final prediction. After an analysis, it turned out that for our model a weighted average ensemble is more suitable and is able to achieve better performance. In a weighted average ensemble, every individual model does not have an equal contribution to the final prediction but each prediction is weighted. Since the performance of the DNN is generally better than the performance of the KNN, the predictions of the DNN should get a slightly higher weight than the predictions of the KNN model. We chose factor 0.6 for the DNN and factor 0.4 for the KNN. As we will see, the ensemble achieves performances similar to the basis DNN model with these weights. By fine-tuning the parameters, the model might achieve even better performance. Within the framework of this thesis, we did not make an exhaustive analysis because the initially chosen values worked very well. With these values, the model produced meaningful results sufficient for the experiments. Further fine-tuning of the parameters might improve the performance even more. Figure 4.7 shows the structure of the ensemble we use in this thesis. We train each of the models independently on randomly composed subsets of the DREBIN dataset. The combination of malicious and benign samples in each subset is the same as in section 4.1.4. After training each model we evaluate the performance of our ensemble on the same kind of test sets as in section 4.1.5. For the final prediction, both models individually evaluate a sample and output their resulting predictions. These are combined to make a final prediction:

$$\max\left((0.6 \cdot \mathbf{F}_0(\mathbf{X}) + 0.4 \cdot \mathbf{KNN}_0(\mathbf{X})), (0.6 \cdot \mathbf{F}_1(\mathbf{X}) + 0.4 \cdot \mathbf{KNN}_1(\mathbf{X}))\right)$$

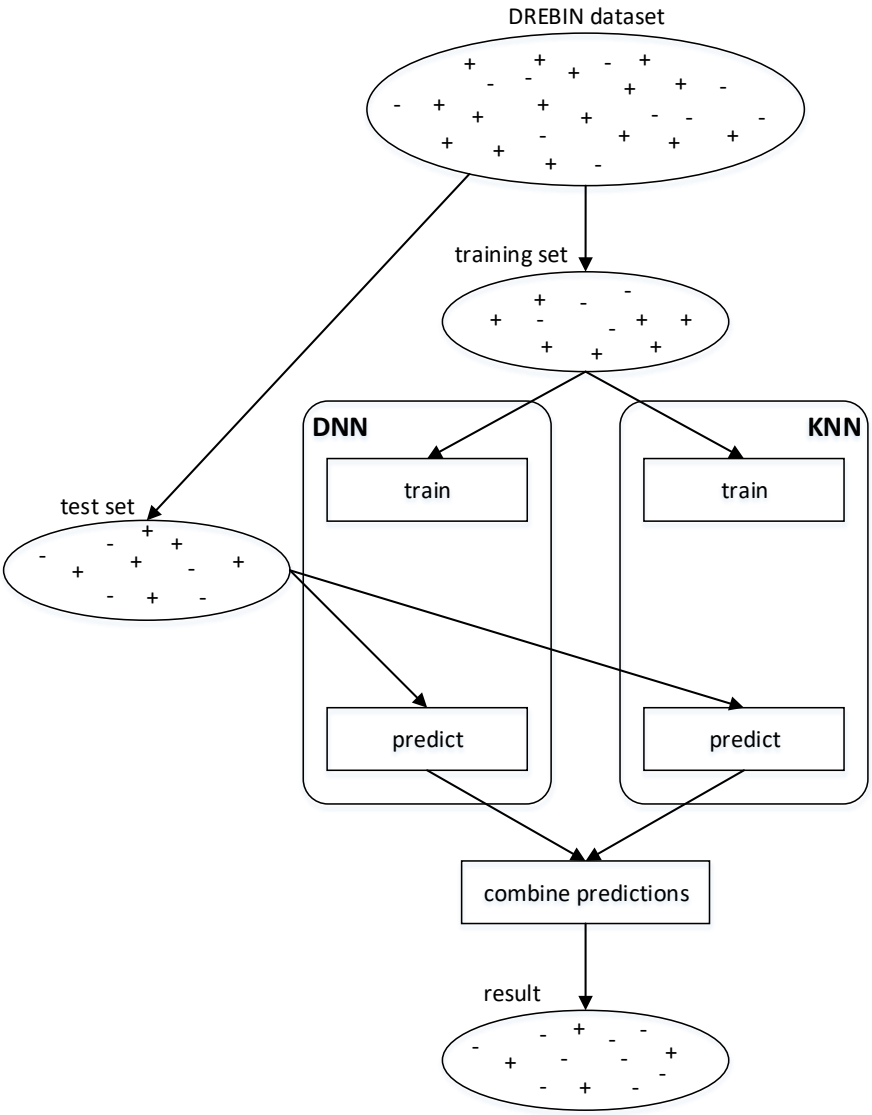


Figure 4.7: Schema of the structure of the model ensemble.

Evaluation

To craft adversarial samples for our ensemble, we use the same algorithm (1) as in the previous sections. This means that each adversarial sample is only based on the DNN alone. Only when evaluating the final classification result, the KNN model is included.

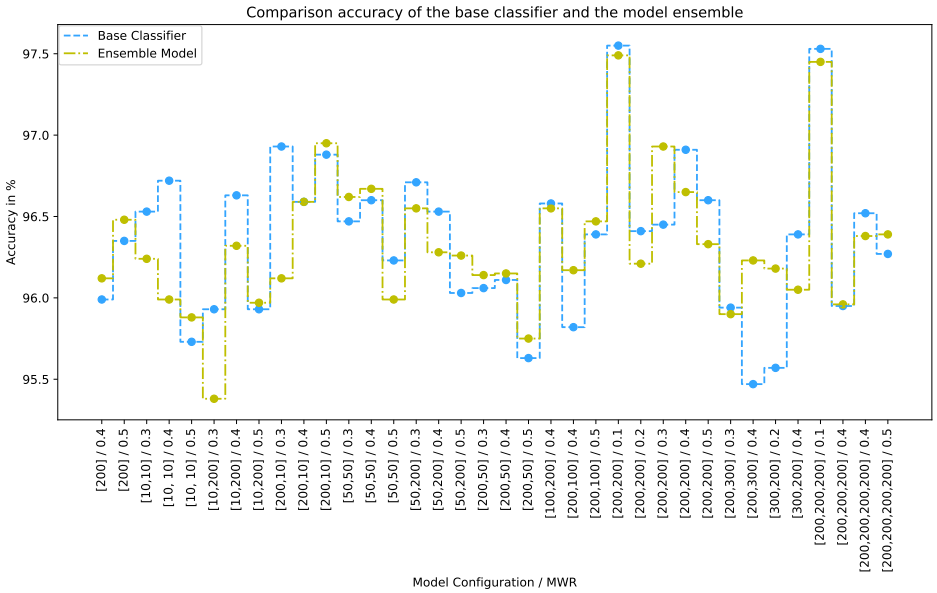


Figure 4.8: Comparison of the accuracy of our base classifier with the accuracy of the model ensemble.

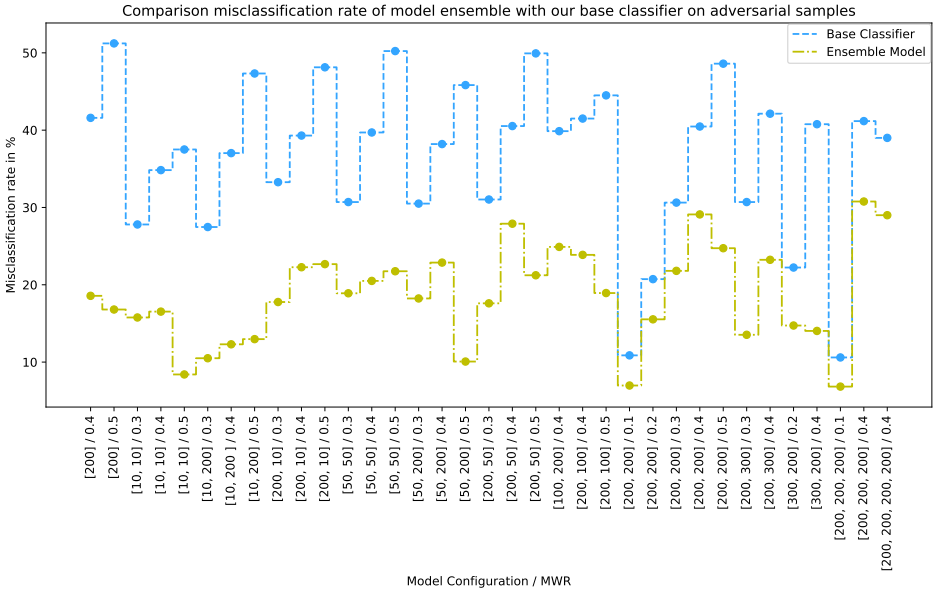


Figure 4.9: Comparison of the performance of our base classifier with the performance of the model ensemble on adversarial samples.

In figure 4.8 we see that the general performance of the ensemble learning is very similar to our base DNN. For most of the tested configurations, the performance of both models is almost the same. Only for a few configurations, the achieved accuracies slightly drift apart. The differences are, however, only marginal. Compared to the feature-reduced model, we can already say that ensemble learning can achieve better performances on regular data. The largest differences between the two models for one configuration is maximally 0.81%. These minor differences probably result from the composition of the training sets and test sets on that the model was trained and evaluated. Different samples within the datasets cause the models to produce slightly different outputs.

Also on adversarial samples ensemble learning does improve the resistance of a malware detector. As figure 4.9 shows, the performance massively improved for every configuration. Overall, we were able to reduce the average misclassification rate of all configurations by nearly 17% compared to the base classifier. The reduction of the misclassification rate is also reflected in the maximum misclassification rate. Compared to the base model - with 51.22% - ensemble learning was able to reduce the maximum misclassification rate by about 12% to 38.93% Since adversarial samples are crafted in the same way as for the base model, the average distortion is comparable to the results of the base model.

These findings are in line with the findings of Papernot et al. [13]. Often, adversarially crafted samples are only able to mislead one of the models, namely the DNN the samples were crafted for. The samples are, however, not able to fool the other part of the ensemble, the KNN model. To successfully cause the whole ensemble to misclassify a sample, the predictions of both individual models are important. The fact that an adversarial sample, which was crafted on a DNN cannot easily be transferred to a KNN model makes an ensemble with a combination of these models so strong against adversarial samples crafted by the proposed algorithm (1).

4.2.3 Combination of Feature Reduction and Ensemble Learning

After we tested both possible countermeasures independently, we now test them in combination. The set up for our ensemble learning stays the same as in the previous section (4.2.2). The only difference is that instead of the entire feature space, we now use the reduced feature space from section 4.2.1 for the input feature vectors.

Evaluation

The performance of the feature-reduced model ensemble shows similar results as the model ensemble from the previous section that was trained on the entire feature space. We constantly achieve a better misclassification rate compared to the misclassification rate of the base model (figure 4.10). On average, we were able to reduce the average misclassification rate from section 4.1 by almost 15%. Even though this is a massive improvement it is still about 2% worse than the model ensemble from the previous section. Figure 4.11 demonstrates the differences between both model ensembles. A vast advantage of the feature-reduced ensemble is, however, its execution time. In our experimental setup (specifications are in Appendix A.1), the model ensemble trained on a reduced feature space needs nearly one second to classify 100 samples. In contrast, the ensemble that was trained on the whole feature space needs about 70 seconds for 100 samples. The minor differences in misclassification rates of the feature-reduced model ensemble, which is on average about 3.2% worse, might, therefore, be negligible. Compared to our base model, the results could be significantly improved.

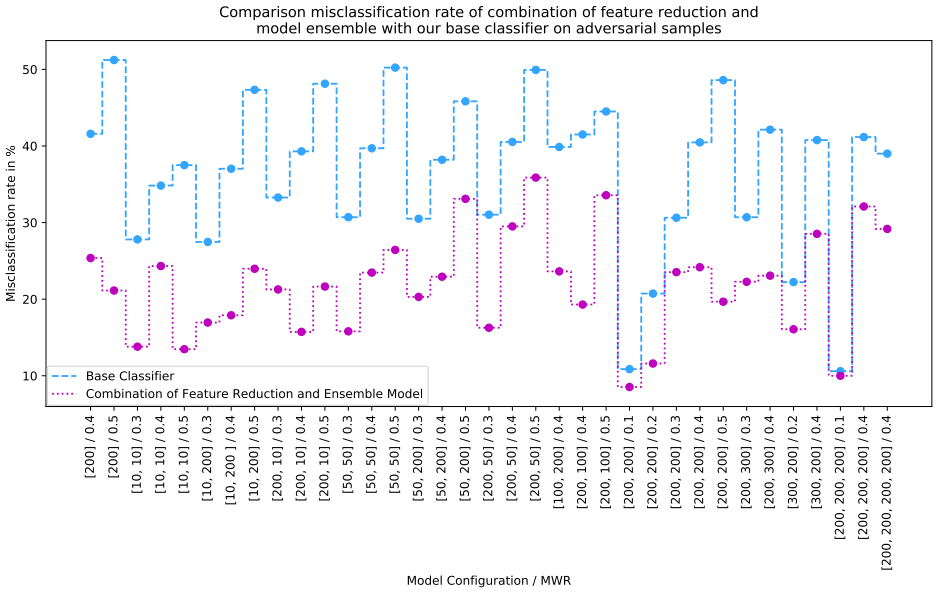


Figure 4.10: Comparison of the performance of our model base model with the combination of feature reduction and model ensemble on adversarial samples.

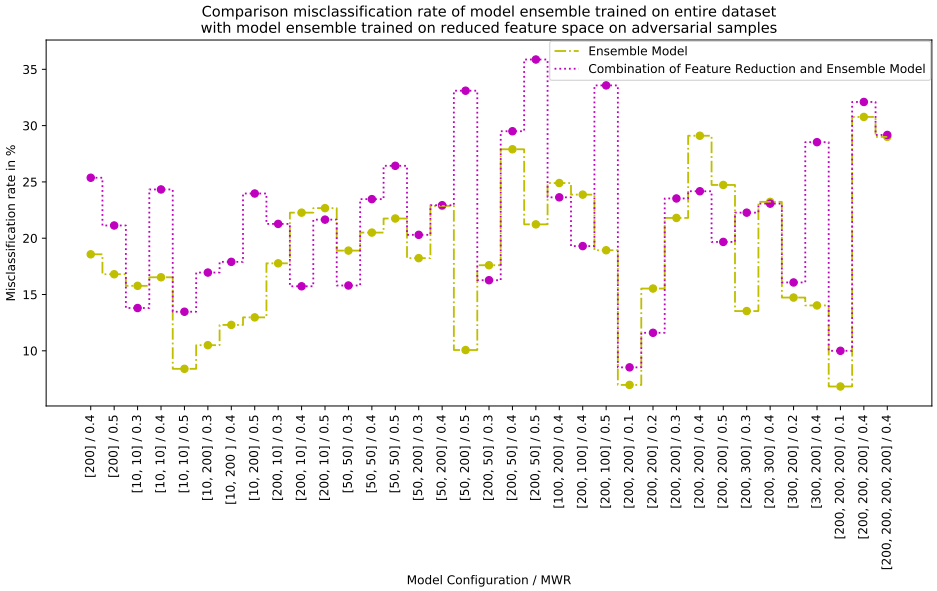


Figure 4.11: Comparison of the performance of both our model ensembles on adversarial samples.

4.2.4 Defensive Distillation

Distillation originally was proposed to deploy bulky machine learning models that can perform complex tasks on smaller devices, like mobile devices. One possibility is to train an ensemble of multiple small networks that can run on devices with limited computational resources. This approach is called *knowledge distillation* and can improve the performance of machine learning models on small devices. Instead of using an ensemble of models, distillation can be used for model compression. Model compression makes it possible to compress the knowledge of an ensemble into one single model [22]. Beyond these two techniques of knowledge distillation and model compression, distillation can be used as a defense mechanism against adversarial crafted inputs [21][1].

Defensive distillation can, in some cases, dramatically improve the performance of DNNs on adversarial crafted samples. Besides that, defensive distillation increases the average minimum distortion, so the number of features that need to be modified to successfully create an adversarial sample [21].

The process of distillation, as it is described in [1], consists of three major steps. First, they craft a new training dataset. Additionally, to the features of applications, the new dataset does contain the output $\mathbf{F(X)}$ of the base model. Second, they build a new DNN **F'** with the same architecture as the base classifier **F**. The aim of the network **F'** is to achieve at least the same

performance than network **F** on the same dataset. Third, they train **F'** on the dataset that was created in the first step. By using the results of the base model (**F**) to train the second model **F'**, the second model's generalization performance improves[21]. This means that the model's performance improves on new data outside the training set.

As their results show, Grosse et al. [1] were able to strengthen their neural network against adversarial samples by using distillation. On the other side, however, although the misclassification rates drop significantly compared to the base model, they are still around 40% and the general performance on regular - with no adversarial crafted samples - data worsens. In comparison to the base model, distillation causes a strong increase in the false positive rate and a slight increase in the false negative rate.

4.2.5 Re-Training

Finally, we take a look at re-training a classifier with adversarial samples [11][1]. As we have seen in section 4.1.1, typically there are one or more datasets on which a classifier gets trained. These datasets consist of representative samples of the target domain within which the machine learning model will operate. During re-training, we not only train a model on the pure dataset (as we did earlier) but use additional training epochs on adversarial samples. Applying adversarial crafted samples already during training aims at improving the generalization of the model and making it less sensitive to small perturbations in the input data. Since adversarial samples take advantage of exactly those small perturbations, re-training aims to make a model more robust against adversarial inputs outside the training set.

Grosse et al. [1] demonstrated that re-training on adversarial malware samples does improve the performance of a DNN on adversarial input data. On several datasets, re-training was able to reduce the misclassification rate and increase the required average distortion. The evaluation shows that the misclassification rate of a re-trained model persists very similar to the misclassification rate of the original model. They argue that a network needs a higher ratio of adversarial samples to properly generalize and improve the resistance of the network against adversarial samples.

Chapter 5

Discussion

We examined five different defense mechanisms and studied the effectiveness of these methods on adversarial samples. During the first method, feature reduction, we considerably reduced the dimensionality of feature vectors by including only the most important features, according to their support. Instead of making it harder for attackers to successfully craft adversarial samples, feature reduction generally weakens the neural networks' resistance. The reduced amount of feature causes a single feature to have more weight. It follows that every feature has a larger impact on the result of the machine learning model, which makes it easier for an attacker to find a suitable feature for crafting adversarial samples. Feature reduction is therefore not a recommendable defensive measure against adversarial inputs.

Ensemble learning, in contrast, is able to strengthen a DNN on adversarial samples. The analysis of both ensembles, trained on the total feature space as well as trained on a reduced range of features, show that they can facilitate a neural network to make it more resistant to adversarial inputs. Compared to the performance of the base model, with ensemble learning, we were able to decrease the misclassification rate by 35% and 30% respectively. The model ensemble performs on average slightly worse on reduced feature vectors compared to the performance on the complete feature space. However, if we consider the differences in execution time, this loss in accuracy might be tolerable. With half a second (0.56s) per 100 samples, the feature-reduced ensemble is about 120 times faster than the ensemble that uses all available features. This needs around 70 seconds to process 100 samples. Of course, this time highly depends on the environment in that the model is executed and especially for KNN it depends on the number of samples on that the model was trained. The higher the number of training samples, the more data points KNN needs to compare to determine the k closest points. In consideration of such a large difference, however, feature reduction might be a decent option. Overall, the ensembles benefit from the fact that a KNN model is less vulnerable to adversarial samples crafted for

DNNs. Another factor that benefits the studied ensemble is that we craft adversarial samples only on the neural network. Other studies propose a method by which adversarial samples for KNNs are crafted using the fast gradient sign method [21][9][23]. The fast gradient sign method slightly differs from the method we used in this research. Despite these differences, ensemble learning shows promising results. Future research will have to investigate ensemble learning under different crafting algorithms that involve both models during the process of crafting.

Furthermore, we investigated two methods that were already suggested in earlier research. First, we took a look at distillation. Distillation was able to reduce misclassification rates up to 38.5% compared to their base model. This drop is in the same range our ensemble learning achieved. A drawback of distillation is, however, that the general performance of the classifier degrades. With distillation, model accuracy ranges in between $93\% - 95\%$. Without distillation, the model achieves performances ranging from 96% to 98%.

Secondly, we considered re-training. Grosse et al. [1] showed that re-training can improve resistance under the right circumstances. The performance of the neural network highly depends on the number of adversarial samples in an input dataset and the training parameters they chose for training their model. On datasets with malware ratio 0.3 and 0.4 misclassification reduces. Further increase of adversarial samples causes the misclassification rate to increase again. With at least roughly 61%, however, the misclassification rate was still rather high.

All in all, we can say that from the defensive measures, we tested in this thesis, ensemble learning, distillation, and adversarial re-training show promising results. All three methods consistently achieve reduced misclassification rates across different model inputs and architectures. Only feature reduction is not a suitable countermeasure for adversarial inputs. Instead of making a model more robust, it weakens a model against adversarial crafted samples.

Chapter 6

Conclusions and Future Research

In this thesis, we evaluated five potential defense mechanisms to increase the robustness of deep neural network based Android malware detectors against adversarial inputs. The experiments provided us with the following results: First, feature reduction does not only make a neural network more vulnerable to adversarial inputs, but it also makes crafting adversarial samples easier. An attacker needs to make fewer changes to a given malware to successfully create an adversarial sample and mislead a malware detector. Second, ensemble learning does improve a model's performance on adversarial inputs. Third, distillation also reduces misclassification rates to a certain extent. Fourth, adversarial re-training can strengthen a model on adversarial crafted inputs. Performance thereby highly depends on the selection of training parameters and the compilation of input datasets. Nevertheless, the experiments also show that none of the studied methods was able to completely diminish the threat of adversarial crafted samples. Adversarial samples on machine learning models remains a topic that requires further research.

Further research should investigate ensemble learning in more detail. The effectiveness of the method should be tested against a modified algorithm, which for example, tries to maximize the change into the direction of the target class. Further, it should be investigated whether the same results can be achieved on a crafting algorithm that takes both parts of an ensemble into account.

Additionally, the defensive measures proposed in this thesis could be tested in a black-box environment as introduced by Papernot et al. [9]. This shows the effectiveness of each method in a more realistic scenario because mostly an attacker does not have entire knowledge about a malware detector.

Bibliography

[1] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," *CoRR*, vol. abs/1606.04435, 2016.

[2] "Smartphone market share." `https://www.idc.com/promo/smartphone-market-share/os`. Accessed: 2019-06-16.

[3] "Android security & privacy 2018 year in review." `https://g.co/androidsecurityreport2018`. Accessed: 2019-05-10.

[4] "Number of available applications in the google play store from december 2009 to march 2019." `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`. Accessed: 2019-06-14.

[5] "Schad-apps: Android-bedrohungen im jahr 2018 so hoch wie nie." `https://www.gdata.de/blog/2019/02/31534-schad-apps-2018-mit-neuem-malware-rekord-fur-android`. Accessed: 2019-06-14.

[6] "Cyber attacks on android devices on the rise." `https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise`. Accessed: 2019-06-14.

[7] "Application fundamentals." `https://developer.android.com/guide/components/fundamentals`. Accessed: 2019-05-07.

[8] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," *CoRR*, vol. abs/1511.07528, 2015.

[9] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against deep learning systems using adversarial examples," *CoRR*, vol. abs/1602.02697, 2016.

[10] R. Zachariah, K. Akash, M. S. Yousef, and A. M. Chacko, "Android malware detection a survey," in *2017 IEEE International Conference on Circuits and Systems (ICCS)*, pp. 238–244, Dec 2017.

[11] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2014.

[12] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," *arXiv e-prints*, p. arXiv:1412.6572, Dec. 2014.

[13] N. Papernot, P. D. McDaniel, and I. J. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *CoRR*, vol. abs/1605.07277, 2016.

[14] 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," *CoRR*, vol. abs/1704.08996, 2017.

[15] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, pp. 83–97, Jan 2018.

[16] Z. Xiong, T. Guo, Q. Zhang, Y. Cheng, and K. Xu, "Android malware detection methods based on the combination of clustering and classification," in *Network and System Security* (M. H. Au, S. M. Yiu, J. Li, X. Luo, C. Wang, A. Castiglione, and K. Kluczniak, eds.), (Cham), pp. 411–422, Springer International Publishing, 2018.

[17] V. Moonsamy, J. Rong, S. Liu, G. Li, and L. Batten, "Contrasting permission patterns between clean and malicious android applications," in *Security and Privacy in Communication Networks* (T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, eds.), (Cham), pp. 69–85, Springer International Publishing, 2013.

[18] Y. Abouelnaga, O. S. Ali, H. Rady, and M. Moustafa, "Cifar-10: Knn-based ensemble of classifiers," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 1192–1195, Dec 2016.

[19] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of android malware in your pocket," *21th Annual Network and Distributed System Security Symposium (NDSS)*, February 2014.

[20] M. Spreitzenbarth, F. Echtler, T. Schreck, F. C. Freling, and J. Hoffmann, "Mobilesandbox: Looking deeper into android applications,"

28th International ACM Symposium on Applied Computing (SAC), March 2013.

[21] N. Papernot, P. D. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," *CoRR*, vol. abs/1511.04508, 2015.

[22] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *arXiv e-prints*, p. arXiv:1503.02531, Mar 2015.

[23] D. Warde-Farley and I. Goodfellow, "Adversarial perturbations of deep neural networks," 2016.

Appendix A

Appendix

Category	Item	Description
Hardware	CPU	Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
	Memory	32 GB
Software	OS	SMP Debian 4.9.130-2 (2018-10-27) x86_64 GNU/Linux
	Python	3.5.3
	Tensorflow	1.13.1
	Numpy	1.16.2
	scikit-learn	0.20.2

Table A.1: **Specifications of the experiment environment.**

A.1 Performance Feature-Reduced DNN

Configuration	MWR	Accuracy	FNR	FPR
[200]	0.4	95.17	4.93	4.76
[200]	0.5	94.75	3.67	6.84
[10, 10]	0.3	94.31	14.78	1.80
[10, 10]	0.4	93.64	9.35	4.37
[10, 10]	0.5	92.87	8.40	3.75
[10, 200]	0.3	94.48	6.22	5.22
[10, 200]	0.4	94.78	7.50	3.70
[10, 200]	0.5	94.97	6.55	3.51
[200, 10]	0.3	95.37	10.69	4.61
[200, 10]	0.4	95.56	6.25	3.23
[200, 10]	0.5	95.27	5.28	4.17
[50, 50]	0.3	94.59	15.98	0.89
[50, 50]	0.4	92.49	17.37	0.94
[50, 50]	0.5	94.63	3.35	7.39
[50, 200]	0.3	95.63	10.86	1.59
[50, 200]	0.4	93.87	13.03	1.52
[50, 200]	0.5	94.71	4.01	6.56
[200, 50]	0.3	95.53	8.49	2.74
[200, 50]	0.4	94.05	5.63	6.17
[200, 50]	0.5	95.20	7.48	2.12
[100, 200]	0.4	95.05	5.47	4.61
[200, 100]	0.4	95.51	4.67	4.36
[200, 100]	0.5	95.57	3.92	4.93
[200, 200]	0.1	96.99	27.4	0.30
[200, 200]	0.2	96.08	12.23	1.84
[200, 200]	0.3	95.73	9.98	1.83
[200, 200]	0.4	95.33	7.33	2.89
[200, 200]	0.5	95.78	5.63	2.81
[200, 300]	0.3	94.77	10.00	3.19
[200, 300]	0.4	94.07	10.08	3.15
[300, 200]	0.2	95.46	17.60	1.28
[300, 200]	0.4	95.59	6.22	3.20
[200, 200, 200]	0.1	95.97	10.60	3.31
[200, 200, 200]	0.4	95.38	8.73	1.88
[200, 200, 200, 200]	0.4	95.11	8.60	2.42
[200, 200, 200, 200]	0.5	94.53	4.32	6.61

Table A.2: **Performance of the DNN on a feature-reduced dataset.** Given are used malware ratio (MWR), accuracy, false negative rate (FNR), and false positive rate (FPR)

Configuration	MWR	MR	Distortion
[200]	0.4	43.10	4.90
[200]	0.5	53.20	5.54
[10, 10]	0.3	36.46	5.05
[10, 10]	0.4	41.91	5.63
[10, 10]	0.5	50.57	7.85
[10, 200]	0.3	31.70	7.38
[10, 200]	0.4	39.90	8.19
[10, 200]	0.5	51.73	5.51
[200, 10]	0.3	31.17	4.59
[200, 10]	0.4	41.10	7.78
[200, 10]	0.5	27.95	7.06
[50, 50]	0.3	30.77	4.43
[50, 50]	0.4	40.60	4.32
[50, 50]	0.5	52.83	6.87
[50, 200]	0.3	30.93	4.71
[50, 200]	0.4	40.70	5.05
[50, 200]	0.5	53.40	5.49
[200, 50]	0.3	31.37	5.14
[200, 50]	0.4	43.70	6.20
[200, 50]	0.5	50.03	6.97
[100, 200]	0.4	42.10	4.84
[200, 100]	0.4	41.97	6.30
[200, 100]	0.5	52.37	5.48
[200, 200]	0.1	13.97	3.69
[200, 200]	0.2	21.63	4.93
[200, 200]	0.3	31.27	4.13
[200, 200]	0.4	41.73	5.76
[200, 200]	0.5	51.47	5.93
[200, 300]	0.3	32.83	5.06
[200, 300]	0.4	41.33	4.73
[300, 200]	0.2	20.43	3.44
[300, 200]	0.4	42.23	5.04
[200, 200, 200]	0.1	12.77	5.25
[200, 200, 200]	0.4	41.13	4.93
[200, 200, 200, 200]	0.4	41.70	5.00

Table A.3: **Performance of the DNN that was trained on a feature-reduced dataset on adversarial inputs.** Given are the different configurations of our malware detector, malware ratio (MWR), misclassification rates (MR), and required average distortion (in number of added features)

A.2 Performance Model Ensemble

Configuration	MWR	Accuracy	FNR	FPR
[200]	0.4	96.12	4.72	3.32
[200]	0.5	96.48	4.27	2.77
[10, 10]	0.3	96.24	8.18	1.87
[10, 10]	0.4	95.99	5.77	2.84
[10, 10]	0.5	95.88	5.24	3.00
[10, 200]	0.3	95.38	11.08	7.15
[10, 200]	0.4	96.32	7.08	1.41
[10, 200]	0.5	95.97	6.09	1.97
[200, 10]	0.3	96.12	3.89	3.88
[200, 10]	0.4	96.59	6.42	1.40
[200, 10]	0.5	96.95	3.92	2.17
[50, 50]	0.3	96.62	6.38	2.09
[50, 50]	0.4	96.67	4.07	2.85
[50, 50]	0.5	95.99	5.01	3.00
[50, 200]	0.3	96.55	9.11	1.03
[50, 200]	0.4	96.28	4.43	3.24
[50, 200]	0.5	96.26	4.15	3.33
[200, 50]	0.3	96.14	8.02	2.08
[200, 50]	0.4	96.15	6.75	1.91
[200, 50]	0.5	95.75	7.48	1.03
[100, 200]	0.4	96.55	5.80	1.89
[200, 100]	0.4	96.17	6.12	2.30
[200, 100]	0.5	96.47	3.13	3.93
[200, 200]	0.1	97.49	19.4	0.64
[200, 200]	0.2	96.21	16.2	0.68
[200, 200]	0.3	96.93	8.31	1.30
[200, 200]	0.4	96.65	6.03	1.55
[200, 200]	0.5	96.33	4.23	3.11
[200, 300]	0.3	95.90	11.27	1.03
[200, 300]	0.4	96.23	3.80	3.75
[300, 200]	0.2	96.18	8.37	2.68
[300, 200]	0.4	96.05	6.73	2.09
[200, 200, 200]	0.1	97.45	20.47	0.56
[200, 200, 200]	0.4	95.96	5.25	3.22
[200, 200, 200, 200]	0.4	96.38	5.80	2.17
[200, 200, 200, 200]	0.5	96.39	5.24	1.97

Table A.4: **Performance of the model ensemble.** Given are used malware ratio (MWR), accuracy, false negative rate (FNR), and false positive rate (FPR)

Configuration	MWR	MR	Distortion
[200]	0.4	18.57	6.65
[200]	0.5	16.80	6.26
[10, 10]	0.3	15.77	10.19
[10, 10]	0.4	16.53	10.62
[10, 10]	0.5	8.40	14.23
[10, 200]	0.3	10.50	11.21
[10, 200]	0.4	12.30	9.35
[10, 200]	0.5	12.97	10.13
[200, 10]	0.3	17.77	10.61
[200, 10]	0.4	22.27	8.20
[200, 10]	0.5	22.67	9.48
[50, 50]	0.3	18.90	8.31
[50, 50]	0.4	20.50	8.66
[50, 50]	0.5	21.75	8.07
[50, 200]	0.3	18.23	5.66
[50, 200]	0.4	22.87	10.16
[50, 200]	0.5	10.07	10.28
[200, 50]	0.3	17.60	8.07
[200, 50]	0.4	27.90	7.63
[200, 50]	0.5	21.23	6.55
[100, 200]	0.4	24.90	7.86
[200, 100]	0.4	23.87	6.62
[200, 100]	0.5	18.93	11.63
[200, 200]	0.1	6.97	4.66
[200, 200]	0.2	15.53	5.51
[200, 200]	0.3	21.80	6.48
[200, 200]	0.4	29.10	6.23
[200, 200]	0.5	24.73	8.30
[200, 300]	0.3	13.53	5.15
[200, 300]	0.4	23.23	8.58
[300, 200]	0.2	14.73	7.56
[300, 200]	0.4	14.03	7.14
[200, 200, 200]	0.1	6.83	6.15
[200, 200, 200]	0.4	30.77	8.81
[200, 200, 200, 200]	0.4	29.00	9.04

Table A.5: **Performance of the model ensemble on adversarial inputs.** Given are the different configurations of our malware detector, malware ratio (MWR), misclassification rates (MR), and required average distortion (in number of added features)

A.3 Performance Combination of Feature Reduction and Model Ensemble

Configuration	MWR	Accuracy	FNR	FPR
[200]	0.4	95.67	5.83	3.32
[200]	0.5	95.07	4.67	5.20
[10, 10]	0.3	94.95	13.76	1.33
[10, 10]	0.4	94.77	9.12	2.65
[10, 10]	0.5	94.57	8.36	2.51
[10, 200]	0.3	95.26	6.72	4.86
[10, 200]	0.4	94.92	8.43	2.85
[10, 200]	0.5	95.17	5.03	4.64
[200, 10]	0.3	95.89	10.53	1.35
[200, 10]	0.4	95.40	7.27	2.82
[200, 10]	0.5	94.99	6.23	3.79
[50, 50]	0.3	94.79	15.22	0.91
[50, 50]	0.4	93.37	15.35	0.82
[50, 50]	0.5	95.27	3.89	5.56
[50, 200]	0.3	95.77	10.60	1.50
[50, 200]	0.4	93.91	13.35	1.25
[50, 200]	0.5	95.05	5.97	3.92
[200, 50]	0.3	95.83	9.55	1.86
[200, 50]	0.4	95.32	5.00	4.47
[200, 50]	0.5	95.23	7.43	2.11
[100, 200]	0.4	95.81	6.00	2.98
[200, 100]	0.4	95.73	5.23	3.08
[200, 100]	0.5	95.69	4.45	4.16
[200, 200]	0.1	96.85	28.73	0.31
[200, 200]	0.2	95.99	14.3	1.43
[200, 200]	0.3	95.79	9.89	1.77
[200, 200]	0.4	95.53	7.23	2.62
[200, 200]	0.5	95.84	5.71	2.61
[200, 300]	0.3	95.23	10.47	2.32
[200, 300]	0.4	94.67	9.77	2.38
[300, 200]	0.2	95.47	18.73	0.98
[300, 200]	0.4	95.46	6.97	2.92
[200, 200, 200]	0.1	96.98	13.07	1.90
[200, 200, 200]	0.4	95.39	8.62	1.93
[200, 200, 200, 200]	0.4	95.61	8.48	1.66
[200, 200, 200, 200]	0.5	94.87	4.79	5.48

Table A.6: **Performance of the combination of feature reduction and model ensemble.** Given are used malware ratio (MWR), accuracy, false negative rate (FNR), and false positive rate (FPR)

Configuration	MWR	MR	Distortion
[200]	0.4	25.37	4.90
[200]	0.5	21.13	5.39
[10, 10]	0.3	13.80	5.00
[10, 10]	0.4	24.33	5.78
[10, 10]	0.5	13.47	8.00
[10, 200]	0.3	16.95	7.41
[10, 200]	0.4	17.90	8.01
[10, 200]	0.5	23.97	5.27
[200, 10]	0.3	21.27	4.58
[200, 10]	0.4	15.73	7.55
[200, 10]	0.5	21.65	14.64
[50, 50]	0.3	15.80	4.42
[50, 50]	0.4	23.47	4.32
[50, 50]	0.5	26.43	6.62
[50, 200]	0.3	20.30	4.69
[50, 200]	0.4	22.93	5.12
[50, 200]	0.5	33.10	5.57
[200, 50]	0.3	16.27	5.07
[200, 50]	0.4	29.50	6.02
[200, 50]	0.5	35.87	6.91
[100, 200]	0.4	23.63	5.90
[200, 100]	0.4	19.30	6.17
[200, 100]	0.5	33.57	5.40
[200, 200]	0.1	8.53	3.17
[200, 200]	0.2	11.60	4.94
[200, 200]	0.3	23.53	4.06
[200, 200]	0.4	24.17	5.70
[200, 200]	0.5	19.67	6.26
[200, 300]	0.3	22.27	5.03
[200, 300]	0.4	23.07	4.69
[300, 200]	0.2	16.07	3.62
[300, 200]	0.4	28.53	4.95
[200, 200, 200]	0.1	10.00	4.95
[200, 200, 200]	0.4	32.10	4.89
[200, 200, 200, 200]	0.4	29.17	5.05

Table A.7: **Performance of the combination of feature reduction and model ensemble on adversarial inputs.** Given are the different configurations of our malware detector, malware ratio (MWR), misclassification rates (MR), and required average distortion (in number of added features)