Network Intrusion Detection in an Adversarial Setting

Report submitted in fulfillment of the requirements
for the B.Tech Project of

Third Year B.Tech.

by

Shreyansh Singh, 16075052

Under the guidance of

Prof. K.K. Shukla

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY (BHU) VARANASI
Varanasi 221005, India
May 2019

Dedicated to

My parents, teachers

Declaration

We certify that

1. The work contained in this report is original and has been done by our team and the general supervision of my supervisor.
2. The work has not been submitted for any project.
3. Whenever we have used materials (data, theoretical analysis, results) from other sources, we have given due credit to them by citing them in the text of the thesis and giving their details in the references.
4. Whenever we have quoted written materials from other sources, we have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Place: IIT (BHU) Varanasi
Date:

Shreyansh Singh, B.Tech.
Department of Computer Science and Engineering,
Indian Institute of Technology (BHU) Varanasi,
Varanasi, INDIA 221005.

Certificate

*This is to certify that the work contained in this report entitled "**Network Intrusion Dtection in an Adversarial Setting**" being submitted by **Shreyansh Singh (Roll No. 16075052)** and carried out in the Department of Computer Science and Engineering, Indian Institute of Technology (BHU) Varanasi, is a bona fide work of my supervision.*

Place: IIT (BHU) Varanasi
Date:

Prof. K.K. Shukla
Department of Computer Science and Engineering,
Indian Institute of Technology (BHU) Varanasi,
Varanasi, INDIA 221005.

Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and our institute. We would like to extend our sincere thanks to all of them. We are highly indebted to our project supervisor Prof. K.K. Shukla for his guidance and constant supervision as well as for providing necessary information regarding the project and also for his support in completing the project.

Place: IIT (BHU) Varanasi
Date: **Shreyansh Singh**

Contents

List of Figures

List of Tables

Abstract

The origin of network-intrusion detection research was using signature-based detection approaches. But since the gain in the popularity of machine learning and anomaly detection techniques based on it, the domain of intrusion detection has also seen such techniques being used. The volume of attacks has been increasing everyday and the techniques adversaries are using for crafting the attacks is also evolving at a very fast pace. Hence with the improvements in big data analytics area, makes machine learning the go-to technique to solve the issue. The aim of this research is to show that although machine learning can prove to be a great tool for network intrusion detection, but the robustness of the classifiers should be evaluated before using such models in production.

The domain of Image classification has seen several adversarial techniques emerge from deep learning research. The main idea in such techniques is to make minor changes in the original input data that is not recognizable by humans but are enough to make a machine learning tool misclassify it. This research explores adversarial machine learning techniques that have emerged from the deep learning domain, against machine learning classifiers used for network intrusion detection.

In this study, we look at the well known and commonly used classifiers and study their performance under attack. The metrics we used are accuracy, F1-score and receiver operating characteristic (ROC). The approach used assumes no knowledge of the original classifier and examines targeted misclassification. Even using very simple methods for generating adversarial examples, we show that it is possible to lower the accuracy of intrusion detection classifiers from 2.5% to 32%. This is achieved by introducing a very small change (9.49% on average) in the original sample to create the adversarial sample, which makes it a candidate for practical adversarial attacks.

Introduction

Although enterprise networks aim to deploy the best security measures, security breaches still remain a source of major concern. Malicious activities within a network can be categorized based on the origin of the attacker as:

- *External users*: These include the activities that are performed by external users and have the intention to get access to the internal network. Such activities could be successfully performed via a breach in the network perimeter using malware, social engineering, phishing attacks and so on. Getting inside the internal network is very dangerous as now it is difficult to distinguish the attacker from normal users since most often they use normal user or administrator credentials. The attackers can use such kind of access to install and run malicious software autonomously like bots, or install backdoors or rootkits on the system of the employees.

- *Internal users*: This is also known as "insider threat". These include the activities that are performed by internal users and have the motive to misuse, attack or steal information.

Intrusion Detection

Intrusion detection is dealing with unwanted access to systems and information by any type of user or software. An intrusion detection system (IDS) is a device or software application that monitors a network or systems for malicious activity or policy violations. An Intrusion Prevention System (IPS) is an IDS which also has the ability to stop attacks. There are two major categories of IDS:

- **Network IDS**, which monitors network segments and analyzes network traffic at different layers in order to detect intruders.

- **Host based IDS**, which are installed in host machines and analyze different indicators such as processes, log files, unexpected changes in the host to determine the presence of malicious activities.

Handling (gathering, storing and processing) the amount of network traffic that is generated on a daily basis by large enterprise networks, is a difficult task. One way to deal with this is to discard parts of the data or log less information, however the emergence of Big Data Analytics

(BDA) as well as the improvement in memory, computing power and the decrease in storage costs, transforms the situation into a big data problem.

Regardless of the specific data set used for Intrusion detection analysis, the nature of the data associated with this class of problems exhibits certain general characteristics:

- Data is generated constantly and there is a time series nature (continuous or discrete) based on the data set and the processing approach.
- *Class imbalance*, i.e. Very few positive labels or lack of labels
- Attack types change a lot over time, with attackers developing novel methods all the time.
- Variety in the type of data: packets, flows, numerical, unstructured text (URLs) and so on.

Traditional approaches in the area of intrusion detection mainly revolve around signature and rule based approaches. The limitations to those is that they work only with known attack patterns and that they require extensive domain knowledge [Chuvakin 2012]. Anomaly detection techniques based on statistical or machine learning approaches promise more flexibility and less dependency in domain knowledge and are more scalable when it comes to big data.

Motivation of the Research Work

Network Intrusion Detection systems play an important role in ensuring the security of the networks of large enterprise systems or critical infrastructures. The aim of this research is to show how Adversarial Machine learning can be used to trick classifiers into misclassifying malicious network packets as legitimate ones. Most of the current work in Adversarial Machine Learning has been done for images like [Szegedy 2013b], [Goodfellow 2014b], [Nguyen 2015b], so our aim is to use the same attack techniques for Network Intrusion Detection systems as well.

Organization of the Report

The organization of the report is as follows:
Chapter 1 gives a description of the past work that has been done in the domain of adversarial machine learning and the research papers we have gone through as a prerequisite for our study. It also gives a description of the attack techniques that we will be using in our study.
Chapter 2 focuses on the dataset we have used and the preprocessing steps.
Chapter 3 provides the implementation details.
Chapter 4 discusses the results we obtained for every step in our study.
Chapter 5 gives an analysis of our results at every step.
Finally, we conclude our report in Chapter 6, and specify our future work.

Chapter 1

Literature Review

1.1 Introduction

We followed the framework proposed by [Vom Brocke 2009] for the literature review process. The first step involved defining the scope and creating a rough outline of the task to perform. This was followed by thorough literature survey and subsequent analysis of the work already done this field. Following these steps helped us to identify the research gaps that existed, and helped to formulate the research questions that we will attempt to answer through our work.

The literature review was conducted using exhaustive search over the following terms: "information security AND machine learning", "machine learning AND IDS", "anomaly detection AND IDS" and "adversarial machine learning", "deep learning AND IDS". Apart from keyword search and relevance, other selection criteria were the chronology of the papers and the quality of sources (peer reviewed journals and conferences).

The search engines utilized for this search were mainly the LTU library search and Google scholar search engines which aggregate results over a number of databases. The majority of the references comes from well known databases such as ACM, IEEE, Springer and Elsevier.

1.2 Intrusion Detection

The methodologies used in NIDS can be divided into various categories. The vast majority of the literature describe the following categorizations:

- **Misuse-based** or **signature based**: These types of systems perform simple signature matching using signatures or indicators extracted from previously known attacks. The problem with these types of systems is that they don't perform well when they encounter new types of attacks and the task of maintaining the signatures given the rising number of attacks today, is also difficult.

- **Anomaly based**: These types of systems try to model normal behavior in a network in contrast to what is anomalous and potentially malicious. These are better than signature based systems because of the fact that they are better at adapting to new attacks. but a major concern with such systems is whether the system “learns” a good definition of what is anomalous or not. The system should be able to classify malicious behaviour as anomalous.

- **Hybrid systems**: These types of systems are the combination of the above approaches.

In many of the research papers we surveyed, we find that the terms “machine learning” and “anomaly detection” are used interchangeably. [Bhuyan 2014] make a broader presentation that includes not only classifiers such as the ones used in Machine Learning and Data Mining but also pure anomaly detection techniques which include statistical methods, clustering and outlier based methods, knowledge based methods and combination learners.

Anomaly Detection based systems promise to solve the issue of adaptation to new attacks, however, the problem of generalization still exists, which makes it difficult to prove whether they can be used widely in practice. [Sommer 2010a] present some challenges that are relevant even today. These challenges include:

- the data used to train the models is very unbalanced, which makes it difficult to apply unsupervised classification techniques,
- High False Positive rate (FPR) can become a problem because that would result in a large number of alarms to be analyzed that are generated by the NIDS, therefore, time and fatigue can be a problem
- interpretation of the results and taking action is not always possible with some ML techniques,
- the lack of high quality representative datasets can lead to problems in the evaluation of different approaches.

[Milenkoski 2015] identified four major categories of evaluation metrics that are used in majority of the studies:

- Attack detection accuracy with most common metrics the False Positive, False Negative, True Positive and True Negative rates, the Positive Predictive Value (PPV) or Precision and the Negative Predictive Value (NPV). The False Positive Rate (FPR) and the True Positive Rate (TPR) are used in the construction of Receiver Operating Characteristic (ROC) curves and the calculation of the Area Under the Curve (AUC).
- Performance overhead which the IDS is adding to the overall network environment.
- Attack coverage, which is the detection accuracy of the IDS without benign traffic.
- Workload processing, which is the amount of traffic that can be processed by an IDS vs. the amount of network traffic the IDS discards.

1.2.1 IDS Datasets

One of the most used dataset is KDD'99 [KDD] which was derived from the DARPA'98 dataset. The dataset was used in a competition that was held during the Fifth International Conference on Knowledge Discovery and Data Mining and the main competition task was to create a predictive model that can be used in network intrusion detection. The KDD'99 dataset had some problems which were analyzed by researchers such as [McHugh 2000], [Sommer 2010b], [Brugger 2007], [Tavallaee 2009a]. The major issues were -

- There is a huge number of redundant records for about 78% and 75% are duplicated in the train and test set, respectively.
- This redundancy makes the machine learning training quite biased.

This led to the creation of an improved version of the KDD'99 dataset, which was called the NSL-KDD dataset [NSL-KDD] by [Tavallaee 2009a]. This dataset did not solve all the problems in the KDD'99 dataset and more importantly it did not erase the fact that it is quite outdated. Still, the NSL-KDD dataset has been used in many Network Intrusion Detection based works due to the lack of public datasets for network-based IDSs. It provides a good analysis on various machine learning techniques for intrusion detection. The advantages of using this dataset are -

- No redundant records in the train set, so the classifier will not produce any biased results
- No duplicate record in the test set which have better reduction rates.
- The number of selected records from each difficult level group is inversely proportional to the percentage of records in the original KDD data set.

1.3 Adversarial Machine Learning

Adversarial Machine Learning (AML) is the study of machine learning in the presence of an adversary that works against the ML system in an effort to reduce its effectiveness or extract information from it.

All aspects and phases of the machine learning process can be attacked by an adversary as can be seen in Figure 1.1.

1.3.1 Attack Types

If we see the problem from the attacker's perspective, we can define two types of attacks as poisoning attacks or evasion attacks. [Biggio 2012] and [Xiao 2015] describe different poisoning attacks. [Xiao 2015] devised attacks against the Ridge and Lasso linear classifiers by maximizing the classification error with regards to the training points. [Biggio 2012] performed poisoning

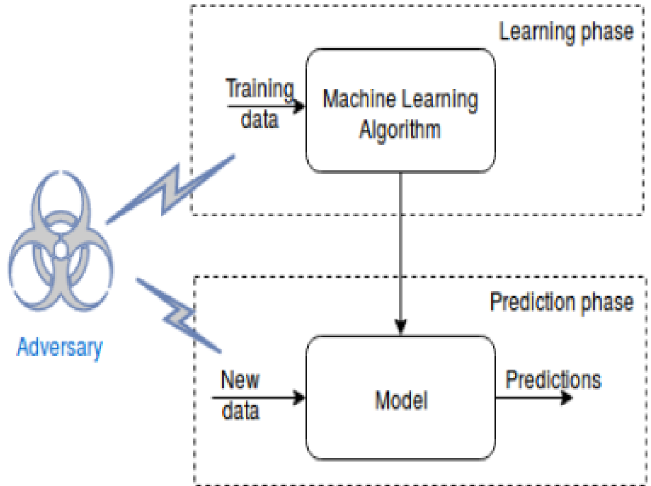

Figure 1.1: Adversarial Machine Learning [Maria Rigaki 2017]

attack on SVMs (Support Vector Machines). They injected samples to the training set in order to find the attack point that will maximize the classification error.

Evasion attacks were studied in the works of [Biggio 2013], [Biggio 2014] and [Ateniese 2015]. The latter, i.e. [Ateniese 2015] proposed a method in which a meta-classifier is created by training several classifiers on multiple training sets. This meta-classifier is used in order to extract statistical properties from the data but not the features themselves, which makes it an attack against privacy.

1.3.2 Adversarial Deep Learning

Deep Learning has become very successful in the recent years in the field of Natural Language Processing and Computer Vision. This also led to the development of Adversarial Deep Learning which was initially centered around the Computer Vision domain.

One of the first breakthroughs came in 2013, when [Szegedy 2013a] successfully demonstrated how one can fool Deep Learning classifiers by introducing small variations in an image. These variations were so small that they were imperceptible to humans but enough to fool the classifiers. Some examples of such images are shown in Figure 1.2. Another work [Nguyen 2015a], generated random images which appeared or had patterns in it, which did not mean anything, but the images were able to fool the Deep Learning classifiers into predicting them into valid object classes. Some reference images have been shown in Figure 1.3.

Although many different explanations have been given for the reason as to why this is possible, [Goodfellow 2014a] explains, contrary to the intuition of many people, that the main cause is the high degree of linearity of the Deep Learning components. The linearity is brought about by the use of piece wise linear activation functions such as Rectified Linear Units (ReLUs). Such functions are used to achieve faster optimization as well as help to create decision boundaries

Figure 1.2: Adversarial example produced by image perturbation. The neural network believes the images on the right are ostriches. [Szegedy 2013a]

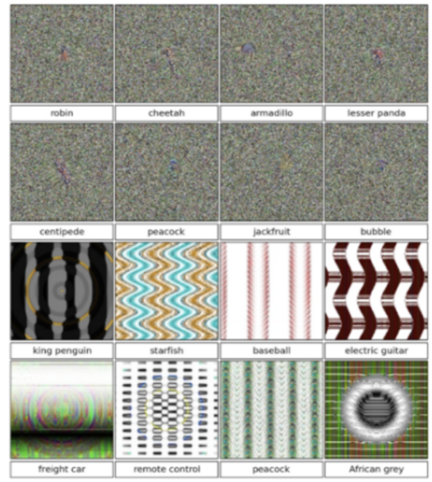

Figure 1.3: Images generated using evolutionary algorithms. [Nguyen 2015a]

that define much larger areas than the training data. This is the reason why when these classifiers encounter new examples (images), that have certain specific properties, they are misclassified ([Goodfellow 2014a] and [Nguyen 2015a]). Another very interesting property that was discovered by [Szegedy 2013a], [Goodfellow 2014a] and [Nguyen 2015a] was that the images that show adversarial properties for one neural network can transfer these properties to other neural networks trained separately.

The only models that have shown some resistance to adversarial examples are the Radial Basis Function (RBF) networks but they are not used often as they don't generalize well [Goodfellow 2014a]. Other than those, even shallow linear models are also affected by the same problem and so are model ensembles.

The methods and algorithms to generate adversarial examples has also been researched upon. There are many such methods which have a trade-off on speed of production, performance and complexity. Some of the methods that have been proposed are given below -

- Evolutionary algorithms, proposed in [Nguyen 2015a]. But this method is very slow compared to the other two alternatives.
- Fast Gradient Sign Method (FGSM) proposed in [Goodfellow 2014a].
- Jacobian-based Saliency Map Attack (JSMA) [Papernot 2016c] is more computationally expensive than FGSM but it has the ability to create adversarial samples with less degree of distortion.

Both the FGSM and JSMA methods try to generate a small perturbation in the original sample so that it will exhibit adversarial characteristics. In FGSM a perturbation δ is generated by computing the gradient of the cost function J in respect to the input x:

$$\delta = \epsilon sign(\nabla_x J(\theta, x, y)) \tag{1.1}$$

where θ are model parameters, x is the input to the model, y are the labels associated with x, ϵ is a very small value and $J(\theta, x, y)$ is the cost function used when training the neural network. This method is very fast because it requires the gradient which can be computed very efficiently using backpropagation. The perturbation is then added to the initial sample and the final result produces a misclassification. An example is shown in Figure 1.4.

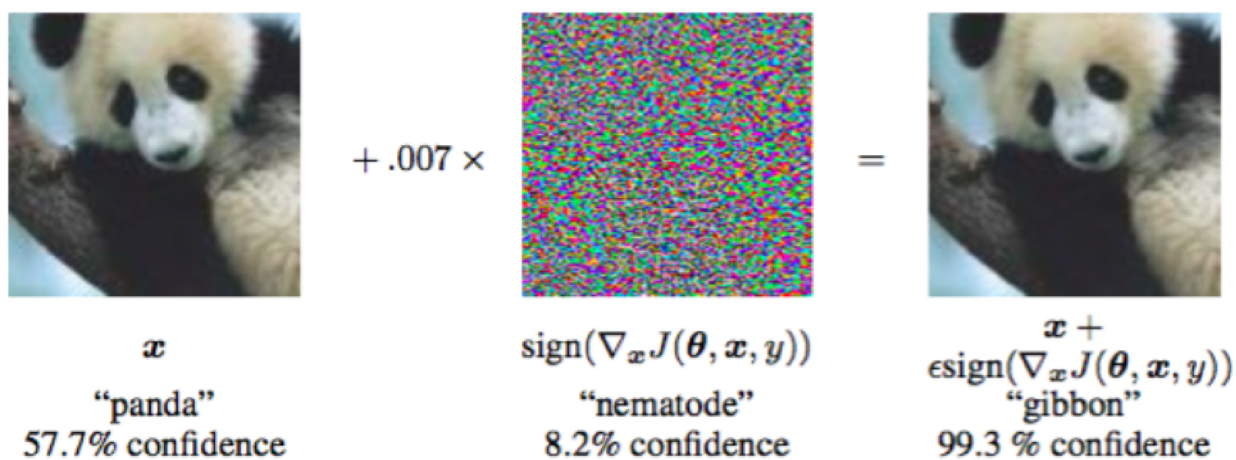

Figure 1.4: Generating adversarial samples with FGSM [Goodfellow 2014a]

JSMA, as the name suggests, generates adversarial sample perturbations based on the concept of saliency maps. The direction sensitivity of the sample in regards to the target class is calculated using a saliency map. [Papernot 2016c] designed an efficient saliency adversarial map under the L_0 distance (i.e. the number of features i such that $x'_i \neq x_i$) . The Jacobian matrix computed for a given sample x is expressed as -

$$J_f(x) = \frac{\partial f(x)}{\partial x} = [\frac{\partial f_j(x)}{\partial x_i}]_{i \times j} \tag{1.2}$$

In this way, the input features of x that made most significant changes to the output can be identified. Basically, the algorithm works by trying to determine which input features will be most likely to create a targeted class change. Using this sensitivity map one or more features are chosen as the possible perturbations and the model is checked to establish whether or not this change resulted in a misclassification. If it does not result in a misclassification, the next most sensitive feature is selected and a new iteration occurs until an adversarial sample that can fool the network is generated [Papernot 2016c]. The process is illustrated in Figure 1.5. Since the method usually takes a number of iterations, it is not as fast as FGSM.

Both FGSM and JSMA operate under the threat model of a strong attacker, e.g. an attacker that has knowledge of at least the underlying model. However if the attacker is not aware of the underlying model, it does not mean that the system cannot be exploited. If the attacker has only access to the model output and has some knowledge of the input to be provided, he can use the output of the model with different inputs to create an approximation of the model. And since the adversarial attacks have the transferability property, it is possible for the attacker to craft adversarial samples on the approximated model which can later be used as attack vectors against the original model [Papernot 2016b].

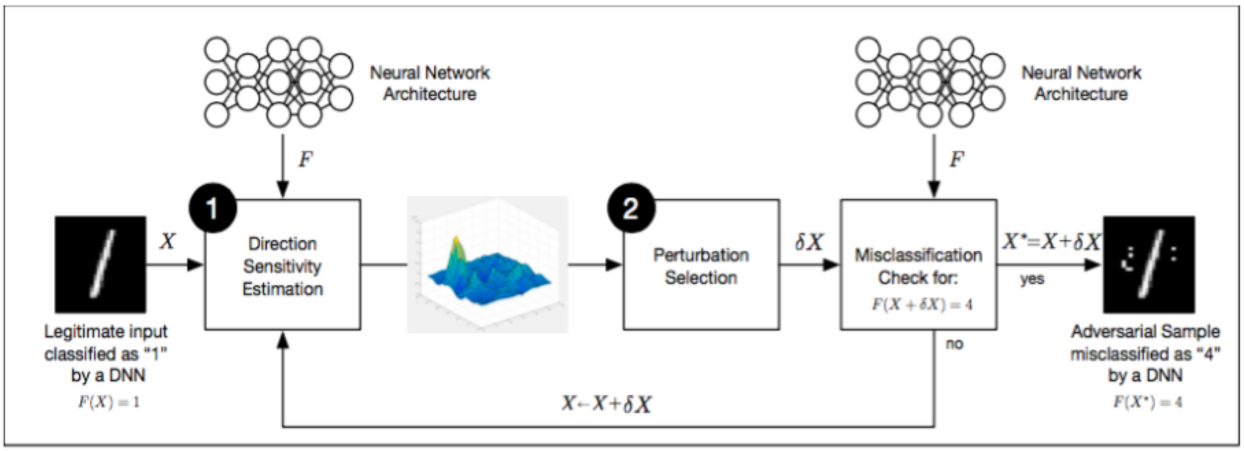

Figure 1.5: Generating adversarial samples with JSMA [Papernot 2016c]

An even weaker threat model is that in which the attacker has no access to the underlying model. These types of situations arise in face or voice recognition systems, i.e. in the physical domain. An attacker that can craft adversarial samples without access or knowledge of the underlying model or system could potentially fool these systems. [Kurakin 2016] demonstrated a successful attempt of such type of an attack.

[Papernot 2016a] thoroughly tested the concept of transferability. The authors tested several classifiers both as source for adversarial sample generation as well as target models. One thing to note, however, is that the testing was confined to image classifiers.

Chapter 2

Data Collection and Analysis

The most popular datasets in the domain of Intrusion Detction as discussed in Section 1.2.1 are the KDD'99 and the NSL-KDD datasets. These are the only ones that are labeled, include a variety of attacks and hence are widely used. Although these datasets are severely outdated, they have been chosen as a basis for this study mainly due to lack of better alternatives and secondly because the purpose of the study is the robustness of classifiers and not to make claims about prediction capabilities and generalization of our models.

2.1 KDD'99 and NSL-KDD

KDD'99 is one of the most widely used datasets in the literature related to Intrusion Detection. The attacks that are present in the dataset can be divided into four major categories: **Denial of Service (DoS) User to Root (U2R)**, **Remote to Local (R2L)** and **Probing** attacks. A short description of these categories are given below -

- **DoS** attacks are an interruption in an authorized user's access to a computer network, in other words, they are attacks against availability. This category contains attacks such as *smurf*, *neptune*, *mailbomb*, *udpstorm*, etc.

- **U2R** attacks indicate attempts of privelege escalation. Some attacks of this type in the dataset are *buffer overflow*, *loadmodule*, *sqlattack* and *rootkit*.

- **R2L** attacks aim to gain remote access to a system by exploiting a vulnerability. Some examples of this type of attacks are *multihop*, *guesspasswd*, *httptunnel* and *xsnoop*.

- **Probe** attacks aim to gather information by using enumeration techniques like scanning or probing different parts of the network, for e.g. the ports. Although strictly speaking, they are not attacks but they are the first set of steps that an attcker will perform before attacking a network. Some examples of such types of attacks are *ipsweep*, *portsweep*, *nmap* and *mscan*.

Feature	Type	Feature	Type
duration	cont.	is_guest_login	sym.
protocol_type	sym.	count	cont.
service	sym.	srv_count	cont.
flag	sym.	serror_rate	cont.
src_bytes	cont.	rerror_rate	cont.
dest_bytes	cont.	srv_rerror_rate	cont.
land	sym.	diff_srv_rate	cont.
wrong_fragment	cont.	srv_diff_host_rate	cont.
urgent	cont.	dst_host_count	cont.
hot	cont.	dst_host_srv_count	cont.
num_failed_logins	cont.	dst_host_same_srv_rate	cont.
logged_in	sym.	dst_host_diff_srv_rate	cont.
num_compromised	cont.	dst_host_same_src_port_rate	cont.
root_shell	cont.	dst_host_srv_diff_host_rate	cont.
su_attempted	cont.	dst_host_serror_rate	cont.
num_root	cont.	dst_host_srv_serror_rate	cont.
num_file_creations	cont.	dst_host_rerror_rate	cont.
num_access_files	cont.	dst_host_srv_rerrorv_rate	cont.
num_outbound_cmds	cont.	is_host_login	sym.

Table 2.1: KDD'99 and NSL-KDD features

The detailed list of features is given in Table 2.1. The original dataset description used the term "symbolic" for categorical variables and the term "continuous" for numerical ones. The features in the dataset can be divided into three main categories: **Basic**, **Traffic** and **Content** related ones, as described in [Tavallaee 2009b].

Basic features are the ones related to connection information such as hosts, ports, protocols and services used.

Traffic features are calculated during a window interval as an aggregate. A further subdivision is "aggregates based on the same host" and "aggregates over the same service". In the NSL-KDD dataset, the time window (in KDD'99) was substituted with a connection window of the last 100 connections.

Content features are extracted from the payload or packet data and they are related to the content of specific applications or protocol used.

The NSL-KDD dataset [NSL-KDD] has the same number of features as the KDD'99, but improved some shortcomings as described by [Tavallaee 2009a] which included the removal of redundant records in the training and testing sets and also adjusting the difficulty of classification for some attacks.

2.2 Data Preprocessing

For data preprocessing, the following steps were followed-

1. One-Hot encoding was used to convert the categorical features to numerical features.
2. All the features (now all numerical) were normalized using Min-Max Scaler as very large values can dominate the dataset and affect the performance of certain classifiers like SVM and the MLP.
3. The dataset had labels consisting of 39 distinct attack categories. These attacks were grouped into four major families - "DoS", "U2R", "R2L" and "Probe". Hence the problem was transformed into a five-class classification problem (including the "normal" class).

After preprocessing, the final number of features are 122. The number of data points in the training set are 1,25,973 and in the test set 22,544.

Chapter 3

Implementation Details

This section will highlight the libraries used and the implementation details. The code is written in Python language.

3.1 Libraries Used

Following are the major libraries used in the project -

- **Keras**: Deep Learning library for creating and training the model
- **Cleverhans**: For implementing adversarial attacks and adversarial example generation
- **Scikit-learn**: For making Machine Learning models for testing purposes
- **Numpy**: For storing the train and test data
- **Pandas**: For reading the train and test data
- **Matplotlib**: For visualisation in the form of plots and graphs

3.2 Description of functions

3.2.1 Jacobian-based Saliency Map Attack

To implement JSMA, we use the *SaliencyMapMethod* function present in the Cleverhans python library.

```
models = KerasModelWrapper(model)
jsma = SaliencyMapMethod(models, sess=sess)
jsma_params = {'theta': 1., 'gamma': 0.1, 'clip_min': 0., 'clip_max': 1., 'y_target': None
    }

for sample_ind in range(0, source_samples):
        sample = X_test_scaled[sample_ind: (sample_ind+1)]
```

```python
        # We want to find an adversarial example for each possible target class
        # (i.e. all classes that differ from the label given in the dataset)
        current_class = int(np.argmax(y_test[sample_ind]))

        # Only target the normal class
        for target in [0]:
                if current_class == 0:
                        break

                print('Generating adv. example for target class {} for sample {}'.format(
                    target, sample_ind), end='\r')

                # Run the Jacobian-based saliency map approach
                one_hot_target = np.zeros((1, FLAGS.nb_classes), dtype=np.float32)
                one_hot_target[0, target] = 1
                jsma_params['y_target'] = one_hot_target
                adv_x = jsma.generate_np(sample, **jsma_params)

                # Check if success was achieved
                res = int(model_argmax(sess, x, predictions, adv_x) == target)

                # Compute number of modified features
                adv_x_reshape = adv_x.reshape(-1)
                test_in_reshape = X_test_scaled[sample_ind].reshape(-1)
                nb_changed = np.where(adv_x_reshape != test_in_reshape)[0].shape[0]
                percent_perturb = float(nb_changed) / adv_x.reshape(-1).shape[0]

                X_adv[sample_ind] = adv_x
                results[target, sample_ind] = res
                perturbations[target, sample_ind] = percent_perturb

print()
print(X_adv.shape)
```

3.2.2 Fast Gradient Sign Method

To implement FGSM, we use the *FastGradientMethod* function present in the Cleverhans python library.

```python
models = KerasModelWrapper(model)
# Craft adversarial examples using Fast Gradient Sign Method (FGSM)
fgsm = FastGradientMethod(models, sess=sess)
fgsm_params = {'eps': 0.3}
adv_x_f = fgsm.generate(x, **fgsm_params)
# adv_x_f = tf.stop_gradient(adv_x_f)
X_test_adv, = batch_eval(sess, [x], [adv_x_f], [X_test_scaled])
```

3.2.3 Adversarial Feature Statistics

To get the statistics of the Adversarial examples that were generated, followed by plotting the most important features in the case of JSMA, we use the following piece of code.

```python
print("============ Adversarial Feature Statistics ============")

feats = dict()
total = 0
orig_attack = X_test_scaled - X_adv
for i in range(0, orig_attack.shape[0]):
        ind = np.where(orig_attack[i, :] != 0)[0]
```

```
        total += len(ind)
        for j in ind:
            if j in feats:
                feats[j] += 1
            else:
                feats[j] = 1

# The number of features that were changed for the adversarial samples
print("Number of unique features changed with JSMA: {}".format(len(feats.keys())))
print("Number of average features changed per datapoint with JSMA: {}".format(total/len(orig_attack)))

top_10 = sorted(feats, key=feats.get, reverse=True)[:10]
top_20 = sorted(feats, key=feats.get, reverse=True)[:20]
print("Top ten features: ", X_test.columns[top_10])

top_10_val = [100*feats[k] / y_test.shape[0] for k in top_10]
top_20_val = [100*feats[k] / y_test.shape[0] for k in top_20]

plt.figure(figsize=(12, 6))
plt.bar(np.arange(20), top_20_val, align='center')
plt.xticks(np.arange(20), X_test.columns[top_20], rotation='vertical')
plt.title('Feature participation in adversarial examples')
plt.ylabel('Percentage (%)')
plt.xlabel('Features')
plt.savefig('Adv_features.png', bbox_inches = "tight")
```

3.2.4 Plotting ROC curves

ROC curves were plotted for four ML based classifiers which are shown later. The general sample code for plotting ROC curves is given below -

```
dt = OneVsRestClassifier(DecisionTreeClassifier(random_state=42))
dt.fit(X_train_scaled, y_train)
y_pred = dt.predict(X_test_scaled)

fpr_dt, tpr_dt, _ = roc_curve(y_test[:, 0], y_pred[:, 0])
roc_auc_dt = auc(fpr_dt, tpr_dt)
print("Accuracy score: {}".format(accuracy_score(y_test, y_pred)))
print("F1 Score: {}".format(f1_score(y_test, y_pred, average='micro')))
print("AUC score: {}".format(roc_auc_dt))

y_pred_adv = dt.predict(X_adv)
fpr_dt_adv, tpr_dt_adv, _ = roc_curve(y_test[:, 0], y_pred_adv[:, 0])
roc_auc_dt_adv = auc(fpr_dt_adv, tpr_dt_adv)
print("Accuracy score adversarial: {}".format(accuracy_score(y_test, y_pred_adv)))
print("F1 Score adversarial: {}".format(f1_score(y_test, y_pred_adv, average='micro')))
print("AUC score adversarial: {}".format(roc_auc_dt_adv))

plt.figure()
lw = 2
plt.plot(fpr_dt, tpr_dt, color='darkorange', lw=lw, label="ROC Curve (area = %0.2f)" % roc_auc_dt)
plt.plot(fpr_dt_adv, tpr_dt_adv, color='green', lw=lw, label="ROC Curve adv. (area = %0.2f)" % roc_auc_dt_adv)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Decision Tree (class=Normal)")
plt.legend(loc="lower right")
plt.savefig('ROC_DT.png', bbox_inches = "tight")
```

Chapter 4

Results

4.1 Baseline Models

A number of different were trained and tested on the NSL-KDD dataset to establish a baseline. The results on the test set are given in Table 4.1.

Method	Accuracy	F1-Score	AUC (normal)
Decision Tree	0.989	0.992	0.992
Random Forest	0.993	0.994	0.994
Linear SVM	0.945	0.958	0.954
Voting Ensemble	0.993	0.993	0.746
MLP	0.985	-	-

Table 4.1: Test set results for 5-class classification

The variation of the accuracy with the epochs while training the MLP on the clean dataset is shown in Figure 4.1.

4.2 Adversarial Test Set Generation

Both the FGSM and JSMA methods were used in order to generate adversarial test sets from the original test set. The underlying model used to generate the adversarial examples was a pre-trained MLP. The architecture of the model is shown in figure 4.2. Table 4.2 below, shows the difference between the two methods in terms of changed features on average as well as the unique features changed for all data points in the test set.

Tables 4.3 and 4.4 show the transformation required for selected features using the JSMA method in order to for the specific data point to become "normal". Some of the altered features for that data point are shown.

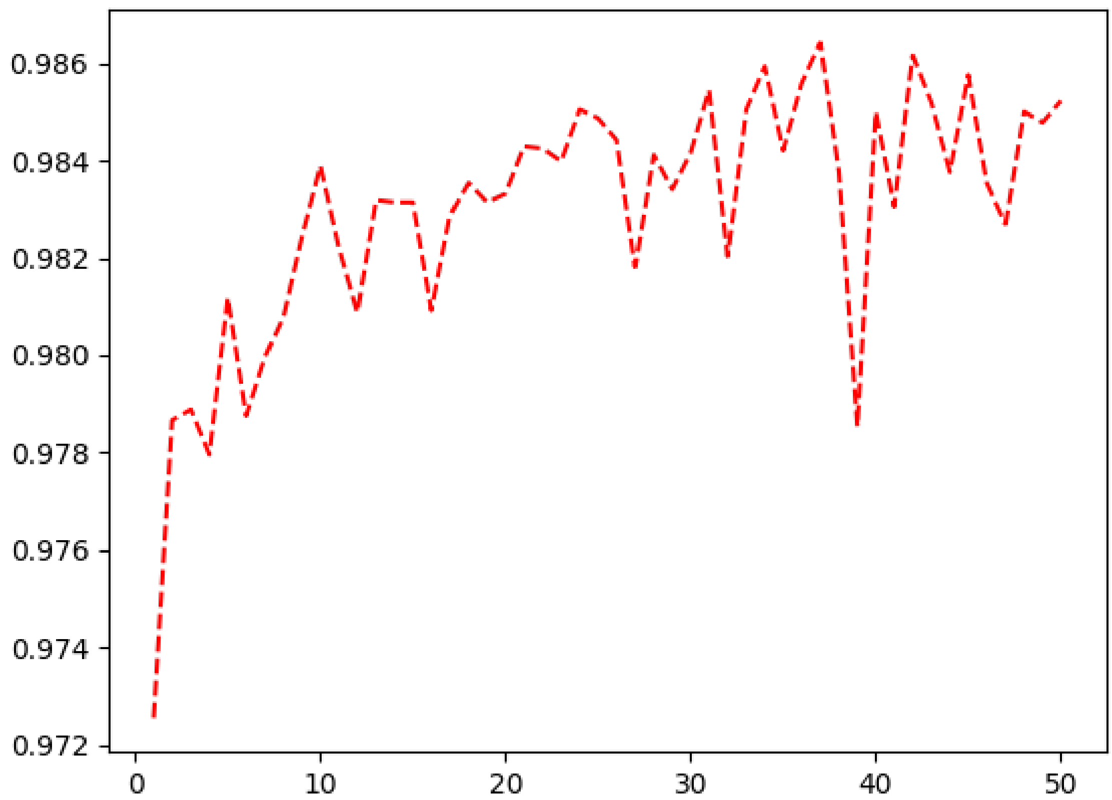

Figure 4.1: MLP Training - accuracy vs. epochs

Method	Num. of unique altered features	Avg. features changed per data point	Percentage of altered features
FGSM	122	76.47	62.68
JSMA	89	11.58	9.49

Table 4.2: Adversarial feature statistics

...	F26	...	F29	F30	...	F41	...	label
...	0.7	...	0.7	0.7	...	0.7	...	dos

Table 4.3: Data point $x^{(17)}$ in original test set

4.3 Model Evaluation on Adversarial Data

This section presents the results of the baseline models on the adversarial test set generated by the JSMA method in terms of Accuracy, F1-score and AUC (Table 4.5). One thing to note here

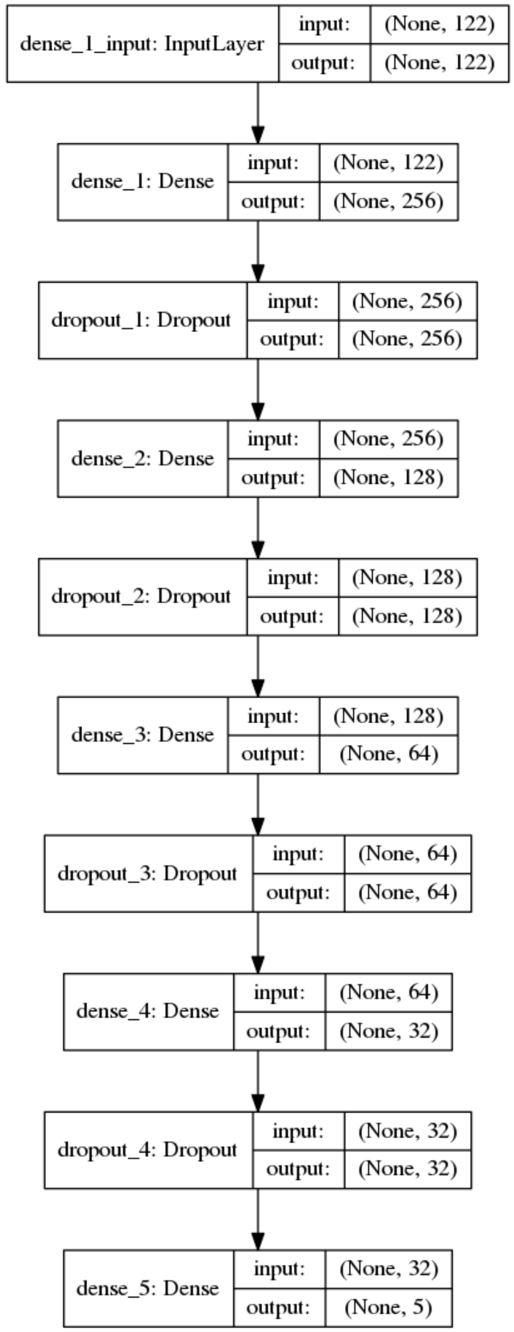

Figure 4.2: Architecture of underlying MLP

...	F26	...	F29	F30	...	F41	...	label
...	1.0	...	1.0	1.0	...	1.0	...	normal

Table 4.4: Transformation of data point $x_{\text{adv}}^{(17)}$ using JSMA

is that both the AUC results as well as the ROC curves in the figures below, are only presented for the the "normal" class, while the F1-score is an average score over all classes.

Method	Accuracy	F1-Score	AUC (normal)
Decision Tree	0.660	0.802	0.744
Random Forest	0.968	0.977	0.986
Linear SVM	0.810	0.846	0.949
Voting Ensemble	0.914	0.914	0.723
MLP	0.670	-	-

Table 4.5: Adversarial test set results for 5-class classification

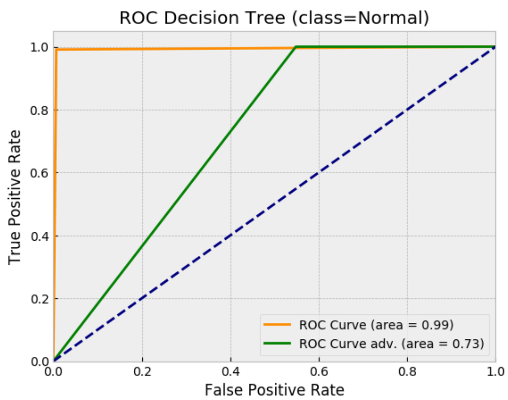

Figure 4.3: Decision Tree ROC curves

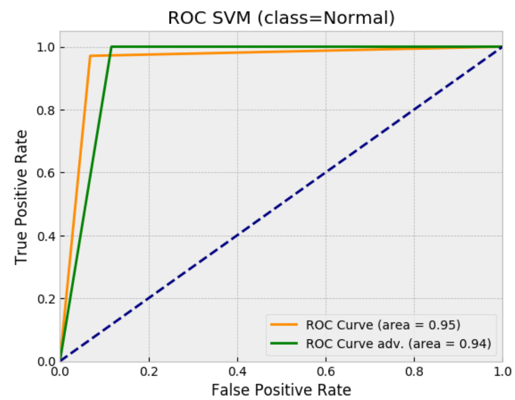

Figure 4.4: SVM ROC curves

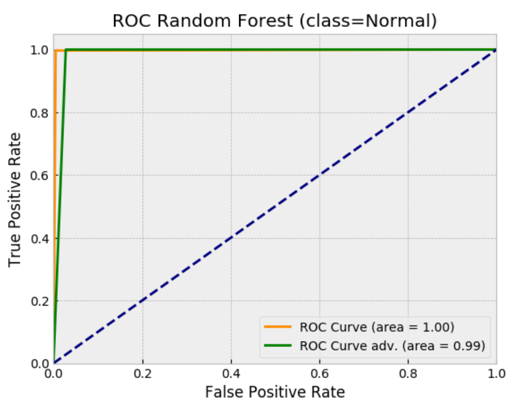

Figure 4.5: Random Forest ROC curves

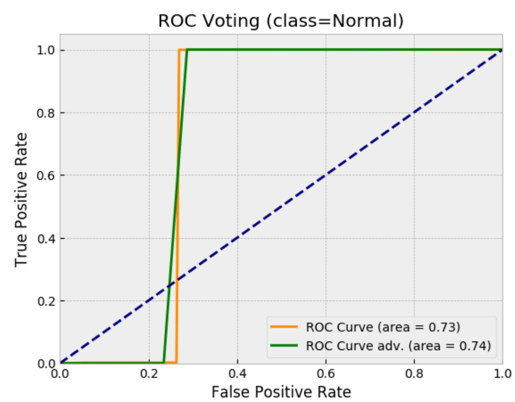

Figure 4.6: Voting Ensemble ROC curves

4.4 Feature Evaluation

After generating the adversarial test set using JSMA, a ranking of the features in terms of frequency with which they appear in the adversarial test set as changed was created. This was calculated by subtracting the original test set from the adversarial test set

$$\delta = X^* - X_{\text{test}}$$

where X^* is the adversarial test set and X_{test} is the original test set. In order to find which features were altered for each data point $\delta^{(i)}$ we need to find the feature indexes j where feature $\delta_j^{(i)} = 0$.

The top ten features and their description are presented in Table 4.6. Figure 4.7 shows the top 20 features and their percentages.

Feature	Description
srv_count	number of connections to the same service as the current connection in the past 100 connections
count	number of connections to the same host as the current connection in the past 100 connections
dst_host_srv_count	number of connections to the same service and destination host as the current connection in the past 100 connections
dst_host_same_srv_rate	% of connections to the same service and destination host
src_bytes	number of data bytes from source to destination
same_srv_rate	% of connections to the same service
dst_bytes	number of data bytes from destination to source
dst_host_diff_srv_rate	% of different services on the current host
dst_host_count	number of connections to the same destination host as the current connection in the past 100 connections
duration	duration of the connection

Table 4.6: Top 10 adversarial features using JSMA

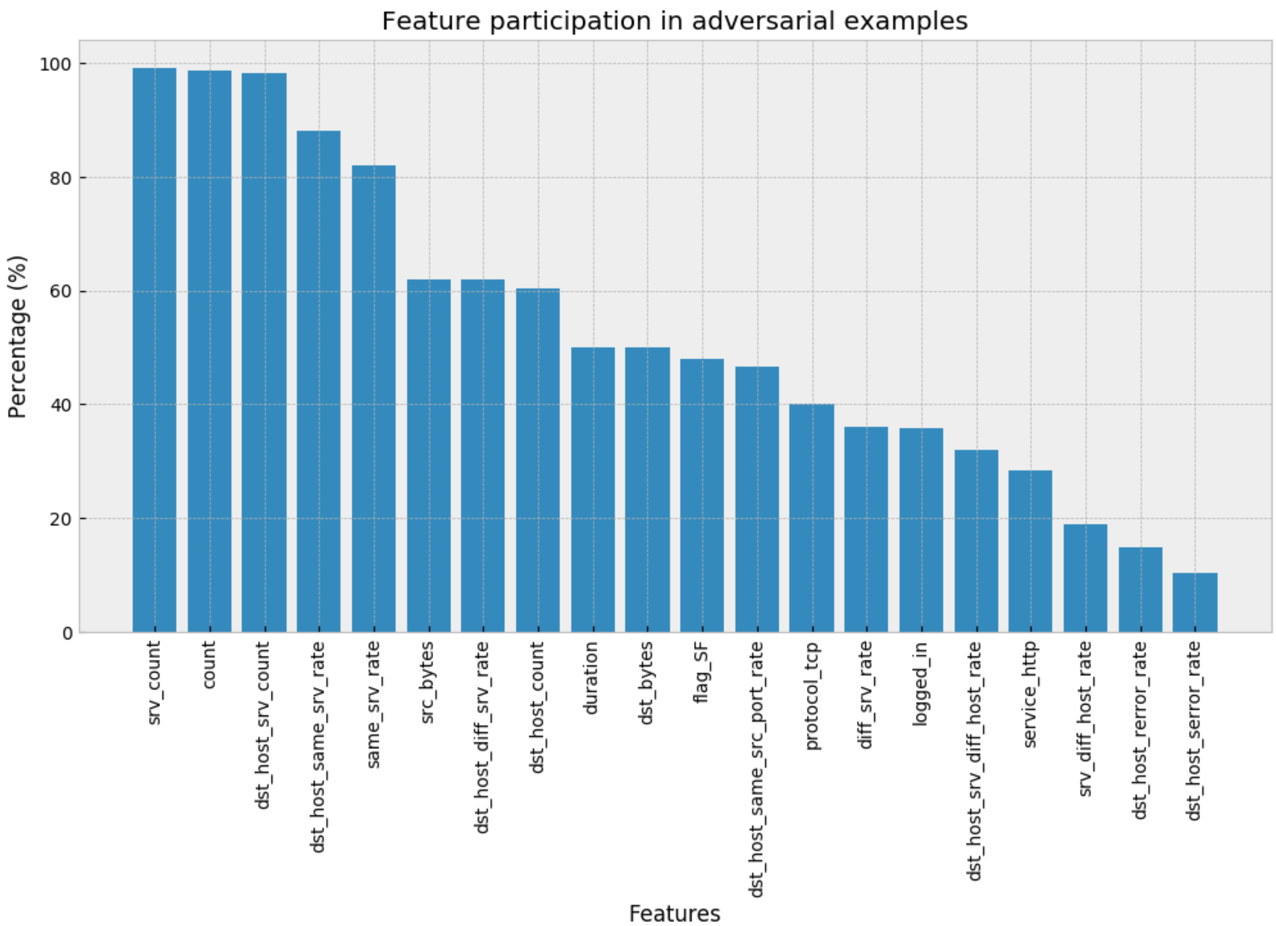

Figure 4.7: Most used features in adversarial sample generation

Chapter 5

Discussion

5.1 Data Modelling

Table 4.1 shows that all the models have an overall accuracy and F1-score around 99%. The AUC scores however show a major difference where we observe that the Decision Tree and the Random Forest classifier outperform the SVM and the Voting ensemble. This implies that the first two methods are performing slightly better in classifying the “normal” test samples exhibiting a lower False Positive Rate (FPR). This can also be observed in the ROC-AUC curves shown in Figures 4.3 to 4.6.

5.2 Adversarial Test Set Generation

As it was explained earlier, the FGSM method changes each feature very slightly while JSMA searches through the features of each data point and changes one at each iteration in order to produce an adversarial sample. This means that FGSM is not suitable for tasks like NIDS since the features are generated from network traffic and controlling them in a fine grained manner would not be possible for an adversary. On the contrary, JSMA changes only a few features at a time and although it takes more time to generate adversarial examples as it is iterative, it can form the basis for a practical attack due to the lower number of features that have to be changed. This is totally inline with the observations in [Huang 2011] where the importance of domain applicability is highlighted as a potential problem for an attacker.

5.3 Model Evaluation on Adversarial Data

In terms of overall classification accuracy all classifiers were affected. The most severely affected is the Decision Tree with a drop of 32.9% and the Decision Tree whose accuracy dropped by 13.5%. The Random Forest classifier showed some robustness with an accuracy drop of 2.5%.

When it comes to F1-score, the Decision Tree was affected the most and its score was reduced by 19%. Linear SVM saw an F1-score reduction of 11.2%. The other two classifiers did not suffer as much and again the Random Forest showed the highest robustness by dropping only 1.7%.

The AUC over the normal class is an indicator of how robust were the classifiers against targeted misclassification towards the normal class. It provides a measure on how much was the increase on the FPR compared to the TPR, in other words, how many attacks were misclassified as normal traffic. The best performing classifier was the the Linear SVM, which only dropped 0.5%, followed by the Random Forest which dropped 0.8%. The Decision Tree classifier was severely affected, losing 24.8% percentage points.

Based on the results, it seems that the only method that was robust across all metrics was the Random Forest. The Decision Tree was the worst performing classifier, which also corresponds with the result of [Papernot 2016a], in which also, Decision Tree was one of the worst performing methods.

5.4 Feature Evaluation

Table 5.6 gives an idea of the top-10 features which contribute most during the generation of adversarial samples. Among those features, the top two are about the number of connections to the same service/host as the current connection in the past 100 connections. The next two features are about the rate and the count of the connections to the same host and port. This tells us that one way an attacker could get around the detection would be to lessen the number of requests they generate. Exploiting this can be very helpful for running bots as one can generate connections to external command and control servers and can hide their traffic under normal traffic that a user creates. A similar type of reasoning can also be applied to other count and rate types of features. This type of discussion is also relevant to the Denial of Service (DoS) type of attacks and while this dataset is quite old, historically, there have been attacks that followed the “low and slow” approach in order to appear as close to legitimate traffic as possible. When it comes to features related to service types, using common protocols like HTTP or HTTPS can be be a good strategy in order to hide into other normal traffic, instead of using protocols that might be easier to get discovered.

Chapter 6

Conclusion

From the results and discussion presented previously we can say that JSMA is more preferable in the intrusion detection domain in terms of applicability of attacks in terms of as compared to FGSM which cannot be used in a practical manner. We also observed the robustness exhibited by different classifiers which is also shown in [Papernot 2016a]. Also, it is clear that using a substitute model to generate adversarial samples can be successful and it is worth looking at adversarial security when deploying machine learning classifiers. This means that even when attackers do not have access to the training data, adversarial samples can transfer to different models under certain circumstances. This means that when machine learning is used, it should be accompanied with relevant adversarial training and testing and strengthening.

Also, one important thing to note is that in our case, we had a preprocessed dataset and not raw data which made it easier to attack. A physical attack would require some idea on how the raw network data are processed and the types of features that are generated. So, if we were to deal with raw data some knowledge would be required about how the data is preprocessed and how features are generated.

Finally, even if we know the features used, it would still require work to adjust the traffic profiles of the specific attack. Contrary to the image classification problem, where each bit in the image can be considered a feature which can be easily altered, not all traffic related characteristics can be changed, even when an adversary has the ability to craft specific network packets and payloads. To protect against adversaries, NIDS classifiers will have to use features that can not be easily manipulated by an attacker.

6.1 Future Work

This study presented an attempt to transfer adversarial methods from the Deep Learning image classification domain to the NIDS domain. While defenses have been proposed against these methods in several studies, these defenses do not generalize very well. A future study would be to examine some of these defenses and establish whether they improve the situation or not especially in the NIDS domain.

In our study a neural net was used as the source model for preparing the adversarial examples. An extension of this study could be to use other models as the source as well.

Finally, further study is also required to understand the effect of the adversarial methods in different attack classes which would potentially yield a better overview of which features are more important for each attack type when it comes to adversarial sample generation. This can eventually be used by adversaries to select strategies that would allow them to hide their malicious traffic depending on the chosen attack.

Appendices

Appendix A

Source Code

```
import numpy as np
import pandas as pd
import sys
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop, adam
from keras import backend as K
from keras.utils import plot_model
from cleverhans.attacks import FastGradientMethod, SaliencyMapMethod
from cleverhans.utils_tf import model_train , model_eval , batch_eval, model_argmax
from cleverhans.attacks_tf import jacobian_graph
from cleverhans.utils import other_classes
from cleverhans.utils_keras import KerasModelWrapper

import tensorflow as tf
from tensorflow.python.platform import flags

from sklearn.multiclass import OneVsRestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC

from sklearn.metrics import accuracy_score, roc_curve, auc, f1_score
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
import matplotlib.pyplot as plt
import pickle
plt.style.use('bmh')

K.set_learning_phase(1)

FLAGS = flags.FLAGS

flags.DEFINE_integer('nb_epochs', 50, 'Number of epochs to train model')
flags.DEFINE_integer('batch_size', 64, 'Size of training batches')
flags.DEFINE_integer('learning_rate', 0.005, 'Learning rate for training')
flags.DEFINE_integer('nb_classes', 5, 'Number of classification classes')
flags.DEFINE_integer('source_samples', 10, 'Nb of test set examples to attack')

print()
print()
print("============================== Start of preprocessing stage ==============================")

names = ['duration', 'protocol', 'service', 'flag', 'src_bytes', 'dst_bytes', 'land', 'wrong_fragment', 'urgent', 'hot', 'num_failed_logins', 'logged_in', 'num_compromised',
```

```
'root_shell', 'su_attempted', 'num_root', 'num_file_creations', 'num_shells', 'num_access_files', 'num_outbound_cmds', 'is_host_login', 'is_guest_login', 'count', 'srv_count', 'serror_rate', 'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv_rate', 'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_srv_diff_host_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate', 'dst_host_rerror_rate', 'dst_host_srv_rerror_rate', 'attack_type', 'other']

df_train = pd.read_csv('../NSL_KDD/KDDTrain+.txt', names=names, header=None)
df_test = pd.read_csv('../NSL_KDD/KDDTest-21.txt', names=names, header=None)
print("Initial training and test data shapes: ", df_train.shape, df_test.shape)

full = pd.concat([df_train, df_test])
assert full.shape[0] == df_train.shape[0] + df_test.shape[0]

full['label'] = full['attack_type']

# Denial-of-Service (DoS) Attacks
full.loc[full.label == 'neptune', 'label'] = 'dos'
full.loc[full.label == 'back', 'label'] = 'dos'
full.loc[full.label == 'land', 'label'] = 'dos'
full.loc[full.label == 'pod', 'label'] = 'dos'
full.loc[full.label == 'smurf', 'label'] = 'dos'
full.loc[full.label == 'teardrop', 'label'] = 'dos'
full.loc[full.label == 'mailbomb', 'label'] = 'dos'
full.loc[full.label == 'processtable', 'label'] = 'dos'
full.loc[full.label == 'udpstorm', 'label'] = 'dos'
full.loc[full.label == 'apache2', 'label'] = 'dos'
full.loc[full.label == 'worm', 'label'] = 'dos'

# User-to-root (U2R) Attacks
full.loc[full.label == 'buffer_overflow', 'label'] = 'u2r'
full.loc[full.label == 'loadmodule', 'label'] = 'u2r'
full.loc[full.label == 'perl', 'label'] = 'u2r'
full.loc[full.label == 'rootkit', 'label'] = 'u2r'
full.loc[full.label == 'sqlattack', 'label'] = 'u2r'
full.loc[full.label == 'xterm', 'label'] = 'u2r'
full.loc[full.label == 'ps', 'label'] = 'u2r'

# Remote-to-local (R2L) Attacks
full.loc[full.label == 'ftp_write', 'label'] = 'r2l'
full.loc[full.label == 'guess_passwd', 'label'] = 'r2l'
full.loc[full.label == 'imap', 'label'] = 'r2l'
full.loc[full.label == 'multihop', 'label'] = 'r2l'
full.loc[full.label == 'phf', 'label'] = 'r2l'
full.loc[full.label == 'spy', 'label'] = 'r2l'
full.loc[full.label == 'warezclient', 'label'] = 'r2l'
full.loc[full.label == 'warezmaster', 'label'] = 'r2l'
full.loc[full.label == 'xlock', 'label'] = 'r2l'
full.loc[full.label == 'xsnoop', 'label'] = 'r2l'
full.loc[full.label == 'snmpgetattack', 'label'] = 'r2l'
full.loc[full.label == 'httptunnel', 'label'] = 'r2l'
full.loc[full.label == 'snmpguess', 'label'] = 'r2l'
full.loc[full.label == 'sendmail', 'label'] = 'r2l'
full.loc[full.label == 'named', 'label'] = 'r2l'

# Probe attacks
full.loc[full.label == 'satan', 'label'] = 'probe'
full.loc[full.label == 'ipsweep', 'label'] = 'probe'
full.loc[full.label == 'nmap', 'label'] = 'probe'
full.loc[full.label == 'portsweep', 'label'] = 'probe'
full.loc[full.label == 'saint', 'label'] = 'probe'
full.loc[full.label == 'mscan', 'label'] = 'probe'

full = full.drop(['other', 'attack_type'], axis=1)
```

```python
print("Unique labels", full.label.unique())
full = full.sample(frac=1).reset_index(drop=True)

# Generate One - Hot encoding
full2 = pd.get_dummies(full, drop_first=False)

# Separate training and test sets again
features = list(full2.columns[:-5])   # Due to One-Hot encoding
y_train = np.array(full2[0: df_train.shape[0]][['label_normal', 'label_dos', 'label_probe', 'label_r2l', 'label_u2r']])
X_train = full2[0: df_train.shape[0]][features]

y_test = np.array(full2[df_train.shape[0]: ][['label_normal', 'label_dos', 'label_probe', 'label_r2l', 'label_u2r']])
X_test = full2[df_train.shape[0]: ][features]

# Scale data
scaler = MinMaxScaler().fit(X_train)
X_train_scaled = np.array(scaler.transform(X_train))
X_test_scaled = np.array(scaler.transform(X_test))

# Generate label encoding for Logistic regression
labels = full.label.unique()
le = LabelEncoder()
le.fit(labels)
y_full = le.transform(full.label)
y_train_l = y_full[0: df_train.shape[0]]
y_test_l = y_full[df_train.shape[0]: ]

print("Training dataset shape", X_train_scaled.shape, y_train.shape)
print("Test dataset shape", X_test_scaled.shape, y_test.shape)
print("Label encoder y shape", y_train_l.shape, y_test_l.shape)

print("================================ End of preprocessing stage ================================")
print()
print()

print("========================== Start of adversarial sample generation ==========================")
print()
print()

def mlp_model():
        """
        Generate a Multilayer Perceptron model
        """
        model = Sequential()
        model.add(Dense(256, activation='relu', input_shape=(X_train_scaled.shape[1], )))
        model.add(Dropout(0.4))
        model.add(Dense(256, activation='relu'))
        model.add(Dropout(0.4))
        model.add(Dense(FLAGS.nb_classes, activation='softmax'))
        model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

        model.summary()
        return model

def mlp_model2():
    """
    Generate a Multilayer Perceptron model
    """
    model = Sequential()
    model.add(Dense(256, activation='relu', input_shape=(X_train_scaled.shape[1], )))
    model.add(Dropout(0.2))
```

```
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(32, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(FLAGS.nb_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    model.summary()
    return model

acc_list = []
count = 0
def evaluate():
        """
        Model evaluation function
        """
        global count
        count += 1

        eval_params = {'batch_size': FLAGS.batch_size}
        accuracy = model_eval(sess, x, y, predictions, X_test_scaled, y_test, args=eval_params)
        global acc_list
        acc_list.append((count, accuracy))
        print("Test accuracy on legitimate test samples: " + str(accuracy))


# Tensorflow placeholder variables
x = tf.placeholder(tf.float32, shape=(None, X_train_scaled.shape[1]))
y = tf.placeholder(tf.float32, shape=(None, FLAGS.nb_classes))

tf.set_random_seed(42)
model = mlp_model2()
plot_model(model, show_shapes=True, to_file='model.png')
sess = tf.Session()
predictions = model(x)
init = tf.global_variables_initializer()
sess.run(init)

# Train the model
train_params = {'nb_epochs': FLAGS.nb_epochs, 'batch_size': FLAGS.batch_size, 'learning_rate': FLAGS.learning_rate, 'verbose': 0}

model_train(sess, x, y, predictions, X_train_scaled, y_train, evaluate=evaluate, args=train_params)

file = open("scores.pkl", "wb")
pickle.dump(acc_list, file)
file.close()

# Generate adversarial samples for all test datapoints
source_samples = X_test_scaled.shape[0]

# Jacobian-based Saliency Map Attack
results = np.zeros((FLAGS.nb_classes, source_samples), dtype='i')
perturbations = np.zeros((FLAGS.nb_classes, source_samples), dtype='f')
grads = jacobian_graph(predictions, x, FLAGS.nb_classes)

X_adv = np.zeros((source_samples, X_test_scaled.shape[1]))

models = KerasModelWrapper(model)
jsma = SaliencyMapMethod(models, sess=sess)
jsma_params = {'theta': 1., 'gamma': 0.1, 'clip_min': 0., 'clip_max': 1., 'y_target': None}
```

```python

for sample_ind in range(0, source_samples):
    sample = X_test_scaled[sample_ind: (sample_ind+1)]
    # We want to find an adversarial example for each possible target class
    # (i.e. all classes that differ from the label given in the dataset)
    current_class = int(np.argmax(y_test[sample_ind]))

    # Only target the normal class
    for target in [0]:
        if current_class == 0:
            break

        print('Generating adv. example for target class {} for sample {}'.format(target, sample_ind), end='\r')

        # Run the Jacobian-based saliency map approach
        one_hot_target = np.zeros((1, FLAGS.nb_classes), dtype=np.float32)
        one_hot_target[0, target] = 1
        jsma_params['y_target'] = one_hot_target
        adv_x = jsma.generate_np(sample, **jsma_params)

        # Check if success was achieved
        res = int(model_argmax(sess, x, predictions, adv_x) == target)

        # Compute number of modified features
        adv_x_reshape = adv_x.reshape(-1)
        test_in_reshape = X_test_scaled[sample_ind].reshape(-1)
        nb_changed = np.where(adv_x_reshape != test_in_reshape)[0].shape[0]
        percent_perturb = float(nb_changed) / adv_x.reshape(-1).shape[0]

        X_adv[sample_ind] = adv_x
        results[target, sample_ind] = res
        perturbations[target, sample_ind] = percent_perturb

print()
print(X_adv.shape)

print("=============== Evaluation of MLP Performance ===============")
print()

eval_params = {'batch_size': FLAGS.batch_size}
accuracy = model_eval(sess, x, y, predictions, X_test_scaled, y_test, args=eval_params)
print("Test accuracy on normal examples: {}".format(accuracy))

accuracy_adv = model_eval(sess, x, y, predictions, X_adv, y_test, args=eval_params)
print("Test accuracy on adversarial examples: {}".format(accuracy_adv))
print()

print("================ Decision tree Classifier ===============")
dt = OneVsRestClassifier(DecisionTreeClassifier(random_state=42))
dt.fit(X_train_scaled, y_train)
y_pred = dt.predict(X_test_scaled)

# Calculate FPR for normal class only
fpr_dt, tpr_dt, _ = roc_curve(y_test[:, 0], y_pred[:, 0])

roc_auc_dt = auc(fpr_dt, tpr_dt)
print("Accuracy score: {}".format(accuracy_score(y_test, y_pred)))
print("F1 Score: {}".format(f1_score(y_test, y_pred, average='micro')))
print("AUC score: {}".format(roc_auc_dt))

# Predict using adversarial test samples
y_pred_adv = dt.predict(X_adv)
fpr_dt_adv, tpr_dt_adv, _ = roc_curve(y_test[:, 0], y_pred_adv[:, 0])
```

```
roc_auc_dt_adv = auc(fpr_dt_adv, tpr_dt_adv)
print("Accuracy score adversarial: {}".format(accuracy_score(y_test, y_pred_adv)))
print("F1 Score adversarial: {}".format(f1_score(y_test, y_pred_adv, average='micro')))
print("AUC score adversarial: {}".format(roc_auc_dt_adv))

plt.figure()
lw = 2
plt.plot(fpr_dt, tpr_dt, color='darkorange', lw=lw, label="ROC Curve (area = %0.2f)" % roc_auc_dt)
plt.plot(fpr_dt_adv, tpr_dt_adv, color='green', lw=lw, label="ROC Curve adv. (area = %0.2f)" % roc_auc_dt_adv)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Decision Tree (class=Normal)")
plt.legend(loc="lower right")
plt.savefig('ROC_DT.png', bbox_inches = "tight")
print()

print()
print("=============================== Random Forest Classifier ===============================")
rf = OneVsRestClassifier(RandomForestClassifier(n_estimators=200, random_state=42))
rf.fit(X_train_scaled, y_train)
y_pred = rf.predict(X_test_scaled)

# Calculate FPR for normal class only
fpr_rf, tpr_rf, _ = roc_curve(y_test[:, 0], y_pred[:, 0])

roc_auc_rf = auc(fpr_rf, tpr_rf)
print("Accuracy score: {}".format(accuracy_score(y_test, y_pred)))
print("F1 Score: {}".format(f1_score(y_test, y_pred, average='micro')))
print("AUC score: {}".format(roc_auc_rf))

# Predict using adversarial test samples
y_pred_adv = rf.predict(X_adv)
fpr_rf_adv, tpr_rf_adv, _ = roc_curve(y_test[:, 0], y_pred_adv[:, 0])
roc_auc_rf_adv = auc(fpr_rf_adv, tpr_rf_adv)
print("Accuracy score adversarial: {}".format(accuracy_score(y_test, y_pred_adv)))
print("F1 Score adversarial: {}".format(f1_score(y_test, y_pred_adv, average='micro')))
print("AUC score adversarial: {}".format(roc_auc_rf_adv))

plt.figure()
lw = 2
plt.plot(fpr_rf, tpr_rf, color='darkorange', lw=lw, label="ROC Curve (area = %0.2f)" % roc_auc_rf)
plt.plot(fpr_rf_adv, tpr_rf_adv, color='green', lw=lw, label="ROC Curve adv. (area = %0.2f)" % roc_auc_rf_adv)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Random Forest (class=Normal)")
plt.legend(loc="lower right")
plt.savefig('ROC_RF.png', bbox_inches = "tight")
print()

print()
print("=============================== Linear SVC Classifier ===============================")
sv = OneVsRestClassifier(LinearSVC(C=1., random_state=42, loss='hinge'))
sv.fit(X_train_scaled, y_train)
y_pred = sv.predict(X_test_scaled)
```

```python

# Calculate FPR for normal class only
fpr_sv, tpr_sv, _ = roc_curve(y_test[:, 0], y_pred[:, 0])

roc_auc_sv = auc(fpr_sv, tpr_sv)
print("Accuracy score: {}".format(accuracy_score(y_test, y_pred)))
print("F1 Score: {}".format(f1_score(y_test, y_pred, average='micro')))
print("AUC score: {}".format(roc_auc_sv))

# Predict using adversarial test samples
y_pred_adv = sv.predict(X_adv)
fpr_sv_adv, tpr_sv_adv, _ = roc_curve(y_test[:, 0], y_pred_adv[:, 0])
roc_auc_sv_adv = auc(fpr_sv_adv, tpr_sv_adv)
print("Accuracy score adversarial: {}".format(accuracy_score(y_test, y_pred_adv)))
print("F1 Score adversarial: {}".format(f1_score(y_test, y_pred_adv, average='micro')))
print("AUC score adversarial: {}".format(roc_auc_sv_adv))

plt.figure()
lw = 2
plt.plot(fpr_sv, tpr_sv, color='darkorange', lw=lw, label="ROC Curve (area = %0.2f)" % roc_auc_sv)
plt.plot(fpr_sv_adv, tpr_sv_adv, color='green', lw=lw, label="ROC Curve adv. (area = %0.2f)" % roc_auc_sv_adv)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC SVM (class=Normal)")
plt.legend(loc="lower right")
plt.savefig('ROC_SVM.png', bbox_inches = "tight")
print()

print()
print("=================================== Voting Classifier ===================================")
vot = VotingClassifier(estimators=[('dt', dt), ('rf', rf), ('sv', sv)], voting='hard')
vot.fit(X_train_scaled, y_train_l)
y_pred = vot.predict(X_test_scaled)

# Calculate FPR for normal class only
fpr_vot, tpr_vot, _ = roc_curve(y_test_l, y_pred, pos_label=1, drop_intermediate=False)

roc_auc_vot = auc(fpr_vot, tpr_vot)
print("Accuracy score: {}".format(accuracy_score(y_test_l, y_pred)))
print("F1 Score: {}".format(f1_score(y_test_l, y_pred, average='micro')))
print("AUC score: {}".format(roc_auc_vot))

# Predict using adversarial test samples
y_pred_adv = vot.predict(X_adv)
fpr_vot_adv, tpr_vot_adv, _ = roc_curve(y_test_l, y_pred_adv, pos_label=1, drop_intermediate=False)
roc_auc_vot_adv = auc(fpr_vot_adv, tpr_vot_adv)
print("Accuracy score adversarial: {}".format(accuracy_score(y_test_l, y_pred_adv)))
print("F1 Score adversarial: {}".format(f1_score(y_test_l, y_pred_adv, average='micro')))
print("AUC score adversarial: {}".format(roc_auc_vot_adv))

plt.figure()
lw = 2
plt.plot(fpr_vot, tpr_vot, color='darkorange', lw=lw, label="ROC Curve (area = %0.2f)" % roc_auc_vot)
plt.plot(fpr_vot_adv, tpr_vot_adv, color='green', lw=lw, label="ROC Curve adv. (area = %0.2f)" % roc_auc_vot_adv)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
```

```
plt.ylabel("True Positive Rate")
plt.title("ROC Voting (class=Normal)")
plt.legend(loc="lower right")
plt.savefig('ROC_Vot.png', bbox_inches = "tight")
print()


# Print overall ROC curves
plt.figure(figsize=(12, 6))
plt.plot(fpr_dt_adv, tpr_dt_adv, label = 'DT (area = %0.2f)' % roc_auc_dt_adv)
plt.plot(fpr_rf_adv, tpr_rf_adv, label = 'RF (area = %0.2f)' % roc_auc_rf_adv)
plt.plot(fpr_sv_adv, tpr_sv_adv, label = 'SVM (area = %0.2f)' % roc_auc_sv_adv)
plt.plot(fpr_vot_adv, tpr_vot_adv, label = 'Vot (area = %0.2f)' % roc_auc_vot_adv)

plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve (adversarial samples)')
plt.legend(loc = 'best')
plt.savefig('ROC_curves_adv.png', bbox_inches = "tight")


plt.figure(figsize=(12, 6))
plt.plot(fpr_dt, tpr_dt, label = 'DT (area = %0.2f)' % roc_auc_dt)
plt.plot(fpr_rf, tpr_rf, label = 'RF (area = %0.2f)' % roc_auc_rf)
plt.plot(fpr_sv, tpr_sv, label = 'SVM (area = %0.2f)' % roc_auc_sv)
plt.plot(fpr_vot, tpr_vot, label = 'Vot (area = %0.2f)' % roc_auc_vot)

plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve (normal samples)')
plt.legend(loc = 'best')
plt.savefig('ROC_curves.png', bbox_inches = "tight")
print()

print("====================== Adversarial Feature Statistics ======================")

feats = dict()
total = 0
orig_attack = X_test_scaled - X_adv
for i in range(0, orig_attack.shape[0]):
        ind = np.where(orig_attack[i, :] != 0)[0]
        total += len(ind)
        for j in ind:
                if j in feats:
                        feats[j] += 1
                else:
                        feats[j] = 1

# The number of features that were changed for the adversarial samples
print("Number of unique features changed with JSMA: {}".format(len(feats.keys())))
print("Number of average features changed per datapoint with JSMA: {}".format(total/len(
    orig_attack)))

top_10 = sorted(feats, key=feats.get, reverse=True)[:10]
top_20 = sorted(feats, key=feats.get, reverse=True)[:20]
print("Top ten features: ", X_test.columns[top_10])

top_10_val = [100*feats[k] / y_test.shape[0] for k in top_10]
top_20_val = [100*feats[k] / y_test.shape[0] for k in top_20]

plt.figure(figsize=(12, 6))
plt.bar(np.arange(20), top_20_val, align='center')
plt.xticks(np.arange(20), X_test.columns[top_20], rotation='vertical')
plt.title('Feature participation in adversarial examples')
plt.ylabel('Percentage (%)')
plt.xlabel('Features')
```

```python
plt.savefig('Adv_features.png', bbox_inches = "tight")

# Craft adversarial examples using Fast Gradient Sign Method (FGSM)
fgsm = FastGradientMethod(models, sess=sess)
fgsm_params = {'eps': 0.3}
adv_x_f = fgsm.generate(x, **fgsm_params)
# adv_x_f = tf.stop_gradient(adv_x_f)
X_test_adv, = batch_eval(sess, [x], [adv_x_f], [X_test_scaled])

# Evaluate accuracy
eval_par = {'batch_size': FLAGS.batch_size}
accuracy = model_eval(sess, x, y, predictions, X_test_adv, y_test, args=eval_par)
print("Test accuracy on adversarial examples: {}".format(accuracy))

# Comparison of adversarial and original test samples (attack)
feats = dict()
total = 0
orig_attack = X_test_scaled - X_test_adv

for i in range(0, orig_attack.shape[0]):
	ind = np.where(orig_attack[i, :] != 0)[0]
	total += len(ind)
	for j in ind:
		if j in feats:
			feats[j] += 1
		else:
			feats[j] = 1

# The number of features that where changed for the adversarial samples
print("Number of unique features changed with FGSM: {}".format(len(feats.keys())))
print("Number of average features changed per datapoint with FGSM: {}".format(total/len(orig_attack)))
```

Bibliography

[Ateniese 2015] Mancini L.V. Spognardi A. Villani A. Vitali D. Ateniese G. et G. Felici. *Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers*. International Journal of Security and Networks 10(3), 137–150, 2015.

[Bhuyan 2014] Bhattacharyya D.K. Bhuyan M.H. et J.K Kalita. *Network anomaly detection: methods, systems and tools*. IEEE communications surveys tutorials 16(1), 303–336, 2014.

[Biggio 2012] Nelson B. Biggio B. et P. Laskov. *Poisoning attacks against support vector machines*. arXiv preprint arXiv:1206.6389, 2012.

[Biggio 2013] Corona I. Maiorca D. Nelson B. Srndi c N. Laskov P. Giacinto G.and Roli F. Biggio B. *Evasion attacks against machine learning at test time*. Joint European Conference on Machine Learning and Knowledge Discovery in Databases, 2013.

[Biggio 2014] Fumera G. Biggio B. et F. Roli. *Security evaluation of pattern classifiers under attack*. IEEE Transactions on Knowledge and Data Engineering 26(4), 984–996, 2014.

[Brugger 2007] S. T. Brugger et J. Chow. *An assessment of the darpa ids evaluation dataset using snort*. UCDAVIS department of Computer Science 1(2007), 22, 2007.

[Chuvakin 2012] Schmidt K. Chuvakin A. et C. Phillips. *Logging and log management: Theauthoritative guide to understanding the concepts surrounding logging and log management*. Newnes, 2012.

[Goodfellow 2014a] Shlens J. Goodfellow I. J. et C. Szegedy. *Explaining and harnessing adversarial examples*. arXiv preprint arXiv:1412.6572, 2014.

[Goodfellow 2014b] Shlens J. Goodfellow I. J. et C. Szegedy. *Explaining and harnessing adversarial examples*. arXiv preprint arXiv:1412.6572, 2014.

[Huang 2011] Joseph A. D. Nelson B. Rubinstein B. I. Huang L. et J. Tygar. *Adversarial machine learning*. Proceedings of the 4th ACM workshop on Security and artificial intelligence, ACM, pp. 43–58, 2011.

[KDD] KDD. *KDD Cup 1999 Data*. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`.

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

[Maria Rigaki 2017] Ahmed Elragal Maria Rigaki. *Adversarial Deep Learning Against Intrusion Detection Classifiers*. ST-152 Workshop on Intelligent Autonomous Agents for Cyber Defence and Resilience, 2017.

[McHugh 2000] J. McHugh. *Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory*. ACM Transactions on Information and System Security (TISSEC) 3(4), 262–294, 2000.

[Milenkoski 2015] Vieira M. Kounev S. Avritzer A. Milenkoski A. et B.D. Payne. *Evaluating computer intrusion detection systems: A survey of common practices*. ACM Computing Surveys (CSUR) 48(1), 12, 2015.

[Nguyen 2015a] Yosinski J. Nguyen A. et J. Clune. *Deep neural networks are easily fooled: High confidence predictions for unrecognizable images*. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015.

[Nguyen 2015b] Yosinski J. Nguyen A. et J. Clune. *Deep neural networks are easily fooled: High confidence predictions for unrecognizable images*. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 427-436, 2015.

[NSL-KDD] NSL-KDD. *NSL-KDD Dataset*. `http://www.unb.ca/cic/research/datasets/nsl.html`.

[Papernot 2016a] McDaniel P. Papernot N. et I. Goodfellow. *Transferability in machine learning: from phenomena to black-box attacks using adversarial samples*. arXiv preprint arXiv:1605.07277, 2016.

[Papernot 2016b] McDaniel P. Goodfellow I. Jha S. Celik Z. B. Papernot N. et A. Swami. *Practical black-box attacks against deep learning systems using adversarial examples*. arXiv preprint arXiv: 1602.02697, 2016.

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

[Sommer 2010a] R. Sommer et V. Paxson. *Outside the closed world: On using machine learning for network intrusion detection*. 2010 IEEE Symposium on Security and Privacy (SP), IEEE, pp. 305–316, 2010.

[Sommer 2010b] R. Sommer et V. Paxson. *Outside the closed world: On using machine learning for network intrusion detection*. 2010 IEEE Symposium on Security and Privacy (SP), IEEE, p. 305–316, 2010.

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

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

[Tavallaee 2009a] Bagheri E. Lu W. Tavallaee M. et A. A. Ghorbani. *A detailed analysis of the kdd cup 99 data set.* Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on, IEEE, pp. 1–6, 2009.

[Tavallaee 2009b] Bagheri E. Lu W. Tavallaee M. et A. A. Ghorbani. *A detailed analysis of the kdd cup 99 data set.* Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on IEEE, pp. 1–6., 2009.

[Vom Brocke 2009] Simons A. Niehaves B. Riemer K. Plattfaut R. Cleven A. et al. Vom Brocke J. *Reconstructing the giant: On the importance of rigour in documenting the literature search process.* ECIS, Vol. 9, pp. 2206–2217, 2009.

[Xiao 2015] Biggio B. Nelson B. Xiao H. Eckert C. Xiao H. et F. Roli. *Support vector machines under adversarial label contamination.* Neurocomputing 160, 53–62, 2015.