A Generative Adversarial Approach for Packet Manipulation Detection

by

Åsmund Kamphaug

Master Thesis in
Information and Communication Technology

FINAL VERSION 5

The University of Agder

Grimstad, June 4, 2018

Abstract

Over the years, machine learning has been used together with intrusion detection systems to protect networks against different threats. The evolution of machine learning has exploded and there are new types of of machine learning algorithms being studied on different fields. Networks security is not one these fields that have the most research, and with the continuous change in the way attacks are appearing, machine learning in network security is more alluring than ever. The intention of this thesis is to present a solution that shows that using machine learning in intrusion detection domain is a way to enhance network security.

Several different generative techniques have emerged from the latest years of deep learning research. One particular that stands out is The Generative Adversarial Network (GAN), that is largely used in the field of image generation. These techniques is based on the idea of two networks competing against each other and trying to be superior than the other. This thesis follows a quantitative methodology and a combination of experimentation and engineering research.

The study focuses on how well the developed solution performs on detecting networks attacks and how well it can learn to recreate networks packets. This approach implements a modified version of the generative adversarial network, by implementing an optimisation training step to the regular algorithm. The results shows that with this new type of generative adversarial network the accuracy increases from 2 % to 100 % when detecting DARPA99 labelled attacks. The results also shows that the solution experiences mode collapse when creating new network packets, but the model is able to create real network packets that are approved by Wiresharks syntax check and also for the human eye looks like normal network packets.

Acknowledgements

I would like to thank my thesis supervisor, Prof. Ole-Christoffer Granmo for his guidance and helpful directions throughout the duration of this study. Furthermore, I would like to thank my friends and family for their good conversations and support over these two years, and to thank my girlfriend Mia Therese Formo Selte, for all the continuous support and immense tolerance for the past two years, without which it would not have been possible to finalize my studies and this work.

Contents

List of Figures

List of Tables

Abbreviations

IDS Intrusion Detection System

NIDS Network Intrusion Detection System

DL Deep Learning

ML Machine Learning

GAN Generative Adversarial Network

CNN Convolutional Neural Network

DCGAN Deep Convolutional Generative Adversarial Network

Chapter 1

Introduction

1.1 Introduction

Internet and computer networking is now a part of our daily life. New platforms that uses computer networks has been developed more and more over these past years. Now everything is connected to your network and to the internet, from your computers to telephones, tablets, cars and all the way down to your fridge and electric doors. But what are protecting all of these things?(Today there are more and more cyber attacks happening worldwide than we can imagine. It's not only companies that are targeted for these attacks. Recently it was discovered a hacker group that had taken control over 500 000 devices on the internet. These devices was not only the devices of companies, but also private persons telephones, fridges, microwaves and computers.

As we can see, the more our life gets online, the more important it is to have good networks security. This is where intrusion detection systems comes into play. These are systems that protect people and their networks by detecting malicious data that flows in the network. Enterprises deploy different security measures on their network, but still security breaches is a major concern.

Machine learning can be a powerful tool to help improve network security. It has been attempted before to use different machine learning techniques in intrusion detection systems with varied result. In the past years there have not been so many attempts on this task, while the breakthroughs on machine learning is happening often. Meaning that there are new techniques haven't been used for this task. One of them being the Generative Adversarial Network.

1.2 Goal

By using the state of the art machine learning techniques, this thesis presents a solution that uses a modified Generative Adversarial Network. Proves that it can be used as an intrusion detection system with very high detection accuracy on real life attacks, as well as creating new network packets.

1.3 Thesis definition

In Chapter 2 an in-depth research concerning intrusion detection systems was contocted and it was discovered that machine learning had been used earlier to enhance different intrusion detection systems. The most recent years machine learning have evolved, but these new breakthroughs has not been used for this task. This thesis will make use the state of the art techniques to try to solve this growing problem on how to solve network security in this modern age.

1.4 Limitations

This project aims on how to create an solution for a generative adversarial network as an intrusion detection system(IDS). It will not however try to solve every problem that a final product may have. The main priority is to make a solution that uses a Generative Adversarial network as an intrusion detection system. This solution will be trained on the DARPA99 dataset, and different experiments will be performed to measure the solutions ability to detect attacks and if it can create genuine networks packets.

1.5 Thesis Organization

The rest of the thesis presents an in-depth review of previous research work in the area of generative adversarial machine learning and its application in information security in Chapter: 2 and the definitions of the research questions. Followed by the research methodology in Chapter: 3. Chapter: 4 is including the detailed process and structure of the the solution. The experiments and the result is reported in Chapter: 5. The thesis closes with the critical review of the results and the final conclusions and proposed future work in Chapter: 6.

Chapter 2

Literature Review

In this section of the study it was conducted an in-depth research on the background as well as the current state of the art of intrusion detection systems and machine learning techniques. It also to explore different intrusion detection dataset. In this study, the purpose of the review was to identify missing research gaps that allowed the formulation of the research questions, that this thesis tried to answer. As well as choose what kind of dataset was going to be used in this study.

2.1 Intrusion Detection

When it comes to categorisation of techniques and methodologies used in NIDS, the vast majority of the literature agrees on the following categorisation.

- **Misuse-based** or **signature based** are the systems that use signatures or indicators extracted from previous documented attacks. These signatures are manually generated by security experts, and they have to updated every time a new attack type of is discovered and the system needs to detect it. So since this process is manual, the maintenance of these types are frequent, especially with the increased rate of attacks we see today, is becoming a concern.
- **Anomaly based** system are system that tries to model normal behaviour in a network in contrast to what is anomalous and potential malicious. While these type of system promise the ability to adapt to new attacks. One of the major concerns is how to define whenever the behaviour are anomalous or not..
- **Hybrid** systems are the different combination of the above approaches

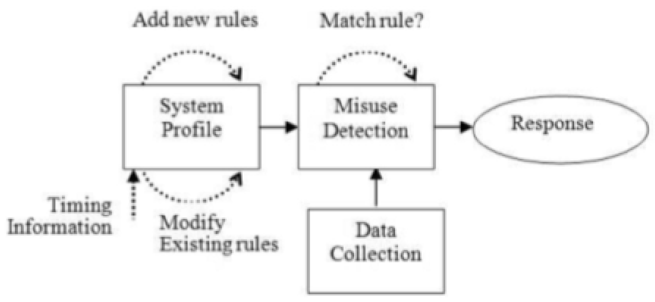


Figure 2.1: A typical misuse/signature based detection system

In Figure 2.1 an typical architecture of misuse based detection system is illustrated. The model consist of four components, data collection, system profile, misuse detection and response. The data can be collected from many data sources including audit trails, network traffic and system call trace. The collected data are transferred to an understandable format for the other components of the system. The system profile is used to characterise normal and abnormal behaviours. This is done by setting rules to see what a normal user behaviour is and what kind of operations that user typical would perform or not on different objects. The profiles are matched with the actual system activities and if it matches it is flagged as an intrusion and the system sends a response to a security expert. [7]

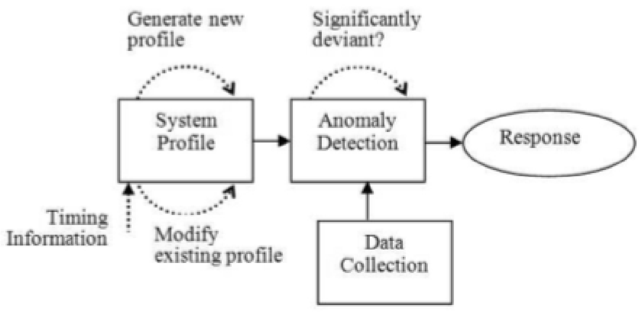


Figure 2.2: A typical anomaly detection system

In Figure 2.2 an typical architecture of anomaly based detection system is illustrated. This model also consist of four components, but instead of misuse detection it has anomaly detection. Normal data and network activity is obtained and saved by the data collection. Then normal system profiles are created using different techniques.

The anomaly detection compares the similarity of the system profiles and the current activities and decides what percentage these activities should be flagged as abnormal traffic. Then a response are send to a security expert. Anomaly detection addresses the biggest limitation of misuse detection, since it can find novel attacks. However the false alarm rate is general very high since there are much assumptions to these models. This is where machine learning can be a tool to help mitigate this problem. [7]

One of the most recent survey on intrusion detection system and machine learning was made by Anna L. Buczak and Erhan Guven in their article A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection[3]. In their work they took quite a comprehensive look and examines a large number of different machine learning algorithms like Artificial Neural Networks, Association Rule Mining, Bayesian Networks, Clustering, Decision Trees, Ensemble Learning, Evolutionary Computation, Hidden Markov Models, Naive Bayes and Support Vector Machines. In this paper there was many different machine learning used as intrusion detection systems, but the authors of the paper choose only to include papers where high number of citation was used. Meaning ignoring the work that had been performed in recent years. For example most of the papers that was used to gather information about Artificial Neural Networks was from the years 1998-2002. While some of the papers about Deep learning was from later years (2013-2014)

It was again made a new survey on machine learning in intrusion detection system by Monowar H. Bhuyan and is coleeges in 2014 that included not only classifiers as the article before, but also pure anomaly detection techniques like; clustering, statistical methods,computing, knowledge based and combination learners.

2.2 IDS Datasets

One of the most known dataset is the DARPA dataset[17]. This dataset was the first standard corpora for evaluation of computer network intrusion detection systems. This was created by The Cyber Systems and Technology Group (formerly the DARPA Intrusion Detection Evaluation Group) of MIT Lincoln Laboratory, under Defense Advanced Research Projects Agency (DARPA ITO) with the sponsorship of the Air Force Research Laboratory (AFRL/SNHS). These people created the first formal repeatable and statistically evaluation of different IDS. This evaluation was carried out in 1998 and 1999. These evaluation measured the probability of the detection and false alarms for every system tested on. Furthermore the dataset this thesis will focus on is the newest of does two, the 1999 dataset. In the 1999 dataset there was two parts, the first one was an offline evaluation and a real-time.

The main reason DARPA dataset was used instead of one of the more common ones like KDD'99 is that unlike the KDD'99 dataset data does not contain the actually packet that made this attack, instead only using features extracted from the the data. The DARPA dateset data uses the raw networks packets together with labelling of these to classify which packets contains attacks. And since this thesis will try to recreate network packets, this is essential. This change did not address all issues that DARPA has and more importantly it did not erase the fact that it is quite outdated. The Table: 2.1 contains different IDS dataset that can be used.

Table 2.1: Intrusion detection systems datasets

Source	Type	Labelled	Reference
DARPA	NIDS	Yes*	[17]
KDD'99	NIDS	Yes	[10]
UNB	NIDS (flows)	Yes	[19]
CTU-13	Botnet	Yes*	[6]
LANL	Multi-source	Yes	[12]
Uni of Victoria	Botnet	Yes	[22]

2.3 Generative Adversarial Network

Ian Goodfellow together with colleagues at the University of Montreal published a paper in 2014 that introduced Generative Adversarial Network's to the word. Through the combination of computational graphs and game theory. It was shown through their research that given enough computational power, it was possible to make two models that fought against each other to co-train through backpropogation.

These two models has two very distinct roles. Given some real dataset **R**, **G** the *generator*, is trying to create some fake data that looks alike as the the real data. While the other part of this architecture, *D* the *discriminator* would get fed either data from the genuine dataset or from the generated data and classify them between fake or real. Goodfellow's metaphor was that the generator was like a team of forgers that was to match real paintings with their output, while the discriminator was the team of detectives trying to tell the difference as illustrated in Figure: 2.3 Except in this particular case where the forgers (the generator) never gets to see the original data, but only the detectives (the discriminator) judgement of the data, meaning that they are blind forgers.

In other words does the discriminator learn the boundaries between the classes while the generator learns the distribution of the classes. In the ideal case is when the discriminator and the generator would get better over time until the generator created so alike sample as the real data so the discriminator was not able to differentiate between does two samples.[9]

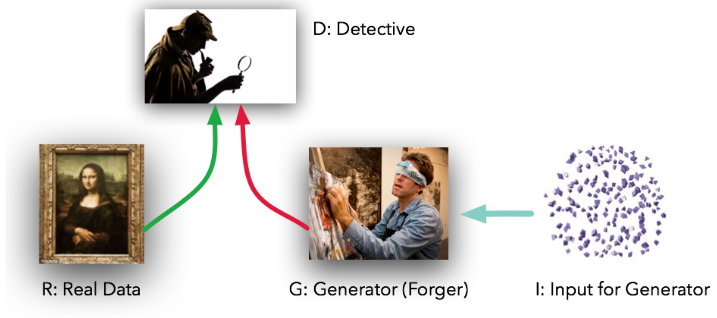


Figure 2.3: Illustration of GAN architecture

For the Generator to learn the distribution, p_g over data x, the input noise of the generator should be defines such as $p_z(z)$. Then $G(z, \theta_g)$ maps z from a latent space Z to a data space and $D(x, \theta_d)$ outputs a probability to choose if x originated from the real data or of the generators distribution p_g

The Discriminator would be trained to maximise the probability of predicting the right label to the real data and the generated sample. The generator on the other hand would be trained to minimise the probability of the discriminator's correct answer, with mathematical terms: $\log(1 - D(G(z)))$. So this architecture training task can be seen as is a minmax games between the generator and discriminator with the function $V(G, D)$:[18]

$$\underset{G}{min}\ \underset{D}{max}\ V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

Meaning that the Generator tries to fool the discriminator and the discriminator becomes better to not be fooled by the generator, that's where the Aaversarial prts comes in. In the best case scenario the training of these network would stop if the discriminator is not able to distinguish p_g and p_{data}, i.e. $D(x, \theta_d) = {}^1/_2$. This is very hard to obtain.[18]

2.3.1 Mode Collapse

One of the problem GANs experience is Mode Collapse also know as **the Hevetica scenario**. Mode collapse is a commonly cause of failure for GANs where the generator learns to reproduce samples with very little variety. This problem apears when the generator learns to map different input noise values to the same output point. In the article realeased by Goodfellow[9] he explains mode collapse and one of the example he used is shown in Figure: 2.4

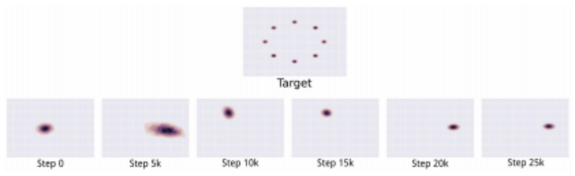


Figure 2.4: This figure illustrates the mode collapse problem on a two-dimensional fictive dataset.

The first row shows the target distribution. In the lower row there are different distribution that the generator has learned over the training process. Instead of converging to a distribution that contains all modes, the generator only produces a single mode that is different each time, and the discriminator learns the features of each one. There are several people that work on solving mode collapse, since this is the biggest problem GAN's have.

2.3.2 Image Generation With Generative Adversarial Network

Generative adversarial Networks (GAN's) are mainly used for images Generating. In earlier days of GAN's the vanilla GAN architecture, that consisted of only fully connected layers, was used to create new images so alike the real dataset that the discriminator could not longer differentiate between the real and from fake images. [8] Meaning that the generator could create new types of images from that dataset. This was the first stepping stone for the generative adversarial network. Later in 2016 Alec Radford, Luke Metz and Soumith Chintala published an article that introduced a new type of GAN, the Deep Conditional Generative Adversarial Nets(DCGAN).[21] DCGAN is one of the most popular types of GANs today. Instead of the network containing fully connected layers, Convolutional Nets (CNN's) was used in it's place.

CNNs was first introduced by Yann LeCun[16] and was a class of biologically inspired neural networks that solved a supervised learning task through a series of convolutional filters and simple non-linearity's. They are remarkable result in a wide variety of machine learning problems.[15].

The architecture for the DCGAN is different from the vanilla GAN, but the objective is the same. In Figure 2.5 the architecture of the generator of the DCGAN is shown. The discriminator is almost the just opposite of the generator meaning that the discriminator takes in an image and then produces a probability if that image is real or fake.

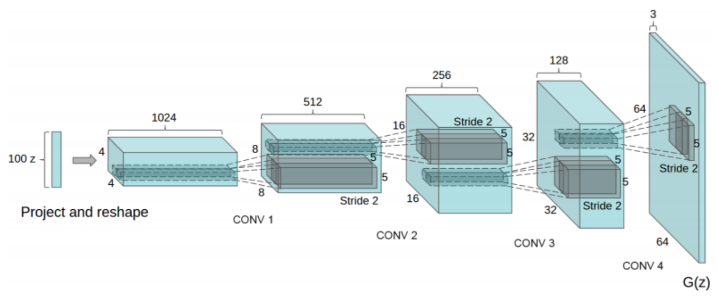


Figure 2.5: Architecture of the generator in DCGAN from this paper [21]

In the paper [21] the Generator in has four convolutional layers, followed by batch normalization and Rectified Linear Unit(ReLU) as the non-linearity's.[1]. The generator is fed a vector z that are drawn from a normal distribution that are reshaped to a 4D shape. Next each network up-sample the 4D tensor by performing transposed convolution operation on it. The difference of a transposed convolution from a regular one is that a regular one typically goes from wide and shallow layers to narrower and deeper. Meanwhile the transposed goes from a deep and narrow layer to a shallow and wide.

The generator start with a very deep and narrow input vector and after the vector goes through a transposed convolution z using a $5x5$ kernel size then the depth of the picture are reduced from 512 to 3 that represents a RGB colour image. The final shape the generator produces is a $32x32x3$ vector.[21]

The discriminator in their method is feed $32x32x3$ images tensor, that the generator created and returns 1 shaped output. Instead of the performing transposed convolutions, the discriminator performs normal convolutions. There each is reducing the feature vector dimensions by half. After these convolutionss, it returns a probability by using Logistic Sigmoid as the activation function in the last layer. The generator and discriminator is trained at the same time using backpropogation where the training steps is written more in detail section 2.3 explaining the minmax game.

As of the result of the DCGAN the architecture was able to generate images that looked alike real images, but some of them was not as good as shown in figure: 2.6

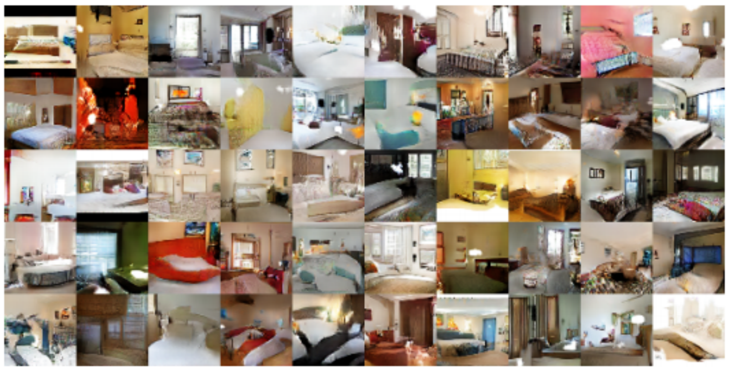


Figure 2.6: Result of generated images of Hotel Room using DCGAN

2.4 Research Questions

Based on the research made in Chapter 2 it come to light that machine learning has been used in different forms in intrusion detection system, with different results. But the latest years it has not been that many approaches to use machine learning with network intrusion detection systems. The generative adversarial network has been deployed in later years to different problems, but mainly on image problems. Through the research made in chapter 2 shows that generative adversarial network has not been used in a intrusion detection system. This thesis will explore the possibilities this generative architecture has as an intrusion detection system. So based on all the gathered data and research the following research questions was created.

- To what degree can a completely trained GAN detect real attacks in computer network?
- To what degree can a Generative Adversarial Network's generate network packets that passes as normal packets?
- To what degree can a Generative Adversarial Network's produce abnormal packets, that passes as normal ones?

Chapter 3

Methodology

3.1 Quantitative method

Quantitative research deals in numbers, logic, and an objective stance. Quantitative research also focuses on numeric and unchanging data and detailed, convergent reasoning rather than divergent reasoning. An essential thing with the quantitative method is classification and categorisations of a problem. Quantitative methods have primary value for research which aims to acquire general knowledge. [2]. A differen way to see research methods, is in the field of computer science. In computer science there are three mayor paradigms which are used to provide a context when writing work in the discipline of computing. These three paradigms all consist of four steps.[5]

The first paradigm *theory* is rooted in mathematics and follows in development of coherent, valid theory. When errors or inconsistencies are discovered, Mathematicians iterates though these steps.

1. Characterise object of study (definition);
2. Hypothesise possible relationships among them (theorem);
3. Determine whether the relationships are true (proof);
4. interpret results;

Experimental scientific method is the root of the second paradigm, *Explanatory* modeling. An other name for this paradigm is ”experimentation”, but ”abstraction” is used for this paradigm since it is the most common used paradigm in this discipline. This method is used when investigating a phenomenon. A scientist has to go though the following steps, when for example a model's prediction does not mach with experimental evidence.

1. Form a hypothesis;
2. Construct a model and make a prediction;
3. Design an experiment and collect data;
4. Analyze results;

The third and final paradigm *design* has its root in engineering and follows these steps when constructing a system to solve a given problem. The engineers have to iterate these steps when for example test reveals that the requirements for latest version of the system is not met.

1. State requirements;
2. State specifications;
3. Design and implement the system;
4. Test the system

3.2 Potential error factors

During research, there will be some sort of error factors. In this section, these error factors will be explained and how they may occur during research. Intentional errors with misleading results and values is an ethical challenge. This involves the researcher with hasty, ill-informed conclutions that there are no scientific foundations to. The emotional involved researcher can represent an risk factor for these wrong end projection. But constructive contributors will often have a strong engagement for his research field and they will both take care of enthusiasm and objectivity. While accidental errors can occur through the whole research process, and it is errors that can attributable to shortfall of competence to used appropriate methods.[2]

3.3 Guidelines

3.3.1 Research Question Guidelines

- **Theoretical Triangulation:** The research questions were formulated after a thorough Background chapter. 2 and are based on research gaps that were discovered during said review.

- **Research Approach** The nature of the underlying ML model goes under a design research approach. The main purpose of the model is produce a classifier that can predict anomalous network packets. But this thesis also explains the data and generate hypotheses, that are in-depth talked about in section 2.4. It will also be conducted examinations on how well the ML model is able to predict. And by doing these things, it uses an experimental scientific approach on the problem. By this it goes under this abstraction paradigm since one makes an prediction that a generative adversarial approach is possible as a NIDS. [2] [5]

  This thesis is mainly focused about creating a new generative adversarial method, but also on exploring the question on if a GAN can be can be employed as a NIDS in a computer network. Since thesis will not focuses on the robustness of predictive algorithms, but still explore it, we treat the work as an combination of explanatory and predictive research.[5]

- **Iterative process:** The methodology in this thesis can be divided into several steps. Several of these steps are iterative in nature, for example the

  Several of these steps are iterative in nature, for example the pre-process step. The whole process is also iterative in the sense that some steps might circle back to a previous step. During this study, several iterations occurred, especially after the result evaluation which led back to experiments, then to the solution and so on to pre-processing.

3.4 Data Collection Guidelines

Following section will define and explain what kind of guidelines this thesis has followed for data collection.

- **Discuss the nature of data in terms of its variability and reliability:** Network traffic data is the primary source used for this study. In terms of variability, the dataset covers a lot of different attack scenarios as well as background traffic, however as discussed earlier in section 2.2, it might not be representative of newer attacks and therefore not as reliable as a newer dataset. However, since the the solution is a proof of concept on if this type of architecture can be used as an detection system. And given the lack of other dataset that uses raw network flow as data, it as reliable enough for the purpose of this study.[2][5]
- Document the data collection in detail: The data collection has been described in depth in Chapter 4.
- Provide access to the data used: The dataset used is a publicly available dataset which can be downloaded from the authors site [17]

Algorithms evolve, ensure their validity based on disciplines such as computer science and machine learning: All machine learning algorithms used are based in well known Python libraries which are widely used in Machine Learning research.

3.5 Analysis Guidelines

- Document the data pre-processing steps in detail: The data preprocessing steps are documented in full detail in chapter 4
- Algorithms evolve, ensure their validity based on disciplines such as computer science and machine learning: All machine learning algorithms used are based in well known Python libraries which are widely used in Machine Learning research.
- All pre-processing was preform with a combination of the module dpkt[11], numpy[23] and native python. The deep learning adversarial methods were based on the paper Generative Adversarial Networks with some modification by Goodfellow [8], and this was created by using Pytorch[20].

Chapter 4

Proposed Solution

4.1 Proposed solution / algorithm

As seen in the information that was gathered in the Literature Review that DL is not as commonly used in IDS, but it has some usages. Upon that, there are no earlier mentioned about using a GAN architecture as an IDS. This thesis will focus on this specific problem, by explore if its possible to use the characteristic of the GAN as a working IDS. To better understand the possibilities the GAN architecture and possibilities to improve open existing IDS solution. So this thesis presents a solution that uses a Generative adversarial approach on an IDS solution. The main solution is based around a GAN, this neural network is a modified neural network that is made by combine two neural network that competing against each other.

4.1.1 The Generative Adversarial Network

One of these network is the Generator (**G**) and the second one is the Discriminator (**D**). For more in depth on how the GAN work see chapter 2. For this solution the DL network will be fed with a network packet presented as a binary string, as the forward pass of the network occurs, the generator network; instead of generating pictures, it will try to create a new binary string from noise that represent a new network packet that will be uses to trick the discriminator. Following is showing on how the generator network work mathematical. Where the batch size is b, the number of input nodes is shown as n and is the length of one network packet and number of hidden nodes as h. Where this process is illustrated in Figure 4.1

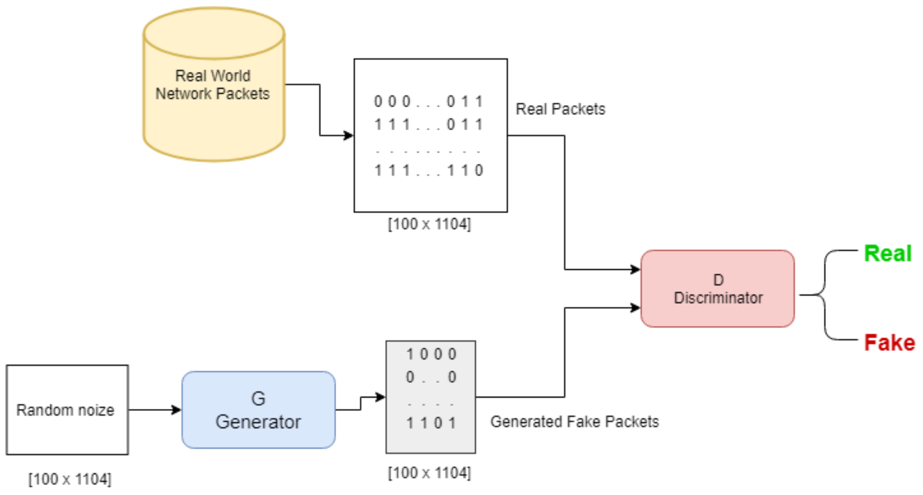


Figure 4.1: The flow of of the proposed solution is illustrated here

Generator Network

The generator takes summation of a randomly created matrix X with the shape of bxn and multiplies it with a weight W1 with a shape of number of (n x h) over the number of batches. Then an bias is added to this equation to make the network able to shift the activation function on the x-axix.

This is illustrated in the first line of in the following equation: 4.1.1. This function will return a new matrix with the shape *b x h*, this matrix is send though a *ReLU* activation function. Where it will return as the same shaped matrix where it returns the max number of ”0” or the number in the matrix illustrated in line two. This matrix *f(X)* will be multiplied with a new weight matrix *W2* where the shape is *h x n* summarised over batches, followed by a new bias added. This step will get repeated the number of hidden layers the generator uses. A *Sigmoid* function is appended to The matrix *z* where each value is squeezed to a number between 0 or 1 since network packets is a long bit string containing 0 and 1. The returned matrix will be the same shape as the first input layer meaning batch size times the length of the packet size; being 100×1104. This was done because the generated network packets had to be the same shape and format as the genuine network packets, that was going to be fed the discriminator.

$$\begin{aligned}
&X = \sum_{i=0}^{b}(\underset{(b\times n)}{X_i} \times \underset{(n\times h)}{W1_i}) + \underset{(h)}{bias}\\
&f(x) = max(0, \underset{(b\times h)}{X})\\
&H = \sum_{i=0}^{b}(\underset{(h\times n)}{X_i} \times \underset{(n\times h)}{W2_i}) + \underset{(h)}{bias}\\
&f(H) = max(0, \underset{(b\times h)}{H})\\
&z = \sum_{i=0}^{b}(\underset{(b\times h)}{f(x)} \times \underset{(h\times n)}{W3_i}) + \underset{(n)}{bias}\\
&\underset{(b\times n)}{Y_{\_}} = \theta(z) = \tfrac{1}{1+e^{-z}}
\end{aligned}$$

Discriminator Network

The discriminator in the DL network is also trained at the same time as the generator, this network will be trained as typical supervised problem. The **D** is fed real network packets in the form as binary strings from the data set, as well as network packets created from the **G**. The **D** then classify the packets as ”real” or ”fake” and update the whole network using propagation. And so by using a GAN architecture it trains a classifier to protect the network as well as a generator to create new attack on the network.

Following is the mathematics operations of the discriminator network where the number of batches is specified as b. Packet size is the same as number of input nodes that are defined as n and the number of hidden nodes in the network as h. The discriminator network's input will be the matrix created from the generator network that contains 100 different generated network packets, with the same shape as the generator. Both the generator and the discriminator use the same formulas the first layer. Matrix X (100 x 1104) is multiplied with a weight matrix $W1$ (1104 x 50), then summed over the number of batches, as in this thesis 100. A random generated bias vector of 50 is then added to this matrix, that returns a matrix that are as the same as the generator send though a *ReLU* activation function. The next layer is identical as the previous except for the shape of the weight W4 that is 50 x 50. The last layer, also known as the prediction layer, consist of the last layers input matrix times the last weight. That is of the shape 50 x 1, where the last output is a vector of one single dimension, for prediction of a network packets are real or fake.

$$X = \sum_{i=0}^{b} (\underset{(b \times n)}{X_i} \times \underset{(n \times h)}{W4_i}) + \underset{(h)}{bias}$$
$$f(x) = max(0, \underset{(b \times h)}{X})$$
$$H = \sum_{i=0}^{b} (\underset{(h \times n)}{X_i} \times \underset{(n \times h)}{W5_i}) + \underset{(h)}{bias}$$
$$f(H) = max(0, \underset{(b \times h)}{H})$$
$$z = \sum_{i=0}^{b} (\underset{(b \times h)}{f(x)} \times \underset{(h \times 1)}{W6_i}) + \underset{(1)}{bias}$$
$$\underset{(b \times 1)}{Y\_} = \theta(z) = \frac{1}{1+e^{-z}}$$

4.1.2 Algorithmic Enhancements

After experimenting with a solution, a new technique was implemented, called the optimiser dataset. This dataset was created by extracting 26 (25 %) of the packets with labelled real attacks, where this dataset was used to optimise the training of the discriminator network. By using this dataset it allowed the discriminator to learn the features of real attacks. By doing so the discriminator was able to use these features to optimise the training for detecting unseen attacks. The optimisation step is explained in detail in Algorithm 4.1

4.1.3 Discussion of the Parameter Space

- **batch_size** = 100 is the parameter that specify how many examples are fed into the network at the same time. In this theses the batch size is 100, meaning that 100 network packets are fed into to network each iteration.
- **g_hidden_size** = 50 This parameter is to specify how many hidden nodes the generator has in it's layer. This makes the hidden layer in the generator has 50 inputs nodes as well has 50 output.
- **d_hidden_size** = 50 This parameter has the same usage as the generators hidden nodes, meaning that the discriminator also has 50 hidden nodes in it hidden layer.
- **d_output_size** = 1 This parameter is to specify the number of output nodes the discriminator has. In this thesis will this parameter be static, since it will only predict if the packets are real or fake.
- **d_learning_rate** = 2e-4 This is to specify the learning rate that the discriminator are using for gradient decent.
- **g_learning_rate** = 2e-4 This is the learning rate for the generator used in gradient decent.
- **new_data_bytes** This parameter is a true false statement that controls if the data should be pre-possessing again with new parameters set by the user.
- **num_databytes** = 100 This parameter is a limit on how many data bytes that are included in the pre-possessing. Meaning that the code limit on how large the packets is padded to and fed into the network.
- **num_packets_limit** = None This is also a limit. This limits how many packets of the data set is taken into the pre-possessing. In this thesis it set to None, meaning that every packets how the training set is included to possessing.

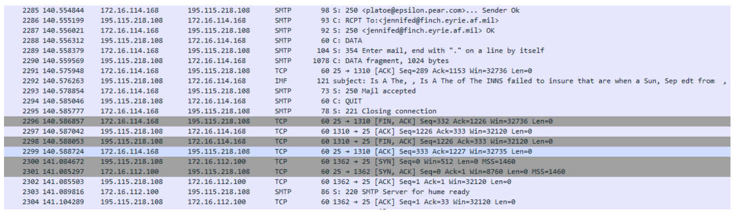
- **pcap_path path** = "./data/training.pcap" This string parameter is for specifying where the training set capture file is located on disk.
- **save_location** = "logs/" This string parameter is to specify where the log files for Tensorboard would be saved.
- **epochs** = 2 Number of epochs is specify how many times the network is to see the whole dataset. So in this thesis the number of times the notwork would the whole dataset is 2 times. Epochs is also used here to take the measurement of the performance over this whole network, as well as using the generator to generate sample packets.
- **d_steps** = 5 This is 'k' steps in the original GAN paper.[8] This is a integer parameter that determent's how frequent the discriminator is trained.
- **g_steps** = 1 This is also a integer parameter, that determent's how frequent the generator is trained.

4.2 Prototype

4.2.1 Pre-processing

In this section the pre-processing of the solution will be explained. When it comes to working with Neural Networks or machine learning in general, pre-processing is very important. This is because a Neural network is a type of supervised machine learning, meaning that the network needs to be told what is wrong and what is right. So by this the data that are fed into the network needs to be right and pre-processed in a way that the network can "understand" it. Since the GAN architecture is a combination of supervised and unsupervised learning, this is also the case for this architecture.

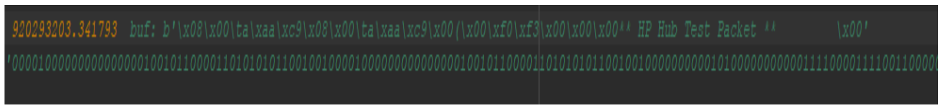
The data used in this thesis is gathered from the DARPA dataset mentioned in section 2.2. DARPA 1999 dataset was downloaded from their site, only one week of data was downloaded, and since this solution occurs inside a computer network, only the inside capture file was downloaded and used. After the dataset was downloaded it was extracted and open in Wireshark[4] to check if it was possible to make it readable for the human eye as well as if it was the type of data needed for this study as shown in figure 4.2.



```
2285 140.554844   172.16.114.168    195.115.218.108   SMTP     98 S: 250 <platoe@epsilon.pear.com>... Sender Ok
2286 140.555199   195.115.218.108   172.16.114.168    SMTP     93 C: RCPT To:<jennifed@finch.eyrie.af.mil>
2287 140.556021   172.16.114.168    195.115.218.108   SMTP     92 S: 250 <jennifed@finch.eyrie.af.mil> OK
2288 140.556312   195.115.218.108   172.16.114.168    SMTP     60 C: DATA
2289 140.558379   172.16.114.168    195.115.218.108   SMTP    104 S: 354 Enter mail, end with "." on a line by itself
2290 140.559569   195.115.218.108   172.16.114.168    SMTP   1078 C: DATA fragment, 1024 bytes
2291 140.575948   172.16.114.168    195.115.218.108   TCP      60 25 → 1310 [ACK] Seq=289 Ack=1153 Win=32736 Len=0
2292 140.576263   195.115.218.108   172.16.114.168    IMF     121 subject: Is A The, , Is A The of The INNS failed to insure that are when a Sun, Sep edt from  ,
2293 140.578854   172.16.114.168    195.115.218.108   SMTP     73 S: 250 Mail accepted
2294 140.585046   195.115.218.108   172.16.114.168    SMTP     60 C: QUIT
2295 140.585777   172.16.114.168    195.115.218.108   SMTP     78 S: 221 Closing connection
2296 140.586857   172.16.114.168    195.115.218.108   TCP      60 25 → 1310 [FIN, ACK] Seq=332 Ack=1226 Win=32736 Len=0
2297 140.587042   195.115.218.108   172.16.114.168    TCP      60 1310 → 25 [ACK] Seq=1226 Ack=333 Win=32120 Len=0
2298 140.588053   195.115.218.108   172.16.114.168    TCP      60 1310 → 25 [FIN, ACK] Seq=1226 Ack=333 Win=32120 Len=0
2299 140.588724   172.16.114.168    195.115.218.108   TCP      60 25 → 1310 [ACK] Seq=333 Ack=1227 Win=32735 Len=0
2300 141.084672   195.115.218.108   172.16.112.100    TCP      60 1362 → 25 [SYN] Seq=0 Win=512 Len=0 MSS=1460
2301 141.085297   172.16.112.100    195.115.218.108   TCP      60 25 → 1362 [SYN, ACK] Seq=0 Ack=1 Win=8760 Len=0 MSS=1460
2302 141.085503   195.115.218.108   172.16.112.100    TCP      60 1362 → 25 [ACK] Seq=1 Ack=1 Win=32120 Len=0
2303 141.089816   172.16.112.100    195.115.218.108   SMTP     86 S: 220 SMTP Server for hume ready
2304 141.104289   195.115.218.108   172.16.112.100    TCP      60 1362 → 25 [ACK] Seq=1 Ack=33 Win=32120 Len=0
```

Figure 4.2: Dataset explored with Wireshark

After the data was extracted and explored, the next part was to read the data and format it in Python, so that the network could work with it. For this task the Python package dpkt[11] was used. dpkt is a pcap parser and creator. This package was then used to read the pcap file (tcpdump) to split the time and data packets into a iterative. In this solution the time of the packets was not going to be used so it got dumped. Further the packets was then converted from tcpdump syntax to a long binary string shown in figure 4.3, since the network could not work with characters. The data was then saved to the drive as a binery string for easier usage of the untrimmed data.



```
920293203.341793  buf: b'\x08\x00\ta\xaa\xc9\x08\x00\ta\xaa\xc9\x00(\x00\xf0\xf3\x00\x00\x00^^ HP Hub Test Packet ^^        \x00'
'0000100000000000000010010110000110101010110010010000100000000000000010010110000110101010110010010000000000101000000000001111000011110011000
```

Figure 4.3: Tcpdump syntax formated to bit string

Since the neural network can't be fed with data that are not the same length, one known technique used in this solution to overcome this, was padding. Padding means that adding some type of value that would be added to the bit string to make them of same length. So for this solution the padding was done by adding **"0"** at the end of each string to match the longest packet. It was also important to check if the chosen padding didn't interfere interfere with the data used. Here is the argument *num_databytes* used in this part of the solution, as explained in the section. 4.1.3. Further is the now trimmed data converted to a numpy array and saved to the disk for faster easier pre-prossesing later. This part of the pre-prossesing will then return a numpy array with the shape of the batch size x packets size, both explained in the 4.1.3 section.

The last part of the pre-possessing for the data used for training, was the matrix from earlier sent into a class that would use the matrix together with the chosen batch size specified in 4.1.3. To split the data into mini batches. First the data are split into a training/testing set with the last 1000 packets being the testing set. The larger part of the data was then split into smaller batches, as this was the format the data was going to be fed into the network. A pointer variable was also created to hold control on which of the batches was active. Later on was a function of this class called to return the next part of the data in the form as a new batch as wall as add to the pointer variable. So when this class was called the next batch was returned. There was also a function for the generator that created the random noise the generator used to create it's packet, this was a lambda function $lambdam, n : torch.rand(m, n)$. This function just takes in to parameters that makes a matrix that is populated of random numbers from a uniform distribution.

The next part of the pre-processing is the gathering and creation of the validation data, this data was used to rate how good the discriminator network was able to detect real attack that was discovered and labels in the DARPA dataset. The data that was going to be used in the validation was also downloaded from the DARPA website. The training dataset consisted of one day in the first week of the time the dataset was created. The validation set was consisted of two whole weeks (4th and 5th week). As explained in section 2.2, these two week had 201 instances of about 56 types of attacks distributed among them. All of the .tcpdump files was downloaded and converted to .pcap files, then from the same site every instance of the different attacks was converted to a .csv file. With the time of the attacks in the .csv file, it was used to go through every .pcap file and look for the correct time stamp. When the time stamp of the attacks match with the packets of the .pcap files it would add it to a list, and follow the same procedure as the training data.

4.2.2 Network Models

The Network Model is the most central section of the machine learning of this thesis. The following section contains all specification and all the connection of the different layers defined in the network. First will the architecture of the first part of the GAN neural network be describe, as it is the first part where the processed data are used. It will describe of the architecture is specified using pytorch API. Secondly will the discriminator network be described on how the data flow though this network after it as gone tough the generator, as well as how data is classified as real or fake data.

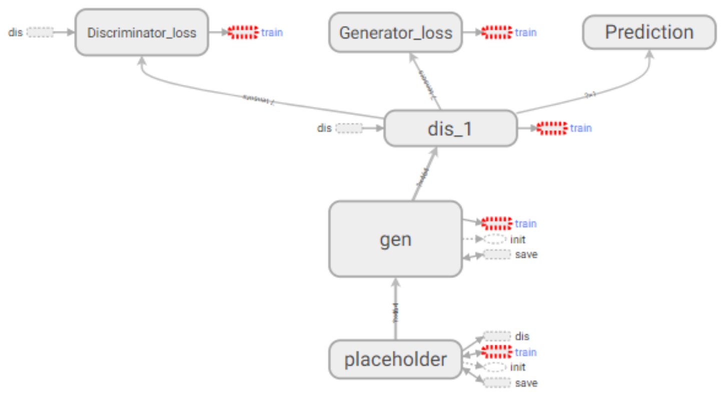


Figure 4.4: Generative Adversarial Network Model

The architecture of the whole the solution is shown in figure 4.4, while a more graphical presentation of the flow of the network is shown in Figure 4.1 and explained in more detail in section 4.1.1. This makes it so that the flow of the network is easier to apprehend. The whole process starts with the generator gets a type of noise, then it tries to generate a packet, that are send to the discriminator alternate with a real packet, to classify it as real or fake. Then the loss is calculated, here binary Cross Entropy is used to calculate the losses explained in the background chapter: 2. Then Using Adam Optimiser [13] to minimize that loss by calculating the gradients and backpropocate. In the training time, the network would also train a small part of the real attacks to utilise fully supervised learning. This was done because the network would have a much faster learning on what kind of packets was real attack, since the discriminator knows what a packet looks like without attacks,and the other way around. Instead of relying fully on the generator model, to generate packet that looks like packets with attacks and so every 10th training step, the model will update the gradients based optimiser dataset from section 4.1.2. The networks training operation is shown in Algorithm 4.1

Algorithm 4.1 Minibatch stochastic gradient decent training of a generative adversarial net [8]. The number of steps to apply to the discriminator and generator, d_steps, g_steps are hyper-parameters as explained in section: 4.1.3. d_steps=5 and g_steps=1 when focusing on training discriminator.

```
for $k$ number of minibatches do
    for $d\_steps$ do
        • Sample minibatch of $m$ noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from noise prior $p_g(z)$
        • Sample minibatch of m examples $\{x^{(1)}, ..., x^{(m)}\}$ from data distribution $p_{data}(x)$.
        • Update the discriminator by ascending its stochastic gradient:
            $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\Big(x^{(i)}\Big) + \log\Big(1 - D\Big(G\Big(z^{(i)}\Big)\Big)\Big) \right].$$
    end for
    for $g\_steps$ do
        • Sample minibatch of $m$ noise examples $\{z^{(1)}, ..., z^{(m)}\}$ from data generating distribution $p_g(z)$.
        • Update the generator by ascending its stochastic gradient:
            $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\Big(1 - D\Big(G\Big(z^{(i)}\Big)\Big)\Big).$$
    end for
    if $k \mod 10 == 0$ then
        • Sample minibatch of $m$ examples $\{y^{(1)}, ..., y^{(o)}\}$ from optimising dataset $p_{data}(y)$.
        • Update the discriminator by ascending its stochastic gradient:
            $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\Big(y^{(i)}\Big) + \log\Big(1 - D\Big(G\Big(z^{(i)}\Big)\Big)\Big) \right].$$
    end if
end for
```

The gradient-based updates can use any standard gradient-based learning rule. In this thesis momentum was used.

In algorithm 4.1 the training steps taken by the network are written in a mathematical terms. To explain in more detail, the training consist of three main part. The discriminator step, the generator step and the optimisation step. First in the discriminator step, it sample a batch of m numbers of network packets from the real dataset, where m are 100 numbers of packets. It also samples the same number of generated network packets from the generators distribution. With each of these samples it updates using stochastic gradient decent by minimising the loss. Where the discriminators choice is regarded higher then the generator specified in algorithm. This will be repeated as many times as the parameter d_steps is specified.

The second part is the training of the generator network. The network samples m number of noise examples from a random distribution. Then uses this to update the generator by ascending its stochastic gradient by minimising the loss where the generator decision is high regarded. This step is repeated as many times as the parameter g_steps is specified.

The last step in the training process is new for normal generative adversarial network, where the details of are in section 4.1.2. This step will only happens once per tenth iteration. An sample of packets where real attack occured extracted from the optimising dataset also specified in section 4.1.2. Then in this step it will again update as the first step by minimising it's loss. These three steps would be repeated the number of batches in the dataset also 14 913 number of batches.

4.2.3 Post-processing

As explained in the previous section pre-processing is important, but so are the post-processing. In this thesis the post-processing consists of the creation of the metrics that are for measuring the accuracy of the network. The logging of the different values used to score the network and also the post-processing of the generated packet from the generator network.

Since the pytorch API not yet has a implementation of visualisation of graphs, a community made implementation of tensorboard to pytorch was used. The syntax of tensorbordX[14] was almost identical to the normal tensorboard. This made it easier for to implement this into the code. First two writers was created, one that would monitor all the values from the training time, and a writer for the values for the testing time. Both of them was saved to the two folders, with slightly different names. The name itself was generated by the "$day, month, year - hour, minutes$". and the same, only with "$\_test$" behind.

After every training step, the writer added three scalars, that contained the different losses of the network; d_real, d_fake and g that are explained in section: 4.1.3. The next writer the prediction on how well the discriminator can predict real packet and generated packets. This is where the accuracy comes into play. As seen in the following function, that defines how the accuracy for all the different test are made.

$$Accuracy_1 = \frac{np.sum(np.equal((np.around(predictions)), np.zeros\_like(predictions)))}{len(predictions)}$$

$$Accuracy_1 = \frac{np.sum(np.equal((np.around(predictions)), np.ones\_like(predictions)))}{len(predictions)}$$

These two equations, are used multiple times in this thesis and both of them compares two arrays. The fist is a rounded array of the prediction the discriminator network, and its compared with a array like the shape filled with zeroes or one, to what is needed.In this thesis so was these function used to calculate for prediction on real and generated packets, as well on prediction on training set and the prediction on the non seen before attacks. Lastly the writer monitored the distribution of the weights and biases of the network, shown in figure: 4.5. This show the weights and bias values distribution over time.

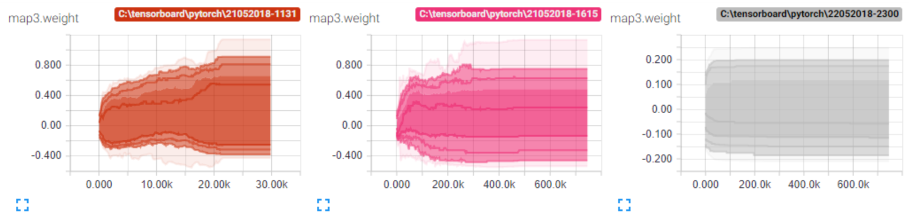


Figure 4.5: Weights distribution over time.

Lastly of the post-processing the conversion of the generated packets to human readable file to open in analyse. After each epoch the generator makes a prediction and tries to create a packet that are alike the real data, since this a bit string it needed to be converted to bytes. Since the output of the generator was between one and zero, it was rounded to its closet. After every bit string was converted to bytes, the module dpkt was used to write the bytes to a .pcap file for later be open in Wireshark.

4.3 Justification of Claim to Originality

As seen in background of this thesis, this type of approach to and IDS is not tried before, as well as using a generative adversarial network to train a IDS on newly generated packets combined with regular supervised learning and optimisation of the algorithm. This thesis explores new possibilities in network security as well as challenging the ability of the generative adversarial network.

Chapter 5

Experiments and Results

5.1 Experiment Setup

Is has been used a wide range of parameter settings and data set configurations to evaluate this approach. In this section, the representative results on the generative adversarial networks reported.

5.2 Data Gathering & Prepossessing

The dataset used in this thesis was gathered from the DARPA web site as explained in 4. The training dataset contained 1.491.300 network packets and the validation dataset contained 105 packets of attacks. The validation set was split into two smaller dataset, the validation dataset containing 75 % of the packets used for validating on of well the network performed. The optimisation dataset containing 26 of the packets or 25 % to optimise the learning of the discriminator that are explained in the solution Chapter 4. The network will only be trained using the training set, and validated using the validation set.

5.3 Experimental Result

The DARPA dataset was used with its 1.491.300 networks packets, the attacks spanning was over 105 to evaluate the ability of a generative adversarial network to predict and detect malicious network traffic in the form as binary strings. The generators ability to recreate network packets similar the real data. The network was tested without the optimisation techniques using real attacks and with. With the focus was on the generator network, the optimisation dataset was used.

In the discriminator experiments all the networks was trained for 2 epochs (the algorithms were allowed to pass over the training dataset 2 times). The reason that number of epochs here is so low is because of the network converges before the 2 epochs are done. However the experiments focusing on generating new network packets, was trained with 10 (the algorithms were allowed to pass over the training dataset 10 times) epoch, since the training of the generator is unsupervised learning, therefore does the training need more time to converge. After training, the performance of the two types of network was validated using a dedicated validation dataset that contained 75% of the labelled attacks

5.3.1 Discriminator Based Intrusion Detection System

In the first experiment, documented in Table 5.1, was not using the optimisation dataset, since this was the first of experiments as well to compare those two. Further network packets with with the number of bits being 1104 was fed into the network in batches of 100 packets each training step. This setup trained the discriminator 5 times more than the generator, for ensure that the discriminator ability was focused in the training. It was observed that this architecture scores significantly high on several of the tests. As shown in the table, this approach provides the highest accuracy on the validation set (100%) in both of the training set and predicting what real packets was, but also in the newly generated packets.

Meanwhile the test it didn't score well in, was the detection of real attacks. This model was not able to detect what a packets that was real attacks. This is where the validation dataset fed fed into the network all at once, then the network tried to classify which of them was packets with and without attacks. This problem was solved in the next experiment, by using the optimisation step in the training.

Table 5.1: Accuracy of packet manipulation detection of Discriminator network after 2 epochs not using optimisation dataset

Training Accuracy	Test Real Data	Generated Accuracy	Real Attacks Accuracy
100%	100%	100%	2%

In the next setup was also trained on the same training set, but also the optimisation technique used, to mitigate the low score of the detection of attacks. This small dataset consisted of 26 real attacks extracted from the validation dataset. Also as the previous experiment, the discriminator was trained for 5 times more often than the generator. The result is reported in Table 5.2. In this experiment the accuracy the training match the previous experiment with a high accuracy on detection on packets real packets, but also on the generated packets, meaning that this setup also had a very high accuracy to detect what was real data and what was generated data. As seen in the table, the accuracy of real attacks went up from 2 % to 100%. Meaning that by introducing the network for data that was labelled real attacks, and trained on this data, the network was able to detect 100 % of all the attacks in the validation set. This shows that this setup was able to detect packet manipulation with a certainty of 100%.

Table 5.2: Accuracy of packet manipulation detection of Discriminator network after 2 epochs using optimisation dataset

Training Accuracy	Test Real Data	Generated Accuracy	Real Attacks Accuracy
100%	100%	100%	100%

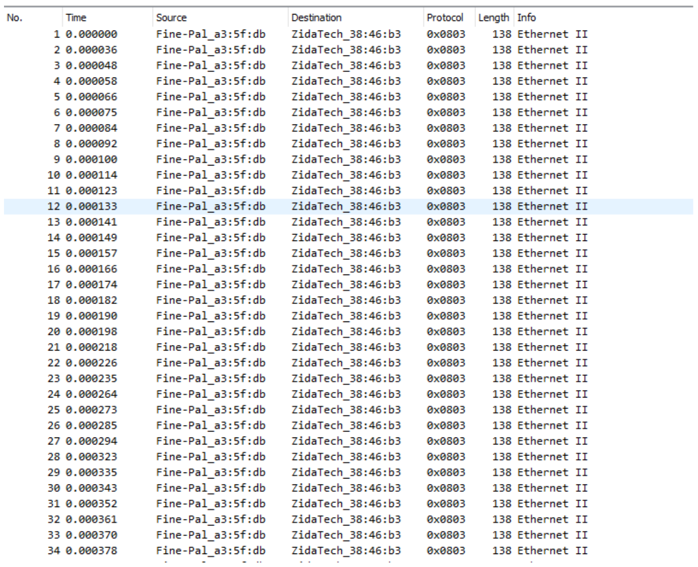
5.3.2 Network Packets Creation With Generator Network

For the next experiment, the focus shifted from the discriminator ability to detect attacks, over to the generative part of the network; the generator. This was done by changing the number of times the generator and discriminator was trained. Unlike the previous experiments, this time the generator was trained 1 times as often as the discriminator to set the training focus more on the generator. The number of epochs in the training was also heighten from 2 to 10, since the generator is only learning from the discriminator, meaning that training would take longer time. The result of the discriminator is shown in Table 5.3. Since this experiment also uses the optimisation dataset, the detection of real attacks also converges as 100 %. meanwhile the accuracy of the generated packets has fallen all the way down to 0 % meaning that the packets the generator is creating is tricking the discriminator to classify them as real packets.

Table 5.3: Accuracy of packet manipulation detection of Discriminator network after 10 epochs focused on generation

Training Accuracy	Test Real Data	Generated Accuracy	Real Attacks Accuracy
99.9%	100%	0%	100%

To analyse the result in this experiment, there was 10000 packets created by the trained generator. These packets was crated by the generator to look as alike as the real data as possible. Some of the packets created is shown in Figure 5.1, As seen there, all the packets are alike. This concerns all of the rest of the generated packets. This is a very known problem when working with GAN's, is mode collapse. Meaning that the generator, was able to learn what kind of packets that's tricks the discriminator every time, and sticks to this packet. Since it will get a lower loss, by tricking the discriminator every time with this type of packet. Further analysing generated data, shows that out of the 10000 packets created, everyone passed Wireshark's syntax check, meaning that the packets are defines as real.



```
No.     Time      Source             Destination         Protocol  Length Info
      1 0.000000  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
      2 0.000036  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
      3 0.000048  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
      4 0.000058  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
      5 0.000066  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
      6 0.000075  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
      7 0.000084  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
      8 0.000092  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
      9 0.000100  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     10 0.000114  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     11 0.000123  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     12 0.000133  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     13 0.000141  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     14 0.000149  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     15 0.000157  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     16 0.000166  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     17 0.000174  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     18 0.000182  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     19 0.000190  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     20 0.000198  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     21 0.000218  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     22 0.000226  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     23 0.000235  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     24 0.000264  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     25 0.000273  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     26 0.000285  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     27 0.000294  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     28 0.000323  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     29 0.000335  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     30 0.000343  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     31 0.000352  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     32 0.000361  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     33 0.000370  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
     34 0.000378  Fine-Pal_a3:5f:db  ZidaTech_38:46:b3   0x0803    138 Ethernet II
```

Figure 5.1: Capture of generated packets in Wireshark

To conclude Figure 5.2 shows the accuracy of the model when the detection was in focus, while Figure 5.3 shows the loss of the generator when the generator training was in focus. As seen in figure 5.2 the loss of both the discriminator and generator is very unstable at the beginning. This is because of the nature of the GAN, meaning that both of them is trying to lower its loss and at the same time heighten the loss of the other part. So this min/max game is the reason that the loss are so unstable. In figure 5.3 the discriminators loss over the fake data has sky rocketed while the generator loss as fallen to a very low value, showing that the generator is creating packets that are tricking the discriminator. Is was observed that the losses of these to converges at the end of the training time.

In the final Figure 5.4 it is observed that when focusing on detection and with the use of the optimisation techniques, the accuracy of the models ability to detect real attacks are much higher (blue graph) compared to the model not using this technique (light blue graph). Here its also observed that the network don't need any more training then 2 epochs, since the network converges before 2 epochs has gone pass.

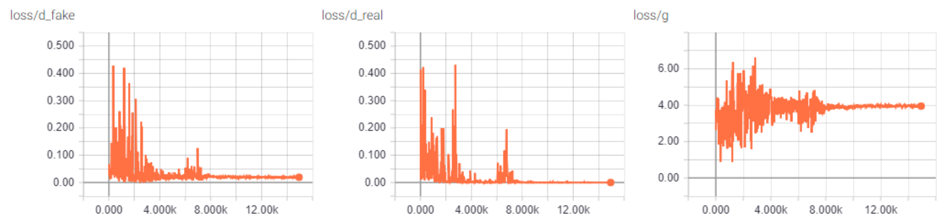


Figure 5.2: Loss over network when detection in focus

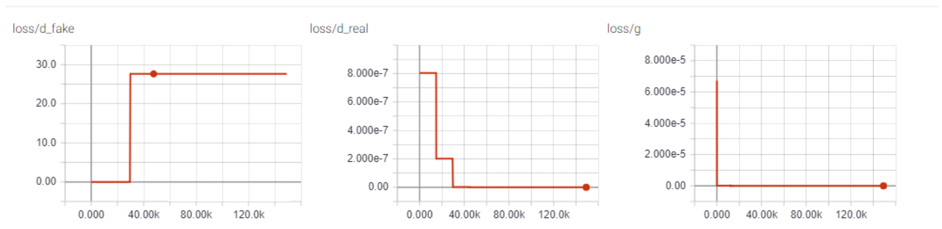


Figure 5.3: Loss over network when generation in focus

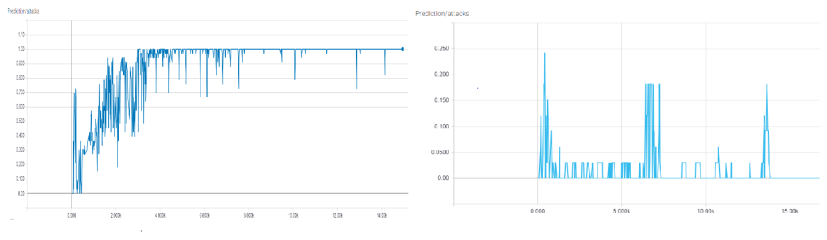


Figure 5.4: Accuracy on real attacks of between the optimised and not optimised

Chapter 6

Conclusion and further work

6.1 Summary of Results

The problem of using a generative adversarial network as an intrusion detection system is hard but still plausible. This is because the when using an generative adversarial network as an IDS the network has to learn how to detect any known attacks. But also be able to use the generator to create new types of packets to trick detection. In this thesis it was proposed an advanced deep learning model, based on generative adversarial networks, that have a very high detection rate of attacks in a computer network. In particular, the experiments that have been reported in this thesis shows that a deep learning network more specifically, a fully connected generative adversarial, performs surprisingly well on the task on detect packet manipulations in a computer network.

It can possible be explained by the capability of the GAN to be able to extract and learn the unique features of the real network packets as well being optimised by learning of packets that are libelled as attacks. Together with the generators ability to created new types of packets based on the discriminators choice, that simply trying to predict if the packet is real or fake. Indeed, this solution is able when using a optimiser dataset to score the highest possible on detection task by achieving a converged accuracy of 100%

The generator on the other hand. The overall detection rate dropped a little but over time because of the optimisation step the detection peeked at 100 % also. While the detection rate of generated packets fell down to 0%. The experiments had the focus on how well the generator was able to generate network packets similar to them found in a real computer network. On this task the this solution was able to created packets that was similar to the packets found in a real computer network. All the packets created by this solutions generator was all validated by Wiresharks's syntax detection. But even if every packets created, passed that test, the generator experienced mode collapse, meaning that its only created exact the same packets each time. This was because the generator at learned that these packets always slipped pass the detection of the discriminator. Mode collapse is a known problem for this type of generative adversarial network, this problem is handled in other types of GAN's. Since this thesis was to explore the possibility to use a GAN as a intrusion detection system and this problem was not attempted before, therefor was this problem not tackled in this thesis.

This solution has both strengths and weaknesses. It has some great results in these experiment and are able to converge at a 100 % detection rate. But this result was achieved in a controlled environment with specific rules and experiments. In a real life scenario it would properly have different results. Since in a real-life scenario unexpected factors could occur, and play a part in the real network. One of these factors can occur while training the network. Since this solution is trained using the normal network traffic to learn what how normal packets look like. If the computer network is attacked under this learning process, the solution could have learned that this attack was part of the network normal traffic. Also for the optimisation dataset, this dataset contains real attacks that are used to optimise training in this solution. Meaning that this dataset need to be populated with real attacks that already is documented and labelled by security experts. The DARPA99 dataset is an old dataset that doesn't contain the newer attacks that are seen today.

These weaknesses can be solved using different methods. The problem that the DARPA99 dataset is old, is not a very strong weakness. Since there are other newer dataset out there, but in this study the task was to explore the possibilities of a GAN as a IDS, so choosing the DARPA99 dataset was most convenient since it is publicly available on the internet and the concept stay the same. About the weakness where it is hard to get hold of updated label attacks, can also be solved. This is where the generator can be used to solve this problem. This thesis has already shown that the generator as the ability to create packets that looks so alike normal packets that it can trick detection. If the problem of mode collapse is address the generator could be used for generating vast different and new packets that the discriminator could learn from. With other word take away the requirement on always having security experts constantly updating databases of known attack.

6.2 Conclusion

So based on the research and the experiments done in this thesis. This novel architecture, and the results is a significant step toward building better intrusion detection systems that are based on machine learning. Or more specific a Generative adversarial Network based detection system. Since this model was able to detect wide range of real computer networks attacks, with a confidence of a 100 %, despite its simple deep learning architecture. This solution was also able to trick the discriminator despite the generator was experiencing mode collapse, by creating computer network packets similar to real data. Meaning that if this problem is solved, this solution as the ability to create even more abstract and new packets. And this could address the problem of creating a intrusion detection dataset, since a trained generator could create thousands of network packets that was alike real attacks so it was no longer needed to capture enormous data to create good dataset of attacks and help train the discriminator detecting attacks. Perhaps even discovering to this date unknown attacks.

So to conclude this, with the abilities of generative adversarial networks has, used in a intrusion detection systems, will make machine learning to a very powerful tool that can help revolutionise the way network security is managed in the future.

6.3 Further Work

In the future, it is intended to train the model on different dataset to see how well this solution is able to detect attacks in a new environment. Also try to solve the mode collapse problem that this solution experienced. By doing so, it may result the generator becoming better at creating different networks packets. This will help with networks security in the form of either training the discriminator by generating large amount of data, that can be used in training.

Or in the form of the generator creating packets, that can help networks analysts discovering new vectors of attacks and how to prevent them. Since network packets are a sequence of bits, that are coherent to the bits around, it may be a good idea to implement a recurrent neural network in both the generator and the discriminator, since this will give the model the ability to remember abstract concepts over the sequence, making it better at creating network packets as well as detecting manipulation of them. At last it is possible to use and other type of generative network for the creation of network packets. Instead of using GAN architecture it can be change to an auto encoder networks instead, to see which of the two models is better at creating network packets.

By doing further work and try to solved these problems that occurred during this study. It will be safe to say that it will have a heavily impact on the way network security is done in the future.

Bibliography

[1] A. F. Agarap, “Deep learning using rectified linear units (relu),” *CoRR*, vol. abs/1803.08375, 2018. [Online]. Available: http://arxiv.org/abs/1803.08375

[2] E. Befring, *Forskingsmetider i utdanningsvitenskap*, 1st ed. Cappelen Damm AS, 2015.

[3] A. L. Buczak and E. Guven, “A survey of data mining and machine learning methods for cyber security intrusion detection,” *IEEE Communications Surveys Tutorials*, vol. 18, no. 2, pp. 1153–1176, Secondquarter 2016.

[4] G. Combs. Wireshark go deep. [Online]. Available: https://www.wireshark.org/

[5] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, “Computing as a discipline,” *Commun. ACM*, vol. 32, no. 1, pp. 9–23, Jan. 1989. [Online]. Available: http://doi.acm.org/10.1145/63238.63239

[6] S. García, M. Grill, J. Stiborek, and A. Zunino, “An empirical comparison of botnet detection methods,” *Comput. Secur.*, vol. 45, pp. 100–123, Sept. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2014.05.011

[7] A. Ghorbani, W. Lu, and M. Tavallaee, *Network Intrusion Detection and Prevention - Concepts and Techniques*, 01 2010, vol. 47.

[8] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative Adversarial Networks,” *ArXiv e-prints*, June 2014.

[9] I. J. Goodfellow, “NIPS 2016 tutorial: Generative adversarial networks,” *CoRR*, vol. abs/1701.00160, 2017. [Online]. Available: http://arxiv.org/abs/1701.00160

[10] Information and I. Computer Science University of California. (1997) Kdd cup 1999 data. [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[11] kbandla. (2017) dpkt. [Online]. Available: https://github.com/kbandla/dpkt

[12] A. D. Kent, "Comprehensive, Multi-Source Cyber-Security Events," Los Alamos National Laboratory, 2015.

[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[14] lanpa. (2018) dpkt. [Online]. Available: https://github.com/lanpa/tensorboard-pytorch

[15] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," vol. 521, pp. 436–44, 05 2015.

[16] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 396–404. [Online]. Available: http://papers.nips.cc/paper/293-handwritten-digit-recognition-with-a-back-propagation-network.pdf

[17] M. I. O. T. LINCOLN LABORATORY. (2017) Darpa intrusion detection evaluation. [Online]. Available: https://www.ll.mit.edu/ideval/data/1999data.html

[18] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks," *CoRR*, vol. abs/1611.02163, 2016. [Online]. Available: http://arxiv.org/abs/1611.02163

[19] U. of New Brunswick. (2017) Intrusion detection evaluation dataset (cicids2017). [Online]. Available: http://www.unb.ca/cic/datasets/ids-2017.html

[20] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[21] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” *CoRR*, vol. abs/1511.06434, 2015. [Online]. Available: http://arxiv.org/abs/1511.06434

[22] S. Saad, I. Traore, A. A. Ghorbani, B. Sayed, D. Zhao, W. Lu, J. Felix, and P. Hakimian. (2011) Detecting p2p botnets through network behavior analysis and machine learning”, proceedings of 9th annual conference on privacy, security and trust (pst2011). [Online]. Available: https://www.uvic.ca/engineering/ece/isot/datasets/

[23] O. Travis E, “A guide to numpy,” 2006. [Online]. Available: http://www.numpy.org/

Appendix A

Source Code

.5

```
# -*- coding: utf-8 -*-
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

import numpy as np

import time
import requests
from os import path,scandir
import datetime


import dpkt
```

```python
from tqdm import tqdm
import tarfile
import logging
import pickle
from tensorboardX import SummaryWriter
import pandas as pd

# ##### Pre-prossesing:

def logging_start():


    # create logger
    logger = logging.getLogger(__name__)
    logger.setLevel(logging.DEBUG)
    logging.basicConfig(level=logging.DEBUG)

    # create console handler and set level to debug
    ch = logging.StreamHandler()
    ch.setLevel(logging.DEBUG)

    # create formatter
    formatter = logging.Formatter('%(asctime)s:
     ↪  %(levelname)s: %(message)s')

    # add formatter to ch
    ch.setFormatter(formatter)

    # add ch to logger
    logger.addHandler(ch)
    return logger

logger = logging_start()

class Data_preporation():
    def __init__(self,data,batch_size):
        self.data=data
        self.batch_size=batch_size
        self.pointer = 0

        self.data=self.data[:-1000]
        self.test_data=self.data[-1000:]
        self.batch_creation()

    def batch_creation(self):
        self.num_batches = self.data.shape[0] //
         ↪  self.batch_size
        #print(self.num_batches)

        self.x_train = self.data[:self.num_batches *
         ↪  self.batch_size]
        self.x_batches = np.split(self.x_train,
         ↪  self.num_batches)
        #print(x_train.shape)
        #print(x_batches)

    def next_batch(self):
        #x,y = db.next_batch()
        self.x_batch = self.x_batches[self.pointer]
        self.pointer += 1
        # #x_test, y_test =
         ↪  self.x_test_batches[self.pointer],
         ↪  self.y_test_batches[self.pointer]
        # return x, y, self.x_test, self.y_test
        return self.x_batch,self.test_data

    def reset_batch_pointer(self):
        self.pointer = 0

    def get_num_batches(self):
        return self.num_batches


def
 ↪  data_to_bitstream(num_packets_limit=None,num_databytes=None,new_nu

                      ↪  pcap_file="./data/training.pcap"):
    if num_databytes is None:
        packet_size = None
    else:
        packet_size = 304 + (num_databytes*8)

    if path.isfile("./data/preprocessed_bits.pkl")and
     ↪  new_number_data_bytes==False:

        logging.info("Using saved processed data...")
        trimmed_bits_numpy =
         ↪  np.load(open("./data/preprocessed_bits.pkl",
         ↪  "rb"))
        return trimmed_bits_numpy,trimmed_bits_numpy.shape[1]
    elif (path.isfile("./data/untrimmed_bytes.pkl")):

        logging.info("Using saved untrimmed processed data...")
        untrimmed_bytes_list = np.load(open("./data/untrimmed_bytes.pkl", "rb"))

    else:
        logger.info("Reading Packets from pcap file %s" % pcap_file)

        pcap = dpkt.pcap.Reader(open(pcap_file, 'rb'))
        TCP_packets = []
        count = 0



        untrimmed_bytes_list=[]
        for ts, buf in tqdm(pcap):
            bit_string=""
           # print(type(buf))
            for bytes in list(buf):

                bit_string += '{0:08b}'.format(bytes)

            #print("packet size",len(bit_string))
            untrimmed_bytes_list.append(bit_string)
            count+=1

            if count ==num_packets_limit and
             ↪  num_packets_limit is not None:
                break
        pickle.dump(untrimmed_bytes_list,
         ↪  open("./data/untrimmed_bytes.pkl", "wb"),
         ↪  protocol=4)


    max_bytes =
     ↪  int(len(max(untrimmed_bytes_list,key=len)))
    if packet_size is None:
        packet_size=max_bytes

    #print(packet_size > max_bytes)
    #print()
    assert packet_size <= max_bytes, "The packets size
     ↪  cant extend maximum bytes, found %s > %s"% (
        packet_size,max_bytes,)


    print("\rLimit on Number of Packets: {}, Limit on
     ↪  Data Bytes: {}, Packet Size:
     ↪  {}".format(num_packets_limit,


    #logging.info()
    trimmed_bits_list=[sublist[:packet_size] for
     ↪  sublist in untrimmed_bytes_list]
    #print(trimmed_bits_list[:2])

    #print(len(max(trimmed_bits_list, key=len)))
    #print(len(trimmed_bits_list[0]))
    #print((untrimmed_bytes_list[0]))

    trimmed_bits_list = [[(x + "0" * (packet_size -
     ↪  len(x)))] for x in tqdm(trimmed_bits_list)]
    # print(trimmed_bits_list[:2])
    assert len(trimmed_bits_list) ==
     ↪  len(untrimmed_bytes_list), "Trimmed bits must
     ↪  be the same len as untrimmed, " \
                                                "found
                                                 ↪  %s
                                                 ↪  >
                                                 ↪  %s"
                                                 ↪  %
                                                 ↪  (len(trimmed_bits_
                                                 ↪  len(untrimmed_byte

    #print(len(max(trimmed_bits_list, key=len)))
    #print(len(max(trimmed_bits_list, key=len)))

    trimmed_bits_numpy =
     ↪  np.asanyarray([list(map(int,string)) for lists
     ↪  in tqdm(trimmed_bits_list)
                                       for string in
                                        ↪  lists])
    print(trimmed_bits_numpy)

    assert trimmed_bits_numpy.shape ==
     ↪  (num_packets_limit,packet_size) or
     ↪  num_packets_limit ==None, "" \
                                        "Trimmed data
                                         ↪  dont match
                                         ↪  origanal
                                         ↪  data,
                                         ↪  found %s >
                                         ↪  %s" % (
        trimmed_bits_numpy.shape,
         ↪  (num_packets_limit,packet_size))


     ↪  #trimmed_bits_numpy.dump("./data/preprocessed_bits.pkl")

     ↪  pickle.dump(trimmed_bits_numpy,open("./data/preprocessed_bits.

    return
     ↪  trimmed_bits_numpy,trimmed_bits_numpy.shape[1]


def get_generator_input_sampler():
    return lambda m, n: torch.rand(m, n)


def create_attack_packets(packet_size):
    if path.isfile("./data/preprocessed_attacks.pkl"):

        logging.info("Using saved processed data...")
        trimmed_bits_numpy =
          ↪ np.load(open("./data/preprocessed_attacks.pkl",
          ↪ "rb"))
        return trimmed_bits_numpy
    else:

        pcap =
          ↪ dpkt.pcap.Reader(open("./data/inside.pcap",
          ↪ 'rb'))
        attack_list = ["080101"]

        df =
          ↪ pd.DataFrame(pd.read_csv(open("./data/attacks.csv",
          ↪ "r"), sep=";", encoding="UTF-8"))
        attack_list.extend(["".join(x[1].split(":"))
          ↪ for x in df["Start_Time"].iteritems()])
        print(attack_list)
        print(len(attack_list))
        untrimmed_bytes_list = []
        count = 0

        attack_packets = []
        for ts, buf in tqdm(pcap):

            ts =
              ↪ str(datetime.datetime.utcfromtimestamp(ts)
              ↪ - datetime.timedelta(hours=5))
            date, time = str(ts)[:10], str(ts)[11:]
            # print(date,time)
          time_long_string =
           ↪  "".join(time.split(":"))

          # print(time_long_string[:6],attack_list)

          if time_long_string[:6] in attack_list:
              bit_string = ""
              # print("Found attack")
              # eth=dpkt.ethernet.Ethernet(buf)
              # print(repr(eth))

              for bytes in list(buf):
                  # print(bytes)
                  bit_string +=
                   ↪  '{0:08b}'.format(bytes)

                  # print(bit_string)
                  # break
              attack_packets.append(bit_string)
              # print(attack_packets)
          # print("%s Attacks found" % count)
          count += 1
          if time_long_string[:6] ==
           ↪  attack_list[-1]:
              break
         # if count >= 10000:
             # break

        trimmed_bits_list = [sublist[:packet_size] for
          ↪  sublist in attack_packets]
```

```
        # print(trimmed_bits_list)
        trimmed_bits_list = [[(x + "0" * (packet_size
         ↪  - len(x)))] for x in
         ↪  tqdm(trimmed_bits_list)]
        # print(trimmed_bits_list[:2])
        trimmed_bits_numpy = np.asanyarray(
            [list(map(int, string)) for lists in
             ↪  tqdm(trimmed_bits_list) for string in
             ↪  lists])
        # print(trimmed_bits_numpy)
        pickle.dump(trimmed_bits_numpy,
         ↪  open("./data/preprocessed_attacks.pkl",
         ↪  "wb"), protocol=4)
        return trimmed_bits_numpy

    #print('Timestamp: ',))

def process_attacks(packet_size):
    if path.isfile("./data/week_4.pkl"):

      #  logging.info("Using saved processed data...")
        trimmed_bits_numpy =
         ↪  np.load(open("./data/week_4.pkl", "rb"))
        return trimmed_bits_numpy
    else:


```

```python
df = pd.DataFrame(pd.read_csv(open("./data/week_4.csv", "r",encoding="utf-8"), sep=";", encoding="UTF-8"))

#print(df.head())
#print(df["id"])
match_dict={}
for entry in scandir("./data/testing"):
    attack_list=[]

    if entry.is_file() and entry.name is not None:
        #pcap = dpkt.pcap.Reader(open("./data/week_4.pcap", 'rb'))
        name= entry.name[:2]
        #print()
        test =[tuple(y.split("."))   for x,y in df["id"].iteritems()]
      #  print(test)
        for x in test:
           # print(x[0],name)
            if x[0] == name:
                print("match")

               # print(x[1])
                attack_list.append(x[1])
                #print(attack_list)
    match_dict[entry.name[:2]]=attack_list
```

```python

        attack_packets = []
        for k,v in match_dict.items():
            pcap = dpkt.pcap.Reader(open("./data/testing/%s.pcap"%k, 'rb'))
            attack_list = v
            #print(v)
            untrimmed_bytes_list = []
            count = 0


            # for ts, buf in tqdm(pcap):
            for ts, buf in tqdm(pcap):

                if not attack_list:
                    break

                else:

                    ts = str(datetime.datetime.utcfromtimestamp(ts) - datetime.timedelta(hours=5))
                    date, time = str(ts)[:10], str(ts)[11:]
                    # print(date,time)
                    time_long_string = "".join(time.split(":"))

                    print(time_long_string[:6], attack_list)

                    if time_long_string[:6] in attack_list:
                        bit_string = ""
                        print("Found")
                        # attack_list.pop()
                        print(attack_list)
                        for bytes in list(buf):
                            #print(bytes)
                            bit_string += '{0:08b}'.format(bytes)

                            #print(bit_string)
                            #break

                            attack_packets.append(bit_string)
                        attack_list.pop(0)

        trimmed_bits_list = [sublist[:packet_size] for sublist in attack_packets]
        # print(trimmed_bits_list)
        trimmed_bits_list = [[(x + "0" * (packet_size - len(x)))] for x in tqdm(trimmed_bits_list)]
        # print(trimmed_bits_list[:2])
        trimmed_bits_numpy = np.asanyarray(
            [list(map(int, string)) for lists in tqdm(trimmed_bits_list) for string in lists])
        # print(trimmed_bits_numpy)
        pickle.dump(trimmed_bits_numpy, open("./data/week_4.pkl", "wb"), protocol=4)
        return trimmed_bits_numpy


# ##### Post-processing:

def sample_to_bytes(sample_array):
    bytes_list=[]
    logger.info(("Formatting {} samples of {} bits to bytes".format(sample_array.shape[0],sample_array.shape[1])))
    print(("Formatting {} samples of {} bits to bytes".format(sample_array.shape[0],sample_array.shape[1])))

    sample_list = sample_array.astype(int).tolist()
    #print("sample_list",sample_list)
    bit_strings = ["".join(map(str, lists)) for lists in tqdm(sample_list)]
    #print(bit_strings)
    for x in tqdm(bit_strings):
        bytes_list.append(bytes([int(x[i:i+8],2) for i in range(0,len(x),8)]))
    logging.debug(bytes_list[:2])
    print(len(bytes_list[1]))
    return bytes_list
```

```


def
  ↪ write_bytes_to_pcap(list_of_bytes,file_path="./data/gan_packets.pca
    logging.info("Writing %s packets to
      ↪ file"%len(list_of_bytes))
    with open(file_path, "wb") as f:
        fd= dpkt.pcap.Writer(f)
        for packet in tqdm(list_of_bytes):
            #print(packet)
            fd.writepkt(packet, time.time())
            #f.flush()

def metric(d_attack,zeroes):
    if zeroes:
        d_attack = np.around(np.sum(

                ↪ np.equal((np.around(d_attack.cpu().data.numpy())),
                ↪ np.zeros_like(d_attack.cpu().data.numpy()))
                ↪ / len(
                  d_attack.cpu().data.numpy())),6)


    else:

        d_attack = np.around(np.sum(

                ↪ np.equal((np.around(d_attack.cpu().data.numpy())),
                ↪ np.ones_like(d_attack.cpu().data.numpy()))
                ↪ / len(
                d_attack.cpu().data.numpy())),)
    return d_attack


# ##### MODELS: Generator model and discriminator
 ↪ model

class Generator(nn.Module):
    def __init__(self, input_size, hidden_size,
     ↪ output_size):
        super(Generator, self).__init__()
        self.map1 = nn.Linear(input_size, hidden_size)
        self.map2 = nn.Linear(hidden_size,
         ↪ hidden_size)
        self.map3 = nn.Linear(hidden_size,
         ↪ output_size)

    def forward(self, x):
        x = F.elu(self.map1(x))
        x = F.sigmoid(self.map2(x))
        return F.sigmoid(self.map3(x))


class Discriminator(nn.Module):
    def __init__(self, input_size, hidden_size,
     ↪ output_size):
        super(Discriminator, self).__init__()
        self.map1 = nn.Linear(input_size, hidden_size)
        self.map2 = nn.Linear(hidden_size,
         ↪ hidden_size)
        self.map3 = nn.Linear(hidden_size,
         ↪  output_size)

    def forward(self, x):
        x = F.elu(self.map1(x))
        x = F.elu(self.map2(x))
        return F.sigmoid(self.map3(x))


# Data params

# Model params
g_hidden_size = 50   # Generator complexity

d_hidden_size = 50   # Discriminator complexity
d_output_size = 1    # Single dimension for 'real' vs.
 ↪  'fake'
minibatch_size = 100

d_learning_rate = 2e-4  # 2e-4
g_learning_rate = 2e-4
optim_betas = (0.9, 0.999)
num_epochs = 2
print_interval = 200
d_steps = 5  # 'k' steps in the original GAN paper.
 ↪  Can put the discriminator on higher training freq
 ↪  than generator
g_steps = 1

new_number_data_bytes = False
```

```
num_databytes=None
num_packets_limit=None
pcap_path="./data/training.pcap"
torch.set_default_tensor_type('torch.cuda.FloatTensor')


bitstream, packet_size =
  ↪  data_to_bitstream(num_packets_limit=num_packets_limit,
  ↪  num_databytes=num_databytes,
                                                           ↪  new_number_data_bytes=ne
                                                          ↪  pcap_file=pcap_path)
print(packet_size)
create_attack_packets(packet_size)
g_input_size = packet_size


data_class = Data_preporation(data=bitstream,
  ↪  batch_size=minibatch_size)

data_class.reset_batch_pointer()


gi_sampler = get_generator_input_sampler()



```

```
G = Generator(input_size=packet_size,
 ↪ hidden_size=g_hidden_size,
 ↪ output_size=packet_size)
G.cuda()
D = Discriminator(input_size=packet_size,
 ↪ hidden_size=d_hidden_size,
 ↪ output_size=d_output_size)
D.cuda()

print(G)
print(D)
criterion = nn.BCELoss()  # Binary cross entropy:
 ↪ http://pytorch.org/docs/nn.html#bceloss
d_optimizer = optim.Adam(D.parameters(),
 ↪ lr=d_learning_rate, betas=optim_betas)
g_optimizer = optim.Adam(G.parameters(),
 ↪ lr=g_learning_rate, betas=optim_betas)


timestr = time.strftime("%d%m%Y-%H%M")
writer =
 ↪ SummaryWriter("C:\\tensorboard\\pytorch\\%s"%timestr)
test_writer =
 ↪ SummaryWriter("C:\\tensorboard\\pytorch\\%s_test"%timestr)

for epoch in range(num_epochs):
    data_class.reset_batch_pointer()
    attacks = create_attack_packets(packet_size)
    seed=len(attacks)//4
    print(seed)
    testing,validate = attacks[:seed*3],attacks[seed:]
    new_attack_data = process_attacks(packet_size)
    for batches in range(data_class.num_batches):
        tick =
         ↪  (epoch*data_class.get_num_batches())+batches

        d_real_data,test = data_class.next_batch()
        d_real_data=torch.from_numpy(d_real_data)
        d_real_data =d_real_data.float()
        #test =)

        d_real_data_tensor =
         ↪  Variable(d_real_data.cuda())
        #print(d_real_data_tensor)
        # print(x)
        for d_index in range(d_steps):
            # 1. Train D on real+fake
            D.zero_grad()

            #  1A: Train D on real

            d_real_decision = D(d_real_data_tensor)
            #print(d_real_decision)
            d_real_error = criterion(d_real_decision,
             ↪  Variable(torch.ones(100).cuda()))  #
             ↪  ones = true
           # print(d_real_error)
            d_real_error.backward()  # compute/store
             ↪  gradients, but don't change params


            #  1B: Train D on fake
            d_gen_input = Variable(
             ↪  torch.rand(minibatch_size,
             ↪  g_input_size))
            d_fake_data = G(d_gen_input).detach()  #
             ↪  detach to avoid training G on these
             ↪  labels
            d_fake_decision = D(d_fake_data)
            d_fake_error = criterion(d_fake_decision,
             ↪  Variable(torch.zeros(100).cuda()))  #
             ↪  zeros = fake

            d_fake_error.backward()
           # d_optimizer.step()

            d_optimizer.step()



        for g_index in range(g_steps):
            # 2. Train G on D's response (but DO NOT
             ↪  train D on these labels)
            G.zero_grad()

            gen_input =
             ↪  Variable(gi_sampler(minibatch_size,
             ↪  g_input_size))
            g_fake_data = G(gen_input)
            dg_fake_decision = D(g_fake_data)
            g_error = criterion(dg_fake_decision,
                                                     ↪  Variable(torch.ones(100).cuda()))
                                                     ↪  # we want to fool,
                                                     ↪  so pretend it's
                                                     ↪  all genuine

            g_error.backward()
            g_optimizer.step()  # Only optimizes G's
             ↪  parameters
            writer.add_scalar('loss/d_real',
             ↪  d_real_error, tick)
            writer.add_scalar('loss/d_fake',
             ↪  d_fake_error, tick)
            writer.add_scalar('loss/g', g_error, tick)









        if tick % 10 ==0:
            d_attack_data = torch.from_numpy(testing)
            d_attack_data = d_attack_data.float()
            d_attack_test =
             ↪  Variable(d_attack_data.cuda())
            d_attack = D(d_attack_test)
            running_attack_ac = metric(d_attack, zeroes=True)
            d_attack_error = criterion(d_attack, Variable(torch.zeros_like(d_attack.data)))
            # print(running_attack_ac)
            d_attack_error.backward()
            d_optimizer.step()  # Only optimizes D's



        if tick % 10 == 0:


            print(tick)


            running_fake_ac = metric(d_fake_decision,zeroes=True)
                                running_real_ac=metric(d_real_decision,zeroes=False)



                                writer.add_scalar('Prediction/real',running_real_ac, tick)
            writer.add_scalar('Prediction/fake', running_fake_ac, tick)
            for name, param in D.named_parameters():
                writer.add_histogram(name,
                 ↪  param.clone().cpu().data.numpy(),
                 ↪  tick)



            d_real_data = torch.from_numpy(test)
            d_real_data = d_real_data.float()
            d_real_test = Variable(d_real_data.cuda())
            d_test = D(d_real_test).detach()
            running_test_ac =
             ↪  metric(d_test,zeroes=False)

            writer.add_graph(D, d_real_test)

            d_gen_input =
             ↪  Variable(torch.rand(minibatch_size,
             ↪  g_input_size))
            d_fake_data = G(d_gen_input).detach()  #
             ↪  detach to avoid training G on these
             ↪  labels
            d_test_fake_decision = D(d_fake_data)
            running_fake_test_ac =
             ↪  metric(d_test_fake_decision,zeroes=True)

            writer.add_graph(G, d_gen_input)


             ↪  test_writer.add_scalar('Prediction/Test/real',
             ↪  running_test_ac, tick)
```

```
                    ↪ test_writer.add_scalar('Prediction/Test/fake',
                    ↪ running_fake_test_ac, tick)




                d_attack_data =
                    ↪ torch.from_numpy(new_attack_data)
                d_attack_data = d_attack_data.float()
                d_attack_test =
                    ↪ Variable(d_attack_data.cuda())
                d_attack = D(d_attack_test).detach()
                running_attack_ac =
                    ↪ metric(d_attack,zeroes=True)



                    ↪ test_writer.add_scalar('Prediction/attacks',
                    ↪ running_attack_ac, tick)


                print("%s/%s: D: %s/%s G: %s (Real: %s,
                    ↪ Fake: %s),  Test Real: %s, Test Fake:
                    ↪ %s " % (
                    batches, data_class.num_batches,
                        ↪ d_fake_error.item(),
                        ↪ d_real_error.item(),
                    g_error.item(), running_real_ac,
                     ↪  running_fake_ac, running_test_ac,
                     ↪  running_fake_test_ac))

                print("Real attack from
                 ↪  dataset:",running_attack_ac)




        new_attack_data_numpy =
         ↪  torch.from_numpy(new_attack_data)
        new_attack_data_numpy =
         ↪  new_attack_data_numpy.float()
        new_attack_variable =
         ↪  Variable(new_attack_data_numpy.cuda())
        new_attack_prediction =
         ↪  D(new_attack_variable).detach()

        running_bob =
         ↪  metric(new_attack_prediction,zeroes=True)

        print(new_attack_prediction.data)
        print(running_bob,"Of DARPA Week 4 & 5 Real Attack
         ↪  Detected!",)


        if epoch % 1 == 0:
            gen_input = Variable(gi_sampler(10000,
             ↪  g_input_size).cuda())
            gen_sample = G(gen_input).detach()
            print(gen_sample)
            gen_sample=gen_sample.cpu().data.numpy()
            gen_sample=np.around(gen_sample)
            print(gen_sample)
            sample_bytes = sample_to_bytes(gen_sample)

            # print()

                ↪  write_bytes_to_pcap(list_of_bytes=sample_bytes,
                ↪  file_path="./data/gen/gan_packets.pcap")
```