Unsupervised Intrusion Detection System

Aykut Öksüz

Kongens Lyngby 2007
IMM-M.Sc.-2007-20

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

This thesis evolves around Intrusion Detection System (IDS) and Neural Network (NN). Intrusion detection systems are gaining more and more territory in the field of secure networks and new ideas and concepts regarding the intrusion detection process keep surfacing.

One idea is to use a neural network algorithm for detecting intrusions. The neural network algorithms have the ability to be trained and 'learn' socalled patterns in a given environment. This feature can be used in connection with an intrusion detection system, where the neural network algorithm can be trained to detect intrusions by recognizing patterns of an intrusion.

This thesis outlines an investigation on the unsupervised neural network models and choice of one of them for evaluation and implementation. The thesis also includes works on computer networks, providing a description and analysis of the structure of the computer network in order to generate network features. A design proposal for such a system is documented in this thesis together with an implementation of an unsupervised intrusion detection system.

Keywords: Intrusion Detection System, Neural Network, Unsupervised Learning Algorithm, Pcap, Network Features.

Resumé

Dette eksamensprojekt omhandler Intrusion Detection Systemer (IDS) og Neurale Netværk (NN). Intrusion detection systemer er blevet mere og mere populære og der optsår hele tiden nye idéer og koncepter til intrusion detection systemer.

En mulig idé er at bruge et neuralt netværk for at opdage indbrud. Et neuralt netværk kan blive trænet til at kunne 'lære' mønstre i et bestemt miljø. Dette egenskab kan bruges i forbindelse med et intrusion detection system, hvor det neurale netværk kan trænes til at opdage indbrud ved at genkende mønstre, der kendetegner et indbrud

Dette eksamensprojekt omhandler en undersøgelse af unsupervised neuralt netvæks modeller og hernæst valg af én for vurdering og implementering. Projektet inkluderer også det arbejde der er forbundet med computernetværk, ved at udarbejde en beskrivelse og analyse af computernetværk for at generere netværkfeatures. Et designsforslag for sådan et system vil dokumenteres i projektet og til sidst vil en unsupervised intrusion detection system implementeres og testes.

Nøgleord: Intrusion Detection System, Neural Netværk, Unsupervised Learning Algorithm, Pcap, Netværk Features.

Preface

This thesis was written at the Department of Informatics and Mathematical Modelling (IMM), the Technical University of Denmark (DTU) in connection with acquiring the M.SC. degree in engineering. It was prepared during autumn 2006 and winter 2007 and has been supervised by Professor Robin Sharp.

The thesis deals with topics like intrusion detections, neural networks, unsupervised learning algorithms and etc. The focus is on evaluating and implementing an intrusion detection system using an unsupervised learning algorithm.

I would like thank everyone who has has contributed in realising this thesis, especially Robin Sharp for his guidance and supervision.

Lyngby, February 2007

Aykut Öksüz

Contents

Chapter 1

Introduction

With the rapid expansion of computer networks the security issues are not to be compromised. The importance of the security measures grows bigger as the protection of data (e.g., company data and research data) and the protection against computer related malicious code or intrusion attacks (virus, worm, trojan horse etc.) are becoming more and more frequent for almost everyone working with computers. Today it is unlikely to find a company, an institution or a private home computer that is not protected against the steadily growing threat from networks (e.g., the Internet). Intrusion events could cost companies great amount of money and time. It could also happen that their precious data could end in the wrong hands, leading to wasted time and effort. There are several proposals to how to enhance security and this project is about one of them.

1.1 The goal of this project

The aim of this project is to make a research on unsupervised neural network models. Roughly, neural networks are specified with many small processors working simultaneously on the same task. It has the ability to 'learn' from training data and use its 'knowledge' to compare patterns in a data set. Especially the unsupervised networks, which do not need categorized training data

set, are more interesting as they are more self-administrative and can learn new patterns within the data set without any interference from outside (i.e., an administrator).
The project also concerns a certain method for protecting computer networks. The *intrusion detection system* (IDS) is one way of protecting a computer network. This kind of technology enables users of a network to be aware of the incoming threats from the Internet by observing and analyzing network traffic. During these processes the IDS will gather information from the network traffic, which will be used to determine whether the traffic holds suspicious content. Upon suspicious behavior in the network traffic, the IDS program can be set to warn its administrator either by mail or SMS, and in most cases write out to log files, which the administrator can read and discover possible intrusion. The IDS does not prevent an intrusion like a firewall which closes ports entirely. The IDS lets the traffic flow but sees the traffic and detects intrusion without really doing anything about it. The rest is up to the administrator or the security policy. Many IDSs are based on some form of pattern recognition, where patterns of intrusions are known to the IDS and the IDS can perform comparison with the patterns of the network traffic in order spot intrusions. A pattern is a certain behavior that characterizes an intrusion. It could be anything from a value to a graphical representation. In order to compare these patterns, the IDS must know them beforehand so the comparison is possible and then intrusions can be detected. But it is a hard process defining all kind of intrusions and some of them are really unknown until they have taken place.
The goal of this project is to develop a system, that does not need any knowledge beforehand but can be trained to detect unknown patterns in network traffic. This unknown pattern does not necessarily need to be an intrusion, it could be a new normal behavior in the network that is just unknown to the system. But the aim is to detect any kind of new and unknown behaviors and later by examining log files determine whether the new patterns indicate intrusions or not. This is here where the unsupervised neural network comes in. A network of this type is capable of detecting any kind of behavior. The behaviors that the network has been trained with is normally representing normal network traffic without any intrusions. And while testing network traffic, the network will distinguish normal data from unknown data and these unknown data are suspicious and potential intrusions. This project is also about finding a neural network model, that is best suitable for tasks like intrusion detection.
The product of this project is a system which implements an unsupervised neural network. The network can be trained and tested while users can follow up on the process through a graphical user interface. In order to prove the efficiency of the system some tests will be carried out after the implementation.

1.2 The structure of the report

The structure of the report is as follows

Chapter 2. Deals with the concept of intrusion detection systems. It will also cover the different types of IDSs, and explain what a network-based IDS is and how it operates.

Chapter 3. Is about neural networks. An introduction to the concept of neural network together with the different learning procedures will be covered in this chapter.

Chapter 4. This chapter is about the different types of unsupervised neural network models, that can be used with the intrusion detection. The models are described and one of them will be chosen based on its qualities.

Chapter 5. Deals with network connection and features. It will also cover subjects like how network traffic can be sniffed, features are constructed and processed in order to make them suitable for the neural network algorithm.

Chapter 6. A specification of the IDS with the chosen neural network model will be described in this chapter.

Chapter 7. This chapter will cover an overall design of the IDS regarding the implementation.

Chapter 8. Deals with the implementation of the IDS.

Chapter 9. this chapter will present result of functional and efficiency test of the implemented IDS.

Chapter 10. Will cover concluding remarks on the IDS and the project as a whole.

Appendix A. User manual to the developed IDS program.

Appendix B. Functional test of the IDS program.

Appendix C. A draft of the log file the developed IDS program computes.

Appendix D. The code of a simple packet sniffer.

1.3 Content of the CDROM

In the attached CDROM, the source code of the developed system is available. The program is also compiled and ready to be run. The execution file is in patch `/IDSnet/idsnet/idsnet` and can be executed from a command shell by writing `[bash]# ./idsnet`. Furthermore, there is a documentation of the implemented system in path `/IDSnet/html/index.html`.

Chapter 2

Intrusion Detection System

This chapter starts with an introduction to the concept of intrusion detection system. Consequently we will look at the different variants of available IDS techniques and at the end specify what kind of properties we would like to have regarding the unsupervised IDS we want to build.

2.1 Intrusion detection system - IDS

Generally speaking, an *intrusion detection system* is a tool for detecting *abnormal behaviors* in a system. An abnormal pattern covers many definitions but in general it is likely described as unwanted, malicious and/or misuse activity occurring within a system. The two main techniques of intrusion detection are called *misuse detection*[1] and *anomaly detection*.

- Misuse detection systems [16] use patterns of known attacks or weak spots of the system to match and identify intrusions. For instance, if someone tries to guess a password, a signature rule for this kind of behavior could be that 'too many failed login attempts within some time' and this event

[1] Some articles [10] refers to it as *signature-based* intrusion

would result in an alert. Misuse detection is not effective against unknown attacks that have no matched rules or patterns yet.

- Anomaly detection [16] flags observed activities that deviate significantly from the established normal usage profiles as anomalies, that is, possible intrusions. For instance a profile of a user may contain the averaged frequencies of some system commands in his or her logging sessions. And for a logging session that is being monitored if it has significantly lower or higher frequencies an anomaly alert will be raised. Anomaly detection is an effective technique for detecting novel or unknown attacks since it does no require knowledge about intrusion attacks. But at the same time it tends to raise more alerts than misuse detection because whatever event happens in a session, normal or abnormal behavior, if its frequencies are significantly different from the averaged frequencies of the user it will raise an alert.

The two general techniques described are two different ways of spotting intrusions. As mentioned intrusion detections can be deployed on different areas, like within a computer to spot users attempting to gain access to which they have no access right, or monitoring network traffic to detect other kind of intrusions like worms, trojan horses or to take control of a host by yielding an illegal root shell, etc.
As for this project we will concentrate our efforts on one type of intrusion detection, that is *Network-Based IDS*. Our interests lie in investigating different potential algorithms and later on using this knowledge to develop an automated and unsupervised approach for building an IDS.

2.2 Network-Based IDSs

In a computer network there are a lot of data exchanges between computers within a local network and between a computer and another network (e.g. the Internet), see figure 2.1. The IDS systems can be deployed in different ares of a network having different detection tasks. Being connected to a large network like the Internet plunges the computers into a world where the risk of getting in touch with harmful network traffic activity is relatively high. Several security precautions can be taken, like deploying antivirus, firewall, access control etc. in order to prevent such activities from intruding upon your computer or network. They all concentrate on different aspects of how to protect and secure a computer/network. Some well-known methods of IDS systems are based on either comparing patterns of network traffic to saved patterns of network activities of known attacks [16] or statistical methods used to measure how ab-

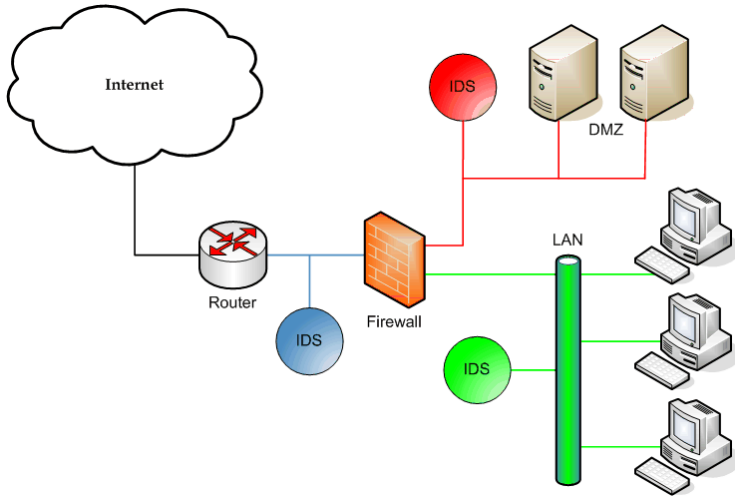

Figure 2.1: A computer network with intrusion detection systems

normal a behavior is [20]. Both methods require that the patterns of activity are known. In the first case, typically, a human expert analyzes the attacks that might come with a network connection and categorizes them depending on their severity level. For each defined attack, the expert will associate the attack with a unique pattern, that is worked out from the attack itself. These patterns could describe behaviors in a network connection. For instance a pattern for an attack like worm could be that the virus tends to use a certain port number to penetrate. Furthermore if we look at packet level, we will see that the order of incoming data packets carrying the virus differs from other packet flows, which may describe an http request. In other words, the expert tries to find certain *features* in that particular traffic that forms the worm. The detection process goes off by comparing the incoming traffic from the network connection to the known attacks.
The IDSs differ in whether they are online or offline [20]. Offline IDSs are run periodically and they detect intrusions after-the-fact based on system logs.
Online systems are designed to detect intrusions while they are happening, thereby allowing quicker intervention. However, offline systems slow down the process of intervention as they are monitoring network connections offline. Online systems continuously monitor network and thereby detect intrusions while they are happening but compared to offline systems they are more expensive in the sense of computation time due to continuous monitoring. But this expense should not scare us from making an online intrusion detection, because it is more important to detect attacks faster and while they are happening, rather than after the attack has taken place and maybe caused harm. Therefore we

choose to make an online intrusion detection system.

2.3 Different types of IDSs

Generally an IDS can be used in three different ways:

- **System IDS**: Large systems or programs which may need protection against illegal user intrusion can be secured with an IDS. The task of the

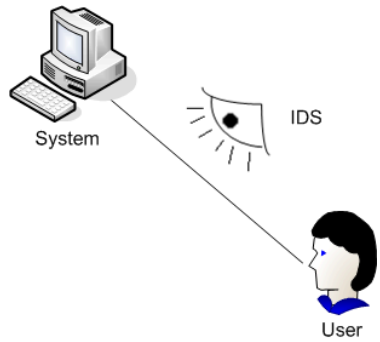

Figure 2.2: An IDS that monitors user behaviors

  IDS is to monitor user behaviors within the system and discover abnormal user actions, see figure 2.2. An example for this could be a scenario where IDS monitors certain users with certain patterns of activities. If a user tries to yield or obtain access to a root shell, which is not in the list of his rights, the IDS can discover this action by comparing it to the typical pattern of behavior of the user and report it to an administrator.

- **Single Network IDS**: Another way of fishing up illegal activities is to monitor network connections for intrusions. This kind of IDS monitors the network connection of a computer or server, see figure 2.3. With this type the IDS detects network-related threats on a single connection.

- **Cooperative IDS**: Cooperative IDS [11] is a system built on information sharing. The principle for this framework is that information sharing takes place both between distinct domains and within domains. These domains may have different or even conflicting policies and may also reside different mechanisms to identify intrusion. Figure 2.4 shows an example of a cooperative IDS. There are two domains, e.g., A and B, connected to a central

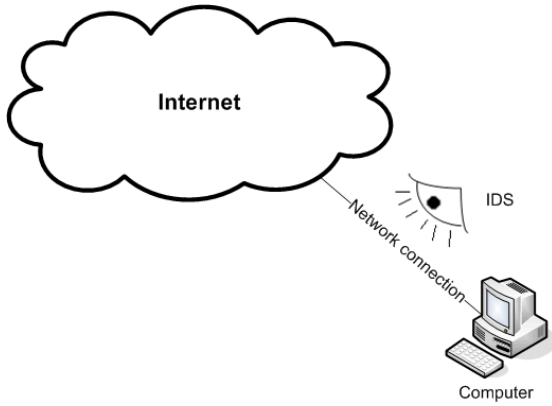

Figure 2.3: An IDS that monitors the traffic between Internet and the computer

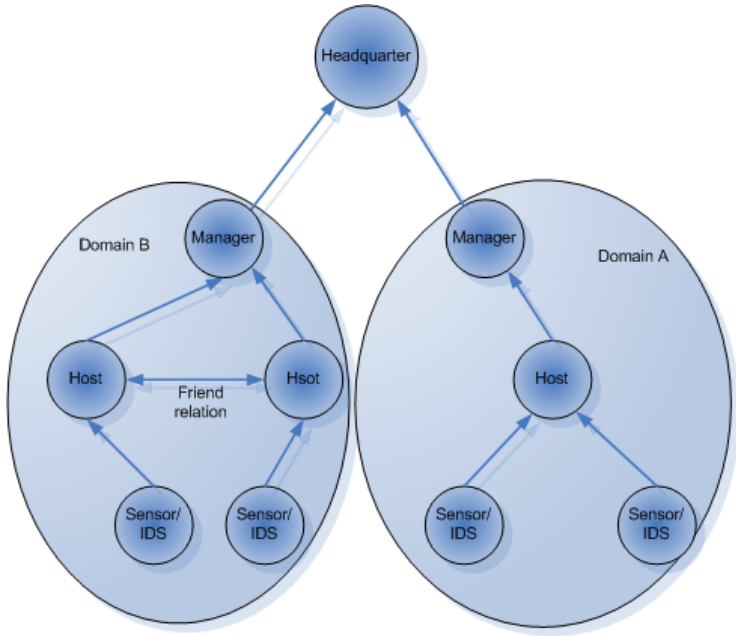

Figure 2.4: Cooperative Network Intrusion Detection

management system, called headquarter. Within these domains there are managers that receive their data from the hosts. The hosts receive their data from sensors/IDSs, which collect their data from outside, i.e., network connections. The collected data travels bottom-up but the hosts and managers do not necessarily need to receive all collected data. The data will be filtered according to the policies of the hosts and managers. Hosts can also send requests or information between themselves without controlling each other. This kind of connection between two hosts is called *friend relationship*.

The above listed different types of intrusion detection operate in almost the same way. The difference lies in the tasks (the task to detect user intrusion, network-related intrusion etc.) of the IDSs. Our project is based on monitoring a single network connection. In the following section we will give an answer to the question of where exactly we intend to use the IDS.

2.3.1 Work on unsupervised IDSs

There are many articles [13][15][22][19][17] about unsupervised IDSs and new ones keep coming, as the development in this field progresses and the need for a more automated and self-administrative IDS systems grows. This need is highly connected to the development of the various and endless attacks related to network connections. Due to this reality, the researches on unsupervised IDS systems are more popular than ever and there are many suggestions on how to make the ideal and optimal IDS system primarily based on unsupervised learning paradigm. Unfortunately there is no general solution for this problem. In the articles [19][13] and [18][2] different solutions are presented, implemented and tested in order to prove the efficiency of the algorithms. Especially the word efficiency rise to the surface over and over again due the aim of building a solution which does not only work, but also uses the least possible expenditure of resources (such as time, CPU power, memory, etc.) The keyword for all of them is the unsupervised learning algorithm, but apart from that their solutions are very different. They all claim, off course, that the solution they present is very effective and works just the way they want it. Which leads us to the dilemma of which algorithm to choose. In the chapter 4 we will introduce some algorithms, by describing them and discuss their strengths and weaknesses.

[2]These works have been used as inspiration for this project and are just the tip of the iceberg in this particular research field

2.3.2 The ideal algorithm for IDS

The choice of an algorithm is difficult depends on many factors and most likely there is no unique solution. Before developing an unsupervised IDS system some considerations must be made:

- **What is needed?** What is the purpose of the IDS system? What is aimed to be detected? IDS systems detect what the developer desires. Mostly the purpose is to detect intrusions like worms, hacker attacks, and other malicious code appearing in a network connection. But this is not absolute, in fact one can set up an IDS system to detect any activity by defining them to the system. For example you could setup the system such that it can detect unnecessary traffic, which is not harmful code but just unwanted. It is up to the developer to make these decisions and from there form a strategy.
- **Where is it needed?** In what context is the IDS system going to be applied? Physically, where is it going to be deployed? As seen in figure 2.1 the IDS systems can be deployed in many different places in a given network. There is a big difference in monitoring a single computer connected to the Internet and a server also connected to the internet but that handles many requests in short a time frame. It could be difficult and time-consuming to train some relatively good algorithms with huge amount of data. Then you are unwillingly forced to choose another learning algorithm that does not have the same skills but is significantly better and faster to train the network.
- **Data preprocessing/representation.** How do you represent your data? Indeed, this is a very important part of the development when using techniques like neural network. The different neural network models require different data preprocessing and it applies to each one of them that there is no standard technique for doing that. Furthermore, the preprocessing also depends on the data to be used for the network. One has to find and pinpoint *features* in the data set in an appropriate way and use these features to train the network with. But finding features in a data set is not an easy task. Tt requires careful analysis and great knowledge about the data set and in some cases can be hard to find.

The above listed questions are of great importance to this project and can be considered the core of the discussions and argumentations constituting this project. Later on in this project we will try to find answers to these questions in order to clarify and use them on scientific basis to build an unsupervised IDS. This project, the evaluation and implementation sections in particular, does not

aim to develop an universal solution proposal to the intrusion detection systems using neural network. Given the purpose and a list of criteria we set our goal to be a development of a specific unsupervised intrusion detection system that is capable of monitoring a network connection traffic (see following chapters) and alerting if necessary.

2.4 Summary and discussion

With this chapter we have made an introduction to the concept of IDSs. Firstly a general definition of an IDS has been elaborated and the two standard techniques of IDSs were presented. The misuse and anomaly detection methods which are considered to be two main methods of the IDS were described together with their strengths and weaknesses. In relation to this project we have pointed out that the aim is to develop an intrusion detection that goes under the anomaly detection category.
We continued with the description of network-based IDS, which relates to our project. We learned how network-based IDSs operate and how they detect intrusions in a network connection.
Network-based IDSs come in two different types, online and offline detection. In order to detect and avoid intrusions faster it is better to construct an online intrusion detection and detect intrusions while they take place. It has been pointed out how important it is to detect intrusions in real time for allowing quicker intervention.
Another field that characterizes an IDS is whether it is supervised or unsupervised. The differences have been presented and it has been decided to build our an IDS based on unsupervised principles. That is due to the fact that an unsupervised IDS is more automated and does not need an expert to tell what is intrusion and what is not.
Finally we have listed some basic questions about the IDS we want to implement. These questions will be answered in the following chapters.
No doubt, that an IDS is an important brick in our wall of safety in the Internet. They are capable of detecting intrusion by examining network traffic and inform us whenever there is an unusual pattern that might be an intrusion. It is the very essence of the IDS that it can be taught, and it learns and acts in a way that is based on its knowledge. The unsupervised IDS we want to build is another proposal to how to make IDSs more automated and with least intervention from experts.

Chapter 3

Intrusion detection using neural networks

This chapter will cover an introduction to *neural networks* by introducing a definition and description of the network. Categorically we will dig into to the essences of neural networks, presenting the core elements like learning processes and paradigms and eventually indicating why these are important to clarify before making a decision on which neural network to use. It will be clarified why a neural network is a good tool for developing IDSs.

3.1 What is a neural network?

The concept of neural networks (also referred to as artificial neural networks) is highly inspired by the recognition mechanism of the human brain. The human brain is a complex, nonlinear and parallel computer, whereas the digital computer is entirely the opposite, it is a simple, linear and serial computer. The capability to organize neurons to perform computation is many times faster than a modern digital computer in existence today. Human vision [12] is a good example for understanding this difference.
There is no universally accepted definition of neural network, but there are some architectural and fundamental elements that are the same for all neural

networks. First of all, a neural network is a network with many simple processors which are known as the *neurons*. These neurons often posses a small amount of local memory. They have the task to receive input data from other neurons or external sources and use this to compute new data as output for the neural network or input data to the neurons of the next layer. The received or computed data is carried by communication channels, better known as the *weights*. The weights which connect two neurons posses certain values and will be adjusted upon network training. The adjustment of the weights is processed in parallel, meaning that many neurons can process their computations simultaneously. The magnitude of the adjustment of the neurons depend on the training data and is carried out with a so-called *training rule* (also known as *learning rule*, see section 3.3). Another common characteristic for most neural networks is that the network can be parted into *layers*. An example of a basic neural

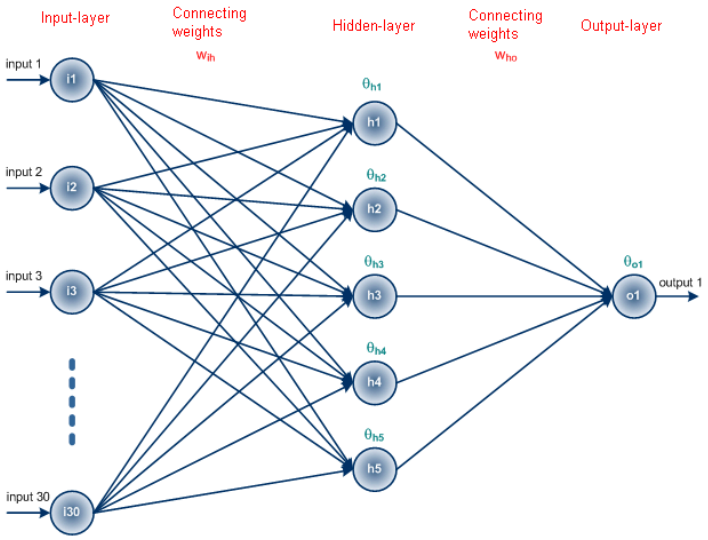

Figure 3.1: Structure of a simple fully-connected neural network with three layers

network model is shown 3.1. I has three layers where the layers are organized as follows; The first layer is the input layer, that receives data from a source. The second is the output layer that sends computed data out of the neural network. The third layer is called hidden layer, whose input and output signals remain within the neural network, see figure. In the particular example the network is *fully-connected*, which means that every neuron in one layer is connected with all neurons in the preceding layer and so on. Although it is not a rule and a

neural network does not need to be fully-connected.
Roughly, the overall task of a neural network is to predict or make approximately correct results for a given condition. Neural networks are trained with training data and the elements (e.g., neurons and weights) of the network will be adjusted on the basis of this training data. When further training does not change the network significantly or a given criterion is fulfilled the network is ready to produce results. Test data can be put into the network, be processed and the network will come up with a result.

3.2 Why use neural network for IDSs?

As described above neural networks posses unique properties, which do not only make them attractive but also a qualified tool. First of all, the intrusion detection systems operate by making results in the sense of predictions based on known as well as unknown patterns. With the use of neural network models it is possible to comply with this process, since these models offer the option to train a custom network and use it as some sort of a strainer for new incoming network connection and thereby detect abnormal behaviors. Several neural network algorithms are capable of fulfilling this requirement and will be described in chapter 4. Furthermore, when working with intrusion detections one will realize that the dimension of the data of a network connection is high. There are many different protocols on different layers of the internet with different services and with destinations and sources and etc. A more detailed description of the network connection is presented in chapter 5.
The property of *dimensionality reduction* and *data visualization* in neural networks can be very useful to reduce the many dimensions of a network connection to 2-dimension. By doing so the features can be represented with 2-dimension and easily visualized on a (X,Y) coordinate system. This will help to visually discover connections which do not fall into the same category or group (clusters) with the trained and trusted ones.

3.3 Learning processes

Learning processes are important to the neural networks. These sets of rules formulate how the weights of a network are to be adjusted. From a higher perspective these rules can be considered to be the mechanism that makes the networks *learn* from their environment and *improve* their performance accordingly. If networks are trained carefully, networks can exhibit some capability for generalization beyond the training data, for example how much is it going to

rain the next year? Based on the rain behaviors for the passed 30 years a neural network, if trained with the proper learning algorithm, which is determined by the analysis and preprocessing of the data, can produce a reasonable prediction on how much it is going to rain the next year. Another field, that is quite more interesting and way more challenging, is the stock market. Undoubtedly, investors will be interested in knowing how the value of a stock is going to develop in the short and long term. Based on the former developments of the stock for the passed years, a neural network model can be trained to predict when a stock will yield the largest profit. But the stock market is a very complex field and it is doubtful that such a model will be invented.
The definition of the learning process implies the following sequence of events:

1. The neural network is stimulated by an environment.
2. The neural network undergoes changes in its free parameters as a result of the stimulation.
3. The network responds in a new way to the environment due to the changes that occurred in its internal structure.[12]

There are numerous algorithms available and as one would expect there is no unique algorithm for designing a neural network model. The difference between the algorithms lies in formulation of how to alter the weights of the neurons and in the relations of the neurons to their environment. In the following sections some of the learning processes will be described.

3.3.1 Hebbian Learning Rule

The Hebbian learning rule [12][14][9] is one of the oldest and most famous of all learning rules and is relatively simple to be coded into a computer program. It is described as a method for determining how to alter weights between neurons. The procedure for altering the weights is to observe two neurons, that are connected via the weight in particular. This weight is increased (or strengthened) if the two neurons are activated simultaneously and decreased (or weakened) if the neurons activate separately. In this sense, the Hebbian learning rule ensures that weights between neurons are adjusted in a way that represents the relationship more precisely. As such, many learning rules can be considered to be somewhat Hebbian in nature.
The simplest version of the Hebbian learning is described by

$$\Delta w_{kj} = \eta y_k x_j \tag{3.1}$$

where η is a positive constant that determines the *rate of learning*. The equation 3.1 can be used in many neural networks as the update rule and as indicated there are many mathematical formulas to express the principal of Hebbian rule and there are many versions.

3.3.2 Competitive Learning Rule

In neural networks with Hebbian learning output neurons can be activated simultaneously. In *competitive learning* [12][14] only one single output neuron can be activated. As the name of the learning implies the neurons undergo a competition, where they compete among themselves to become active. In statistics competitive learning help discovering salient features that may be used to classify a set of input patterns. There are some basic elements to the competitive learning. The weights are randomly distributed and therefore respond differently to a given set of input patterns. There is a limitation imposed on the strength of each neuron and a mechanism that allows neurons to compete for the right to respond to a given subset of inputs, such that only *one* output neuron, or only one neuron per group, is active at a time. The winning neuron is called a *winner-takes-all neuron*.
In order to become a winning neuron, the induced field v_k of the neuron k for a specified input $\mathbf{x}$ must be the largest among all the neurons in the network. The output neuron y_k of winning neuron k is set equal to 1 while the output of the neurons that lose the competition are set equal to 0. Thus we write

$$y_k = \begin{cases} 1 & \text{if } v_k > v_j \text{ for all } j, j \neq k \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

where the induced local field v_k is the combined action of all the forward and feedback inputs to the neuron k. The update of the weights progresses as follows; let w_{kj} denote the weight between neuron k and the input source j. Each neuron is allocated a fixed amount of weight which is distributed among the input nodes. We write

$$\sum_j w_{kj} = 1 \tag{3.3}$$

where all weights are positive. A neuron then learns by shifting weights from its inactive to active input nodes. If a neuron does not respond to a given input, no learning takes place. On the other hand, if a neuron wins the competition each input node of that neuron gives up a proportion of its weight and those weights are distributed equally among the active input nodes. The update rule stating the change, Δw_{kj} applied to the weights can be written by

$$\Delta w_{kj} = \begin{cases} \eta(x_j - w_{kj}) & \text{if neuron } k \text{ wins the competition} \\ 0 & \text{if neuron } k \text{ loses the competition} \end{cases} \tag{3.4}$$

where η is the learning-rate parameter. The overall effect of this update rule is to move the weight vector $\mathbf{w}_k$ of winning neuron k toward the input $\mathbf{x}$.

3.3.3 More learning rules

Beside the learning rules described above, there are several others, like *the error-correction learning rule* and *the Boltzman learning rule* [12]. The rules that have been described in this section are related to chapter 4, where several unsupervised neural network models will be described and analyzed. The learning rules described in this section are somehow included in those network models, for example they appear in the model as the update rule or a part of the learning algorithm.

3.4 Learning paradigms

In neural networks there are two different overall learning paradigms. The first one is *supervised learning*, also known as *learning with a teacher*. The second one is called *unsupervised learning* also referred to as the *learning without a teacher* [12][15]. This project concerns working with unsupervised models and the reason for this will be elaborated in this section.

3.4.1 Supervised learning

In conceptual terms the supervised learning can be seen as a teacher having knowledge of the environment derived from input-output examples. The teacher provide consultancy to the neural network telling it what is normal and abnormal traffic pattern, in the sense of what is classified as *malicious* and *non-malicious*. Basically the supervised learning operates as depicted in figure 3.2.
A portion of network connection is to be analyzed and labeled with the help of the teacher. Afterwards the labeled training data is used by the learning algorithm to generalize the rules. Finally the classifier uses the generated rules to classify new network connections and gives alert if a connection is classified to be malicious.

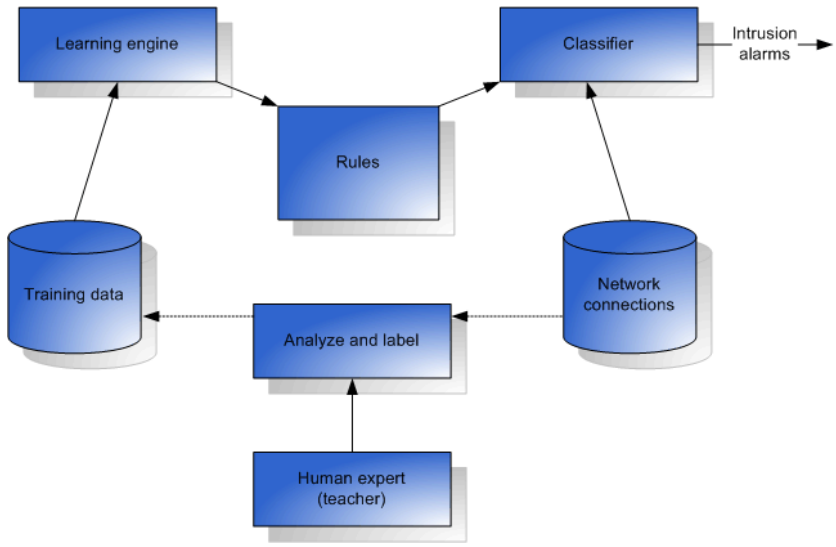

Figure 3.2: Supervised intrusion system with labeled data

3.4.2 Unsupervised learning

Unlike the supervised learning, unsupervised learning does not have a teacher to tell what is a 'good' or 'bad' connection. It has the ability to learn from unlabeled data and create new classes automatically. In figure 3.3 with the use of a clustering algorithm it is illustrated how unsupervised learning operates. First, the training data is clustered using the clustering algorithm. Second, the clustered weight vectors can be labeled by a given labeling process, for example by selecting a sample group of the data from a cluster and label that cluster center with the major type of the sample. Finally, the labeled weight vectors can be used to classify the network connections.

3.4.3 Supervised or unsupervised?

Monitoring network traffic shows a lot of activities in the sense of different data packets being sent forth and back constantly. Of course the magnitude of this activity depends on the network monitored. If a network of a home computer, which is only used for e-mail checking and internet browsing, is monitored, it will show little traffic activity, but if a busy server on the Internet is monitored, it will show a great deal of activity. Intrusion detection systems should be able

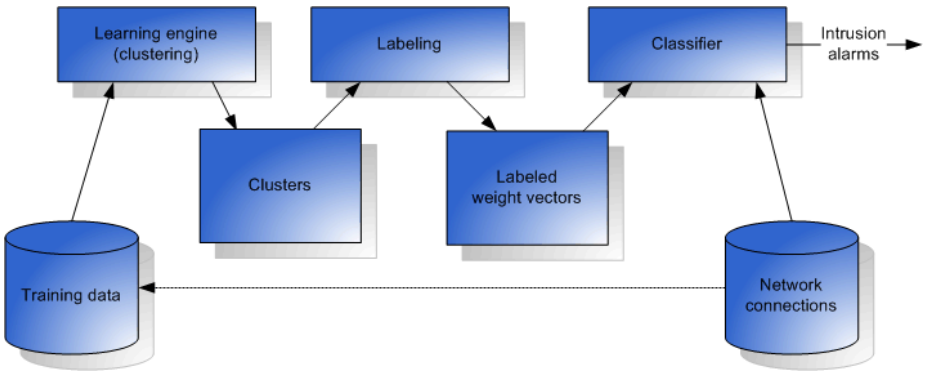

Figure 3.3: Unsupervised intrusion system with unlabeled data

to monitor and categorize (or label) traffic at the same time regardless of the size of the traffic activity. But in networks with large traffic rate, labeling data becomes a tough task. It is time-consuming and usually only a small portion of the available data can be labeled [17]. At packet level it may be impossible to unambiguously assign label to data. On the other hand in real application one can never be sure that a set of labeled data examples are enough to cover all possible attacks. These considerations are important and should be taken into account when choosing network paradigm.
Some serious work on this area, see [17][15], shows the tradeoffs between supervised and unsupervised techniques in their application in intrusion detection systems. The outcomes of these articles show that the supervised algorithms exhibit excellent classification accuracy on the data with *known* attacks. However evaluating supervised algorithms on data sets with *unknown* attacks shows a deteriorating performance. On the other hand unsupervised algorithms show no significant difference in performance between known and unknown attacks. In fact, in all data sets the attacks are unknown to the unsupervised algorithms. Finally, comparing these two paradigms exhibits that the accuracy of the unsupervised algorithms on both data sets is approximately the same as that of supervised algorithms on the data set with unknown attacks.
In relation to this project, we may say that the paradigm that fulfills the criteria defined in chapter 2 best is the *unsupervised learning paradigm*. The fact that there is no significant difference between supervised and unsupervised algorithms on data sets with unknown attacks and that there is no need for laborious labeling process when working with unsupervised algorithms plays a decisive role for choosing the unsupervised learning paradigm for this project. In practical use this paradigm is also easier and cheaper to maintain, since there is no need for a human expert to define new attacks to the system.

3.5 Summary and discussion

In this chapter we have introduced the very basics of the the neural networks. It has been briefly pointed out what a neural network consists of and what it is capable of. Some examples of how and where the neural network principle is used have been listed in order to show the practical use of it and which fields have benefit of this unique technique. In relation to this we have laid down the foundations for some of the different learning processes, which are the most important part of a neural network. Under this section we have described and discussed 3 learning processes which are basic for the design of the neural network. There are of course other learning processes, which are omitted here but are available in [12]. The reason for doing that is simply because they do not form a potential design decision for the implementation of the intrusion detection system. That is mostly because they are not used in unsupervised models.
One of the most important mark of this project is the network paradigm used to solve the main problem. After an introduction of the 2 different paradigms, we have discussed and argued why we found the unsupervised network paradigm to be more suitable for this project.
We complete this discussion with some concluding remarks on the worth of the neural network. The unusual properties of the network, e.g. being almost the complete opposite of a serial computer paradigm, opens up new opportunities in the sense of solving tough tasks containing major amount of data. Therefore neural network models are widely used in different both scientific, commercial and military fields and the development does not seem to end here. In the future with the development of new algorithms and improvements we will see more use of neural networks in different areas of our life.

Chapter 4

Neural Network Algorithms

In this chapter the different algorithms in neural networks that are suitable for IDS systems (only the unsupervised ones) and may be a potential candidate for later evaluation and implementation will be described. For every described algorithm there will be a simple example, showing how the algorithm operates. As an ending section for this chapter it will be discussed and pointed out, which algorithm is most qualified in the sense of building a unsupervised learning system with the given criteria.

4.1 Cluster Detection - CD

Clustering is an unsupervised learning technique which aim is to find structure in a collection of unlabeled data. It is being used in many fields such as data mining, knowledge discovery, pattern recognition and classification [13] and etc. The concept of clustering algorithms is to build a finite number of clusters, each one with its own center, according to a given data set, where each cluster represent a group of similiar[1] objects. In figure 4.1 4 clusters can be observed, due to the way data is spread. Each cluster encapsulates a set of data and here the similarities of the surrounded data is their *distance* to the cluster center.

[1] similar in the sense of position, distance, size, structure etc.

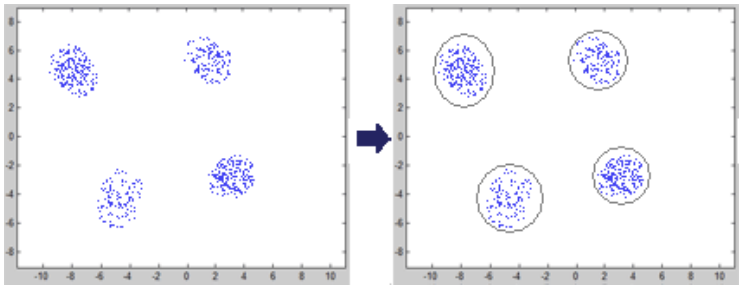

Figure 4.1: Clustering of unlabeled data

4.1.1 The *k*-means clustering

The k-means clustering algorithm [13] is one of the most famous and simplest techniques within unsupervised learning. The popularity of this algorithm is largely due to its simplicity and fast convergence. It clusters objects based on attributes into k partitions (clusters). The goal of the algorithm is to minimize an object function, which is a squared error function. The function is defined by:

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n(j)} \|x_i^{(j)} - \mu_j\|^2 \tag{4.1}$$

where there are k clusters. The term $\|x_i^{(j)} - \mu_j\|^2$ is a distance measure between $x_i^{(j)}$ and the cluster center, μ_j, and is an indicator of the distance of the n data points from their respective cluster center.
The algorithm starts by partitioning (clustering) the input points into k initial sets. Then it calculates the center of each set. Here after a binding takes place between each point and the closest center. In other words centers are being associated with the closest data points. This step is repeated and centers are recalculated and new bindings are constructed until convergence, which is obtained when the new input data points no longer change the centers.

4.1.1.1 An example of the *k*-means

This example demonstrates how the k-means under certain assumptions operate and how centers are moved until there is no change in the mean. Suppose that we have n sample feature vectors, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ and we know that they all fall into k clusters, $k < n$. Let m_i be the center of the vectors in cluster i. Assuming that the clusters are well separated, we can say that $\mathbf{x}$ is in cluster i, if $\|\mathbf{x} - m_i\|$

is the minimum of all k distances. The procedure for finding k-means is as follows:

- Make initial values for the centers $m_1, m_2, \ldots, m_k$
- Repeat until there is no change in any center
  - Use the estimated centers to classify the samples into clusters
  - For i from 1 to k
    - Replace m_i with the center of all of the samples for cluster i
  - End for
- End repeat

In figure 4.2 the procedure of the k-means is depicted. Hence this is a simple

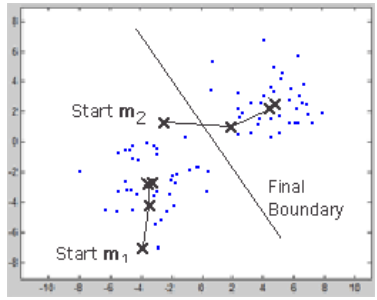

Figure 4.2: The procedure of the k-means algorithm

version of the k-means algorithm and is in its very basic formulation. The idea of the k-means algorithm is widely used in other neural network algorithms, like Single Linkage Clustering [15], online k-means algorithm [22], Lloyd algorithm [19].

4.1.1.2 Weaknesses of the *k*-means

Despite the simplicity and fast convergence, k-means has some remarkable drawbacks. The first limitation does not only concern k-means, but all other algorithms, using clusters with centers. The inevitable question is how many centers should be there and how many does your data set really need? In k-means the free parameter is k and the results depend on the value of k. Unfortunately,

there is no general theoretical solution for finding an optimal value of k for any given data set. We saw in the specific situation depicted in figure 4.2 that the value of k is 2, meaning that there are 2 clusters, each with its own mean (or center). But what if the same algorithm was applied to the same data set

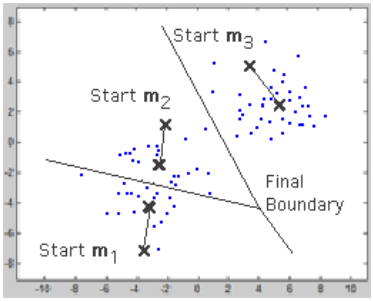

Figure 4.3: 3-means clustering

producing 3-means clustering, see figure 4.3. Is it better or worse than 2-means clustering? There is no specific answer to this question but one way to deal with this is to compare the results of multiple runs with different values of k and then choose the best one according to a given criterion. One way or another the free parameter k is still to be defined by a user and this would raise a discussion on how 'unsupervised' the k-means algorithm is. Then there is a smaller limitation, like how are the means initialized. It frequently happens that suboptimal clusters are found. To avoid this the centers should be initialized with different values so that they don't overlap other clusters. Yet these kind of limitations could be considered less important when compared to the question about how to determine the number of k.

4.2 Self-Organizing Map - SOM

The Self-Organizing Map [12][1] is a competitive network where the goal is to transform an input data set of arbitrary dimension to a one- or two-dimensional[2] topological map. SOM is partly motivated by how different information is handled in separate parts of the cerebral cortex in the human brain [12]. The model was first described by the Finnish professor Teuvo Kohonen [9] and is thus sometimes referred to as a *Kohonen Map*. The SOM aims to discover underlying structure, e.g. *feature map*, of the input data set by building a topology preserving map which describes neighborhood relations of the points in the data

[2]SOM can also transform into three- or more-dimension, but this is rarely used [1]

set. The SOM is often used in the fields of data compression and pattern recognition. There are also some commercial intrusion detection products, that uses SOM to discover anomaly traffic in networks by classifying traffic into categories. The structure of the SOM is a single feedforward network [14][12], where each source node of the input layer is connected to all output neurons. The number of the input dimensions is usually higher than the output dimension. The neurons

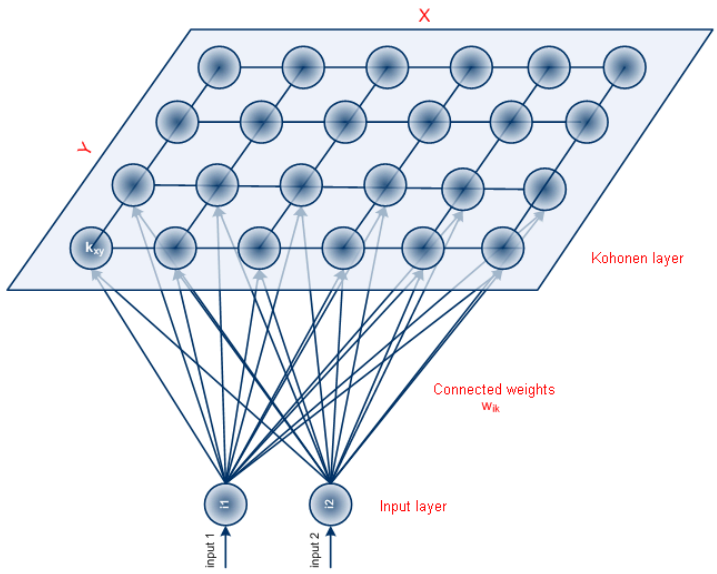

Figure 4.4: The self-organizing (Kohonen) map

of the Kohonen layer in the SOM are organized into a grid, see figure 4.4 and are in a space separate from the input space. The algorithm tries to find clusters such that two neighboring clusters in the grid have codebook vectors close to each other in the input space. Another way to look at this is that related data in the input data set are grouped in clusters in the grid.
The training utilizes competitive learning, meaning that neuron with weight vector that is most similar to the input vector is adjusted towards the input vector. The neuron is said to be the 'winning neuron' or the *Best Matching Unit* (BMU) [9]. The weights of the neurons close to the winning neuron are also adjusted but the magnitude of the change depends on the physical distance from the winning neuron and it is also decreased with the time.

4.2.1 The learning algorithm of the SOM

There are some basic steps involved in the application of the SOM algorithm. Firstly the weights of the network should be initialized. This can be done by assigning them small values picked from a random number generator; in doing so, no prior order is imposed on the feature of map. The only restriction is that the weight vector, $w_j(0)$ should be different for $j = 1, 2, \ldots, l$, where l is the number of neurons in the lattice. An alternative way of initializing the weight vector is to select from the available set of input vectors in a random manner. The key point is to keep the magnitude of the weights small, because the initial weights already give good approximation of the SOM weights. Next step is the *similarity matching*. With the use of the Euclidean minimum-distance criterion, the distance from the training data set to all weight vectors are computed and based on these computations the BMU is found. The Euclidean formula is given by

$$i(\mathbf{x}) = \arg\min_j \|\mathbf{x}(n) - \mathbf{w}_j\|, \quad j = 1, 2, \ldots, l \tag{4.2}$$

where $i(\mathbf{x})$ identifies the best matching neuron to the input vector $\mathbf{x}$. In words this formula finds the weight vector most similar to the input vector, $\mathbf{x}$. This process sums up the essence of the competition among the neurons. In the sense of network topology there is a mapping process involved with this competition; *A continuous input space of activation patterns is mapped onto a discrete output space of neurons by a process of competition among the neurons of the network* [12].

After having found the winning neuron the next step of the learning process is the *updating*. The weight vector of the winning neuron and the neurons close to it in the SOM lattice are adjusted towards the input vector. The update formula for the neuron j at time (i.e., number of iteration) n with weight vector $\mathbf{w}_j(n)$ is

$$\mathbf{w}_j(n+1) = \mathbf{w}_j(n) + \eta(n) h_{j,i(\mathbf{x})}(n)(\mathbf{x} - \mathbf{w}_j(n)) \tag{4.3}$$

where $\eta(n)$ is the learning-rate parameter and $h_{j,i(\mathbf{x})}(n)$ is the time-varying neighborhood function centered around the winning neuron $i(\mathbf{x})$. A typical choice of $h_{j,i(\mathbf{x})}$ is the *Gaussian* function [17][12], which is given by the formula

$$h_{j,i(\mathbf{x})}(n) = \exp\left(-\frac{d_{j,i}^2}{2\sigma^2(n)}\right) \tag{4.4}$$

where $\sigma(n)$ is a width function and $d_{j,i}$ represents the distance between the winning neuron i and its neighbor neuron j.

The whole process is repeated for each input vector over and over for a number of cycles until no noticeable changes in the feature map are observed or a certain number of epochs is reached.

Input vector	(1,-1)	(1,-1)	(1,-1)
Weight vector	(2.2, -1.3)	(-0.6, 1.9)	(3.1, -2.7)
Distance	(1.237)	(2.927)	(2.749)

Table 4.1: Calculation of distances between input and weight vector

4.2.2 An example of the SOM

In this section we will present a simple example of how SOM operates.
Assume that we have a network with two input neurons (neuron 1 and 2) and a Kohonen layer with 4 rows and 4 columns. Two input neurons and 4x4=16 neurons in the Kohonen layer gives 32 connecting weights in the network. The weights connecting the input and Kohonen layer neurons will be initialized to random numbers. Let weights have initial values that lies in the interval $[-\pi, \pi]$. There are two main processes during the training. Firstly, the distance between the input vector and the weight vectors will be calculated in order to find a winning neuron. Off course, the winning neuron will be the one with the shortest distance to the input vector. We use the Euclidean minimum-distance formula (formula 4.2) and get the results shown in table 4.1. To have an idea of how the calculations are carried out and avoid messy calculations we use three weights vectors with random values. The distance is calculated by

$$distance = \sqrt{(1-2.2)^2 + (-1-(-1.3))^2} = 1.237$$

which is the distance between input vector (1,-1) and weight vector (2.2,-1.3). In table 4.1 we see that the weight vector with the shortest distance to the input vector is the first one, which makes the neuron 1 in the Kohonen layer the winning neuron. In figure 4.5 we can see how the values are located in the network.
Secondly, the weights are to be updated. For that we use formula 4.3 and below is an example of how it is calculated. Updating weights of neuron 2 in the Kohonen layer (neuron 2 has the position (1,2) in the Kohonen layer). The winning neuron has the position (1,1)

$$d = \sqrt{(1-1)^2 + (1-2)^2} = 1$$

$$h = \exp\left(-\frac{1^2}{2 * 0.2^2}\right) = 3.73 * 10^{-6}$$

$$w_{1,2} = -0.6 + 0.5 * 3.73 * 10^{-}6 * (1 + 0.6) = \underline{-0,599997019}$$

where $w_{1,2}$ is the weight going from input neuron one to neuron 2 in the Kohonen layer.

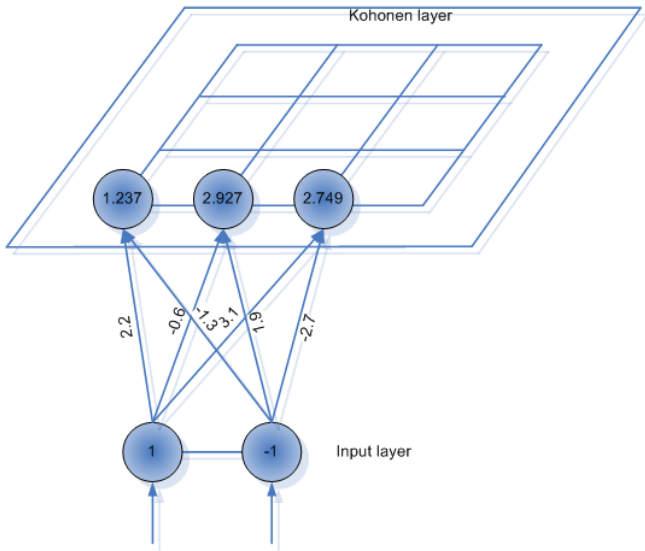

Figure 4.5: Self-organizing map with calculated values

4.2.3 Advantages and disadvantages of the SOM

The self-organizing map is an easy-to-understand algorithm due to its simplicity. It is also therefore an algorithm that can be easily implemented in a computer environment. Moreover the SOM is an effective algorithm that works. The excellent capability to visualize high-dimensional data onto 1- or 2-dimensional space makes it unique especially for dimensionality reduction.
On the other hand there are some serious drawbacks. The number of neurons affects the performance of the network. Increasing the number of output neurons will increase the resolution of the map, but computation time will dramatically increase. To obtain a better clustering result, various numbers of neurons must be evaluated and from these observations an optimal number of the neurons can be decided. Regarding the time consumption the SOM is one of the most time-consuming algorithms. The more the dimension of a data set increases the more time it takes to compute a result. This is because every time an input vector is given to the network, the distance between every single element in that vector to every single neuron in the network must be computed and compared subsequently. Relating this phenomena to intrusion detection system, it could be a significant vulnerability. For example if retraining the network, that operates as a detection system, takes a couple of days, then it becomes inconvenient in practice due to the damages that could be caused by the attack during the

retraining of the network. However the question about how big the dimension of the data should be, must be answered and with proper data analysis and preprocessing this problem can be avoided to some degree.

4.3 Principal Component Analysis - PCA

The PCA (also known as the *Karhunen-Loève transformation* in communication theory) is commonly used in statistics in the fields of pattern (image) recognition, signal processing and data compression. In this area of science statistical analysis becomes very problematic when data has too many features (variables). Cases like this give poor statistical coverage and thus poor generalization. PCA is a linear transformation technique that transforms multidimensional data to lower dimension while retaining the characteristics of the data set. This process is known as *dimensionality reduction* [12], which is a common characteristic for the most unsupervised learning systems. Data is transformed to a coordinate system so that the greatest variance of the data by a projection of the data ends up on the first component (coordinate), the next one in line on the magnitude of the variance ends up on the second component and so on [The Peltarion Blog].

4.3.0.1 An example of the PCA

The following simple example illustrates how PCA transforms data and thereby constructs its components. Suppose that we have 2 dimensional samples (x_1, x_2) as plotted in figure 4.6. We can easily observe that x_1 and x_2 are related, meaning that if we know the value x_1 we can make a reasonable prediction of x_2 and vice versa since the points are centered around the line $x_1 = x_2$. To see how data is spread the data is encapsulated inside an ellipse and vectors P1 and P2 are plotted. These vectors are achieved by rotating the axes over $\pi/4$. This situation is depicted in figure 4.6. P1 and P2 are the principal component axes, the base vectors ordered by the variance of the data. With the given 2D example we have made a $[X, Y] \rightarrow [P1, P2]$ transformation. Generally PCA is used with high dimensionality problems.

4.3.1 PCA with Hebbian learning

As mentioned the PCA is widely used in statistics and there are traditional ways of calculating it. Some of this is based on covariance matrices and some on singular value decomposition. However, these methods require great amount

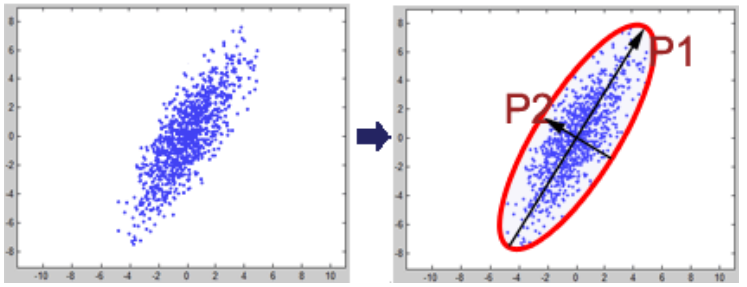

Figure 4.6: Data encapsulated and components produced with PCA

of processing power and memory and are useless for larger data sets [12]. Using Hebbian Learning (see section 3.3.1) with the right choice of update rules is a good alternative for calculation. Hebbian in its basic form has the following update rule:

$$w_i(n+1) = w_i(n) + \eta y(n) x_i(n) \quad i = 1, 2, \ldots, m \tag{4.5}$$

where n denotes discrete time and η is the *learning-rate parameter*. The regular Hebbian rule would make the weights grow uninhibitedly, which is unacceptable on physical ground [14] and will give unreasonable values. To solve this, Erkki Oja introduced what is called Oja's rule that normalizes the weights so that they don't diverge. The normalized update rule will look like (the proof is omitted here but can be found in [14][12]):

$$w_i(n+1) = w_i(n) + \eta y(n)[x_i(n) - y(n)w_i(n)] \tag{4.6}$$

which is also know as the 'Oja learning rule'. This learning rule modifies the weight in the usual Hebbian sense since the first product term is the Hebbian rule, $y(n)x_i(n)$. The second negative term, $-y(n)w_i(n)$ is responsible for stabilization (normalization) of the weight vector.

4.3.2 Limitations of the PCA

First of all the PCA is a linear method, which means that PCA can be used to solve problems that involves linear data set, like in the presented example in section 4.3. In cases where relations are not fairly linear, PCA fails when it produces the largest variance as it is not along a single vector but along a non-linear path [12]. Figure 4.7 shows a case where data is not related linearly. However, this raises discussion about how to use and represent data. If a data set contains both linear and non-linear data, would it still be possible to use

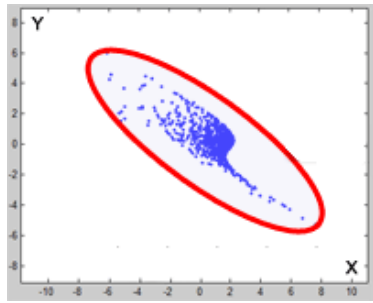

Figure 4.7: Non-linear data set.

PCA? The answer lies in the distribution and amount of linear and non-linear data in the data set. If the amount of non-linear data is insignificant it is likely to be omitted and linear data can be used to build a unsupervised network with. Depending on the analysis of dataset to be used, PCA can be suitable for data sets involving non-linear data.

4.4 An Intrusion detection system with SOM

So far we have gained insight in the various models of the unsupervised learning algorithms. Furthermore in relation to the intrusion detection systems we have pointed out the advantages and disadvantages of the algorithms in question. Based on these considerations we can now point out an algorithm for evaluation and implementation. This choice is also inspired by the works and tests in the articles of [15][22][18][17]. The Self-Organizing Map is chosen to be used as the learning algorithm for the desired IDS. The choice is made according to the following statements

Simple and easy-to-understand algorithm that works. We have analyzed the good and bad sides of this algorithm and came to realize that it is capable of dealing with large problems that require reckoning and comparing without any complexity. As for the IDS that we want to construct, we need an algorithm that can manage to transform a high dimensional data sets into a 2-dimensional data set. The simplicity of the self-organizing map makes it easy to implement and manage.

Topological clustering. The self-organizing map has the ability to construct a topological result. This feature will be useful during the training and

test phase of the IDS in order to observe the validity of the result from the algorithm and follow up on the clustering process to check whether same patterns (e.i. features in this case) fall into the same cluster.

Unsupervised algorithm that works with nonlinear data set. As the traffic from a network connection can be a huge amount and is most likely representing nonlinear data, we will need an algorithm which is operational regardless of the amount and the linearity of the data sets. The self-organizing map has the ability to handle such data set. Another noticeable character of this algorithm is that it is unsupervised, which makes it capable of detecting intrusions without being introduced to it.

So, do these statements exclude the choice of another algorithm? The answer is no, because the other algorithms with certain conditions can also work as the learning algorithm for intrusion detection. Our choice is based on the specifications and requirements for our IDS and therefore we decide to use the self-organizing map due to its properties listed in the statements from above. We can not tell for sure that the self-organizing map is the best choice. The only way of finding out its quality is to compare it with the other unsupervised models covered in this chapter. A reliable way is to evaluate and implement so many models as possible, and then compare them on the efficiency. But since we are going to implement only one algorithm, we have tried to build our choice around the properties and some of the articles listed in the beginning of this section.

4.5 Summary and discussion

We have in this chapter described and analyzed some examples of unsupervised learning algorithms, that can be used to construct an intrusion detection system with. The algorithms in question are chosen due to the fact that they are the most known and widely-used ones when it comes to intrusion detection. Firstly we have introduced the cluster detection, which is an overall model that involves several techniques using the idea of clustering data. As an example to this model, the k-means was presented, which is a simple and popular algorithm in neural networks. In practical, using k-means in its simple and basic form is not a convenient technique, therefore there are lot of enhancement possibilities, some of them can be found in the bibliography. Then we moved to the next algorithm, the self-organizing map, which aims to discover features within a given data set and builds up its clusters accordingly. The principle behind SOM is easy to understand and consequently easy to implement. The challenge lies in the process of data preprocessing, where a proper way of representing data must

be formulated. Finally, we have described the Principal Component Analysis algorithm, which differs from the clustering algorithms. Here, the goal is not to cluster data on the basis of their similarities in the data set but to build components in order to represent data. We also saw that the PCA is a good forerunner for linear data sets, but when it comes to non-linear data sets it runs into problems and it becomes difficult to represent data.
There are many other neural network algorithms which operates best in their own field. The algorithms in this chapter are some of the good alternatives to solve a problem of intrusion detection kind. Especially, the self organizing map, which looks for features in a given data set, can be useful since most intrusion detection systems operate by recognizing and comparing patterns (i.e., features). In the next chapter we will investigate network traffic and aim to find features that come with network connections.

Chapter 5

Network connections and features

In this chapter we will look into the structure of a network connection. A description of the content of IP packets (such as protocols, services etc.) and what kind of features the connection holds will be presented. Based on these observations we will work out a strategy for representing these features and use them for the neural network chosen in the previous chapter. In order to do all this a socalled 'sniffer' will be needed. As the name implies, a sniffer is used to sniff network traffic in a given environment. In this chapter we will look for ways of sniffing network traffic and form a solution for how to use such sniffer and how to process the outcomes of it.

5.1 Protocols in the Internet

Communicating with the Internet happens by using different protocols. Various Internet protocols are used in communication on the different levels of the Internet layered architecture, see figure 5.1. The protocols shown are the most common ones implemented in each layer. A *packet* (i.e., *IP* packet, see figure 5.2) travels top-down at the sender's end. For example, a data packet (e.g. a *http* packet [21]) gets encapsulated in a TCP packet which then gets encapsu-

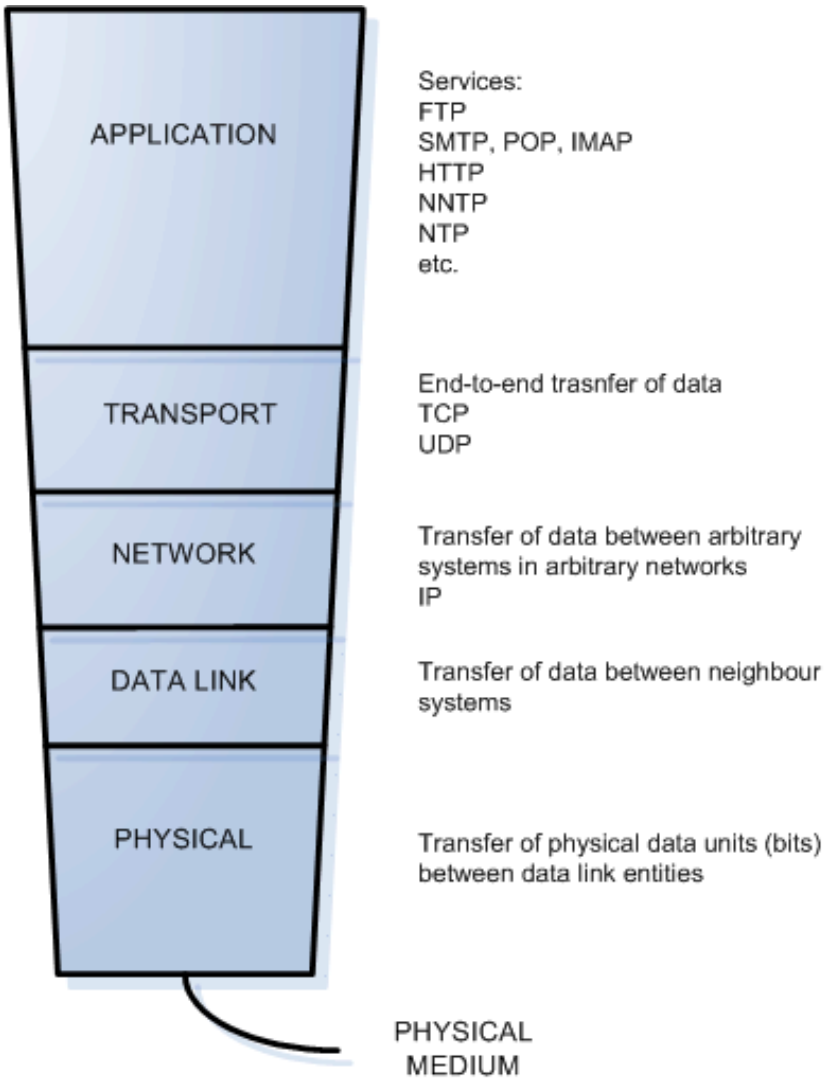

Figure 5.1: Protocols in the Internet layered architecture[21]

lated in an IP packet. In each encapsulation, headers are added to the packet and finally the packet gets encapsulated in an ethernet frame (data link layer) and moves on to the communication media for transmission. At the receivers end, the process is reversed and the packet goes bottom-up. The received packet gets stripped off at each level until it reaches its destination in the Internet layer. An IP packet contains the following types of information

1. **Version** - Indicates the version of IP currently used.
2. **IP Header Length (IHL)** - Indicates the datagram header length in 32-bit words.
3. **Type-of-Service** - Specifies how an upper-layer protocol would like a current datagram to be handled, and assigns datagrams various levels of importance.
4. **Total Length** - Specifies the length, in bytes, of the entire IP packet, including the data and header.
5. **Flags** - Consists of a 3-bit field of which the two low-order (least-significant) bits control fragmentation. The low-order bit specifies whether the packet can be fragmented. The middle bit specifies whether the packet is the last fragment in a series of fragmented packets. The third or high-order bit is not used.
6. **Fragment Offset** - Indicates the position of the fragment's data relative to the beginning of the data in the original datagram, which allows the destination IP process to properly reconstruct the original datagram.
7. **Time-to-Live** - Maintains a counter that gradually decrements down to zero, at which point the datagram is discarded. This keeps packets from looping endlessly.
8. **Protocol** - Indicates which upper-layer protocol receives incoming packets after IP processing is complete.
9. **Header Checksum** - Helps ensure IP header integrity.
10. **Source Address** - Specifies the sending node.
11. **Destination Address** - Specifies the receiving node.
12. **Options** - Allows IP to support various options, such as security.
13. **Data** - Contains upper-layer information.

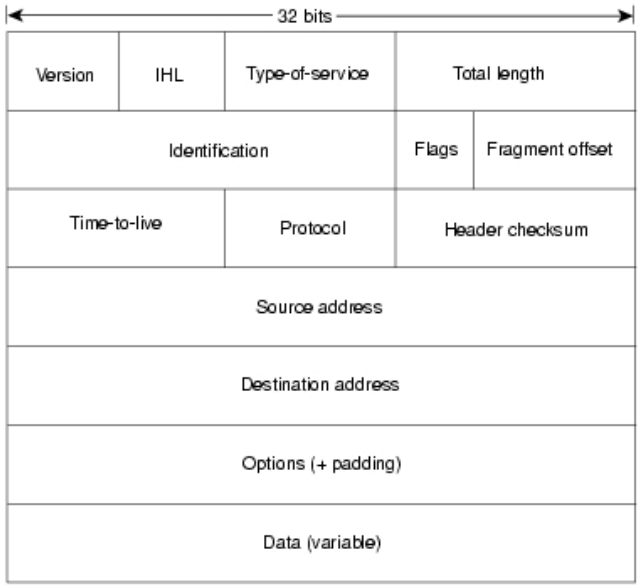

Figure 5.2: The different fields that comprise an IP packet

In figure 5.2 an IP packet is illustrated containing all previous mentioned information. There are several properties within a IP packet and more properties can be formed by combining some of the attributes. In order to form a strategy for finding features within a network connection, these properties will be examined and analyzed along with the packet stream in a network connection.

5.2 Sniffer tools

Within this section we will explore the available sniffing tools for Linux systems. The list is not long and we will look at two noticeable methods. Some short samples of sniffed traffic will be presented for each method. By the end of the section the methods will be evaluated and one will be chosen for the implementation of the IDS system.

5.2.1 TCPDump

A packet is any message that has been encapsulated in various headers that uses the IP protocol to communicate. Tcpdump [8] is a tool that does the capturing. It prints out the headers of packets on a network interface that match a certain built-in boolean expression. By doing so the tcpdump filters the packets and

```
[root@pcpro23 ~]# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode listening on eth0, link-type EN10MB (Ethernet), capture size
96 bytes 18:28:42.485775 IP csfs3.imm.dtu.dk.58962 >
pcpro23.imm.dtu.dk.5900: P 145062350 2:1450623512(10) ack 1728262234
win 63712 <nop,nop,timestamp 316133388 363302451
>
18:28:42.491657 IP pcpro23.imm.dtu.dk.5900 > csfs3.imm.dtu.dk.58962:
P 1:872(871 ) ack 10 win 1448 <nop,nop,timestamp 363302457
316133388> 18:28:42.488369 arp who-has www2.imm.dtu.dk tell
pcpro23.imm.dtu.dk 18:28:42.488529 arp reply www2.imm.dtu.dk is-at
00:02:b3:23:d0:ed 18:28:42.488534 IP pcpro23.imm.dtu.dk.32770 >
www2.imm.dtu.dk.domain:  54499+ PT R? 113.68.225.130.in-addr.arpa.
(45) 18:28:42.488985 IP www2.imm.dtu.dk.domain >
pcpro23.imm.dtu.dk.32770:  54499* 1/ 2/2 (143)

29 packets captured
111 packets received by filter
0 packets dropped
by kernel
```

Figure 5.3: Raw tcpdump output data

does not print out every piece of information that goes through the network card (i.e., the ethernet). It also happens that packets are dropped by tcpdump. Packets are dropped due to the buffer space in the packet capture mechanism overflowing which is caused by tcpdump not being able to read packets fast enough. Tcpdump simply can not keep up with the network traffic and decode it at the same time. Furthermore one must have root rights to be able to use tcpdump. In the following section a raw tcpdump output is shown, see figure 5.3 (as can be seen, the shown data is from a small tcpdump session). This little piece of raw data of the tcpdump illustrates the structure of the tcpdump.

5.2.1.1 Decomposing tcpdump output data

The shown example of tcpdump is in its raw format. Tcpdump comes with many options to refine the output and therefore the command line can become messy. For better understanding of the tcpdump lets observe the first line from figure 5.3:

```
18:28:42:485775 IP csfs3.imm.dtu.dk.58962 >
pcpro23.imm.dtu.dk.5900:  P 145062350 2:1450623512(10) ack
1728262234 win 63712 <nop,nop,timestamp 316133388 363302451>
```

The information within this line is as follows

- The black part is the time the packet came across the network card (not part of the packet)
- The blue part is the source and source port and the destination and destination port of the communication taking place
- The red part is TCP flags (the `ack` flag indicates acknowledgement of the receipt of the data from the sender)
- The orange part is the byte sequence/range
- The olive part is the window size of bytes that the source (sender) is currently prepared to receive
- The purple part is the TCP type of service

Packet structure and information is dependent on the nature of the packet. This example packet involves TCP, port 5900 (used for remote desktop connection i.e., VNCViewer).
In the particular example (in figure 5.3) no packets are dropped, but if (and normally the tcpdump sessions tend to be active longer time) the tcpdump were running for a longer time space, it will drop some of the packets that do not satisfy the build-in boolean expression or simply because tcpdump can not keep up with the traffic. But do we need the dropped packets anyway?
If we are going to monitor a network connection for intrusions we can not afford dropped packets. What if some of the dropped packets contain malicious code which is exactly what we are looking for? Undoubtedly, the fact that tcpdump sometimes is unable to keep up with the network traffic and resulting in a series of dropped packets makes the tcpdump unreliable.
In relation to the IDS, we want to build, there is one thing that should be taken into account. Raw tcpdump output must first be summarized into network connection records using preprocessing programs (i.e., Bro [16]). As shown in figure 5.3, tcpdump has its own way of organizing the sniffed packets. However combined with different command options and using additional packet filtering and reassembling engines it is possible to organize the sniffed traffic in a more personalized way.

5.2.2 PCAP

Pcap [8] stands for Packet Capture Library and provides a high level interface to packet capture systems. For Unix-like systems the implementation of the pcap is known as libpcap[1]. The libpcap library provides implementation-independent access to the underlying packet capture facility provided by the operating system. Also the library enables users to program a *user-specific* sniffer, by using the methods and data structures (i.e., `ip.h`, which is a structure representing an IP header along with its attributes, recall figure 5.2) provided by the libpcap library. The available methods, which can be seen in the manual page for *pcap*, can only be accessed with root privileges. So, what does pcap offer? Basically the pcap allows us to use it with a program to capture packets traveling over a network, to transmit packets on a network at the link layer and gives us a list of network interfaces that can be used with pcap. More detailed, pcap provides an interface that can grab packets in their raw format from a network interface (i.e., ethernet).
To illustrate how pcap works we will demonstrate a small program that sniffs a single packet. The program can be found in appendix D. The output of this pro-

```
Network Devices found
---------------------
Device Name ::eth0

Packet number 1:
       From: 64.233.183.99
         To: 130.225.69.163

Protocol: TCP      Src port: 80    Dst port: 44443
TCP flags: 0x10    Payload (1430 bytes):
```

Figure 5.4: Capturing of a single packet using pcap

gram is shown in figure 5.4. The program discovers the first available network interface device, which in this case is called `eth0`. Subsequently the program grabs a packet and displays some information regarding the packet. We can see where the packet is from and where it goes to (the `From` and `To` fields). Furthermore we are being told that the packet in question is a TCP packet, which means that more information can be revealed as the TCP packets have their own packet structure and attributes (e.g., `src_port`, `dst_port`, `TCP flag`, etc.) In the given pcap example we saw how we can program the sniffer to provide exactly the information we want it to. This makes pcap a flexible tool when it

[1]Files and documentation can be downloaded from www.tcpdump.org

comes to the implementation of the IDS.

5.2.3 Using pcap as sniffer

Not surprisingly, we choose to use pcap as the sniffing tool for the IDS system. The reason behind this decision is related to the flexibility that comes with pcap so that we can implement our own specific sniffer device based on the requirements we have defined. During the implementation phase this will enable us to code the sniffer in a way that will be fully operational with the rest of the system (e.g., the GUI and neural network). By doing so there will be no need for a preprocessing program to process traffic in order to obtain intrinsic features of the packets. With pcap we are able to retrieve these features as soon as the packet is captured.
During traffic sniffing, the sniffer should be set to promiscuous mode. Normally the sniffer program will not capture all packets, unless you tell it to do so. We are interested in capturing all packets to be sure that we do not miss any packets that might have an important value to be used in feature construction. During the activation of the network interface in the implementation phase we will tell the program to operate in promiscuous mode (this is done by giving the second parameter of the method `pcap_live_open()` a true value).

5.3 Feature construction

Once the sniffer starts collecting the passing packets from the ethernet they will be examined and processed in order to construct *features*. Lee and Stolfo [16] describe in their article a strategy for finding features within a network connection from the attributes of packets through some procedures that could identify an intrusion. Within this section we will shortly outline the principles behind feature construction. The following section describes how features will be represented for the IDS program.

5.3.1 Making general features

It is important to define features as generally as possible before presenting them to the self-organizing map algorithm. Some of the features Lee and Stolfo have described in their article represent specific events and some of them are intended to reveal attacks like SYN flood (a DoS attack, where a client sends several

packets (`SYN`) to a server, the server responds (`SYN-ACK`) and awaits confirmation (`ACK`) from the client, but the client never sends back confirmation [6]), which is still possible but most unlikely nowadays. Newer TCP implementations do not take any effect from the SYN flood attack and this type of attacks (that exploits bug in TCP implementation) are mostly historical. Our strategy for defining features will differ from the one Lee and Stolfo uses. While they tend to define features for specific attacks (i.e., `failed_login`, `overflow`, `SYN flood` etc.) we will try to define our features more generally. This has also something to do with the self-organizing map we have chosen for the IDS. The SOM is an unsupervised learning algorithm, which means that it does not know if some input data set is malicious or not. It only knows the patterns of the data set and classifies data according to their patterns. Therefore by making general features we will cover a wide range of general network traffic patterns, so that SOM can identify and distinguish these patterns by letting different patterns activate different neurons. Also when using SOM every cluster in the map represents data sets that somehow are similar. The cluster becomes a general representation of these data sets. Therefore instead of having two almost-similar features (i.e., `failed_logins` and `logged_in`, `root_shell` and `su`, etc. [16]) activating two different clusters, we will try to merge them into a more general feature and activate only one cluster. For example all login related features (i.e., `failed_logins`, `logged_in`) will be merged into one single feature called logins. We will still use some of the features defined by Lee and Stolfo and in addition define some new features which are more general and collective.
Before defining features we should be aware of defining dependent features that will be statistically inappropriate and affect the training and testing of the SOM algorithm. The features should be independent of each other, meaning that a feature should not be built on other features.

5.3.2 Frequent episodes

We will use an automatic procedure for parsing socalled *frequent episodes* [16] and thereby construct features. Frequent episodes describe the study of the frequent sequential patterns of network traffic in order to comprehend the temporal and statistical nature of the many attacks as well as the normal behaviors. This is why we use frequent episodes to represent the sequential traffic records in order to calculate values for the features. The generated frequent episodes will be applied to data sets containing network traffic (i.e., IP packets). The idea is to identify intrusion patterns through comparison with normal traffic data. But before we can compare patterns we need to represent the features in an appropriate way so comparison is possible. One way of doing so, is to convert the features into 'numbers' such that 'similar' features are mapped to closer numbers. In this way feature comparison and intrusion identification is

accomplished by comparing the numbers.

5.3.2.1 Procedure for parsing frequent episodes

The following procedure (the same procedure described in [16]) describes how features are built by using different intrinsic features in a given time space:

- Assume F_0 (e.g., dst_host) is used as the reference feature, and the width of the episode is w seconds.
- Add the following features that examine only the connections in the past w seconds that share the same value in F_0 as the current connection:
  - A feature that computes the count of these connections;
  - Let F_1 be *service*, *src_dst*, or *dst_host* other than F_0 (i.e., F_1 is an essential feature). If the same F_1 value (e.g., http) is in all the item sets of the episode, add a feature that computes the percentage of the connections that share the same F_1 value as the current connection; otherwise, add a feature that computes the percentage of different values of F_1;
  - Let V_2 be a value (e.g., S0) of a feature F_2 (e.g., *flag*) other than F_0 and F_1 (i.e., V_2 is a value of a nonessential feature). If V_2 is in all the item sets of the episode, add a feature that computes the percentage of connections that have the same V_2 value; otherwise, if F_2 is a numerical feature, add a feature that computes the average of the F_2 values.

This procedure parses frequent episodes and computes values to represent temporal (because connections are measured in time windows and share the same reference feature value) features by using three operators, *count*, *percent*, and *average*. The difference between essential and nonessential features is that essential features describe the anatomy of an intrusion, for example the same *service* (i.e., *port*) is targeted while the actual values (i.e., http) are not important since the same attack method can be applied to different targets (i.e., ftp). The values of nonessential features indicate the invariant of an intrusion (i.e., *flag* = S0) because they summarize the connection behavior according to the network protocols.

5.3.3 Intrinsic features

Some of the features that Lee and Stolfo describe in their article are called the *intrinsic* features. These features are extracted from the ip packets (or perhaps from a preprocessing mechanism) and do not identify intrusions. It other words they are intrinsic for a single connection and can not tell whether a connection is an intrusion or not. In table 5.1 the intrinsic features and their descriptions are listed. These features are directly read from the ip packets without any

Feature	Description
`duration`	Length (number of seconds) of the connection
`protocol_type`	Type of the protocol, e.g. TCP, UDP, etc.
`service`	Network service on the destination, e.g. http, telnet, ftp, etc.
`flag`	Normal or error status of the connection
`src_bytes`	Number of data bytes from source to destination
`dst_bytes`	Number of data bytes from destination to source
`wrong_fragment`	Number of 'wrong' fragments
`urgent`	Number of urgent packets

Table 5.1: Intrinsic features of Network Connection

preprocessing. The sniffing tool, Pcap, has a structure definition implemented so that the features defined in table 5.1 can easily be fetched.

5.3.4 Features from intrusion patterns

The features defined so far do not expose any intrusions. In order to expose intrusions that we will use the the procedure for parsing frequent episodes from section 5.3.2.1 to build up 'time-based traffic' features. These features will identify attacks like SYN flood, Port-Scan, etc. containing calculated values. Table 5.2 summarizes time-based features. The time-based features are constructed as follows:

- the 'same host' features that examine only the connections in the past 2 seconds that have the same destination host as the current connection:
  - the count of such connections, the percentage of the connections that have the same service as the current one, the percentage of different destination services, the percentage of SYN errors, and the percentage of REJ errors.

Feature	Description
count	Number of connections to the same host as the current connection in the past 2 seconds
serror	% of connections that have 'SYN' errors
rerror	% of connections that have 'REJ' errors
same_srv	% of connections to the same service
diff_srv	% of connections to different service
srv_count	Number of connections to the same service as the current connection in the past 2 seconds
srv_serror	% of connections that have 'SYN' errors
srv_rerror	% of connections that have 'REJ' errors
srv_diff_host	% of connections to different hosts

Table 5.2: Traffic Features of Network Connection

- the 'same service' features that examine only the connections in the past 2 seconds that have the same service as the current connection:
  - the count of such connections, the percentage of different destination hosts, the percentage of SYN errors, and the percentage of REJ errors.

By using this procedure we are able to find the features listed in table 5.2 and use the computed values to examine network traffic in order to find intrusion behaviors.

5.3.5 Other potential features

Some other features can be defined by observing the packets. The features defined in table 5.1 and 5.2 does not reveal anything about the payload (e.g., the actual data within the IP packet). The data within a packet may give us a clue about whether the packet is a part of an intrusion or not. If we look at the size of the payload in IP packets, we can to some degree determine what the purpose is or what kind of packet it is. For instance, if the size of the payload is huge, it will basically tell us that the IP packet contains data of a big file that is being downloaded/uploaded. But it is a fact that some IP packets with huge amount of payload do not necessarily represent a part of a big file. The payload could be full of junk data or empty spaces. On the other hand if the data is zero or nearly zero we would assume that the IP packet might be an ICMP packet used for pinging hosts or a probing attack (e.g., port scan). However, we can not always be sure of these assumptions. One way of clarifying this is

to compress the payload and make the following rules.

- If the size of compressed payload is the same or nearly the same size of uncompressed payload, it might be an indication of that the IP packet carrying that payload is a part of big file being transmitted.
- If the size of the compressed payload differs remarkably from the uncompressed payload, it might be an indication of an IP packet containing junk data.

This could be a feature called `compressed_payload`, which is general (can be applied to all packets and does not represent a specific event) and maybe help detecting probing attacks, as the probing attack send IP packets with very small payload and concentrate more on discovering services (e.g., port numbers) they can break into.
Another feature description that might be interesting is time of the day the IP packet is captured. Most of the attacks happen to take place during the night. We could define a feature called `time_of_day` that records the clock and date of the captured packet. This is another general feature that by itself does not detect an intrusion but combined with other features could give valuable information. Picture a scene where packets with small or no payload is detected during the night. Again, it does not mean that this *is* an intrusion but it is suspicious and there is a possibility of it being an intrusion.
Using the same approach it is possible to define many general features that could provide valuable information and make it easier to detect intrusions.

5.4 Intrusion detection process with the given features

The defined features gives us a clue about behaviors in network traffic. As for the SOM algorithm, it does not tell whether a packet or a series of packets is an intrusion but can provide information about all kind of behaviors including intrusions. Assume that the algorithm is well-trained with normal and non-attack network traffic. For example, we start testing the algorithm with normal traffic data including a probing (e.g., port scan attack) attack. As result we should have a map where the normal traffic data is spread around certain points and attack traffic data is spread around other certain points in the map. The IDSnet will provide log files about all traffic data that has been used for testing and in this log file we can read what kind of traffic data activates a point in the map. If everything goes well, we should be able to read features values of a

point, that has been activated by the probing attack, but we do not know that the point represents an attack yet. Then the feature values will be examined. If we look at the values of the features, `rerror` and `diff_srv`, we should get high percentages. That is due to the fact that a probing attack sends a lot of raw IP packets to different port numbers (e.g., services) in order to find open ports. And normally it will get a lot of rejections as the most port numbers are not available. From this kind of behavior we can define the following rule;

`rerror` $\geq 83\%$, `diff_srv` $\geq 87\% \rightarrow$ probing attack.

The rule says; if for the connections in the past 2 seconds that have the same destination host as the current connection, the percentage of rejected connections is at least 83%, and the percentage of different services is at least 87%, then this is a probing attack [16].
This example also illustrates how attacks are detected with the SOM algorithm. The algorithm will classify data by letting similar patterns activate the same neuron (e.g., point in the Kohonen map). Then all other patterns that do not activate neurons that represent normal traffic data are suspicious and potential attacks.

5.5 Scaling and transformation of the features

Most of the defined features have values from 0 to 100 expressing a percentage value. Others are count values and they can have values from 0 to the size of the count. Two of the intrinsic features (e.g., `protocol_type` and `service`) have values that start from 0 and end at 65535. Given these conditions we find ourselves in a situation where our values of the input data sets to the algorithm of the self-organizing map vary from each other. From a statistical point of view this is not an appropriate way of providing data to the algorithm. The optimal will be presenting input values having the same scale. Especially the SOM algorithm will be doing fine when all the values of the input data have the same scale (i.e., a scale from 0 to 100). The weights of the algorithm will then have the same scale and both weight initializing and propagating will become easier. A transformation of some of the features is needed.

5.5.1 Transformation of features

The intrinsic feature `service` (see table 5.1) can have a value from 0 to 65535, there are 65536 port numbers in total. Since some of the port numbers (i.e., 80,

25, 443, etc.) are so commonly used with the Internet, we will point out the port numbers that are most used and most important and transform them into smaller numbers. The argumentation for this decision is simple. In figure 5.5

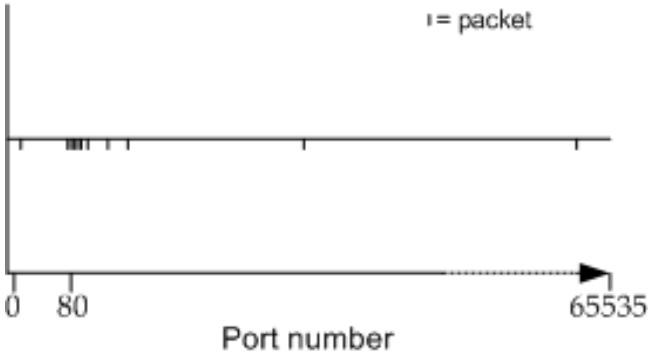

Figure 5.5: Packet concentration around port numbers

we see a horizontal line that represents the port numbers from 0 to 65535. When using the Internet some port numbers are more often used than others. For instance the port number 80 (e.g., *http*) is the world wide web service and

Port Number Range	Description
[0, 1023]	Officially assigned for use by the standard Internet application servers
[1024, 49151]	Can be registered for use with specific applications
[49152, 65535]	Can be used freely, for example, when ports have to be dynamically allocated

Table 5.3: Port number assignments [7]

for sure the most used service when surfing on the Internet. If we choose to give every port number without scaling or transformation as input to the self-organizing map, we will see that the algorithm will cluster inputs that have port numbers close to each other. This will not give any significant result as there is a big difference between using port 80 and 81. On the other hand, there is no big difference between using port, say, 45678 and 45679. In table 5.3 we see how port numbers are assigned and we also observe that the most important port numbers are in the range [0, 1023], which approximately is 1/60 of all the port numbers. In order to represent the important port numbers and avoid those ports to fall into the same cluster, we make the following transformation, see table 5.4. In this table there is a list of Internet services which are the

Port Number	Service Type	Description	Transformation
1	tcpmux	TCP Port Service Multiplexer	1
7	echo	Echo	2
20	ftp	File Transfer	3
21	ftp	File Transfer Control	4
22	ssh	SSH Remote Login Protocol	5
23	telnet	Telnet	6
25	smtp	Simple Mail Transfer Protocol	7
53	domain	Domain Name Server	8
80	http	World Wide Web HTTP	9
110	pop3	Post Office Protocol - Version 3	10
119	nntp	News Service	11
123	ntp	Clock Synchronization	12
143	imap	Internet Message Access Protocol	13
389	ldap	Lightweight Directory Access	14
443	https	http protocol over TLS/SSL	15

Table 5.4: Transformation of important Internet port numbers

most common ones. Each of these services is transformed into a number. For example port number 80, which is the *http* service is transformed into number 9. The rest of the port numbers are transformed in table 5.5. As mentioned

Port Numbers	Transformation
[0, 1023]	16
[1024, 49151]	17
[49152, 65535]	18

Table 5.5: Transformation of the remaining port numbers

before we are only interested in TCP, UDP and ICMP packets, which means that the `protocol_type` features can have 3 different values only. These values are presented in table 5.6.

5.5.2 Scaling of the features

The total number of the transformed port numbers are 18. This means that the `service` feature can have a value from 1 to 18. The count features (e.i., `count`, `srv_count`, etc.) start from 0 and end at the size of the count. In order to have the same scale for all of our features we will need to scale them, so that

Protocol type	Feature value
ICMP	1
TCP	6
UDP	17

Table 5.6: Feature values of the protocol types

every feature has a value within the interval [0, 100]. This is more or less due to the fact that SOM does not label records, it operates and classifies better when there is a good variation in the input data set.
There are several models for scaling values. Below is a list of these models and later, it will be pointed out which method suits our features best.

- Linear scaling. Using the linear model $(a * x + b)$ features values can be scaled linearly
- Exponential scaling. Another way of scaling features values is exponentially (e^x)
- Logarithm scaling. It is also possible to scale values by using the logarithm function $(log(x)$ (base e))

The feature values that need to be scaled, will be scaled linearly.

5.6 Summary and discussion

We have covered maybe the most essential topics of the project in this chapter. The chapter gave an introduction to the very basics of Internet protocols and IP packets. These topics are also very important to understand as we are going to construct an intrusion detection system that monitors network traffic.
Afterwards, we presented the sniffer tools. This is also one of the essential parts of the project and we have gained insight to the potential sniffers and how they operate. There are not many different sniffer tools, but the available ones are enough. Pcap was chosen to be used because of its flexibility and the fact that is implementation-dependent. TCPDump is not a bad candidate, it has many options and the output can get more personalized with the use of some kind of preprocessing mechanism.
Then we moved on to discuss feature construction. We have pointed out the importance of making general features in order to give the SOM algorithm a chance to discover all kind of behaviors. If the features were as specific as the

ones in the article of Lee and Stolfo, we would not be sure of providing a good and varying input data to the SOM, and thereby could not be sure of the outcome. Therefore we have tried to define our features as general as possible.
Finally, the need of a scaling and transformation mechanism to the features in order to represent the features in a certain interval was discussed. We set up a model for transforming some of the features like the `service` feature that has an upper-bound at 65535. The most important services were found and transformed into a smaller value together with the rest of the services in intervals. The scaling assures that all feature values end up in the same interval. It has been decided to use a linear scaling so that features are scaled to a value from 0 to 100.

Chapter 6

Specifications and requirements for the IDS

The content of this chapter covers an introduction to the main problem of this project with regard to the development of the IDS, describing the purpose of it and introducing a specification of a prototype tool that is to be implemented and which is the product of this project. It will be pointed out what a such tool should be capable of and based on these considerations a proper description of the desired IDS will be formulated. Furthermore, it will be elaborated what kind of an IDS system we want to develop in the sense of where to apply the IDS. Since the tasks and data traffic depend on the location and the type of the IDS, we will specify an exact point to deploy the IDS.

6.1 The purpose of the IDS

The purpose of the IDS is to monitor network traffic, examine the IP packets from these packets compute feature values that could be used to compare with other values in order to detect intrusions. The users can observe the result by plotting it in a matrix or coordinate system. Furthermore, the purpose of the IDS program is to provide a graphical user interface with the possibility of user interaction where users will have access to the different functionalities and can

follow up on the process of training/testing.

6.2 The overall system

The IDS program consists of three parts:

The algorithm: Describes the neural network algorithm to be used in the process of intrusion detection. All calculations and computation will take place within the algorithm.

The packet sniffer: An implementation of a tool to sniff network traffic for packets. It will collect all the packets (i.e., IP packets) that goes in and out, convert them to an appropriate object representation and present them to the algorithm.

The GUI: The graphical user interface will be the platform where users can activate the sniffer, start training/testing the neural network algorithm and perform other GUI related operations. Furthermore the GUI will be independent of the algorithm. This means that the GUI can be extended with other algorithms, which will make the whole system more flexible and most important extendible.

In the following sections we will define specifications and requirements for the three parts.

6.2.1 The algorithm

We have outlined in section 4.4 the reasons behind using the self-organizing map as the learning algorithm for the IDS. It is a fact that the implementation of the SOM should be more or less painless and is an easy-to-implement algorithm. The two main functionalities of the SOM algorithm is the training (learning) and testing (detection). The testing operation should not be available before the training of the algorithm is complete. The training operation operates by taking packets (which are processed, e.g., feature values are computed) and adjust the weights of the algorithm according to the packets. Once the algorithm has reached the maximum number of epochs, it will stop adjusting weights and start passing the packets to the test operation. For every incoming packet the test operation will use the trained neural network to find a winning neuron, which will represent the packet and other packets alike in the Kohonen layer.

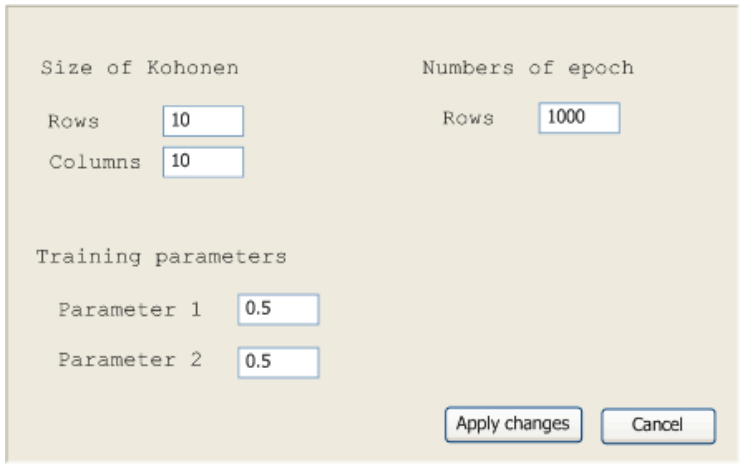

Figure 6.1: A template for changing variables in SOM

The implementation of the SOM algorithm should be dynamic. By that we mean that users of the IDS program can change some of the variables involved with the SOM, see figure 6.1. Below is a list of variables which users can change:

- *The size of the Kohonen layer.* There is no definition or theoretical proof for an optimal size of the Kohonen layer in the SOM. However there is a connection between the dimension of the input data set and the size of the Kohonen layer. The higher dimension the input data set has the larger should the kohonen layer be in order to classify and represent the input. Therefore it should be possible for users to redefine the size (e.g., rows and columns). By doing so we will be able to give users the possibility to find their own optimal size of the Kohonen by trying different values.

- *The training parameters.* During the calculation of the distances between winning neuron and neighboring neurons some parameters are used. It should again be possible for the users to change these parameters and try out different values in order to observe how good the algorithm performs. However changing the training parameters requires some knowledge of the SOM algorithm and to fully understand the purposes of them.

- *The number of the epochs.* Users should also have the opportunity to change the number of the epochs (e.g., the number of the epochs the SOM algorithm should run before stop training and become ready to test). Once again, there is no optimal value for this as the captured packets can be very different from each other and this may require a large number of

epochs before the SOM is fully trained.

With the possibility of changing some of the parameters, we give the users a choice to define their own algorithm. Off course, there will be default values to these parameters but users can try out different setups to see how the algorithm performs.

6.2.2 Packet sniffer

We have discussed in chapter 5 which sniffer tool we are going to use. The requirements for the packet sniffer is that it should capture every single packet that goes in and out of the network interface. This is possible by setting the the sniffer device to promiscuous mode. Furthermore, we are only interested in certain packets, which are:

- *IP packets.* The Internet Protocol is a network layer protocol that contains addressing information and some control information that enables packets to be routed. IP is the primary network-layer protocol in the Internet protocol suite. It provides facilities for segmentation and reassembly.
  - *TCP packets.* The Transmission Control Protocol is a connection-mode protocol which provides reliable transmission of data in an IP environment. Some of the service TCP offers is point-to-point stream service of data transfer, full-dublex operation, multiplexing, etc. TCP makes it possible to set up a large number of connections distinguished by *port numbers.*
  - *UDP packets.* User Datagram Protocol is an alternative Transport layer protocol which provides connectionless-mode service. UDP uses the same concept of ports as TCP to provide multiplexing of several streams of data. Unlike TCP, UDP adds no reliability to IP.
  - *ICMP packets.* The Internet Control Message Protocol is a network-layer Internet protocol that provides message packets to report errors and other information regarding IP packet processing back to the source.

IP is on the network layer while TCP, UDP and ICMP are on the Transport layer, recall figure 5.1. We want the sniffer to capture IP packets from the network layer. Later on, we will examine the IP packets and use only those who have protocol type TCP, UDP or ICMP. From these packets we will calculate feature values and train the SOM algorithm with them.

6.2.3 The GUI

As mentioned the GUI should be independent of the two other parts. It should provide an interface that is capable of being extended with different modules (e.g., the SOM, packet sniffer, another neural network algorithm, etc.). In the GUI users can setup their own algorithm by changing parameters, start/stop the sniffer, observe the results, etc. The look of the GUI should be as simple and user-friendly as possible. Once the program, is started there should be a list of different devices (e.g, the sniffer, SOM, etc.) on the screen. The operations of

Figure 6.2: A template for GUI

the packet sniffer is `start` and `stop` while the operations of the SOM is `record` and `stop`. Once the sniffer is started (e.g., capturing packets), the SOM can be set to record and receive the packets. Some basic information like start time, end time, number of handled packets should be placed next to each device, see figure 6.2. By double click on the SOM device a panel will pop up and parameters of the SOM algorithm can be changed here, see figure 6.1. The GUI

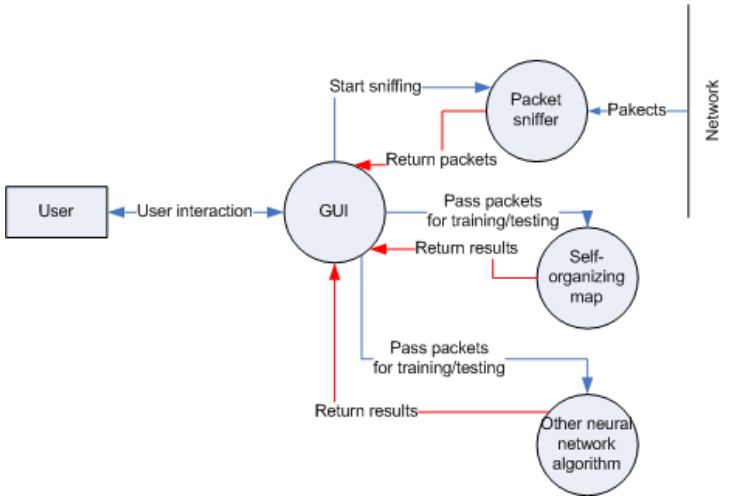

Figure 6.3: An overview of the interacting parts in IDS

will also establish the connection between the different devices, where all data exchange will go through the GUI, see figure 6.3. In other words, the GUI will be the 'centralized control center'.
The GUI will also have the task to convert the received packets from the sniffer device into appropriate object representations. For each packet an instance of this packet object will be created and will have almost all of thee same attributes as the IP packet. Accordingly the attributes of these packets will be used to calculate feature values with as described in section 5.3. Once the feature values are calculated, they will be passed on to the SOM algorithm and depending on the status of the algorithm, the feature values will be used for training or testing. The GUI will receive response from the algorithm as soon as the test starts and will also start receiving the points of the winning neurons for each tested packet. These points will be plotted and written out to log files for further examination.

6.3 Where to use the IDS

Our intention is to use the developed IDS to monitor the network connection of a computer in a socalled GRID system [9]. Shortly, the purpose of the GRID is to connect computers (in GRID these are called *clusters*) working on the same projects and provide massive computational power. By doing so the GRID can

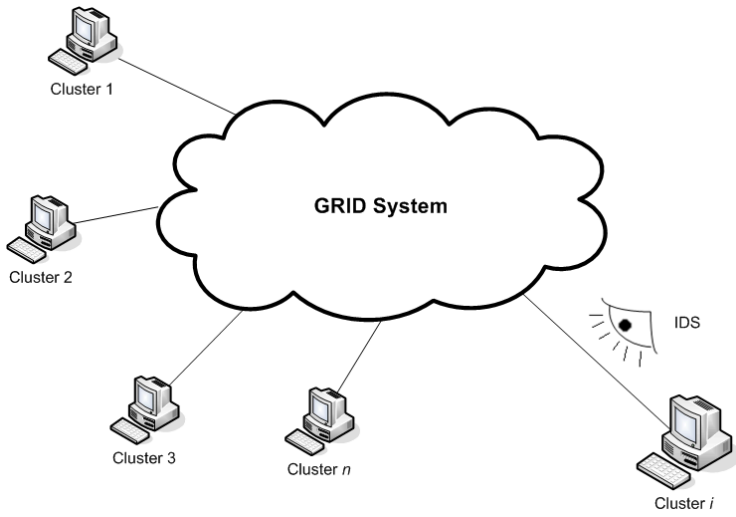

Figure 6.4: GRID system with n clusters, monitoring the ith cluster

solve exceedingly large tasks that can not be solved on a single cluster[1]As can be seen in figure 6.4 the task of the IDS is to monitor the line that connects cluster i with the rest of the system.

6.4 Summary and discussion

In this chapter we have defined requirements for the interacting parts of the IDS program. Firstly, we have clarified the purpose of the IDS program. It should act as a tool for monitoring network traffic, process the packets with the algorithm in order to train the network and detect intrusions and provide a graphical user interface, where users have access to these operations. It has been outlined that the IDS program as an overall system consists of three parts, which are more or less independent from each other. Especially the GUI, which operates as the main base for the other parts. Accordingly, we have described these parts and their tasks. Some specific requirements regarding the algorithm and the packet sniffer are also made in the chapters that deal with these topics. Finally, we have pointed out where we intend to use the IDS program. The GRID systems consists of many clusters, which are connected with each other through a big

[1]SETI@home, FOLDING@home etc. are some of the well-known projects that make use of GRID computing.

local area network or the Internet. In both cases clusters might be in danger for intrusions and need to be protected as well. Given these specifications and requirements we are now able to design the IDS program.

Chapter 7

Design of the IDS

The design chapter includes decisions on the design of the IDS. Based on the specifications made in the previous chapter, we will define an overall theoretical design together with the design of the implementation of the system. Some of the design decisions in the sense of which tools to use, are madein previous chapters, like chapter 3, 4 and 6. Due to that we will briefly summarize these decisions and concentrate more on making a design regarding the implementation.

7.1 Theoretical design

In this section we will present an overview of the theoretical design. Much of this design has been determined in terms of which concept to use in which area in previous chapters. In the following sections a summary of these decisions is presented.

7.1.1 Type of the IDS

In chapter 6 three different types of IDS were introduced. From the beginning of this project it was defined in the problem description that the aim of the

IDS we want to build has to detect intrusions that come with a network connection. This automatically excludes the system IDS type. The two remaining types are capable of detecting network intrusions. The cooperative IDS is based on information sharing, meaning that several systems cooperate on the same task. But our goal with this project is to develop a single connection IDS that monitors network traffic and detects intrusions without any contribution from other systems. So, the type of the IDS is a single connection IDS, that monitors one single network connection and detects intrusions that might occur in that connection.

7.1.2 Packet sniffer

An introduction to the potential network traffic sniffers has been made in section 5.2. It has been decided to use Pcap library as the sniffer device.

7.1.3 Design of the neural network

In chapter 4 we have introduced some candidates of neural network having the capability to fulfil the criteria of the unsupervised learning algorithm. The neural network, which will be used as the learning algorithm to the IDS, is the self organizing map, or more preciously self organizing feature map (SOFM). In the following section the different variables of the algorithm (e.g., the number of neurons in the layers, the variables involved with the equations such as the update rule and etc.) will be defined with values that will also be the default values.

7.1.3.1 Variables in SOM

Below is a list of the variables that need to have default values but some of them can be changed any time during the program execution.

- **Size of input layer**. For every IP packet the corresponding feature values will be calculated. We have defined 17 different features in total, recall sections 5.3.3 and 5.3.4. This means that the size of the input layer will be 17, consisting of 17 neurons (e.g., one feature corresponds to one neuron in the input layer). This value can not be changed.
- **Size of the Kohonen layer**. It is in this layer we will observe the results

of testing the incoming IP packets with the SOM. As mentioned earlier, there is no optimal size that would make the size of Kohonen layer the best. Based on the size of the input layer we define the Kohonen layer to have 10 rows and 10 columns. This value can be changed during the program.

- **The number of epochs**. Since there are many different packets in a network connection, there will a need for a large number of epochs. The default value will be set to 10000 epochs which means 10000 IP packets. This value can be changed.

- **The training parameters**. There are two training parameters, α and σ (the α is the learning rate and σ is the width function, recall formula 4.3 and 4.4). Both are used in calculation of the distance between neurons and will have default values, $\alpha = 0.5$ and $\sigma = 0.2$. These values can also be changed.

With the default values we get the a neural network as depicted in figure 7.1. In the following sections we will introduce a design of how to change these

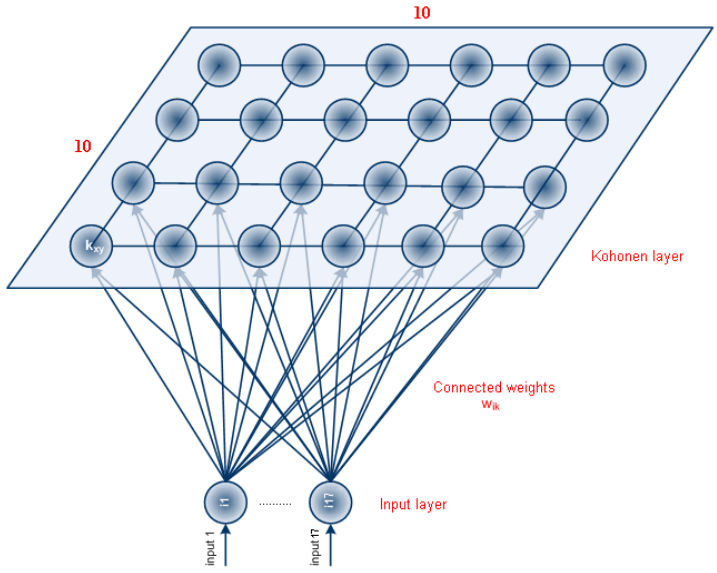

Figure 7.1: SOM with default values

parameters.

7.2 Design of the implementation

This section will cover the topics regarding the design of the implementation of the IDS. It will be decided with tools and technologies that will be used.

7.2.1 The IDSnet

Two former project students at IMM have made a system named *IDSnet*. This program is an intrusion detection system but it makes use of another type of neural network model to detect intrusions. It is implemented in C++ and QT (a cross-platform application development framework, primarily used for the development of GUI programs [2]) The system includes some functionalities that we can use in our project. It is built in a way that fulfills our specifications and requirements and is designed to be extended with other features like implementing an another neural network algorithm. Two of the three main parts of our system, recall section 6.2, the packet sniffer and the GUI, is implemented in the IDSnet system and we can use these with our own SOM algorithm. The

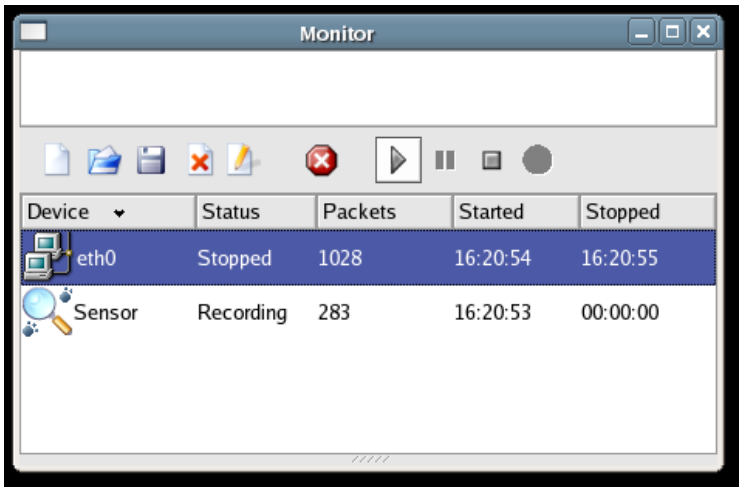

Figure 7.2: The main screen of the IDSnet

following functionalities in the IDSnet will be used:

Packet sniffer: The IDSnet has a built-in packet sniffer, implemented with the pcap library. It is designed to detect the first available network interface

and activates it to sniff all kind of packets. It is also set to promiscuous mode. We need to make a little change to the implemented sniffer. As stated before we are only interested in those IP packets that have either TCP, UDP or ICMP as the protocol type. The sniffer will be redesigned to filter all other packets.

The GUI: The IDSnet has a convenient and simple graphical user interface, where the main screen provides access to the devices and their functionalities, see figure 7.2. We will use this layout and integrate our own algorithm to it.

The IDSnet consists of many lines of source code and is very complicatedly implemented. Without changing to much of the original code of the IDSnet, the SOM algorithm will be implemented separately and attached to the IDSnet so that it appears in the list of devices.

7.2.2 Modelling of the overall system

Since the task is to develop an extension module to the IDSnet, we will first design the classes that represent the SOM algorithm. In figure 7.3 we see a

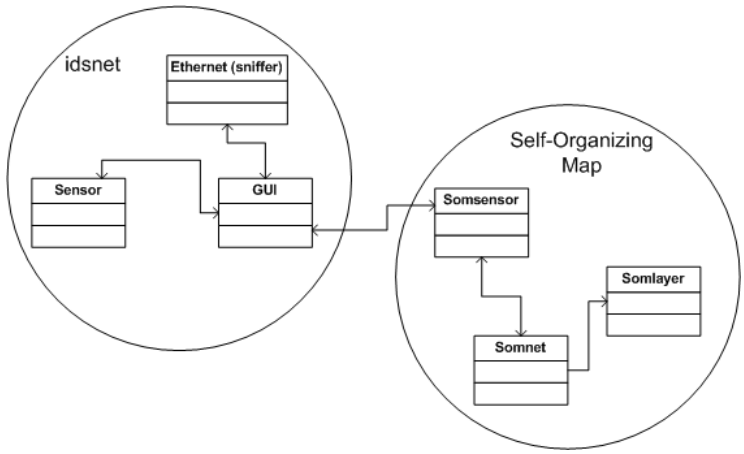

Figure 7.3: Class diagram of the IDSnet + SOM

rough sketch of the classes involved with the IDSnet. Furthermore, we extend the IDSnet with three new classes that represent the SOM algorithm.
Now, with the extension module in order, we define the classes of SOM algorithm

by defining their methods and variables. In figure 7.4 the classes representing

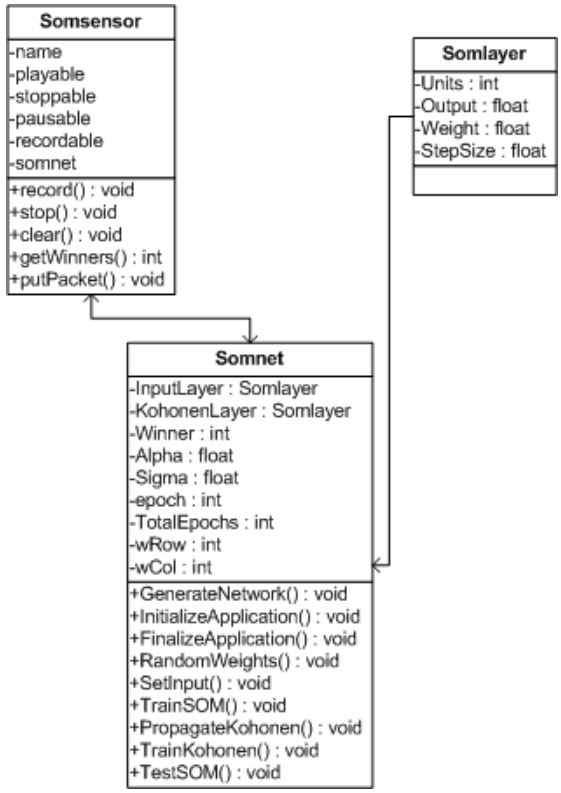

Figure 7.4: Class diagram of SOM

the SOM algorithm is depicted as an UML class diagram.

7.2.3 Design of extension module - SOM

In order to avoid any complexity and to fully benefit from the functionalities of the IDSnet, the extension module describing the SOM algorithm will be written in C++ and QT. The neural network implemented in the IDSnet (which is represented by the device called *Sensor* in figure 7.2) is implemented in such way that the concept can be reused and applied to another algorithm, in this case it will be the SOM algorithm. In this way a new device (called *Somsensor*) will be added to the list of the devices in the IDSnet and will have similar operations like record, stop and clear.

7.3 Summary and discussion

We have in this chapter proposed a design solution to an IDS program that operates with the self-organizing map algorithm. As mentioned in the introduction to this chapter, much of the design decisions regarding the theoretical aspect of the IDS are decided in previous chapters and we have made a summary of these decisions in this chapter. The design of the SOM algorithm involves defining some default values to the parameters of the algorithm. The given default values are not optimal but reasonable values to start training the algorithm. Users can measure the efficiency of the SOM algorithm by trying different parameter values.
Later, we presented the intrusion detection system, called *the IDSnet*, which is a product of two students at IMM. The IDSnet is based on detecting intrusions by finding socalled *signatures* in network traffic and analyzing these signatures with the use of a neural network algorithm. The IDSnet provides an interface which can be extended. Much of the functionalities in the IDSnet, especially the packet sniffer, can be reused with the SOM and this will save us some time. In relation to, this we have extended the class diagram the IDSnet with new classes that will form the SOM algorithm and gave descriptions of these classes with their methods and variables.
The design proposals will be used as basis when we start the implementation of the SOM algorithm. Luckily, the IDSnet program is extendible but unfortunately, it is also a big project, which will take time to familiarize with.

Chapter 8

Implementation

It has been clarified in previous chapter (e.g., the design chapter) how to implement the SOM algorithm as an extension to the IDSnet system. In this chapter, we will briefly outline how we have implemented the extension and attached to the IDSnet. Since the packet sniffer and graphical user interface are provided by the IDSnet, we will in this chapter concentrate more on the implementation of the SOM algorithm. The other two parts will briefly be explained in terms of how they are implemented and how they operate.

8.1 Development environment

First of all we present the tools and their versions numbers, that have been used in implementing the SOM algorithm.

Programming language : The standard C++ version 4.0.0 20050519

GUI language : QT version 4

Sniffer tool : Libpcap version 0.8.3

Development tool : KDevelop integrated development environment version 3.2.0

Operating system : Fedora Core version 4 kernel 2.6.11-1.1369_FC4smp

These tools are also used with the implementation of the IDSnet, but the version numbers the developers have used turned out to be older than the ones we have used. This resulted in incompatibilities as we started to make the IDSnet fully operational with the new versions of the packages. Especially the C++ version 4.0.0 caused some problems during compilation, as it turned out that version 4.0.0 is more intolerant and strict[1] compared to the older versions of C++. But after a while we got the IDSnet running and started off on our project.

8.2 Implementation of IDSnet

Without going too far into details of the implementation of IDSnet, we will briefly outline how it works. IDSnet is a big project that has many features. The GUI forms a big part of the entire code. It operates as follows; The program starts and the engine of the IDSnet discovers the first discovered ethernet device and use it to sniff packets. The neural network algorithm is also listed in the list of devices. The algorithm is represented by the device called sensor. Double click on the sensor a panel pops up giving the users the opportunity to setup the sensor. In this panel users can follow up on the process of intrusion detection.
The IDSnet has an object representation describing the captured packets. This means that every captured packet will create an instance of this object, copying the information into an appropriate header structure. It is designed to represent TCP, UDP and ICMP packets and to have the attributes, that can be easily accessed and used during feature value calculation. More information on the IDSnet can be found in [5].

8.3 Implementation of the SOM algorithm

The class diagram of the SOM algorithm from the design chapter shows the structure and relations of the classes. We have used the same organization and created three classes that represent the new device named *Somsensor*. This class extends a namespace called *Devices*, which includes the packet sniffer and a neural network algorithm as well. It has record (records packets from the sniffer), stop and clear operations.
The Somsensor device is automatically added to the list of devices in the IDSnet

[1] In terms of obeying the syntax rules, that former versions did not complain about

when it starts. Whenever the record button is pushed, an instance of the object, *Somnn*, is created. This object organizes the SOM algorithm by generating a

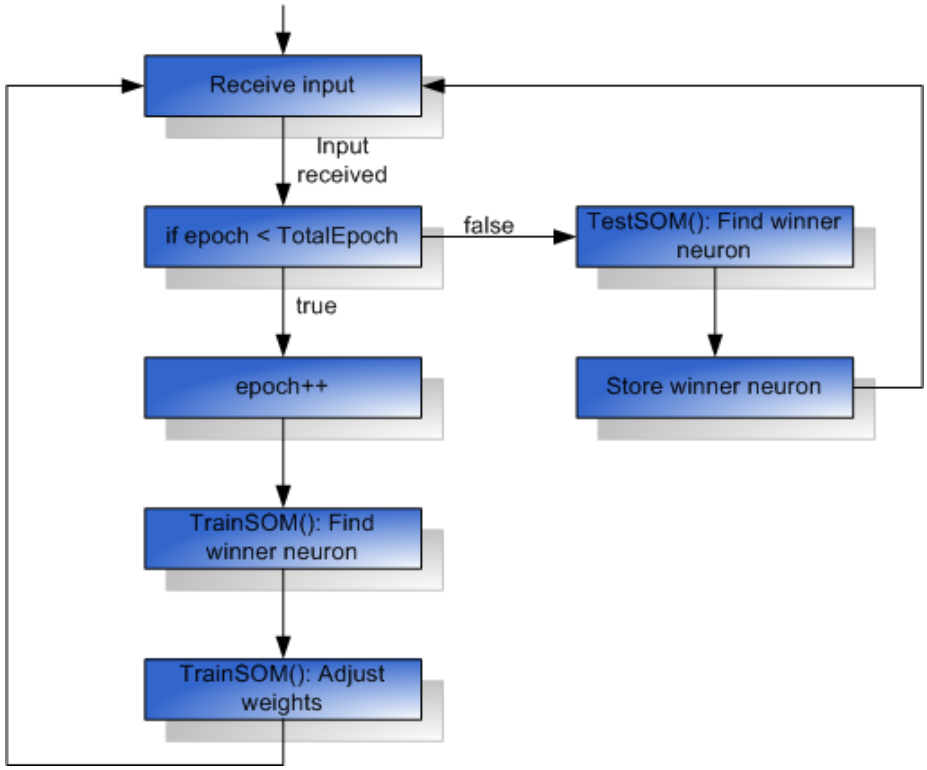

Figure 8.1: Train and test procedures of SOM

network, initializing weight values and etc. To create the input and Kohonen layer, two instances of the object *Somlayer* is created. At this point the SOM algorithm generated and ready to be trained. The training and testing procedures is depicted in figure 8.1, which also illustrates how the implementation of SOM works as a whole.

8.3.1 Receiving packets

Once the Somsensor is set to record, the sniffer (called ethernet in IDSnet) can be set to play (e.g., start capturing packets). The Somsensor has a method that will be called every time a packet is captured by the sniffer and passed on

to the Somsensor. Within this method the packet will be processed in order to calculate feature values that will characterize the packet in question. Once these values are calculated, they will be passed on to the Somnn, which will be ready to receive them. According to whether the status of the algorithm is training or testing the received packet will be processed through the neural network. If the status is training, the packet will given to the net as input and weight adjustments will be made. But if the status is testing, the packet will be sent through the network and the winning neuron, that matches the input best, will be recorded and stored in a list (i.e., a vector).
Off course, before any test can happen the network should be trained. Therefore when the record button is pushed the network will start training until it reaches the total epochs and accordingly starts training. During the test for each received packet a winning neuron (e.g., the activated neuron) will be found and stored in a list. These winning neurons will be plotted on the screen, see following section how it is done.

8.4 Implementation of the SOM GUI

We have extended the IDSnet GUI with a new panel that is reserved for the SOM. Double click on the Somsensor device pops up a panel divided in two sections. The first section (the one placed on the right side) is the panel where users can change parameters of the of the SOM algorithm. The changeable parameters are defined in section 7.1.3.1. Once the parameters are changed the apply-button should be pushed in order to apply the new parameters to the SOM algorithm. This change of the parameters should be carried out before the training. In the other section of the panel (the one on the left side) is a map representing the Kohonen layer. This map is dynamic and can be changed according to user wishes. During the training the map stays empty, because the algorithm is still adjusting its weights. Whenever the test starts, the list of stored winning neurons will be plotted on to this map by clicking the update-button in the first section. The GUI of the SOM algorithm can be seen in figure 8.2. The map shows how may packets activate the same neuron and according he theory the packets with same patterns should activate the same neuron the Kohonen layer.

8.5 Summary and discussion

The implementation of the SOM algorithm and GUI is now complete. Within this chapter we have given an explanation of how we have implemented the three

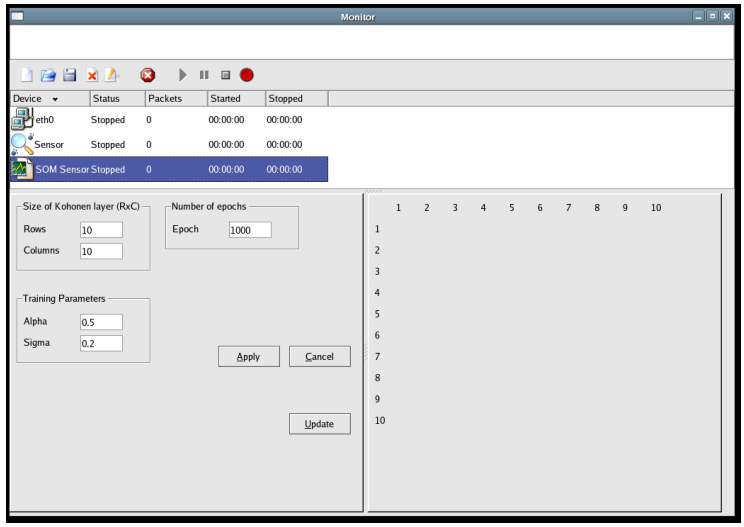

Figure 8.2: The GUI of the SOM

parts without digging to much into the technical side. A brief introduction to the implementation of the IDSnet was presented. A little note to this; as mentioned a couple of times before getting familiar with the IDSnet project required many hours of studying and editing. It is a complicated and huge project that involves many theories and technologies and one has to know these theories as much as possible in order to extend the IDSnet program. Fortunately we have managed to gain enough knowledge about the IDSnet project and successfully added an extension. Once the knowledge was acquired it became easier and the implementation of the classes of the SOM algorithm was realized.
Roghly, we have described how we have implemented the algorithm and the GUI of the self-organizing map and in broad outline explained the ideas. The IDSnet has been extended without any noticeable intervention to the original code and the original neural network and other parts of the IDSnet work fine as before. The IDSnet system has been a great tool to benefit from. It is open for extensions and we have used this option to extend the IDSnet with a new neural network algorithm, a whole new feature to the system, which will make it better and more flexible.
During the implementation of the SOM it was intended to organize classes and objects in the same way as in the IDSnet. We managed to realize this by copying the class and object structures whenever it was possible and ended up with an extension which does not violate with any concepts of the IDSnet system.

Chapter 9

Test of IDSnet with SOM

In this chapter two areas of the implemented IDS will be tested. These areas are the functionality test and the efficiency test. For both cases an overall test strategy will be formed that will aim for providing solid test result in order to make any conclusions on our unsupervised intrusion detection system.

9.1 Test strategy

The first test, the functionality test, is a test where the system is looked at from a user's view. Basic operations will be tested, primarily the GUI, to acknowledge that the program does what it is intended. The focus in this test will be on the SOM algorithm. It will be tested whether the implemented SOM device is fully operational with the IDSnet and that it can perform its task using the facilities provided by the IDSnet. Furthermore the GUI of the SOM device will be tested in order to confirm that the changes made in the
On the other test we will test the efficiency of the algorithm. The result of this test will give us an implication of how good and efficient the SOM algorithm works in connection with intrusion detection. This test will also give us a clue about whether the unsupervised algorithm has been a good choice or the implementation of the algorithm is not complete. By this we mean that the implementation of the SOM could need improvements, in terms of making the

SOM more suitable to the task of intrusion detection. To achieve test results in this area we will use two data sets, where one of them is for training and without any intrusion and the other one is for testing having normal traffic data together with intrusions.

9.2 Functionality test

The functional test can be found in appendix B.

9.3 Efficiency test

With the efficiency test we aim to test the efficiency of the SOM algorithm in terms of how well it performs in detecting intrusions. In order to do that two scenarios will be described. But before the description of the scenarios we will introduce a tool that will be used to create intrusion.

9.3.1 Nmap

Nmap [3] (Network Mapper) is a free and open source utility that is being used for network exploration and security auditing. It can perform scan on large networks and reveals what hosts are available, what services to those hosts are offered, what operating system the host uses and etc. It is a sophisticated port scanner that sends raw IP packets to scan hosts.
Altough Nmap is meant to be used for security reasons it is a common prelude to an intrusion attempt, a way of finding out if any vulnerable service is running. Hackers are easily tempted by the Nmap to quickly find out vulnerabilities on a victim host. Our intention is to use Nmap to perform port scan on a host, where the IDSnet program with SOM algorithm is running. The goal is to detect this intrusion (e.g., port scan intrusion also known as PROBING) and see if the SOM algorithm is capable of classifying the attack differently than normal traffic.

9.3.2 Description of the test

The test is carried out by collecting network traffic in two different data sets. The first data set will consist of normal traffic. By normal we mean ordinary traffic

which has no intrusion traffic but simple TCP/UDP/ICMP packets sent trough different port numbers. The second data set will also have normal data but also a probing attack created with nmap. The idea is to train our algorithm with the normal traffic data, so that the neural network is trained to know normal traffic and can recognize them during the test phase. Once the algorithm is trained we will do two tests of the trained algorithm. The first test is to test the trained algorithm with the normal traffic data. This test will show how normal data traffic is spread out in the Kohonen layer and we can observe which neurons in the map are activated. In the second test we will test the algorithm with the normal network traffic data including a probing attack. The expectation is that the trained SOM algorithm will activate other neurons for the attack and thereby prove its capability of detecting intrusions.

9.3.3 Creation of traffic data

As the traffic generator we will use a 15-days trial software called Nsasoft Network Security Auditor [4], which has a built-in traffic emulator, see figure 9.1. The emulator will be used to create normal traffic data by sending TCP/UDP/ICMP packets to the target host with the IDSnet. We will then

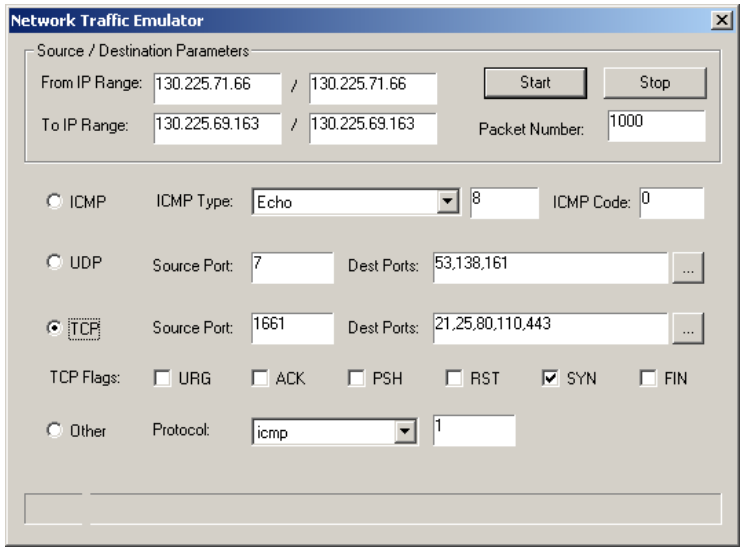

Figure 9.1: Nsauditor Network Traffic Emulator

collect these packets and save them to a file. In the first round 10000 packets

will be collected and used for training. Secondly we will use the same traffic data set and extend it with a probing attack created with nmap. Nmap has a command option to scan all ports on the target destination, which the IDSnet program can be used to sniff those packets representing the probing attack.

9.3.4 Test setup

The SOM algorithm will have a Kohonen layer with 10 rows and 10 columns. The number of runs (e.g., epochs) will be set to 10000 which also means 10000 IP packets. The connecting weights will be initialized to random numbers in the range $[0, 100]$. The SOM algorithm will be trained with the first normal data set of 10000 packets. Accordingly, it will be tested with the same data set. And finally, the data set with probing attack will be tested.

9.3.5 Test results

The algorithm was trained with the first data set. Then we tested the trained algorithm with the same data set and got the following Kohonen layer depicted in figure 9.2. We see 5 neurons that have been activated by the data set. Some neurons (e.g, point (5,9)) has been activated 5250 times which means that the feature values of 5250 packets must have been so similar that they activated the same neuron. On the other hand only 10 packets have activated point (5,6). In the project CDROM there is a log file in the path `/IDSnet/idsnet/WinnersAndFeatures.txt` where it is possible to see what kind of packets activated the different neurons and in appendix C there is a draft of that file. The content of the file is sketched in table 9.1. In this table only one packet is shown but in the log file there are much more packets and information. Features with value 0 are not included in the log in order save space. In this log file we see that the packets that have activated point (5,9) are mostly TCP and ICMP packets. Point (6,1) is activated by UDP packets using big port numbers, point (10,3) is also activated by UDP packets using even bigger port numbers and so on. See the log file for further inspection.
With the trained algorithm we started testing it with the second traffic data set with a probing attack. The result can be seen in figure 9.3. Here, we observe that other neurons are activated. These are (2,2), (6,5) and (9,8), which so far meets our expectation of seeing other neurons getting activated by the probing attack. By examining the log file we see that the packets that activate for example neuron (2,2) are some UDP and ICMP packets using port numbers over 45000.

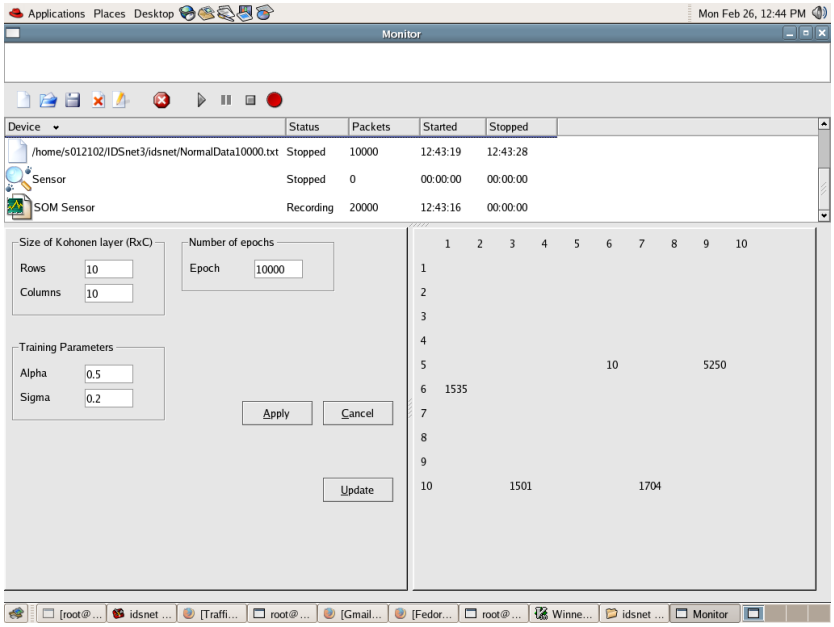

Figure 9.2: Test of the SOM with normal traffic data

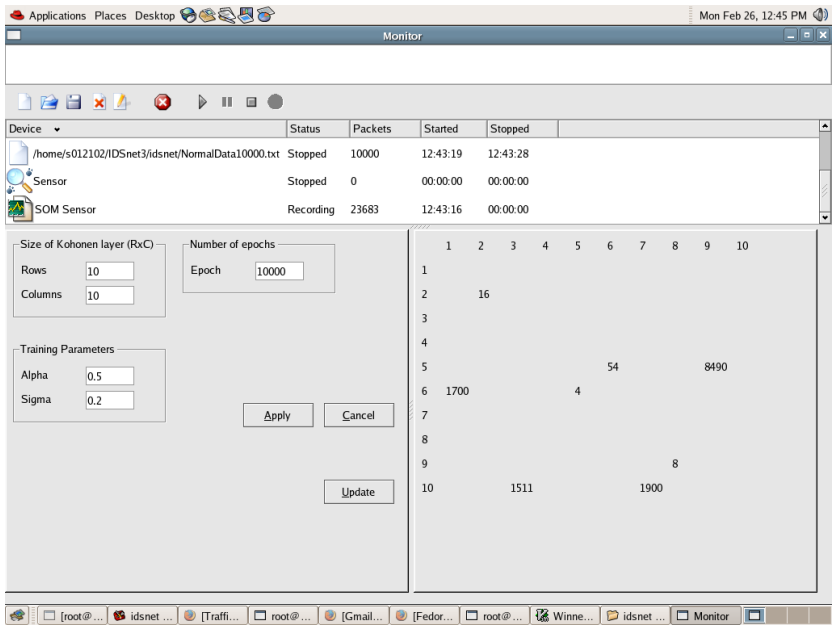

Figure 9.3: Test of the SOM with normal traffic data + probing attack

(5, 9)	Point
f[0] = 0	duration
f[1] = 6	protocol_type
f[2] = 80	service
f[3] = 100	src_bytes
f[4] = 3.086	dst_bytes
f[5] = 0	flag
f[6] = 0	wrong_fragments
f[7] = 0	urgent
f[8] = 1	count
f[9] = 0	serror
f[10] = 0	rerror
f[11] = 100	same_srv
f[12] = 0	diff_srv
f[13] = 100	srv_count
f[14] = 100	srv_diff_host
f[15] = 0	srv_serror
f[16] = 33.333	srv_rerror

Table 9.1: Log file

9.3.6 Concluding remarks on the efficiency test

We have observed how the SOM algorithm behaves to the two different traffic data sets. In the first one we saw a good variation of the data in the Kohonen layer. Similar packets did activate same neurons. In the second test other neurons were activated due to the probing attack. From the visual perspective, we see that the SOM algorithm finds and activates new neurons for unknown behaviors in the data set, recall figure 9.3. But unfortunately, when we investigate the log file, that tells which packets activate which neurons, we do not really acknowledge the precision of the implement SOM algorithm. Some of the packets from probing attack did activate other neurons than the normal traffic data did. But most of the packets from probing attack fell into the neurons that describe normal traffic data, because the total amount of packets with the probing attack was nearly 1500 packets and only 28 packets activated new neurons in the kohonen layer. This means that the rest of the packets must have activated the same neurons as the normal traffic data. This could be caused by different things. Maybe the algorithm was not properly implemented, or the collected data was not the best, or the features generated were simply not enough for a task of this kind. We will try to find answers to these questions in the last chapter.

9.4 Summary and discussion

In this chapter we have made tests of the IDSnet program with the SOM algorithm. The first test was a functional test, that proved the (mostly graphical) functionalities of the SOM algorithm to be working well. The efficiency test aimed to show the precision of the SOM algorithm and how data is classified in the Kohonen layer. The results showed that the algorithm somehow failed to classify the probing attack as it classified the packets representing the probing attack together with the normal traffic data. However, on the visual plan we did see that new behaviors were detected and to some degree the SOM could identify the probing attack. In order to fully understand how the implemented SOM algorithm works, the log file containing information about features values of the packets must be examined carefully. And in this file we see that most of the packets of the probing attack were similar to the packets of normal traffic data. This is probably the reason for classifying packets of probing attack together with packets of normal traffic.

Chapter 10

Concluding remarks

In this chapter we will summarize the achievements we have obtained in this project describing them whether they have been successful or not.

10.1 The SOM algorithm

We have in this project investigated 3 different unsupervised neural network algorithms. Based on what we have read in various articles about these algorithms and the research we have made in chapter 4 we decided to implement the self-organizing map as the learning algorithm. The implementation of the algorithm was successful, however, the efficiency test did not really confirmed the precision of the algorithm. We were able to observe how the SOM algorithm performed during the test and could also see that it was capable of detecting new behaviors. But during the test most of the intrusion packets activated the same neuron as the normal traffic, which shouldn't happen. One reasonable explanation to this could be that the implemented SOM algorithm was not properly implemented. With this we mean that the implemented algorithm may not have been specified precisely to handle such a task. An investigation of the implemented algorithm could maybe reveal the need for enhancements in the implementation. But we have a strong feeling that the problem lies with the

features (see following sections). As conclusion to the choice of the SOM algorithm we may say that it has been a reasonable choice, which was mostly based on the articles and the properties that came with the SOM. Surely, this choice could have been more justified if we had implemented the other algorithms as well and compared their efficiencies.

10.2 Working with the IDSnet

We have been given an intrusion detection system program called *IDSnet*, which is a project prepared by former students at IMM. The good thing about this program was that it had implemented a packet sniffer and could construct features from those packets, which we needed for the preparation of our project. But the IDSnet program was very difficult getting familiar with due to its complexity and many lines of code. Unfortunately, the familiarization process cost us many days and we had hard time extending the IDSnet. Making the program run did also caused us troubles. Finding the right packages and installing them took long time. Once the packages were installed we discovered some bugs in the code. We didn't expect such problems when we started working on the code, but we kept working on it and tried to solve the problems as they started appearing. The motive power for continuing working on the IDSnet was the preimplemented tools that we could use for our project and to end up with an IDSnet program, that would have two different neural networks implemented that could be used for intrusion detection. As conclusion, we did extend the IDSnet program with a new neural network algorithm but unfortunately it seems that it still needs some work.

10.3 Network features

The network features we have defined in this project was aimed to be as general as possible. We have defined 17 features in total and they all are identical to some of the features defined by Lee and Stolfo [16]. The features defined in the original IDSnet program were also identical to the features of Lee and Stolfo. Therefore we decided to use these features without really digging into the complex code. We saw in the efficiency test (e.g., in the log file) that many of the features were the same for most of the packets. This might be the reason to the problem with the clustering in the SOM algorithm. And it is also obvious, packets with similar feature values activate the same neuron. One reason as explanation to this could be that the features are maybe not calculated correctly or maybe not calculated as we intended in this project. We

did trust the IDSnet program with regard to the feature value calculations and by the time we got suspicious about this situation, it was to late to investigate the code responsible for feature values calculations. However, this is only a qualified guess and does not need to be correct.
Due to time pressure, we did not manage to implement other features like the ones described in section 5.3.5. It could have been a worthy effort if we tried to implement those features and some more in order widen the list of general features.

10.4 Remarks on project progress

The developed intrusion detection system was intended to be used in a GRID environment. It was also intended to investigate what kind of network vulnerabilities there are for the GRID system and construct GRID-related features to detect the attacks that occur in the GRID. However, we did not manage to realize that due to the time pressure.
As concluding remark to the project we may say that it has been truly exciting and fascinating working with this project. It is an open project covering a wide range of subjects and there are a lot of extension possibilities. We did manage to cover the necessary theoretical fields and tried to give reasons for our choices as explanatory as possible. It is though a bit annoying that we didn't get fully satisfying test results to justify our work and choices in the project. But we are sure that is not insoluble and with further inspection, it would be possible to fix these and achieve a better result.

Appendix A

User manual

We present a manual for the SOM algorithm part implemented in the IDSnet program.

Somsensor
A somsensor is an input device and represents the self-organizing algorithm in the IDSnet program. It's task is to receive packets from an output device and make an analysis to set up a network model. The somsensor will be ready to receive packets when the Record button is pushed.

When the IDSnet is started, a somsensor device together with an ethernet and sensor device are created by default. By double click on the somsensor device, a panel will pop up from the button of the screen (sometimes the panel gets 'stuck', and you may need to scroll the pane up manually).

Somsensor panel
In the somsensor panel changes can be made to the learning algorithm. In the left section you can design your own algorithm by giving new values to some of the variables of the algorithm. Once the new values are defined, they will be applied to the algorithm when the Apply button is pushed. The Cancel button will close the somsensor panel.

Training and testing

The somsensor can start the training when the somsensor device is selected and the Record button is pushed. It will only collect IP packets of type TCP, UDP and ICMP from an output device. Once the collected packets equal or exceed the number of epochs defined in somsensor panel, the somsensor device automatically starts testing and the Update button in the left section gets enabled. By pushing the update button, the collected packets that have been tested with the trained algorithm, will be plotted in the right section of the somsensor panel, which represents the Kohonen layer of the algorithm. The training/testing can be stopped by pushing the Stop button.

Appendix B

Functional test

Below is a list of basic operations, which will be tested.

- *Program start.* Test of program start to observe the SOM algorithm (known as Somsensor in the IDSnet program) is also initiated and available on the main screen

- *Record/stop/clear operation.* Test of Somsensor device with the basic operations and see if the status of the Somsensor changes

- *Training and testing.* Test of training and testing of the Somsensor. Observe whether packets are received by the Somsensor and are processed.

- *The GUI of the SOM.* It will be tested to see that the GUI of the SOM works (e.g., the changed parameters are applied)

These tests will help us to find out if the functionalities are operational regarding the GUI.

B.0.1 Program start

Here we simply start the program and observe whether the Somsensor is listed. The result is in table B.1.

The test	Expectation	Observation	OK
Start of the IDSnet program (either by typing `./idsnet` in console or directly run from KDevelop)	IDSnet should initiate and display main screen with the Somsensor device	IDSnet is initiated and the Somsensor device is listed in the device list	✓

Table B.1: Test: Program start

B.0.2 Record/stop/clear operations

These basic operations will be tested in order to confirm that the Somsensor is compatible with the IDSnet. In order to test the clear operation the Somsensor should first receive packets from the ethernet device. So, we assume when we test clear operation that the Somsensor has received packets. In table B.2 exhibits the test result of the basic operations. The test shows that the Somsensor we have implemented is compatible with the operations in the IDSnet and works fine as it should.

B.0.3 Training and testing

The tests in this section will help us to observe that the Somsensor representing the SOM algorithm actually trains and tests when it is meant to. In table B.3 we see that the training and testing with Somsensor is carried out as it should. It may be a little difficult to observe that the test confirms that the SOM algorithm is trained and tested. In the next section we will have other test results that will also confirm the ability of training and testing of the implemented SOM algorithm.

The test	Expectation	Observation	OK
In the main screen the Somsensor device is selected and the record button is pushed	The status of the Somsensor device should change to 'Recording'	The status is changed to 'Recording'	✓
In the main screen the Somsensor device is selected and the record button is pushed, accordingly the stop button is pushed	The status of the Somsensor device should change to 'Stopped'	The status is changed to 'Stopped'	✓
In the main screen the Somsensor device is selected and the record button is pushed. Packets are received, which can be seen in the 'Packets' column that increases on each received packet. The clear operation will dispose packets so far and reset	The status of the Somsensor device should change to 'Stopped' and the 'Packets'-field is reset (e.g., set to 0)	The status is changed to 'Stopped' and the 'Packets'-field is set to 0.	✓

Table B.2: Test: Record/stop/clear operations

B.0.4 Test of the GUI

Some basic functional tests regarding the GUI of the SOM algorithm is tested. Table B.4 shows the test results of the GUI of the SOM algorithm.

B.0.5 Conclusion on the functionality test

The functionality test of the IDSnet program with the implemented Somsensor device shows that the functionalities regarding the Somsensor as device and the graphical-user interface works satisfactory. This was important to test because the original IDSnet program involves many lines of code implementing the GUI and the basic operations. And making an extension to a system like the IDSnet is not easy and requires the acquaintance of the whole system. But it has been proved with the functionality tests that the Somsensor has been integrated

The test	Expectation	Observation	OK
Select the Somsensor device, push the record button. The select the Ethernet device and push the play button to test that Somsensor is receiving the captured packets and trains	The 'Packets'-field should start counting the received packets	The 'Packets'-field starts counting	✓
Select the Somsensor device, push the record button. The select the Ethernet device and push the play button. Once the received packets exceeds the epoch size, the Somsensor will start testing. Click on the 'Update'-button to observe the testing takes place and winning neurons are displayed.	As soon as the received packets size is larger than the total epoch size, the packets will be used for testing and is displayed by clicking the 'Update'button.	Once the packet size has reached the total epoch size, the new packets are displayed	✓

Table B.3: Test: Training and testing of SOM

successfully and performs the required operations satisfactory.

The test	Expectation	Observation	OK
Double click on the Somsensor device	A new frame will pop up from the button of the main screen, displaying the properties of the SOM algorithm and a matrix representing the Kohonen layer	The frame pops up from the button, displaying properties and the Kohonen layer matrix	✓
On the property frame, the size of the Kohonen layer is changed from 10x10 (default values) to 7x7 and the 'Apply'-button is pushed	The matrix representing the Kohonen layer on the right side of the property frame should be changed, having 7 rows and 7 columns	The matrix is changed and now has 7 rows and 7 columns	✓
On the property frame, the size of the epochs is changed from 10000 to 5000 epochs and the 'Apply'-button is pushed. Once the packets size has reached 5000 the 'Update'-button (upon pushed) should displaying winner neurons on the matrix.	After 5000 packets, the 'Update'-button should start displaying the winner neurons	The winner neurons are displayed after 5000 packets	✓

Table B.4: Test: the GUI of the SOM algorithm

Appendix C

A draft of the log file

```
(5, 6)
f[1] = 17.000000
f[2] = 28416.000000
f[3] = 100.000000
f[6] = 100.000000
f[8] = 10.000000
f[11] = 100.000000
f[13] = 100.000000
f[14] = 100.000000
(10, 7)
f[1] = 17.000000
f[2] = 16206.000000
f[3] = 100.000000
f[6] = 100.000000
f[8] = 30.000000
f[11] = 100.000000
f[13] = 100.000000
f[14] = 100.000000
(10, 7)
f[1] = 17.000000
f[2] = 13568.000000
f[3] = 100.000000
```

```
f[6] = 100.000000
f[8] = 100.000000
f[10] = 29.655172
f[11] = 100.000000
f[13] = 100.000000
f[14] = 100.000000
(5, 9)
f[1] = 1.000000
f[6] = 100.000000
f[8] = 5.000000
f[11] = 100.000000
f[13] = 100.000000
f[14] = 100.000000
(6, 1)
f[1] = 17.000000
f[2] = 35328.000000
f[3] = 100.000000
f[6] = 100.000000
f[8] = 100.000000
f[10] = 29.655172
f[11] = 68.965517
f[13] = 100.000000
f[14] = 100.000000
(10, 3)
f[1] = 17.000000
f[2] = 41216.000000
f[3] = 100.000000
f[6] = 100.000000
f[8] = 100.000000
f[10] = 29.655172
f[11] = 68.965517
f[13] = 100.000000
f[14] = 100.000000
```

Appendix D

Source code of a simple sniffer using pcap

```
/***************************************************************************
                          main.c  -  description
 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <pcap.h>
```

```
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h> /* default snap length (maximum bytes per
packet to capture) */
#define SNAP_LEN 1518

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN  6

/* Ethernet header */
struct sniff_ethernet {
  u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */
  u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */
};

/* IP header */
struct sniff_ip {
  u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
  u_char  ip_tos;                 /* type of service */
  u_short ip_len;                 /* total length */
  u_short ip_id;                  /* identification */
  u_short ip_off;                 /* fragment offset field */
  #define IP_RF 0x8000            /* reserved fragment flag */
  #define IP_DF 0x4000            /* dont fragment flag */
  #define IP_MF 0x2000            /* more fragments flag */
  #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
  u_char  ip_ttl;                 /* time to live */
  u_char  ip_p;                   /* protocol */
  u_short ip_sum;                 /* checksum */
  struct  in_addr ip_src,ip_dst;  /* source and dest address */
};

#define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef u_int tcp_seq;

struct sniff_tcp {
  u_short th_sport;               /* source port */
```

```
  u_short th_dport;               /* destination port */
  tcp_seq th_seq;                 /* sequence number */
  tcp_seq th_ack;                 /* acknowledgement number */
  u_char  th_offx2;               /* data offset, rsvd */
  #define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
  u_char  th_flags;
  #define TH_FIN  0x01
  #define TH_SYN  0x02
  #define TH_RST  0x04
  #define TH_PUSH 0x08
  #define TH_ACK  0x10
  #define TH_URG  0x20
  #define TH_ECE  0x40
  #define TH_CWR  0x80
  #define TH_FLAGS        (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
  u_short th_win;                 /* window */
  u_short th_sum;                 /* checksum */
  u_short th_urp;                 /* urgent pointer */
};

void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet);
void print_payload(const u_char *payload, int
len);

int main(int argc, char **argv) {
  int dev; /* name of the device to use */
  char *net; /* dot notation of the network address */
  char *mask;/* dot notation of the network mask    */
  int num_dev;   /* return code */
  struct pcap_pkthdr header;
  const u_char *packet;           /* The actual packet */
  char errbuf[PCAP_ERRBUF_SIZE];
  bpf_u_int32 netp; /* ip          */
  bpf_u_int32 maskp;/* subnet mask */
  struct in_addr addr;
  pcap_if_t *alldevsp,*temp_alldevsp;
  char sniff_dev[10];
  int num_packets = 10;

  /* ask pcap to find a valid device for use to sniff on */
   pcap_t *handle;
   num_dev=pcap_findalldevs(&alldevsp,errbuf);
   temp_alldevsp=alldevsp;
```

```
   if(num_dev==0)   /* device lookup success */
     {
     printf("\n\tNetwork Devices found\n\t--------------------\n");
     while(temp_alldevsp!=NULL){
       printf("Device Name ::%s\n",temp_alldevsp->name);
       temp_alldevsp=temp_alldevsp->next;

       }
     }

   temp_alldevsp=alldevsp;
   printf("\n\tNetwork Device Information\n\t--------------------\n");
   while(temp_alldevsp!=NULL){
     printf("\n\nDevice Name ::%s",temp_alldevsp->name);
     if(temp_alldevsp->description!=NULL)
           printf("\nDevice Description ::%s",temp_alldevsp->description);
     else
           printf("\nNo description available for this device");

     if((temp_alldevsp->flags) & (PCAP_IF_LOOPBACK==1))
           printf("\nDevice is a Loopback device\n\n");
     temp_alldevsp=temp_alldevsp->next;
   }

   handle = pcap_open_live("eth0", BUFSIZ, 1, 1000, errbuf);
   if (handle == NULL) {
     fprintf(stderr, "Couldn't open device %s:\n",errbuf);
   }
   else
   {
     pcap_loop(handle, num_packets, got_packet, NULL);
    }
   pcap_close(handle);
  return 0;
}

void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet)
{

  static int count = 1;                   /* packet counter */

  /* declare pointers to packet headers */
  const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
```

```
const struct sniff_ip *ip;              /* The IP header */
const struct sniff_tcp *tcp;            /* The TCP header */
const char *payload;                    /* Packet payload */

int size_ip;
int size_tcp;
int size_payload;

printf("\nPacket number %d:\n", count);
count++;

/* define ethernet header */
ethernet = (struct sniff_ethernet*)(packet);

/* define compute ip header offset */
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
if (size_ip < 20) {
  printf("   * Invalid IP header length: %u bytes\n", size_ip);
  return;
}

/* print source and destination IP addresses */
printf("       From: %s\n", inet_ntoa(ip->ip_src));
printf("         To: %s\n", inet_ntoa(ip->ip_dst));

/* determine protocol */
switch(ip->ip_p) {
    case IPPROTO_TCP:
        printf("\nProtocol: TCP\t");
        break;
    case IPPROTO_UDP:
        printf("   \nProtocol: UDP\n\t");
        return;
    case IPPROTO_ICMP:
        printf("   \nProtocol: ICMP\n\t");
        return;
    case IPPROTO_IP:
        printf("   \nProtocol: IP");
        return;
    default:
        printf("   \nProtocol: unknown\n");
        return;
}
```

```
  /*  This packet is TCP. */

  /* define/compute tcp header offset */
  tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
  size_tcp = TH_OFF(tcp)*4;
  if (size_tcp < 20) {
    printf("   * Invalid TCP header length: %u bytes\n", size_tcp);
    return;
  }

  printf("   Src port: %d\t", ntohs(tcp->th_sport));
  printf("   Dst port: %d\t", ntohs(tcp->th_dport));
  printf("   TCP flags: 0x%x\n",(tcp->th_flags));

    /* compute tcp payload (segment) size */
  size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);

  /* Print payload size */
  if (size_payload > 0) {
    printf("   Payload (%d bytes):\n", size_payload);
  }
  return;
}
```

Bibliography

[1] *ftp://ftp.sas.com/pub/neural/FAQ.html.*

[2] *http://doc.trolltech.com/.*

[3] *http://insecure.org/nmap/.*

[4] *http://nsaditor.com.*

[5] *http://www2.imm.dtu.dk/IDSnet/.*

[6] *http://www.cert.org/.*

[7] *http://www.iana.org/assignments/port-numbers.*

[8] *http://www.tcpdump.org.*

[9] *http://www.wikipedia.org.*

[10] Damiano Bolzoni, Sandro Etalle, Pieter H. Hartel, and Emmanuele Zambon. Poseidon: a 2-tier anomaly-based network intrusion detection system. In *Proceedings of the 4th IEEE International Workshop on Information Assurance, 13-14 April 2006, Egham, Surrey, UK*, pages 144–156, 2006.

[11] D. A. Frincke, D. Tobin, J. C. McConnell, J. Marconi, and D. Polla. A framework for cooperative intrusion detection. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 361–373, 1998.

[12] Simon Haykin. *Neural Networks, A Comprehensive Foundation.* Prentice Hall, 2nd edition, 1999.

[13] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An efficient k-means clustering algorithm: analysis and implementation. In *IEEE Transactions on Pattern Analysis and Machine Intelligence 24(7): 881-892.*, 2002.

[14] Ben Kröse and Patrick van der Smagt. *An introduction to neural networks.* The University of Amsterdam, 1996.

[15] Pavel Laskov, Patrick Düssel, Christin Schäfer, and Konrad Rieck. Learning intrusion detection: Supervised or unsupervised?. In *Image Analysis and Processing - ICIAP, 13th International Conference, Cagliari, Italy*, pages 50–57, 2005.

[16] Wenke Lee and Salvatore J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Trans. Inf. Syst. Secur.*, 3(4):227–261, 2000.

[17] John Zhong Lei and Ali A. Ghorbani. Network intrusion detection using an improved competitive learning neural network. In *2nd Annual Conference on Communication Networks and Services Research (CNSR 2004), 19-21 May 2004, Fredericton, N.B., Canada*, pages 190–197. IEEE Computer Society, 2004.

[18] P. Lichodzijewski, A. Zincir-Heywood, and M. Heywood. Dynamic intrusion detection using self organizing maps, 2002.

[19] Giuseppe Patanè and Marco Russo. The enhanced lbg algorithm. *Neural Networks*, 14(9):1219–1237, 2001.

[20] Jake Ryan, Meng-Jang Lin, and Risto Miikkulainen. Intrusion detection with neural networks. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[21] Robin Sharp. The poor man's guide to computer networks and their applications. April 2004.

[22] Shi Zhong, Taghi Khoshgoftaar, and Naeem Seliya. *Clustering-based Network Intrusion Detection.* Department of Computer Science and Engineering, Florida Atlantic University.