Implementation of a Computer Immune System for Intrusion- and Virus Detection

Markus Christoph Unterleitner
office@unterleitner.info

February 13, 2006

Contents

List of Figures

List of Tables

1. Introduction

The number of reported incidents on local area networks (LANs) and further on individual computer systems are continually increasing since mid of the last decade. The governments, companies and home users are typical targets for malicious activities of hackers. Most of the incidents, especially on home computer systems are not reported. It is important to deploy new defence mechanisms in the computer security field. System administrators have to ensure that their systems are up to date and not vulnerable. They constantly have to take steps towards more secure computers and networks. But it seems that some hackers are not affected by these steps, because some of them are still successful exploiting computer systems. They are able to bypass the systems security perimeter and find paths through the present intrusion detection systems. By this they can access most of the computers connected to a network.

Internet worms and computer viruses use advanced infection techniques. They spread over the internet and infect a high number of computer systems within a few days. Because of the global connection of computers, the infectious agents may be distributed all over the world. It possibly takes some weeks for the computer administration staff to sanitize all infected computer systems in a network. After an outbreak it could cost billions of Euros to reestablish the productivity of infected systems.

Malicious agents can spread easily over the Internet because of weaknesses in the protection of individual systems. It is difficult to ensure network-wide security on large networks. Every day new vulnerabilities and security holes are discovered which have to be repaired immediately. Especially home users often do not download patches or make updates of their antivirus database right after they are available. Systems which are not kept up to date are in danger to be infected. A high percentage of today's internet worms and viruses are undetected until they have a negative effect on the system. Therefore, users often do not notice that their system is infected.

The currently used methods often suffer from the need of a database which has to be constantly kept up to date. Those methods are not effective enough to detect new malicious agents when they try to infect computer systems. Current systems show weaknesses in preventing their spread and in preventing intrusions. This can be seen by the increasing rate of reported incidents.

1.1 Strategies of intrusion detection systems (IDS)

Generally, there are two main strategies for the implementation of an IDS:

- **Misuse detection** is based on the detection of already known attack patterns. The information is analysed by comparing it with signatures in a database. An IDS that uses misuse detection protects the system as soon as installed with a very low false positive rate. If the IDS raises an alarm it is directly referred to a specific kind of network activity. But those systems do not detect new types of attacks which are not included in the database. Therefore, the database has to be kept up to date.
- **Anomaly detection** is based on the detection of deviations of the system behaviour which is defined by a profile for normal behaviour. This profile is learned by the IDS during a long time observation of the network.
    - ◊ This approach has several **advantages**. Because the IDS is based on a learned profile of normal behaviour, attackers do not know what activity will lead to an alarm of the IDS. The detection of intrusions is not based on known malicious activities like in misuse detection systems. Anomaly detection enables the IDS for the detection of new attacks which probably were used for the first time.
    - ◊ An IDS using anomaly detection has also some **disadvantages**. The profile for the normal behaviour must be defined and therefore, the IDS does not protect the system for a time period after installation. The definition of what is "normal" must be configured in the profile. Anomaly detection systems have a high rate of false positives and a generated alarm is more difficult to associate with a specific network activity. Furthermore, if a specific attack does not violate with the defined behaviour profile it passes unnoticed.

Therefore, anomaly detection systems are not often implemented in IDS's. Actually a lot of research is done about the anomaly detection strategy. The combination of both strategies to a hybrid detection system architecture will use the benefits of both strategies. This thesis proposes a **hybrid detection system** with an anomaly detection system based on the biological immune system.

1.2 Overview about the thesis

The human body shows us how a robust infection defence system can be built. The biological immune system has the job to keep the body healthy. This system is very complex and uses advanced mechanisms for detecting and eliminating infectious microorganisms (pathogens). The immune system learns to recognize new pathogens which are intruding the body and then produces the right kind of response to fight them. It has many properties which can be adopted for the design of artificial systems in the computer security field. The immune system uses anomaly detection for the recognition of pathogens.
This thesis presents the implementation of a computer security system that is inspired by the biological immune system. The computer immune system (CIS) is applied to the network security domain. It is able to perform the detection of malicious code in the payloads of transferred network packets. This system applies several match algorithms for the inspection of packets. The results of the experiments with the CIS are presented and analysed.

This thesis has the following structure:

Chapter 2 gives a short introduction to the biological background of the computer immune system. It analyses the complex biological learning mechanism for the adaptive detection and elimination of pathogens. The important properties of the human immune system which are influencing the design of the CIS are presented.

Chapter 3 presents the model of the computer immune system. Related work is mentioned as well as a description of important mechanisms in the model. It is presented how relevant mechanisms of the human immune system are applied in the CIS.

Chapter 4 describes the five implemented match algorithms. These algorithms are used to perform the data discrimination in the anomaly based detection system.

Chapter 5 presents the components and the implementation of the CIS framework. It provides a short overview about the intrusion detection system Snort. This chapter also shows how matching algorithms are applied. It gives a detailed description of the experiments and their results.

Chapter 6 explains why the data representation of the packet payload is changed from byte code to machine instructions. The content of the payload is decoded by a disassembler. When applying the disassembler an entry point has to be chosen. The problem of finding the right entry point in the content of the payload is analysed. Finally some further experiments and their results are presented.

Chapter 7 contains the conclusions of this thesis as well as proposed future work.

Appendix A analyses the byte spectrum of some text based and binary based file formats.

Appendix B presents the tables of data from the experiments presented in chapter 5 and 6.

Appendix C analyses the instruction spectrum of some text based and binary based file formats.

Appendix D uses code examples to explain the “entry point effect” presented in chapter 6.

2. The Immune System of the Human Body

This chapter is an introduction into the biological background of immune systems. It explains how parts of the human immune system are working. For further information the reader is referred to [19, 20, 21]. The main intention of this chapter is to understand the basic architecture and to define algorithms which can be mapped to computer immune systems.

The human immune system is a very complex defence system with the intention to protect the body from various types of threats. During lifetime the body is permanently exposed to harmful substances (**pathogens)**. Pathogens are harmful microorganisms like bacteria or viruses. If pathogens are left undetected, they lead to a rapid deterioration of the body's health. The immune system has the job to fight those pathogens to keep the body healthy.
The immune system performs two elementary tasks. One task is to detect pathogens. As soon as pathogens are detected, another task is to eliminate or neutralize these pathogens. Therefore, the immune system deploys a great amount of cells and molecules which are able to recognize and eliminate pathogens. The cells interact with the local environment via chemical signals. A cell that binds to foreign proteins is signalling a detection event. This cell will then attract other cells for assistance and is therefore initiating a sequence of reactions. Short after the detection of pathogens, a high number of cells will interact at the site of infection. The immune system reacts with a **primary immune response** which leads to the destruction or neutralisation of the pathogens. During the primary response, the immune system produces a variety of cells which are able to detect particular kinds of pathogens with an increasing accuracy. The adaptation of these cells takes some time. But as soon as the immune system recognizes the new pathogens they are killed. After the successful elimination of the intruding pathogens the immune system memorizes a small fraction of the adapted cells (**immunological memory**). These cells are high specific and enable the immune system to respond much faster and more efficient during future occurrences. This mechanism is called a **secondary immune response**. It is based on the immunological memory of the immune system. This secondary response is often fast enough to fight pathogens before they are reaching a sufficient number to harm the body. The immunological memory protects the body from infectious diseases which the immune system has encountered before.
Cells of the immune system can detect subtle chemical differences between pathogens. They can distinguish between harmful pathogens and elements of the body. Thus, the immune system is able to distinguish between **self and nonself** [1]. The self set consists of harmless substances which include elements of the body. Whereas, the nonself set contains harmful substances.

The immune system maintains a huge amount of cells doing their job. These cells are working independently in the local environment and are not under centralized control [1]. This provides a system that is highly distributed and robust against attacks.

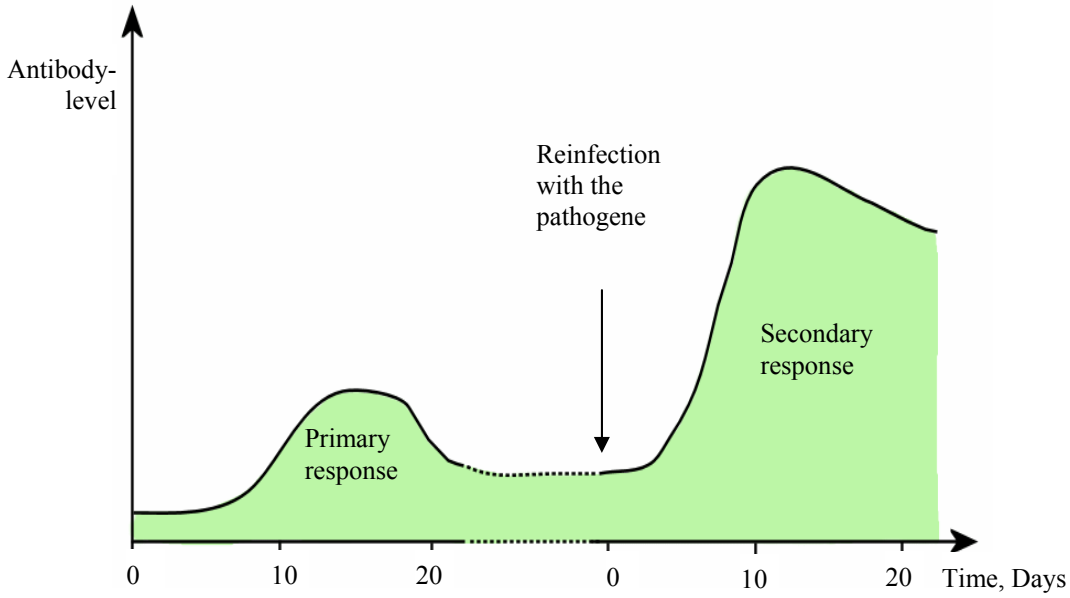

Figure 2: This figure shows the difference between the primary and the secondary response to a pathogen reflecting the principle of the immunologic memory [21, 22]. When a pathogen is detected the immune system reacts with a primary response having a low level of antibodies. If a second infection with the same pathogen occurs, the immune system will respond through a secondary response with a much higher antibody level. The antibody level reaches its peak much faster. This secondary response is more efficient than the primary response.

2.1 Architecture overview

The immune system provides defence mechanisms on several levels. First, the pathogens will meet an anatomic barrier, the skin. The skin blocks all substances which have not the right shape to pass through. It forms an effective barrier to many microorganisms. But some of them will reach the next level of defence, especially if the skin is compromised by wounds. Wounds provide broad paths to infections. The next level maintains physiologic barriers where a low pH value and a high body temperature are mechanisms to prevent the pathogens from evolving. When pathogens have entered the body, then two further mechanisms are engaged: the innate and the adaptive immune system. Both mechanisms are based on a big number of cells circulating in the bloodstream.

Level of defence	**Type**	**Mechanisms**
Skin	Anatomic barrier	Physical shield, blocks entry of microorganisms
Temperature	Physiologic barrier	Prevent pathogens from evolving, normal body temperature provides a inhospitable environment
Low pH value	Physiologic barrier	Kills many pathogens by the acidous living conditions within the body
Innate IS cells	Bloodstream content	Static defence mechanisms
Adaptive IS cells	Bloodstream content	Dynamic defence mechanisms

Table 2: The skin, the body temperature, the low pH value and the innate immune system (IS) cells are non-specific defence mechanisms. These mechanisms are equal among a population and they do not change during the lifetime of an individual. The adaptive immune system is able to develop a specific response to an encountered pathogen and retains an immunologic memory. It changes often during lifetime and gets more and more sophisticated.

2.1.1 The innate immune system

The innate immune system forms the evolutionary evolved resistance of the body to already known pathogens. It is a first line of defence to already known pathogens, but it cannot detect new kinds of pathogens. It is mainly based on a chemical signalling system called the **complement system**. The complements, also called **antibodies**, are free floating molecules that can bind to pathogens. Here it is important to note that the antibodies do not bind to cells of the body. They help to eliminate pathogens in an early stage of infection. The antibodies which are bound to pathogens are used as markers for special killer cells like **macrophages**. Macrophages are eliminating the detected pathogens (Figure 2.1).
When macrophages detect pathogens they are activated. Activated macrophages will then release molecules called **cytokines**. Cytokines are not only released by activated macrophages, but also by other cells of the immune system. Cytokines raises a kind of alarm signal to the system, which is called the **inflammatory response**. This results in a higher local blood flow in order to attract other cells for assistance to the site of infection. They induce the increase of the body temperature and are responsible for activating other immune system cells.
The innate immune system integrates several molecules that are capable to recognize molecule patterns typical to pathogens. There are many types of molecules that can only be found in pathogens, but not in the body (self). The fast response to such patterns is a strong feature of the innate immune system.

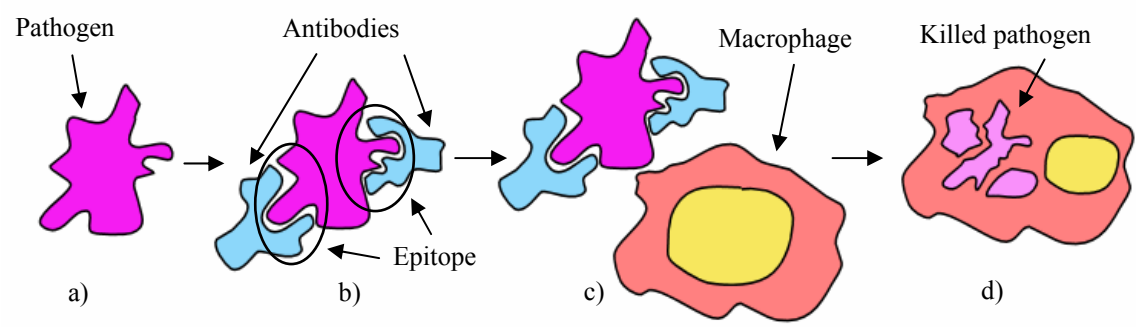

Figure 2.1: The process of a pathogen that is coated with the antibodies. In a) a pathogen, which was discovered by the immune system has entered the body. In b) the pathogen is detected by two antibodies. A pathogen has different epitope structures (see 2.2) on the surface. That's why several different antibodies can bind to pathogens. In c) the coated pathogen is detected by a floating killer cell, the macrophage. Finally in d) the macrophage engulfed the pathogen.

2.1.2 The adaptive immune system

In contrast to the innate immune system, the adaptive immune system reacts with a high specific immune response to the threat. It is able to recognize special pathogens with an increasing efficiency. This is a **learning process** performed on pathogens which have not been seen by the system before. After the immune system had cleared the infection, it will keep a fraction of the recognizing cells in the **immunologic memory**. The response of the immunologic memory to a pathogen is called the secondary response of the immune system. This feature makes the individuals of a species diverse. The response will not be the same for all members of the species - unlike the innate immune response. A need for the adaptive immune system is the ability to distinguish between self and nonself. This feature is essential for all immune systems, because the response to elements of self will have fatal consequences. If cells of the immune system are responding to elements of the body, they are causing an autoimmune reaction [23, 24].
The adaptive immune system is mainly based on parts of white blood cells, called **lymphocytes**. Lymphocytes are mobile and independent detectors which are circulating in the bloodstream and the lymphatic systems of the body. The huge number of distributed detectors builds a strong system capable to detect and eliminate the invading foreign microorganisms efficiently.

The main features of the adaptive immune system are:

- Lymphocyte and receptor diversity
- Increasing specificity to pathogenic structures
- Complex interactions between immune system cells
- Discrimination between self and nonself elements
- Immunologic memory

In the following, the adaptive immune system and its features are described in more detail.

2.2 The adaptive detection of pathogens

The detection of pathogens is based on the recognition of protein structures. A cell of the immune system has **receptors** on its surface which can bind to various kinds of **epitopes** (also called antigenic protein structures or **antigen**). Epitopes are protein structures on the surface of a pathogen. A detection event occurs when the chemical binding between the complementary receptors and epitopes succeed. The binding mechanism of receptors to epitope structures can be explained as a **pattern match process**, where a lymphocyte tries to match epitopes on pathogens. All receptors on a single lymphocyte are the same, but they are not the same on different lymphocytes. A lymphocyte is a **detector** for the recognition of a small subset of epitopes. In contrast to the lymphocytes, pathogens have different epitopes on their surface. Therefore, many lymphocytes can bind to a single pathogen. The immune system does not need to maintain lymphocytes for each pathogen to cover the whole space of possible epitope structures.
It is very important for developing the immune response to have lymphocytes that have many **equal receptors** on their surface. When they are activated by the detection event, it is clear on which kind of pathogen the immune system should react on. Otherwise, the detection of a pathogen can result in an unclear immune response. A detection event can be defined as the number of receptors on the surface of a lymphocyte which are needed to establish a binding to a pathogenic epitope. This number is a threshold and measures the affinity between the receptor and the eptiope.

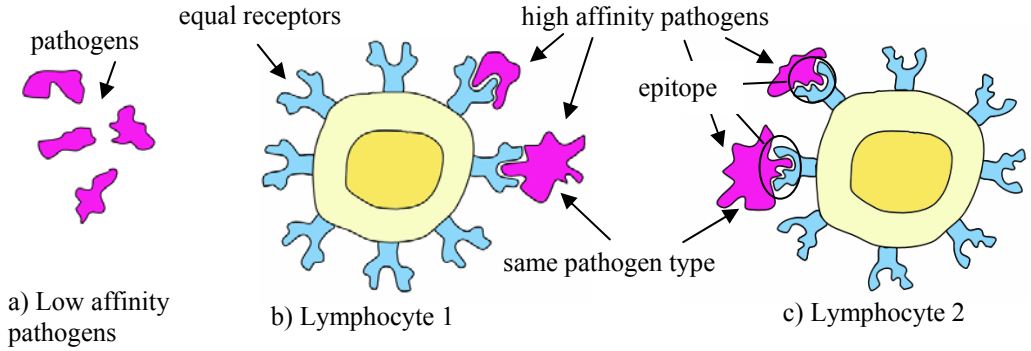

Figure 2.2: Shows the structure of lymphocytes and successful detected pathogens. In a) pathogens are presented, which have low affinity to the receptors of both lymphocytes. These do not bind to either lymphocyte 1 or 2. The lymphocytes receptors are not specific to a single pathogen. As it can be seen in b) and c) they can bind a small subset of pathogens which have a similar structure. The same type of pathogen can also be bound by several lymphocytes.

The immune system maintains a **dynamic detector population** [22]. Lymphocytes have a short lifetime and if they do not match to any pathogen, they will be replaced. New lymphocytes will be produced in the bone marrow each day. The number of continually new lymphocytes is about 10 percent of their total number. Only a few of them are successful and the rest is replaced again after some time. Thus, there is a continual turnover of detectors which increases the protection capability of the body.

2.3 The learning mechanism of the adaptive system

The development of an immune response needs the cooperation between two different types of cells, which are the **T lymphocytes** and the **antigen presenting cells**. Antigen presentation occurs in various cells of the body (see 2.3.3). The lymphocytes are part of the white blood cells which are produced in the bone marrow. As described before, lymphocytes have receptors on their surface that bind to antigen structures. They can be divided into two main subgroups that are **B lymphocytes**, also called **B-cells** and **T lymphocytes**, also called **T-cells**.

2.3.1 The B-Cells

B-cells are the detectors of the immune system which are able to specialize to encountered pathogens. They are able to adapt to a specific pathogen and enable their efficient detection by the immune system. B-cells are essential for the detection of pathogens. If the B-cell receptors bind to an antigen it is defined as a match event of the B-cell detector. This match event will lead to the activation of the B-cell. The process of the B-cell activation is shown in figure 2.3.2 on page 17. Activated B-cells are developing the adaptive immune response. The adaptation of B-cells to pathogens includes the following cycle of mechanisms:

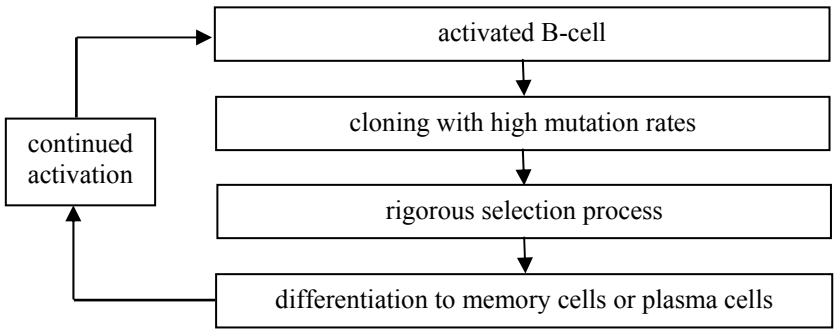

Figure 2.3.1: The adaptation cycle of B-cells.

When B-cells are activated they clone themselves by a cell division process. The receptors of the clones mutate with unusual high mutation rates (also known as **hypermutation**) [25]. The high mutation rates of the receptors ensure that clones may have a higher affinity to the encountered pathogen, but it could also be worse. The clones do not live long and they have to pass a rigorous selection process. B-cell clones having a higher affinity to the detected pathogen are more likely to survive the selection process. The others, which are more than 90 percent of the cloned B-cells, will die through programmed cell death (**apoptosis**) after a short time.
The clones that survive the selection will differentiate into two B-cell types. This is an important step for the adaptive recognition of pathogens. Those B-cells become **memory cells** or **plasma cells**. Memory B-cells are equipped with higher affinity receptors and have a long life span. Plasma B-cells are short lived and have no or only a few receptors. Their purpose is to produce **antibodies**. The antibodies are a soluble form of the high affinity receptors. The plasma B-cells releases the antibodies to the bloodstream where they act as free floating recognition molecules.
In each adaptation cycle the B-cell clones become more and more specific to the threat. The cloning and selection process is repeated until the infection has been cleared. This takes time and represents the primary response of the immune system. It shows how a very general

response evolves to a very specific response. The adaptive immune system works as a dynamic defence system which uses the interactions of many cells. Initially, when pathogens are detected, it trains new cells needed to fight a specific kind of pathogen. Afterwards, it memorizes those detectors with the highest affinity to the pathogen, while all lower affinity cells will die by apoptosis. When the body has defeated the intruding pathogens the adaptive learning process always results in a more robust defence system.
The new fraction of memorized B-cells is an addition to the cells of the **immunologic memory**. It retains the population of B-cells that are specific in detecting a special kind of pathogen. If the same pathogen appears in future infections then the memorized B-cells react with a fast and efficient secondary response. Figure 2 compares the antibody level and speed of occurrence in the primary and the secondary response.
The use of only B-cell detectors for developing the immune response has an until now unmentioned problem. B-cells are maturing in the bone marrow. When they leave the bone marrow they possibly could bind to self cells. This is because the selection process in the bone marrow is not sufficient. In addition, in the adaptation cycle the receptors of B-cells are cloned with high mutation rates. Those B-cells could also initiate autoimmunity. But usually, mature B-cells are tolerant to self cells. To prevent B-cells from initiating an autoimmune reaction the T-cells come into play. A B-cell must receive two different signals to become activated. The first signal occurs when the B-cell receptors bind to an antigen. The second signal, a **costimulatory signal,** is provided by the T helper cells (TH-cells, see 2.3.2). If the B-cells receive only the first signal they will die. The need for an approval signal from the TH-cells prevents the B-cells from causing autoimmune reactions.

2.3.2 The T-Cells

T-cells are also produced in the bone marrow. To get matured, they have to migrate to the thymus. In the thymus they are made tolerant to almost all self protein structures. T helper cells (**TH-cells**), a subgroup of the T-cells are entrusted with the task of ensuring tolerance to the self cells. If a T-cell binds to self cells, it will die by apoptosis in a process known as **negative selection** [11]. The negative selection process finally chooses those T-cells to join the population which are not binding to self.
Matured T-cells have on their surface a unique antigen binding receptor (T-cell receptor). B-cells receptors are membrane bound antibodies which can only bind to the antigen structures on pathogens. In contrast to the B-cells, the T-cells are equipped with receptors that can only recognize antigen in combination with molecules of the Major Histocompatibility Complex (**MHC**). The MHC molecules are proteins bound to the surface of self cells. MHC molecules exist in two major types. The one can only be found in antigen presenting cells. It has the purpose to show other cells what they have detected. This type is recognized by TH-cells. The second type is found in almost every cell of the body. It is needed to display the cells genetic code. If the cells genetic code is changed by an infection this MHC type is recognized by T killer cells (**TK-cells**). TK-cells are another subgroup of the T lymphocytes.
A B-cell uses the antigen presenting cell MHC type. If a B-cell recognizes a pathogen the combined molecule is presented to TH-cells in order to get the necessary approval for its detection – this is called the **costimulatory process**.

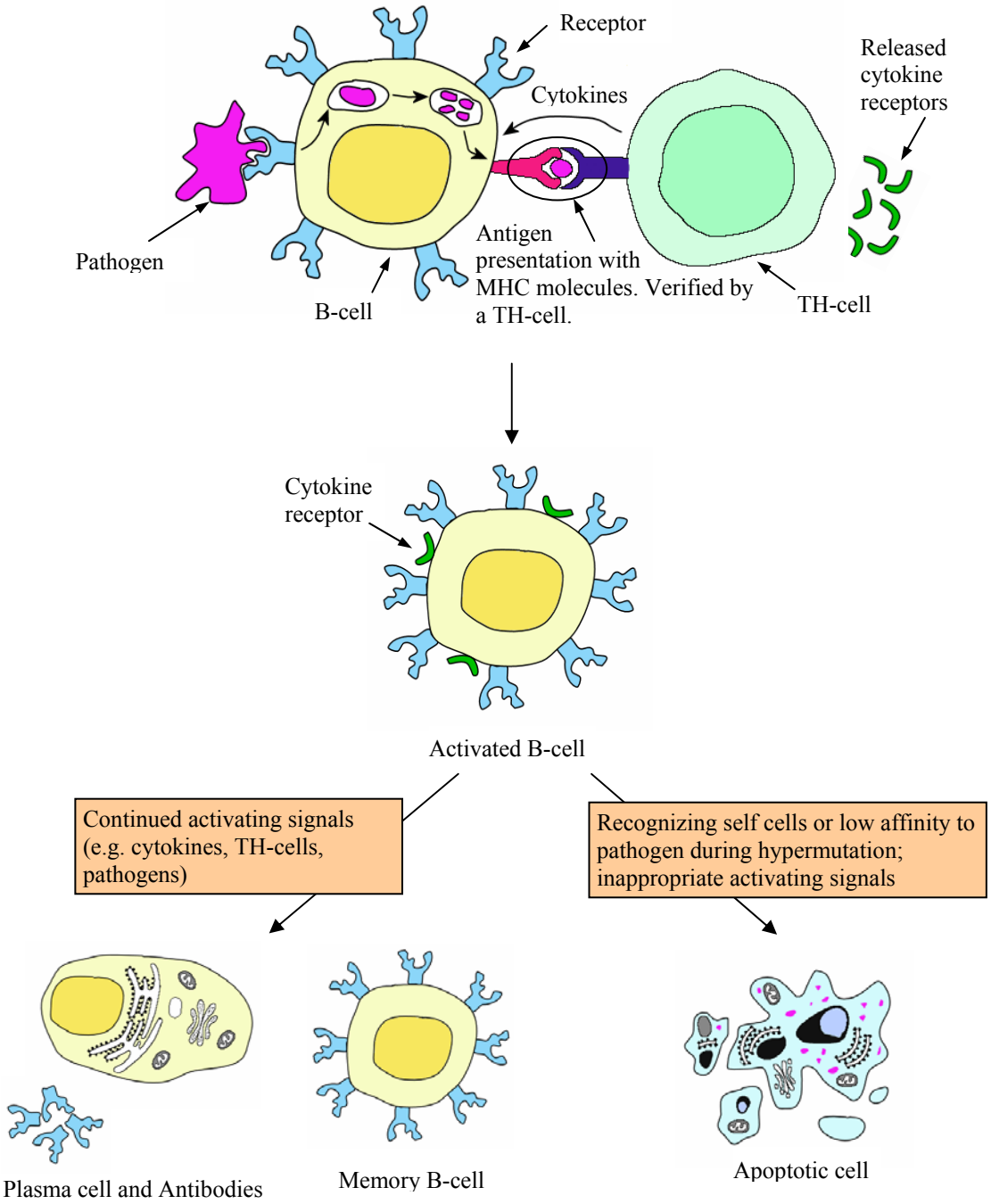

Figure 2.3.2: The cycle of a B-cell that recognizes a pathogen [21]. A B-cell recognizes a pathogen and gets approved by a TH-cell. The activated B-cell then proliferates and differentiates with high mutation rates known as **hypermutation**. The clones of the B-cell with a higher affinity to the recognized pathogen will be selected for Memory B-cells and Plasma cells. Plasma cells are producing antibodies.

2.3.3 The costimulatory process

As already mentioned, the T-cell receptors recognize pathogenic antigens in combination with MHC molecules. The TH-cell must somehow verify what the B-cell has detected to provide it with the necessary costimulation. The MHC molecules allow the B-cells in a process called **antigen presentation** to show TH-cells what they have recognized. For that purpose, the B-cells must combine antigen with the MHC molecules (see 2.3.4). Antigen presentation occurs in different cells of the body, like in cells that bind to antigenic structures of the pathogens (B-cells). Further antigen is also presented in killer cells which engulf and eliminate the pathogens (macrophages).
The TH-cells are binding only to antigen presenting cell MHC molecules. If they recognize antigen in combination with MHC molecules they release a high number of cytokines. The cytokines produced by activated TH-cells lead to the activation of other immune system cells. That's why the activation of TH-cells must be done carefully. Otherwise their response will initiate major autoimmune reactions. This is because the cytokines are also causing immune reactions by the antigen presenting cell. The activated B-cells for example will differentiate and proliferate, whereas activated macrophages eliminate the found pathogen.
TK-cells recognize MHC molecules on altered self cells. They are essential for the detection and elimination of contaminated host cells. TK-cells are activated by binding to cells of the body that present the MHC in combination with the cells altered genetic code. The infection of self cells alters their genetic code. The MHC then displays pathogenic structures on the cells surface which came from a pathogen living inside the cell. These cells are for example virus infected self cells. Activated TK-cells will eliminate infected self cells.

2.3.4 Antigen processing and presenting

Pathogens in intracellular regions: As mentioned before, the recognition of pathogens is done by binding of foreign surface proteins (epitopes) to the lymphocyte's receptors. But some pathogens, so called intracellular pathogens, do not display their epitopes to the B-cells. B-cells are not able to recognize intracellular pathogens.
The immune system uses another recognition mechanism to detect infected self cells: The MHC is used to carry fragments of proteins from inside the cell to the surface of the cell. Self cells can be infected by viruses or bacteria which are altering the structure of the cells. The complex foreign protein structure contains also fragments that can be combined with the MHC. This composition is transported to the surface of the presenting cell, where it can be detected by the TK-cells. They eliminate the contaminated host cell.
Pathogens found by antigen presenting cells: When pathogens are bound to the receptors of the antigen presenting cells they trigger a presenting process. The antigen presenting cells are decomposing the complex foreign protein structure into some fragments which will enter the cell by phagocytosis or endocytosis. The antigen structure will then bind to the MHC of antigen presenting cell types. The composition is then also transported on the surface of the cell, where it can be detected by the TH-cells.

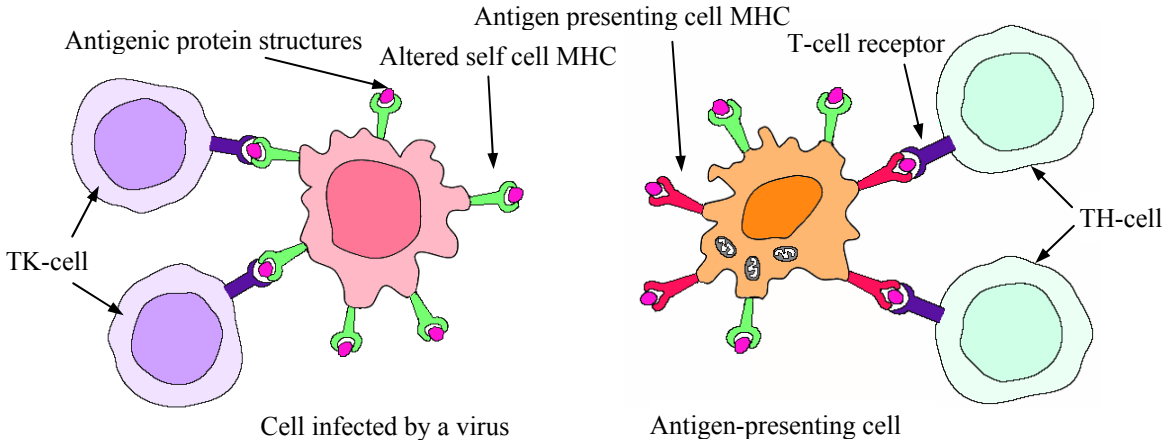

Figure 2.3.3: Here both T-cell subgroups are binding to their recognized classes of MHC molecules (see [21]). TH-cells can only bind to antigen presenting cell MHC molecules. TK-cells have specific receptors for the altered self cell MHC molecule.

2.4 Detection failures made by the immune system

The human immune system is a strong defence system against invading pathogens. But the immune system sometimes fails. There are two kinds of detection errors which can occur. These failures of the immune system can also appear in a computer defence system. Both of them are harmful to the system and can result in a fast deterioration of the body's health.

- A **false positive** occurs, if the immune system responds to a molecule which is not harmful (e.g. self cells)
- A **false negative** occurs, if the immune system does not respond to a molecule which is harmful

In the normal case if a single immune system cell kills a healthy cell it is not a problem. This immune system cell will not receive the costimulatory signal from the TH-cells and it will soon die by apoptosis. The body maintains a huge amount of cells and there are many other cells which can fulfil the function of the killed cell. The problem emerges if the immune system cell gets stimulated by the TH-cell. In this case the immune system develops a continually increasing population of antibodies which are responding to the detected healthy cell type. The response of the immune system on elements of the body is an autoimmune reaction. This can be defined as false positives in the detection system. The development of a method for the prevention of false positives in biological systems needs the understanding of mechanisms which are responsible for cells death. Further it needs a clear look at mechanisms how and why cells die in the known autoimmune disorders. A deep insight in autoimmune diseases is provided by [24].

The immune system always faces new types of pathogens. By evolution pathogens permanently are changing their genetic codes. Infectious agents which are already known by the immune system possibly mutate and evolve to some new types. Therefore beside the already known pathogens which are easily detected, the adaptive immune system always has to learn how to fight new ones. One of the central mechanisms for the detection of a pathogen is antigen presentation by the immune system cells. As mentioned before this process uses MHC molecules. If pathogens are altering their structure in such a way that MHC molecules cannot

bind to them then these pathogens cannot be recognized by the immune system. However, the antigen presenting cells possibly recognize those pathogens. But they never get stimulated by TH-cells for the detection and therefore those cells will die after a short time. This error is termed as a false negative by the immune system. The immune system must provide a big repertoire of MHC molecules to enable the binding to a huge amount of foreign antigenic structures. The body has a fixed set of MHC types which does not change during the lifetime. This repertoire is most diverse within a population. Because individuals do not have the same MHC repertoire the robustness to some special kind of pathogen is improved by evolution. The individuals that will withstand a virus infection have MHC types which can bind to its antigenic structure.

2.5 Summery about the tasks of immune system cells

This chapter provides a short introduction into the sophisticated mechanisms of the human immune system. The immune system maintains a huge amount of cells where each of them is entrusted with a minor task. The immune system gets its performance by the cooperation of these cells. The focus of the proposed approach lies on the adaptive immune system, since it is able to perform anomaly detection. This is done by the ability to identify elements that are not part of the body (nonself). After the recognition of those elements, the adaptive learning process (by high mutation rates of B-cells) leads to their elimination. A strong feature of the adaptive system is the immunologic memory. It enables the immune system to react on a reinfection with a pathogen.
The tasks of lymphocytes during the development of the immune response can be summarized as follows:
Many types of cells involved in the immune system are specialized to enable a speed up of the immune response in a repeated infection with the same pathogen. The response is developed in a way that it avoids autoimmunity. This indicates that the immune system can distinguish between elements of self and everything else. TH-cells are responsible for the tolerance to the self elements. At the same time they allow B-cells to evolve and mutate with high rates. By continued costimulation they guarantee that the cloned B-cells have an increasing specificity to the invading pathogens (all others die). This ensures a fast adaptation to a new pathogen. The task of B-cells is to diversify to become effective detectors, whereas TH-cells are not specific detectors. TH-cells are able to verify an antigen that is displayed by the B-cells. But they are not able to recognize a specific kind of pathogen on their one.
The adaptive immune system uses the two subgroups of lymphocytes to recognize and respond to two different types of pathogens. The B lymphocytes react on extracellular pathogens by establishing a binding between the receptors and epitope structures on the pathogens. The T lymphocytes react on intracellular pathogens where antigenic structure - that is bound to a MHC molecule - is transported to the surface of the cell.

3. Modelling a Computer Immune System

The sophisticated defence mechanisms of the human immune system are developed by evolution. In analogy to this system a model for a computer immune system is going to be designed. Up to now, for intrusion detection and virus detection several approaches have been implemented. Some of them are reviewed and analysed in [7, 54, 55, 57, 59, 60]. The proposed model in this thesis is influenced by the architecture of a computer immune system which was developed at the University of New Mexico [1].

It is important to understand how the main parts of the immune system are working and cooperating to ensure the elimination of harmful elements. By applying useful architectures, algorithms, and design principles, this knowledge will help to improve the design of computer systems. Systems which are able to adapt to different kinds of threats will make those systems more unique. In biology, some individuals are able to fight an infection before they will notice it, whereas some others are getting sick. The diversity of individual immune systems among a population is a feature that should be considered in the design of computer systems. It will automatically improve the robustness of computer systems. But it is not only the diversity feature we are interested in. There are more features we think about in this model.
It is difficult to design a useful defence system by copying the human immune system. It is not the goal of the design to apply the human immune system to the computer security domain [28]. Instead, the design would follow biology only as far as the implementation of the mechanisms appears to be useful for the special problem domain. Now, the main features of the immune system are pointed out [4]:

- The immune system is a **multilayer defence** system. Several layers fulfil their part of resistance against the pathogens. From the skin to the adaptive layer the immune system provides several barriers of defence and a high overall security.
- The *adaptive layer* ensures that the systems are individual and **diverse**. It learns to withstand a specific kind of pathogen and enables a population to be more **robust** to infections. An infection will have different effects on the individual systems. Some of them are getting sick, others withstand. It is a characteristic of unique systems, that the individual systems are susceptible in a different amount. The immune system supports a huge amount of diverse and redundant components. Its architecture ensures the continued functioning even if a few of these components fail.

- Since control is localized and not centralized, the immune system is robust and **error tolerant** against failures of some components. The immune response is the result of the interaction of a huge amount of independent components which are distributed all over the body.
- In combination with the multiple layers, the immune system provides a **distributed protection** mechanisms of the body. There is no single point in the system where an attack will cause the whole system to fail.
- The immune system **learns to respond** to a special kind of pathogen with increasing speed and efficiency. It is very important for the survival of the population that the system is able to perform a detection based on the recognition of anomalies. Individuals adapt to the new encountered pathogens and will detect new infections very often in their lifetime. The adaptive immune system learns to detect the new pathogens and finally generates an efficient adaptive response.
- It retains an **immunological memory** which enables the system to respond more rapid on known pathogens in future.

3.1 Related Work - UNM's Computer Immune System

A Computer Immune System called ARTIS [1] (artificial immune system) was proposed by Hofmeyr and Forrest (2000). It was developed at the Department of Computer Science, University of New Mexico (UNM). This system was inspired by the biological immune system. The main aspects of the Computer Immune System in this thesis are based on this architecture.

3.1.1 Overview of ARTIS

ARTIS consists of a huge amount of cells. The cells are the detectors of the computer immune system. Their function is comparable with the one of the lymphocytes in the biological immune system. They interact dynamically with the environment to establish an **anomaly based detection system**. The architecture follows the biological immune system to enable the detection and elimination of pathogens on a computer system. For this purpose the detectors are distributed over the system. Like in the biological systems there is no centralized control. The detectors are communicating and interacting in each node of the computer system only locally. Detecting and eliminating of pathogens is performed in each node independently. As a consequence, the ARTIS is very robust and error tolerant. It provides a distributed protection of the computer systems and is therefore robust to direct attacks.
The main problem of detecting infectious agents in Computer Systems is to distinguish between "**self**" and "**nonself**". In anomaly based detection systems, "self" is defined as all elements of the computer system that do not damage the system. "Nonself" are the remaining elements. The "nonself" set includes all elements that have a negative effect on the continuous functioning of computer systems. If "nonself" elements are detected by ARTIS, some action can be taken to eliminate them. Different elements may need to be eliminated through different mechanisms. The elimination is a separate problem and is up to the particular problem domain where ARTIS is applied.

3.1.2 Architecture of ARTIS

The **detectors** in ARTIS are designed as binary bit strings with defined length. The detectors comprise the main properties of the lymphocytes in the biological immune system. They contain an indicator for the detectors state which can be immature, mature or memory. Further they contain an activation flag which shows if the detector is activated and also the number of matches it has accumulated. The binary bit string of the detector is created by random and represents the receptors of the lymphocytes which bind to pathogens.
ARTIS discriminates between self and nonself through matches of binary bit strings. The applied match algorithm is called "**r-contiguous bits**" [3]. The r-value defines the threshold for the specificity of a detector. It also gives the number of contiguous bits that two strings must have in common to raise a match event. Chapter 4.2.3 gives a detailed description of this match rule.
Detectors contain a randomly created bit string and therefore they can respond to self or nonself. After their creation they are in the immature state, but only for a limited time. For the training purpose, the immature detectors are exposed to the local environment. This time is called **tolerance period**, because immature detectors should learn to respond only to nonself elements. All detectors that match a self string will be eliminated during this period. Detectors which do not match to anything during the tolerance period will reach the mature state. **Mature detectors** have to match several strings until the number of matches reaches the activation threshold. Thereafter, they will be activated and indicate an anomaly.
During the tolerance period usually not all elements of self can be presented to the detectors. The training of the detector set is based on **incomplete self sets**. The training of lymphocytes in the biological immune system is, as mentioned in chapter 2, also based on incomplete self sets. It is possible that mature detectors still are activated by elements that do not harm. ARTIS uses two methods to solve this problem. There are two kinds of thresholds implemented which influence the activation of the detectors:

- A detector becomes activated when the string matches exceed a threshold within a defined period of time. The **activation threshold** of detectors can be influenced with the local sensitivity level.
- The **local sensitivity level** is an adaptive adjustment of the activation threshold. A higher local sensitivity level corresponds to a lower local activation threshold. The sensitivity level will be incremented when a mature detector in the set detects a string and its match count is incremented from 0 to 1. The sensitivity level rises when a detector encounters a nonself string which was never seen before. After some period of time the sensitivity level is reduced again. This ensures that nonself strings which occur within a short time period are still detected. But rare self strings that may be present in the traffic do not trigger the detectors.

A human user has to verify the strings which are causing the activation of the detector. If the user decides that the detector has detected malicious activity, he will send a **costimulation signal** to the detector. This indicates to the detector that it has correctly identified a harmful nonself string and it will then become a memory detector. Memory detectors have a lowered level of the activation threshold and are therefore able to react with a fast and efficient secondary response. This feature models the immunologic memory of the biological immune system.

3.1.3 Using ARTIS for Network Intrusion Detection

A Computer Immune System for Network Intrusion Detection called LISYS (**L**ightweight **I**ntrusion detection **SYS**tem [1, 2, 9, 56]) was implemented at the Computer Science Department of the University of New Mexico. This system is based on the architecture of ARTIS.
In LISYS, network traffic is monitored. The elements representing self and nonself are modelled by **binary strings** containing the header information of the TCP network traffic. In LISYS, only the start of the TCP connection is monitored. This is done by processing captured TCP/SYN packets. The binary string which is representing a network connection is compressed. It consists of the **datapath information** that is extracted from the TCP/IP header. It is a string of 49-bit length and contains the source IP address and the destination IP address which are extracted from the IP-Header, and the used TCP service or port received from the TCP-Header.
The **self set** is defined as frequently occurring TCP network connections during the systems training period. The elements of the **nonself set** are then defined as TCP connections which are not occurring during the training period. The assumption was that nonself traffic is rare.
The model of the distributed environment is defined as a graph with vertices, where each vertex is represented by a computer in a LAN. The graph is represented by the connections of the computers in a LAN. The model of the local environment is defined by the detection node including many detectors. The number of the detector used in a detection node of LISYS is set to 100 at the same time.
The network topology is assumed to be a **broadcast network** and therefore, every computer receives every packet transferred over the network. In LISYS, mobile detectors, their replication, or an immune response is not implemented at this time. Also the elimination mechanisms for detected intrusion attempts are not implemented. In summary, LISYS implements most of the components and techniques given in ARTIS and uses network connections to determine the set of self and nonself.

3.1.4 Results from LISYS

A prototype of LISYS was implemented at the Computer Science Department of UNM. For the training of the detectors connection information was collected from 50 computers. The self set was represented by 2.3 million TCP/IP connections which were collected in a period of 50 days. From the logged TCP/IP headers 0.8 million headers have been filtered out. This was caused by noisy traffic sources from ftp or web servers. They usually have widely varying connection information and therefore, they do not provide a stable definition of self. Finally, 3900 unique connections were extracted. The system was trained on the self set over a time period of 30 simulated days. Afterwards for the remaining 20 days, it was exposed to the traffic in which seven different intrusion attempts did occur. These intrusion attempts were real incidents on the network and had been logged. The logged intrusion attempts were several types of port scans and IP address probes. LISYS has correctly recognized the seven incidents with an average of 1.76 false positives per day. This was a very encouraging result for a Network Intrusion Detection System. These results have been reported from the off-line prototype. An on-line prototype was implemented and tested but the results have not been published.

3.1.5 Limitations of LISYS

The implementation of LISYS has several limitations. They can be summarized as follows:

- Some limitations of LISYS result from its design to be used in **broadcast networks**. The current LAN technologies are moving towards switched Ethernet and broadcast networks are becoming rare. In modern network topologies, a computer receives only the packets that are intended for it. This speaks against the design philosophy of LISYS.
- In LISYS the **replication** of memory detectors and the proposed **mobility** of the detectors in the network are not implemented yet.
- The designers decided to monitor only **TCP datapaths**. But intrusions can also be executed at different protocols and layers. Examples are intrusion attempts by using the UDP protocol. LISYS does not detect them. At this time LISYS only looks for TCP/SYN packets. Therefore, also the detected attack range is very limited. Looking at the header of the TCP/SYN packets and extracting a 49 bit connection information can only result in the detection of connection related attacks, like different kinds of port scans.
- Another problem of LISYS is the correct setting of the **activation threshold** for the detectors. A higher threshold is important for reducing false positives. But in general, the use of a threshold will miss attacks if the attacker stays under a certain number of connections. There would be two possibilities for an attacker to sneak in undetected: by infrequent anomalous connections or when the number of connections stays lower than the activation level.
- LISYS will also not work on computer systems where applications like web browsers are going to establish connections to other computer systems, like WWW or FTP servers on the internet. Their possible IP range is not providing a **stable definition of the self set**.
- LISYS is trained on real traffic, assuming that **self occurs more frequently** than nonself. This is an assumption made in most anomaly based systems. If nonself is presented frequently, it would be treated like self and the system will fail to detect the particular nonself elements.
- Finally a limitation of LISYS is also that it is not completely autonomous, because **human operators** are needed. However, the architecture of ARTIS allows a learning process that is usually independent from user interactions.

3.2 Modelling a hybrid system architecture

As it is mentioned in chapter 1.1, for the detection of attacks mainly two techniques are implemented [38]. The first approach supports a large signature database that is kept up to date. It contains thousands of attack or virus signatures with the favour of detecting signatures by pattern matching. The second does not maintain a large database, but it watches the traffic for anomalous behaviour. It uses no signatures and detects attacks through protocol anomaly detection.
The **misuse detection** or **pattern matching approach** is reacting on the occurrence of typical byte sequences or combinations of them as the traffic flows by. It detects known malicious code patterns by inspecting the packets of the network traffic. The byte sequences stored in the database are compared with bytes sequences in the packets. If there is a significant match, an

attack has been detected. The advantage of this technique is that it is possible to detect all known malicious activities based on the signatures stored in the database with no false positives. A disadvantage of this approach is that it looks for known signatures and potential dangerous code patterns which have not been identified before will not be listed and can slip through.
The other approach, known as **protocol anomaly detection**, inspects the network traffic for flow and protocol anomalies. It can detect protocol violations or their misuse and can also scan the traffic for statistical abnormalities. The advantage is that it is effective at discovering new types of attacks, but it does not incorporate the knowledge about malicious code patterns.
The next step in developing a highly efficient detection system is to combine both techniques to a **hybrid system architecture** [11] using rock solid techniques and an adaptive learning technique to achieve an intelligent intrusion detection architecture. Such a system will achieve a significant better detection, especially for the currently growing number of polymorphic attacks [26]. Some of the commercial application firewalls are already trying to implement protocol anomaly detection. But in this field there are still many open issues. The successful detection of future threats will require mechanisms that are capable for the detection of polymorph attacks.

3.2.1 Defence mechanisms of the human immune system

In biology the immune system of the human body is equipped with defence mechanisms on **several layers**. Each layer is intended to build a barrier for a special kind of intrusion attempts. The outer layer, which is the first line of defence, is responsible for rejecting very basic kinds of intrusion attempts. Each inner layer has the task to filter a part of the intruding elements until the inner most layer which should face only the most advanced intrusions.

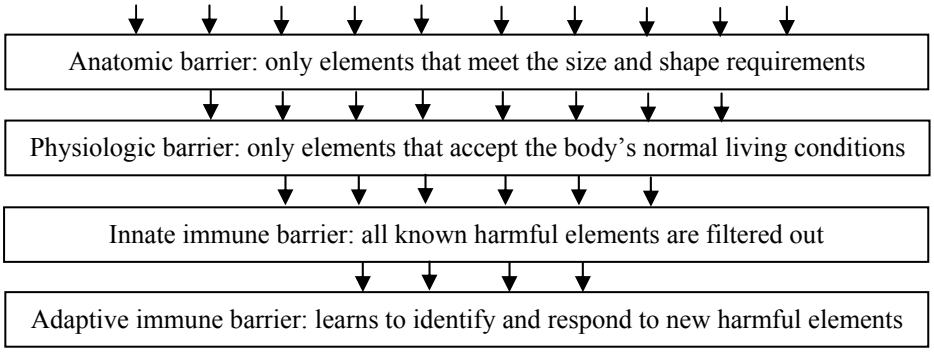

Figure 3.2.1: The defence mechanisms of the immune system on different layers.

The layered defence system of the human body consists of the skin which is the **first line** of defence. It is responsible to filter out microorganisms with some very basic methods. If it is not compromised by wounds, it provides an effective barrier to microorganisms that do not have the required size to reach the next layer. Only elements that meet the size and shape requirements are able to pass through. All other elements will be rejected. When the pathogenic microorganisms have entered the body they have to multiply themselves to reach a sufficient high number to invade the body which then can lead to disease.
The growth of the pathogens will be affected by the environmental conditions of the body. These are forming the **next barrier** for the invaders. This defence layer consists of physiologic barriers like low pH-value, high temperature and some kinds of soluble or cell associated protein structures. All of them are responsible for slowing down the growth of pathogens. The elements belonging to the self set are adapted to these living conditions and they are treated as

normal. But for some of the intruders, the prevailing conditions in the body are not normal. Due to that fact they will be blocked.
The **third layer** of defence is activated when the pathogens have already entered the body. There the innate immune system will eliminate all known bad elements. The innate immune system was learning this information to distinguish between good and bad during the evolution. This knowledge is given to us as a static defence mechanism. It enables the immune system to protect the body against all the already known pathogens.
The **fourth layer** of defence is the adaptive immune system. On this level the body can distinguish between self and nonself. It can identify and adapt to pathogens which have not previously met by the system. The adaptive immune system memorizes them to strike next time when they are met even faster and harder. This is the reason why every system will be unique.

3.2.2 Defence mechanisms of the computer immune system

In analogy to the layered defence mechanisms of the biological immune system, the proposed framework should include defence mechanisms on **several levels**. Each level is responsible for blocking a part of the intrusion attempts. As it is shown in the following figure they are handled on four different levels.

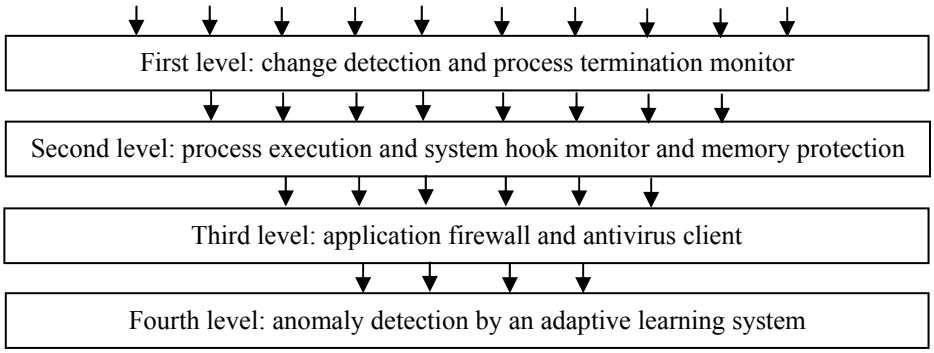

Figure 3.2.2: The defence mechanisms of the computer immune system on different levels.

3.2.2.1 The defence mechanisms on the first level

The first layer consists of an **applications and component control (ACC)** and a **process termination monitor**. The security of this layer will be increased if not just applications are monitored. It is important to inspect also the components of the active applications in a system. The defence mechanisms on this layer are able to protect applications from the termination through other processes. These mechanisms can also prevent the system from executing applications or their components when they have been modified. The ACC is monitoring changes on the local harddisk [37].
If a process crashes then the system creates some error messages. In contrast, when a process is terminated it is usually not noticed by the user. The process termination monitor alerts the user if a process, which has not been recognized and approved before (for example the Task-Manager), tries to terminate another running process [39].
The ACC provides also a significant improvement in security due to detecting changes on a computer system. A typical Win32 application uses many modules. These modules can easily be substituted by modules created from virus or trojan developers. The ACC will inspect the applications when they are starting up for possible changes in the files. If the application itself

or its components were modified and the application tries to access the local network, then the ACC will raise an alarm. It also warns the user if the application tries to build up a connection to other computers on the internet. The user has to decide if he wants to allow or stop the activities for this changed component. Through that the ACC ensures that the executable files and their modules are not altered and malicious.
Trojans are not always standalone applications. They are sometimes installed as a component used by a well known Win32 application (just to mention one example - the web browser). In Microsoft Windows systems the security privileges of applications are directly applied to their components. If the host application has network access rights then its components will also receive automatically the privileges needed for establishing a connection to other computer systems.
The ACC on this level allows the user to view the components and the applications it is currently monitoring. It also should provide the possibility for the user to remove any of the monitored applications and components from the trusted list. The list contains components which are used by several applications in the Windows operating system. The user will not be called for attention when a shared component is going to be used by applications to which it initially has not been registered. The trusted list is a database of executable files and their components which is continually kept up to date. Initially, all components of the Win32 platform will be added to the database because these shared components are requested by most applications. But this initial setting can be modified as mentioned before. Every executable file that is not listed and trying to establish a connection is going to raise an event in which the user is asked to allow or to forbid activities from this file. If the user decides to allow the connection of this file, it will be added to the trusted list of files with a signature to make it possible to monitor also changes of this file.

3.2.2.2 The defence mechanisms on the second level

The second level of defence is the **monitoring of hidden process execution** and **global system hooks**, and a control for the **protection of the physical memory**. On Windows systems, several applications are capable for transferring data over the network, but do not implement a routine for the network access. They use external references to establish the connection through separate processes. This technique enables the applications to evade the detection by traditional firewalls. The applications are executing additional processes which carry out the desired task. These processes can be separate applications or components which provide the needed functionality. If separate applications are launched then the firewall security restrictions of the parent application are not applied to these child processes. Furthermore, these processes are not visible to the user. Thus, the actions which are really performed by the application are effectively hidden. This technology is e.g. implemented by commonly needed applications, like the Microsoft Internet Explorer. It is used to carry out repeated and automatically performed maintenance operations without the need of user interaction. Such maintenance operations are checking if updates or patches on some servers are available. But on the other side trojan applications may use this mechanism to initiate unauthorized activities in the system, such as stealing passwords and sending them to a remote host. The second layer of defence should give the user the ability to control hidden processes. Especially this applies to processes that are intended to perform some specific tasks for the trusted application. Thus, unwanted activities in the background can not be fulfilled without the notice of the user.
Many trojans, viruses and especially shellcodes use sophisticated techniques. These techniques allow them to modify the executable code of the trusted application when it is actually running and loaded into the memory [39, 62]. They can bypass the security definitions on a system and

launch their malicious activities with the privileges of the hijacked process. One method to bypass a firewall is to directly inject (without the use of threads) some code into the web browser. This is known as copycat attack. There are several other methods to modify the physical memory of running processes. For example thermite attack which injects code through a DLL to the target and creates a thread in the attacked process. The firehole attack uses a global system hook to load a DLL into all processes which are using the systems user32.dll. It gets notified when its own component is injected into the web browser and then it can perform malicious activities by using the browsers network connection.
This second level of defence should therefore monitor and control functions which can be used to write bytes to the memory and functions which are used for global system hooks. On this level it is possible to watch these functions and prevent them from injecting malicious code into the address space of trusted applications.

3.2.2.3 The defence mechanisms on the third level

The third level of defence will be an **application firewall** in combination with an **antivirus client** [40] that holds a database of already known attacks or viruses. This databases need to be kept up to date. Installing both of these applications on a computer system is an important step to a higher security level. The database lists definitions in form of signatures of known attacks or viruses. These signatures are a kind of fingerprint, which exactly identifies both through a short or compressed description of the contents. The description may include a special sequence of instructions or bytes, which can only be found in this kind of attack or virus. The description contains also important information about the configuration of a received packet. Such information can be a set flag, the port or service which is going to be used or a significant part of the payload which is also constantly the same in this kind of attack. Microsoft has released the Service Pack 2 for Windows XP systems in summer 2004 [41]. It enables by default a simple firewall which is a good step in the right direction.

3.2.2.4 The defence mechanisms on the fourth level

The fourth and last level of defence is the **adaptive learning system**. In a training period the computer system should learn to distinguish between self and nonself. It should learn to be tolerant to the normal network traffic which is defined as self. But the computer system should also recognize and react on nonself packets. Nonself packets are infrequent occurring parts of the traffic. These should be detected by the system that has learned to be tolerant to frequently occurring packets. Nonself elements are defined as anomalies in the network traffic and therefore, the implemented adaptive system is an anomaly detection system. This work focuses on the implementation of the fourth level of defence, the adaptive learning system.

3.3 The representation of self and nonself

In the human immune system the discrimination between self and nonself is based on chemical bonds that are set up between complementary formed protein structures. These structures are receptors of the immune cells and the epitopes of pathogens. The decision what is self or nonself in the human immune system relies on fragments of proteins.
The representation of self and nonself in the computer immune system should use a comparable model [5]. The protein fragments in the computer immune system are designed as **binary byte**

strings of fixed length. The universe U of strings of defined length is divided into two subsets. The set of self S and the set of nonself N. U equals to the union of S and N, but their intersection results in the zero set. The computer immune system then has to discriminate between the sets S and N. Usually the set N includes much more elements than set S. This is, since the set S of a single computer system represents only a small fraction of the possibilities in U. Further, N represents everything that is not self. The system must decide for a given arbitrary string in the universe if it actually belongs to the self set or the nonself set. The nonself set contains all packets that are classified as being anomalous.
The decision is based on the actual defined contents of the self set. The problem is that neither the immune system of the body nor the computer immune system have complete self sets. The training in both systems is based on an **incomplete definition of self**. Therefore, in the immune system mature lymphocytes possibly cause an autoimmune response. But this is prevented by an additional stimulation signal (see chapter 3.5.2 for details).
In the computer immune system the autoimmune response is denoted as a **false positive** of the detection system. False positives appear in a system when the description of self patterns is incomplete [43]. The system will somewhen observe new patterns which belong to self. But since these were not included in the definition of self patterns they are treated as nonself and are producing false positives.
The other kind of failure occurs when the system classifies a nonself string as being normal. These **false negatives** are also occurring in the human immune system, when pathogens are not detected and eliminated. These types of detection errors are harmful to either system. The goal is to minimize the appearance of both errors. Therefore, it is important to the computer immune system to think about the selection of proteins structures which are able to classify elements as self or nonself [17, 18].

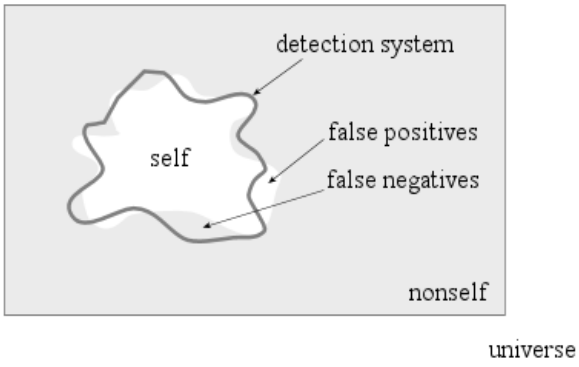

Figure 3.3: Shows the representation of the all strings belonging to the universe (see [1]). The nonself is defined as the complement to self and each string either is part of self or nonself. The immunological detection system tries to reflect the self set accurately.

3.4 The detectors

The immune system of the body uses a multitude of immune cells for the detection and elimination of foreign organisms. The detectors in this system are based on the group of immune cells called lymphocytes. The detectors combine the functionality of both lymphocytes subgroups. The implemented system is similar to the human immune system. The detectors are like the lymphocytes in the body acting in the local environment. They are changing their states and carrying out different kinds of actions. During the lifetime of a detector it can achieve **different states**. These states are introducing different phases and abilities. The system consists

of many **independent and mobile** detectors which can circulate in a distributed environment. Detectors with a certain state can spread out to all computer systems present in a network. But they are communicating and interacting only in the **local environment** of a computer system.
The lymphocytes and the antibodies in the human immune system are responsible for the detection of pathogens. The detection event occurs when a sufficient high number of receptors of a lymphocyte bind to epitope structures on pathogens. The likelihood for an occurring bond is expressed through the affinity between the receptor and the epitope.
In the computer immune system the receptors and epitopes are represented as character sequences of defined length. A **detection event** is produced by approximate string matching algorithms. Each detector is equipped with a binary string which is modelling its receptors. Some possible approximate matching rules are presented in chapter 4. All algorithms are going to be implemented and tested on the ability of discriminating between self and nonself. To define and influence the specificity of the detector, a **threshold** is needed. With this threshold, the specificity of the detectors can be adjusted. The specificity is the size indicator of the subset of similar strings which can be detected by a single detector. For example, if the threshold is equal to the length of the binary string, the detector can only match a single string. A completely general detector is defined through a threshold value that is zero. Such a detector will match every binary string. In practice it will be useful to apply a partial matching rule with a threshold somewhere between these two values. This value is strongly influenced by the used match algorithms. To determine the specific threshold for each match algorithm, some different parameter configurations will be tested. But in general, the **level of detection** in a computer system depends on the combination of two values:

- the used **threshold for the detector specificity**
- and **the number of detectors**.

The same level of detection in a computer system is kept if both values are changed to the same direction. It is necessary when the threshold is increased to use also a higher number of detectors.

3.5 Training the detection system

Lymphocytes are created in the bone marrow and their diversity results from a pseudo random process. Different lymphocyte genes are created in a DNA recombination process and are causing diversity among their receptors. The new generated lymphocytes can then bind to self cells or to the pathogens. Lymphocytes in the body are trained to bind to the epitopes of pathogens. During the training the lymphocytes should learn to bind to nonself. In this period if the lymphocytes are recognizing anything they will be dieing by apoptosis (negative selection process). All lymphocytes that do not match during this time will be selected to survive. The time in which the lymphocytes are learning not to detect the cells of the body is called tolerance period. Afterwards they have to be tolerant to the self set. The human immune system is training the lymphocytes on self proteins structures. These structures are expressed to them in the thymus to get detectors which are able to discriminate between self and nonself. After the training period the lymphocyte will be mature. When it is activated, the body reacts as if a pathogen was detected.
The training of the lymphocytes in the immune system is done by the **negative selection process** which ensures the recognition of nonself elements in the body. The advantage of using negative selection for the detectors is explored and analysed by [6, 54, 58]. An efficient negative selection algorithm is proposed by [61]. It is especially designed to be used by a system that

implements the r-contiguous match rule with a binary representation of the data. The architecture of ARTIS includes this algorithm. But also non-binary alphabets have been used to achieve a self/nonself discrimination in a computer system [5]. This mechanism is adopted in the computer immune system for the training of the detector set to achieve anomaly based detection.

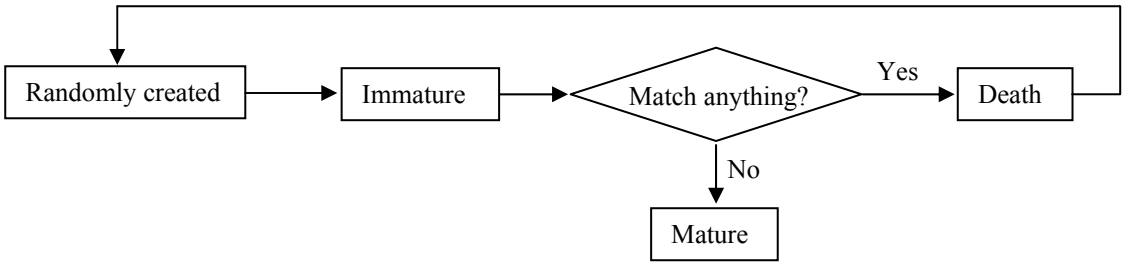

Figure 3.5: The negative selection process for a detector in the tolerance period. The state of new randomly created detectors is set to immature. This state is kept for a time period in which the detector is exposed to the environment. If it matches anything it will die and is replaced by a new one. Every detector that does not match to a string in the tolerance period is released in the mature state.

This computer immune system implements the negative selection algorithm. Each detector is created analogous to the receptors of the lymphocytes with a **randomly generated string**. Its state is set to immature. Detectors remain immature over the **tolerance period**. In the computer immune system the detectors are not trained centrally like the lymphocytes in the thymus, but more distributed in the system perimeter. During the tolerance period, the detector is released to the environment of the local computer system. There it can detect self and maybe a few nonself strings. If the detector matches any of the presented strings it is eliminated and replaced by a new randomly created **immature detector**.
For the training it is important to assume that nonself will occur infrequently and most of the observed traffic contents are part of self. Otherwise the system would also become tolerant to the frequently expressed nonself. The decision what are **self patterns** is based in the anomaly detection system on the frequency they are observed by the system. All nonself strings that are observed during the tolerance period will be treated by the system as part of the self set. Therefore, the detectors will not detect these patterns. They would become tolerant and remain tolerant for the whole lifetime. But this is not a problem for the detection of nonself by the system, because the detectors do not life longer than a few days. Therefore, a fraction of the detectors is always in the immature state. This provides like in the human immune system a **dynamic detector population**. Consequently, for detectors in the mature state, a nonself pattern will only escape the detection if it is frequently present in the tolerance period. The system faces only false negatives if nonself is continually observed and this is against the assumption made for anomaly detection systems. Furthermore, a dynamic detector population enables the system to adapt to slowly changing self sets. When the self set changes, the system releases new mature detectors from the tolerance period. The other already matured ones that are causing false positives will be replaced due to the missing costimulation or because of their age.
Like the B-cells in the body, if the detector does not match anything during the tolerance period, it will change the state and is upgraded to a **mature detector**. All detectors have a probability of dying once they have matured. The number of detectors in a system always stays the same. When detectors die they will be replaced by randomly created and new immature ones. In general all detectors in the set will be dieing, unless their state has been set to the memory status. If mature detectors match a string they will be activated and signal the immune system

that they have detected an anomaly. The mechanism of **anomaly detection** relies also on the frequency based assumption. Immature detectors that react on self elements have a high chance to observe these strings during their tolerance period. On the other hand when an immature detector reacts on nonself elements it has a low chance to detect these strings during its tolerance period. Activated mature detectors reach the possibility to get upgraded to **memory detectors** which are long-lived and not eliminated over time.

3.5.1 Memory based detection

The human immune system has an adaptive defence layer. The adaptive immune system evolves a set of lymphocytes that specifically adapt to some kinds of pathogens that it had encountered. It memorizes this information to speed up the response in future. When there is a reinfection of the body the immune system does not have to learn how to recognize the pathogens again. It remembers the specific kind, and it will react with a much **faster and effective secondary response**. The secondary response is based on the detection effect with a memorized number of former trained lymphocytes. The memory based detection in the immune system is capable for detecting also different pathogens which have a related structure to previous encountered ones. This is for example the case at evolving and mutating pathogen populations. When the infection is eliminated, a fraction of high affinity cells are retained as memory cells in the body. Memory cells are not replaced over time. The fraction of memory cells includes enough lymphocytes to combat the pathogen with the secondary response if it is met again.
The computer immune system supports memory based detection in a similar way. When a nonself string activates more than one detector at a node then those detectors with the highest match value, according to the match rule, are upgraded to the memory detector status. Memory detectors are also activated by **nonself strings with related structure**. This is ensured by the threshold for their specificity which is adjusting the value for partial string matching. The computer immune system can also detect new nonself elements that have similar structures. Every match algorithm mentioned in chapter 4 supports this feature.
Activated detectors are asking for a costimulation signal (see chapter 3.5.2). The detectors which finally receive the costimulation become memory detectors. These detectors will make a copy of themselves. The copy then is sent to a **signature database**. From the database, those **successful detectors can spread** out to all nodes of the network. A representation of the detected string will be distributed over the network. Then all future occurrences of it are fast detected in any node of the network. This imitates the features of the second response in the human immune system.
As mentioned, the exception to the finite lifetime of a detector will be if it achieves the memory status. Similar to the memory cells in the immune system of the body. Detectors which reach the memory state have a **long lifetime** and their patterns are remembered to provide a lifelong protection against threats. Memory detectors can only die when they are also responding on self strings. In that case they will not receive the costimulation signal again.
A problem in the memory based detection is that all detectors of the set sooner or later could achieve the memory detector state. This would result in a more and more shrinking number of immature and mature detectors and lowers the effect of having a dynamic detector population. This problem is solved by defining a **maximum number of detectors** that can be in the memory state at the same time. This number is set to some fraction of the total number of detectors. When a new activated detector receives the costimulation it becomes a memory detector. If the fraction of the memory detector population exceeds the defined limit, then the upgrade is done with the least-recently-used (LRU) mechanism, as mentioned in [44]. The

memory detector which was activated least recently will change his state to a simple mature detector with again having a finite lifetime.

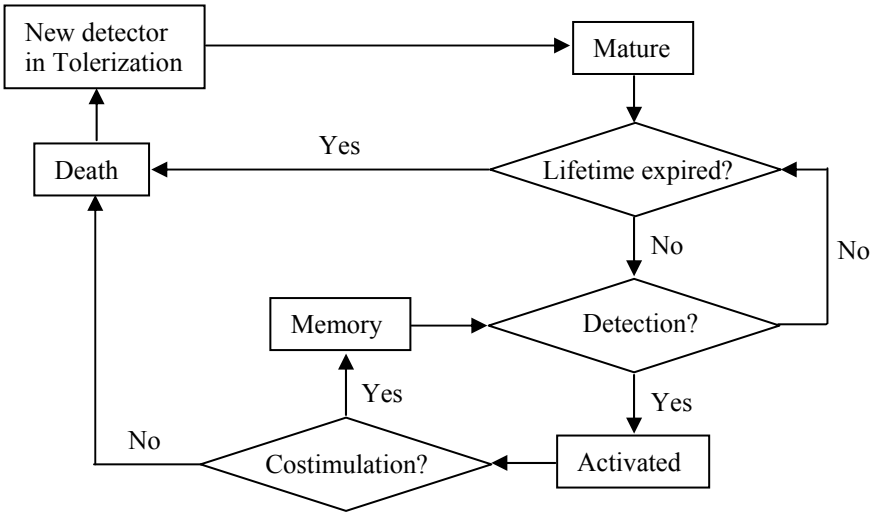

Figure 3.5.1: The figure shows the life-cycle of the detectors. Once a detector is released from the tolerance period, its state is set to mature. A mature detector has a finite lifetime. When its age is exceeding this time it will be replaced by a new random detector. If a mature detector detects a possible nonself string it will be activated and requests a costimulation signal. If the detector has detected a string of the self set it will not receive the costimulation signal and it dies. But if it has detected nonself the costimulation signal is sent back and its state is upgraded to a memory detector.

3.5.2 The costimulation signal

T lymphocytes are matured centrally in the thymus. But unfortunately the set of self proteins in the thymus is incomplete. They will be tolerant to most but not all proteins structures contained in the human body. Therefore, the negative selection process in the thymus maybe is not complete. T lymphocytes could get matured but possibly still be activated by a self protein which was not presented in the thymus. Because of this the T lymphocytes can cause an **autoimmune reaction**. The immune system however combats that problem by an additional stimulation procedure. An autoimmune reaction is **prevented by a required costimulation**. There are two signals needed to activate the T lymphocytes. The one occurs automatically when binding to antigen presenting cells. The second signal is provided by cells of the innate immune system and some other cells when the tissue of the body is damaged somehow. T lymphocytes are dieing with lack of receiving the costimulation. When autoreactive T lymphocytes bind to self proteins in absence of tissue damage they will be eliminated. In general, these lymphocytes will have a high probability of dying before reaching an area with damaged tissue, because most of the tissue is not damaged when circulating through the body.
Also the training of the detectors in the computer immune system takes place on incomplete self sets. If the detector survives the tolerance period its state is changed from immature to mature. The mature state indicates that the detector is capable of detecting nonself. But it cannot be assumed that a detector has encountered every possible self string during its tolerance period. Resulting in mature detectors that may match a few strings in the self set. A **human operator** can provide the second signal within the costimulation delay. Detectors that are activated by

strings will send a costimulation request to the human operator which then verifies what the detector has recognized. If it has detected nonself the operator sends the second signal back to the detector. Otherwise no signal will be sent and the detector dies and is replaced by a new randomly generated detector. The human operator therefore does not have to respond when the system has incorrectly recognized something. Some occurring false positives are corrected by the system itself. This system is therefore **largely autonomous**.

4. Matching algorithms

The immune system must be able to distinguish between self and nonself. The discrimination job is the central task in anomaly detection systems based on the immune system. Many approaches can be used to identify nonself elements. The question is: Which algorithms are better suited for this kind of application?
The algorithms are independent from the data representation and can be applied to binary byte strings as well as to e.g. DNA chains. They return a value which is representative for the affinity between two sequences. In our particular problem domain, the algorithm must measure the strength of the bond between the detector sequence and any data sequence under investigation.

Two different methods are checked if they are able to significantly separate self elements from nonself elements. The first method is based on associations between data sets and the algorithm is the widely used Pearson correlation coefficient. The second method includes four algorithms based on approximate string matching [42]. The source code is presented for all of them to analyse the complexity of the needed algorithms.

4.1 Pearson Product-Moment Correlation Coefficient

This method is often used to determine the association or correlation between data sets which are measured through the consistence in the relative position for pairs of values [35, 36]. The correlation coefficient assumes in a coordinate system a straight line dependence between the two data sets. It measures how good a deviation in data set #1 is matched by the deviation of the corresponding point in dataset #2. It is called product moment correlation because it can be calculated by the sums from the products (crossproduct) of the deviation of the values from their mean. The correlation coefficient results in a value between -1 and +1, where -1 indicates the best negative correlation, 0 means no correlation and +1 stands for the best positive correlation. It is in practice quite cumbersome to calculate it from the deviation of individual datapoints. But another formula is easier to implement and it is used to compute Pearsons r:

$$r = \frac{n \cdot \sum x_i y_i - \sum x_i \cdot \sum y_i}{\sqrt{\left[n \cdot \sum x_i^2 - \left(\sum x_i\right)^2\right] \cdot \left[n \cdot \sum y_i^2 - \left(\sum y_i\right)^2\right]}} \qquad (4.1)$$

The variable n is the number of values in the set and x_i,y_i are the specific values in data sets at position i. The significance of the correlation coefficient can be tested by the formula:

$$t = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}} \qquad (4.2)$$

The resulting value t gives a statistical match value which is then compared with cutoff-scores in the so called T-table. For a defined error probability the T-table shows which t value the Pearson coefficient must reach to be significant.

4.2 Approximate string matching approaches

In general the approaches return values for the similarity of sequences. But there is no universal and precise mathematical definition for the notation of similarity. It very much depends on what needs to be compared. When comparing some specific words in the dictionary they may sound and are spelled very similar, but they have a completely different meaning. The same problem occurs in microbiology: Chains of proteins may be similar concerning the functions, the molecular structure, or also the contained sequence of special amino acids. The focus lies on the shape of molecular chains, which is determined through their sequences. But the shape is the indicator for their specific function. The search of similarity of sequences in biology has the purpose to find similarities in shape and so the function of proteins. But unfortunately many times sequences have low or even no similarity in their sequence and nevertheless their molecules fit to same shape or fulfil the same functions.
The computer immune system mimics biology. Therefore, it is of interest to match code patterns which have similar functions. To measure the similarities of symbol sequences concerning their functions some flexible and more general approaches are needed. The approaches presented here differ in their flexibility and can be divided in two methods for determining the relation between two given sequences [32]. The first method is the measure of similarity. It assigns a representative numeric value to the two sequences that are compared. A higher value is associated with a greater similarity. The second method measures the distance of two sequences. In general, distance and similarity values are dual to each other, because a higher distance value will result in a lower similarity value.

4.2.1 Hamming distance

Several algorithms are measuring the distance between strings. The Hamming Distance [34] is a simple form for determining the distance. It compares two strings with equal lengths and counts how many symbols are different at the corresponding locations in the strings. It can also be seen as the number of symbol substitutions which are needed to change one sequence until it is equal to the other. If l is the length of the strings, then l comparisons for the calculation of the Hamming distance are needed. This is resulting in an asymptotic running time of $\mathcal{O}(l)$. For example:

```
A = 01011011
B = 01010110
```

The Hamming distance between A and B is 3.

The calculation of the Hamming distance needs strings with equal lengths. This measure of the Hamming distance is often useful, but sometimes a more flexible algorithm is needed. For example if strings have different lengths, it can be solved by the sliding window in the algorithm below:

```
String1: A[0..m]
String2: B[0..n]

for x:=0 to m do
      tmpmatch := 0
      z:=0
      for y:=0 to n do
            if A[x+z] = B[y] then
                  tmpmatch++
            z++;
            if (x+z)>m then break
      if tmpmatch>matchlen then
            matchlen := tmpmatch

return n - matchlen
```

Algorithm 1: Calculation of the Hamming Distance

When two strings are compared, they usually do not have equal length. In this case the smaller string (assume it is B), is slided like a window byte per byte towards the end of string A. For every sliding step all bytes in the window are compared. Every match increases the temporary match counter. When the window processing is finished, the actual temporary match counter is compared with the maximum value recorded so far. When the new value is bigger it is set as the maximum value. Then the window B slides to the next position in the A string. After the window has reached the end of the A string, the Hamming distance is given back as the length of B minus the maximum recorded match count.

If m is the length of the data sequence and n is the length of the string B then the string B is needed to be shifted and compared m times with n symbols, resulting in an upper asymptotic running time of $\mathcal{O}(nm)$.

The affinity P of the occurred match is defined by:

$$P = 1 - \frac{d}{l}, \tag{4.3}$$

where d is the value returned by the Hamming distance and l is the length of the comparison window.

Probability of a match

$HAMM_{l,r}$ (*string1*, *string2*) defines the occurring match when using the Hamming distance. Both strings are of length l and match if they have at least r equal bytes in corresponding positions. The match probability of two randomly selected byte strings is defined by:

$$P(HAMM_{l,r}(string1, string2)) = b^{-l} \cdot \sum_{i=r}^{l} \binom{l}{i} \qquad (4.4)$$

as presented in [28]. b^{-l} defines the probability for the occurrence of a sequence of length l in a alphabet of b members, and l over i counts the number of strings having i equal bytes in the comparison windows of length l.

The Hamming distance allows only substitutions on fixed byte positions. More flexible are algorithms with additional operations like insertions or deletions.

4.2.2 Levenshtein or Edit Distance

The Levenshtein distance or edit distance [33] is a more sophisticated method to determine the distance between two strings. The algorithm is returning the minimum number of required edit operations to change one of the two strings that it is equal to the other. The allowed operations are deletions, insertions or substitutions.
With more allowed operations edit distance becomes more general than the Hamming distance. The difference is shown with the following two strings A and B:

```
A = CBEGAFDK
B = BEGNFDKK
```

The Hamming distance between A and B is 7 characters. When measured with the Levenshtein algorithm a distance of only 3 characters is returned. Three edit operations are needed that both strings will be equal to BEGAFDK. The operations are the deletion of C from A and the last K from B, and the replacement of N in B with A from A.

The edit distance uses following definitions of one-character operations, which are necessary to change a sequence A to another sequence B.

(x,x) defines a character match and results in no change from A to B
(x,y) defines a character replacement of x in A by y from B, where x is not equal to y
(x,-) defines a character deletion, x in A
(-,y) defines a character insertion, y in A

These definitions are symmetric in A and B. When a character is inserted in the sequence A also one could have been deleted from the sequence B. The edit distance will not require two sequences of the same length. The size of A and B can be altered by insertion or deletion of characters to get finally the same length.

```
A: CBEGAFDK-
B: -BEGNFDKK
```

This example shows one Delete (C,-), one Replacement (A,N) and one Insert (-,K), thus 3 needed operations. The operations are processed with characters at corresponding positions in

both sequences. The pairwise applied edit operations will transform the sequence A finally into B.

```
String1: A[0..i]
String2: B[0..j]

for x:=0 to i do
      d[x,0]:= x
for y:=0 to j do
      d[0,y]:= y

for x:=1 to i do
for y:=1 to j do
    if A[x] = B[y] then cost:= 0
    else cost:= 1
    d[x,y] = min( d[x-1,y  ] + 1, // insert operation
                  d[x  ,y-1] + 1, // delete operation
                  d[x-1,y-1] + cost)// character substitution
return d[i,j]
```

Algorithm 2: Calculation of the Levenshtein or Edit Distance

This algorithm uses a two dimensional array with the size ij. A character of string A is compared at a time with each character of string B. If the current position B[y] is equal to A[x] then the edit cost is zero. Otherwise an edit operation is needed. The algorithm selects that operation which is resulting in the minimum value for the current cost.

The affinity P of the occurred match is defined by:

$$P = 1 - \frac{d}{\max(i, j)}, \tag{4.5}$$

where d is the distance returned by the Levenshtein algorithm and i and j is the size of the strings.

The calculation needs $\mathcal{O}$(ij) operations and also $\mathcal{O}$(ij) space. The algorithm can be optimized in its space requirements if the value for the edit distance should be calculated. Then the allocation of two rows of the previously two dimensional array is enough and the needed space is $\mathcal{O}$(i).

4.2.3 R-Contiguous Symbols

This is a matching approach which is more related to the immune system of the body [1, 8, 10]. It returns the number of common contiguous symbols in two sequences. For example:

```
A = 01011011
B = 10010101
```

The sequences A and B match with the four contiguous symbols. This match rule can also be applied on two symbol sequences which have not the same length. Then the shorter sequence is slided over the longer sequence of symbols. The algorithm returns the maximum number of contiguous symbols found in both sequences. The variable r is the discriminator, the threshold value which determines if two sequences are matching or not. If in the above example r is given with the value 4 then the sequences A and B are matching. If r is going to be 5, then the

sequences A and B do not match. Because the compared sequence A does not have 5 contiguous symbols with B in common.
In terms of an immune system, the threshold describes the specificity of a detector. The higher the threshold, the more selective a detector will be in its matches.

The first sequence is defined to be A = (a[1], a[2],…, a[n]) and the second sequence is denoted by B = (b[1], b[2],…, b[m]), and the variable match holds the maximum number of the contiguous symbol sequences of (a[1], a[2],…, a[n]) and (b[1], b[2],…, b[m]).

```
for x:=1 to n do
      c[x,0]:= 0
for y:=0 to m do
      c[0,y]:= 0
for x:= 1 to n do
      for y:= 1 to m do
            if a[x] = b[y] then
                  c[x,y]:= c[x-1,y-1]+1
                  if match < c[x,y]
                        match = c[x,y]
return match
```

Algorithm 3: Calculation of the R-Contiguous Symbols

The complexity of the matching algorithm is determined for sequences with different length by the needed shift operations. If m is the size of the longer sequence and n is the size of the shorter sequence then the shorter symbol sequence is shifted and compared m times with n symbols. The upper asymptotic run time of the algorithm is $\mathcal{O}(nm)$. When both symbol sequences have equal length then the run time is $\mathcal{O}(n^2)$.

The affinity P of the occurred match is defined by:

$$P = \begin{cases} 0 & \text{if match} < r \\ \dfrac{match}{\max(n,m)} & \text{if match} \geq r \end{cases} \qquad (4.6)$$

whereas `match` is the variable defined above.

Probability of a match
$CONTIG_{l,r}$ (*string1*, *string2*) defines the occurring match under the r-contiguous symbols rule. Both sequences *string1* and *string2* are of length l and match if they have at least r equal contiguous symbols in corresponding positions. The match probability of two randomly selected sequences is defined by:

$$P(CONTIG_{l,r}(string1, string2)) = 1 - b^{-l} \cdot T(l,r) \qquad (4.7)$$

$$T(l,r) = \begin{cases} b^l - b^{l-r} - (l-r) \cdot b^{l-r-1} & if\ r \leq l \leq 2r \\ 2T(l-1,r) - T(l-r-1,r) & if\ l > 2r \end{cases} \qquad (4.8)$$

as presented in [29]. b^{-l} defines the probability for the occurrence of a sequence of length l in a alphabet of b members. T(l,r) gives the absolute number of strings of defined length l that have not r symbols in common. So T(l,r) calculates the complement of the matching strings.

4.2.4 Longest Common Subsequence

The longest common subsequence [30, 31] solves the problem of finding the longest consecutive sequence which is common across multiple strings even with the possibility of having other symbols between corresponding symbol positions. It is an important application in the bioinformatics field when similarities among protein sequences need to be known. Genes are changing with the evolution of the species and the proteins they encode change with them. The task is now to search different species for regions in the DNA chains which fulfil the same functions. This search is done under the assumption that there must be a similarity in the chain for proteins functions which were preserved during evolution. When looking for the longest common subsequence of molecular structures within corresponding proteins among different species it is possible to detect molecules that have not been changed through evolution.

Computer systems can apply such a method to the search of similarities in polymorph evolving threats. Polymorphic viruses consist of a decryption engine and the corresponding encrypted code. For their detection it is very important to analyse the mechanisms how differences in the decryption engine can be generated. Generators for polymorph viruses are using many different mechanisms to evade the detection. For example when garbage instructions are inserted between the instructions belonging to the decryption engine then this could still be detected by this match algorithm. This is because the sequence of the instructions belonging to the decryption engine often stays the same.
The longest common substring algorithm can be seen as a modification of the edit distance. It allows two edit operations on the sequences, which are the insertion and the deletion of symbols. The character substitution is not allowed and it can be computed faster than edit distance.
For example:

A = BCEAEAEK
B = BEAEAENK

The longest common subsequence of A and B is BEAEAEK. Thus the result is 7.

The first sequence is defined to be A = (a[1], a[2],…, a[n]) and the second sequence is denoted by B = (b[1], b[2],…, b[m]). The two dimensional array c of size [n,m] holds the maximum length of the longest common subsequence at index c[n,m].

Then

$$c[x,y] = \begin{cases} 0 & \text{if } x = 0 \text{ or } y = 0 \\ c[x-1, y-1] + 1 & \text{if } x, y > 0 \text{ and } a[x] = b[y] \\ \max(c[x-1, y], c[x, y-1]) & \text{if } x, y > 0 \text{ and } a[x] \neq b[y] \end{cases} \quad (4.9)$$

In the implementation of the equation of c[x,y] the algorithm looks like this:

```
for x:=1 to n do
      c[x,0]:= 0
for y:=0 to m do
      c[0,y]:= 0
for x:= 1 to n do
      for y:= 1 to m do
            if a[x] = b[y] then
                  c[x,y]:= c[x-1,y-1]+1
            else
                  c[x,y]:= max(c[x-1,y],c[x,y-1])
return c[x,y]
```

Algorithm 4: Calculation of the Longest Common Subsequence

This solution does not return the longest common subsequence itself but its length. The subsequence can be recovered by a trace back function from the matrix c. The upper asymptotic run time of the algorithm is $\mathcal{O}(nm)$.

The affinity P of the occurred match is defined by:

$$P = \frac{c[n,m]}{\max(n,m)} \tag{4.10}$$

where c[n,m] is the maximum length returned by the algorithm and n and m is the size of the symbol sequences.

5. Implementing the Computer Immune System using SNORT

In this chapter the implementation of the computer immune system is presented. It is shown how it succeeds at detecting malicious code in packets of the network traffic. The targets for this system are Internet worms, different kinds of viruses and shellcodes which are possibly polymorph. The targets are further all other pieces of code that have a negative effect on the continued functioning of computer systems and on larger scale local area networks (LANs).
Today's computer systems are running (mostly) one kind of operating system which makes it easy for a new virus to spread rapidly. Independently evolving immune systems will make computers in a network more diverse. Because of that, systems will become more robust to today's common threats. The implementation in this work is intended to shield computers in a LAN from new network driven attack attempts (see [12]).
Each network node is equipped with a sensor which is used to train an individual set of detectors. This set evolves in response to the network traffic at this node. The set of detectors will be different in each node. That makes the whole network system highly diverse. A vulnerability in one system refers not automatically to the same kind of security hole in another system. Applying the computer immune system with multiple independent sensors across a network ensures a distributed detection system which is not centrally or hierarchically controlled.

In this chapter the framework of the computer immune system is described in detail. The implementation takes advantage of the Open Source Network Intrusion detection system called Snort [16]. It provides the basis for processing the packets of the network traffic. Since it is an open source intrusion detection system it also has a large community of users from which newly developed plug-ins can be tested. Now, the following chapter gives a short overview about the Snort Framework. For further information about Snort the reader is referred to [13, 14, 15].

5.1 Overview on the Snort Framework

Snort is a popular multilayer packet inspection system [16]. It is a network based intrusion detection system which is sniffing and processing the transferred packets. Since it is based on signature detection, it searches in the packets for known attack patterns. During packet processing all packets are examined by several independent component layers.

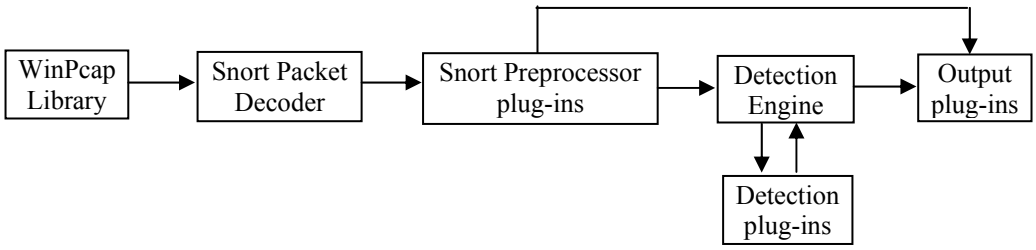

Figure 5.1: Schema of the Snort Framework. It shows the packet processing flow.

The component layers in the Snort Framework function as follows:

- Snort receives all the traffic by using the packet capture library. In Windows Systems this is done by the **WinPcap library** which is used to get access to Ethernet packets. First, Snort receives unprocessed packets from the Data Link Layer (OSI-Model [63]).
- Then, the **packet decoder** processes the header information. It will decode the data link frame, which can be for example 802.3 (Ethernet) or 802.11 (Wireless LAN). Thereafter, the IP protocol header and the TCP or UDP packet header will be decoded. Snort's decoder fills up internal data structures with the information provided by the packets and protocols. By this it enables the start of the inspection at the next level. The data structures are accessible from all higher layers and provide the needed information for further processing.
- A **preprocessor module** follows the packet decoder. It includes several plug-ins and operates on the data structures of the packet decoder. It implements various detection mechanisms which are inspecting the packets on this low level. In the preprocessor module also data transformations are performed which should simplify the further data inspection. Preprocessors can directly access the output modules and they are able to alert on suspect packets immediately. They also can classify the packets and possibly drop them if one of the preprocessors has encountered an attack.
- The next component is the more CPU-intensive **detection engine**. It executes the main pattern inspection techniques. Here the packets passed on from previous layers are now inspected at the transport and application layers, which are the layers 3 and 4 in the TCP/IP model [63]. The packet contents are examined by rule-based detection plug-ins. They use signatures of already known attacks.
- The **output module** is placed on top of all. It generates and logs an alert if one of the previous layers has encountered significant patterns in a packet. The detection engine and the preprocessor are directly connected with the output module. If a preprocessor or a rule of the detection engine responds on the contents of a packet then the output module receives a description of the incident. In the generated alert also some optional information about the detection activity can be provided.

The Snort framework has two positions for a plugin development: In the preprocessor and in the detection engine. For the decision where the computer immune system should be implemented a closer look on both layers is needed.

5.1.1 Snort's Detection Engine

Snort has a clever plugin architecture and therefore, it is more than a simple pattern matching system. The implemented rules ensures that it is flexible to configure and easy to keep it up to date. The detection engine analyses the contents of packets to **find patterns and match signatures**. These signatures are stored in rule files. Snort provides a flexible and robust system of predefined rules. At startup, it reads the attack signatures by processing the rule files line by line. The contents are parsed and loaded to an internal data structure. The detection engine checks traffic packets by comparing them with the attack signatures. Overall, it processes packets at a high level with a rule based system which makes it not to the right place to perform deep data analysis.

5.1.2 Snort's Preprocessor

In preprocessors special pattern detection mechanisms can be implemented and they will work as the network traffic flows by. For example one existing plugin analyses packets for the Back Orifice magic cookie. It uses a weak encryption and therefore, it cannot be detected by comparison with patterns. The used encryption can be broken with the brute force method by trying all possible keys. This example needs something different than a rule based detection engine. Here, the traffic will not be scanned for possible attack signatures. The purpose is to implement some deeper data analysis.
The preprocessor is the first module where a packet passes through the components of Snort. It removes packets from their way to the detection engine which needs much CPU time to verify all signatures. An advanced packet preprocessor detects inconsistencies or wrong header information in packets which is typically an indication for suspect behaviour.
In the OSI model the preprocessor is placed at the Data Link Layer. A big advantage of the preprocessor plugin layer is that it is possible to **analyse packets at the lowest level**. There, the implementation of other sophisticated techniques like the detection of anomalous protocol behaviour or other statistical related approaches will be possible. The extension of Snort with an anomaly detection preprocessor will make it capable for detecting malicious activities. The implementation of a preprocessor plugin can make use of some more advanced mechanisms than the port and pattern matching applied in the detection engine.

5.2 Architecture of the computer immune system

The Snort framework consists of several layers of processing routines installed on each network node. The focus lies on the **sensor** installed in the preprocessor plugin layer and the corresponding **monitor application**.

- The **sensor** processes all the network traffic of the network node. It represents the main logic of the computer immune system. The sensor incorporates all the mechanisms for training and reacting on various activities of the detector set.

- The **monitor** of the network node presents the status of the sensor to the user. It shows the ongoing activity in the detector set and provides a visual overview of all changes. The monitor also allows the user to react on a possible detected malicious code fragment in a packet. Furthermore, it is responsible for the communication with a central database.
- This **MYSQL-Database** collects the most successful detectors of the whole computer immune system. Further it receives statistical information about the status of each network node. The monitor sends successful detectors, trained by the local sensor, to the central database. It eventually receives new ones which were trained by other network nodes. The monitor passes such newly received detectors to the sensor for implementation.
- The database can also be read by **external frameworks** with a similar kind of environment. In this way successful detectors and incident reports can be provided to different work groups. The common data format for the exchange of information with the database is **IODEF** (Incident Object Description and Exchange Format), a XML based format.
- The statistical information in the database is used to update the **Web-Frontend**. There, the status of the computer immune system can be viewed.

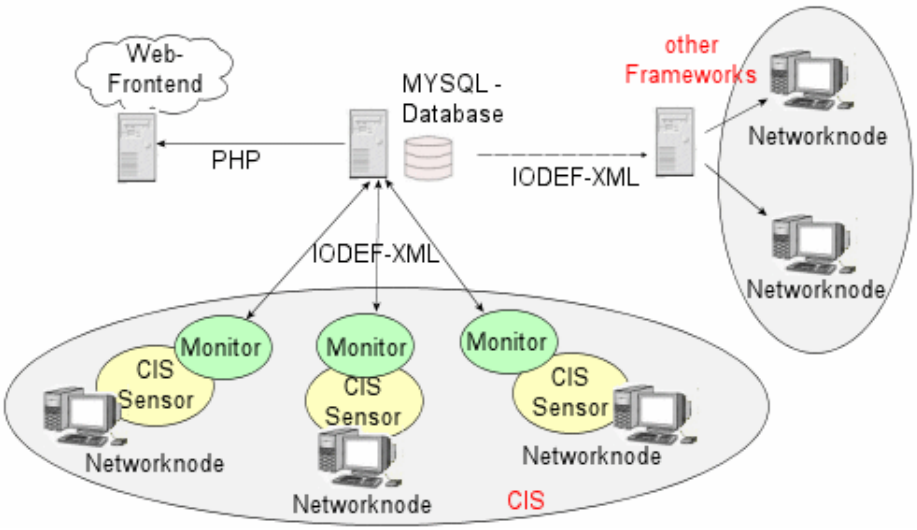

Figure 5.2: Architecture of the computer immune system (CIS). Each network node is equipped with a CIS sensor and a monitor application. The monitor applications are communicating with a database. The purpose is to update the statistics of the local training process and to upload successfully trained detectors, so that they can spread over the CIS.

5.3 Architecture differences to ARTIS and LISYS

The concept behind the implementation of this computer immune system is mainly based on ARTIS. Most mechanisms of ARTIS are implemented. This system has only some minor changes from the architecture, like a lower activation threshold, or the use of more match algorithms.
The LISYS (**L**ightweight **I**ntrusion detection **SYS**tem [1]) from the Computer Science Department of the University of New Mexico (UNM) is also based on ARTIS. How the proposed architecture in ARTIS is implemented depends on the particular problem domain. This system should find malicious code in the payload of network packets. The design of LISYS is

intended to find some port scan attacks. Beside the fact that LISYS was based on ARTIS it has major differences the system implemented in this thesis. This implementation needs the inspection of all TCP and UDP packets of the network traffic. In contrast, the design of LISYS needs only the inspection of packet headers. The packets which are intended for inspection are included in the TCP-handshake procedure which is needed for the connection establishment between two computers in a network.
A more detailed description of the main differences between both systems is listed below:

- **Packet payload processing:** A major difference is, that LISYS monitors only the start of a TCP connection. For this purpose, the connection information (Source IP, Destination IP and Port or Service) from the header of TCP/SYN packets are extracted. This information is then compressed to a 49-Bit string on which all match operations took place. In contrast to LISYS, the proposed approach in this thesis is not to inspect the headers, but the payloads.
- **Inspection of the whole network traffic:** In addition, LISYS processes only TCP packets which have set the SYN flag. In contrast the Snort framework allows a simple access to the network traffic through WinPcap. In this system not only TCP/SYN packet headers are watched, but also the whole network traffic. That includes all TCP and UDP packets.
- **No realtime packet inspection capability:** Assuming a 100Mbit network connection, then there will be a huge amount of data to process. LISYS was tested by UNM over a period of 50 days. In that time period about 2.3 million TCP/IP connections were detected. That is an average of 32 connection requests per minute. The point is that LISYS doesn't need to process much data. Because the immune system in this thesis will not focus on a particular part or type of the network protocols it will require much more computing time than LISYS. Therefore, it is not possible for the preprocessor to act in real-time on the packets that are captured.
- **Online prototype:** For LISYS only results of an offline prototype have been published by using idealized assumptions. The system described in this thesis was tested in an online environment.
- **Data representation at byte level:** Representing the payload at bit level and comparing it, by using a sliding window, with the detectors is too CPU-intensive. This approach also doesn't make sense when the payload is inspected. Applications are working with the payload data at byte level and therefore this will also be used in this thesis.
- **No restriction about the network topology:** The design of LISYS was based on a broadcast network topology, where all nodes in the network are confronted with the same network traffic. Therefore, also the detector sets of all nodes were trained on the same traffic. The underlying topology for the system presented here is not restricted to a broadcast network.
- **More autonomic network nodes:** LISYS was designed for a distributed environment, although that feature wasn't implemented. For example, mobile detectors or the exchange of successful detectors are not implemented. The system of this thesis should work in a more independent way with a higher number of detectors. It can be distributed over a large network without loosing autonomy of each individual system. An additional network node does not increase the communication overhead for the whole system. In the present work the number of network nodes is only limited by the maximum number of users allowed by the database.

- **Higher number of detectors:** LISYS uses 100 detectors on each node. The decision to use a higher number of detectors in this thesis is not only based on the non-broadcast environment. A reason is also the much greater spectrum of possibilities of the payload data compared to the 49 bit string of LISYS.
- **Five match algorithms:** ARTIS proposed the r-contiguous bits match rule and in LISYS the designers where using it with a window size of 12 bits. With the huge amount of data to process and the higher number of detectors it needs to be investigated which kind of match rule is the best approach. In this work five different match algorithms are implemented and compared in their results.
- **Different symbol representation of the detector:** Because of the byte level representation of the data it is also necessary to change the symbol representation of the detector. Instead of a bit string the detector holds a vector of bytes (or instructions).
- **Low activation threshold:** In LISYS, the activation threshold was initially set to 10 and was then adapted dynamically to the current traffic. For LISYS, the implementation of an activation threshold was useful. Because for detecting portscans, the extracted header information will remain largely the same and portscans are usually occurring within a short period of time. In this thesis the activation threshold is set to 1 which should ensure that the system is more sensitive to infrequent occurring suspect packets. By setting the activation threshold to 1 every match of a detector will lead to its activation. This is based on the observation that the same substring of a payload will occur very infrequent in time or even not at all. If they are appearing within a certain period of time its more up to coincidence and mostly in another context of data. Therefore, it makes no sense to wait for another occurrence by using a higher threshold. In contrast to that a port scan will occur as burst of related packets. A disadvantage of a threshold of 1 is that the chance for false positive increases.

5.4 The sensor of the computer immune system

For the construction of the sensor it was very important to consider the kind of information the set of detectors should be trained on. The detectors will face text based data as well as binary data in the network traffic. Two kinds of matching approaches are implemented: These are the group of **approximate string matching** algorithms and the **Pearson correlation coefficient**. In the approximate string matching algorithms the payloads of the packets can be seen as a chain of strings, where some of them are characteristic. The Pearson correlation is implemented by building the spectrum of the byte distribution of the data in the payload. Some analysis was done to show whether the spectrum of the payload will give a good discrimination of the data. This provides the basis for the training of the detector set.
For the training, **self** is defined as the contents of the payloads which are frequently observed on the LAN over a long period of time. **Nonself** on the other hand is then defined as the set of the contents of the payloads which are not frequently present in the traffic on the LAN.
Packets can be transferred between two internal computers on the LAN, or between an internal computer on the LAN and an external computer outside the LAN. Here we do not differentiate between the locations of computer systems or between different sources where packets can come from. This is because potentially malicious code can also be contained in packets from computers inside the LAN. An assumption in the design is that the **systems do not trust any other computer** that is transferring data to them.

Computers, which are part of the immune system do not transfer any data related to the immune system between them. The design of the immune system does not need a communication between computers in the LAN. The only transfer of data generated by the immune system is done through the monitor application which communicates with the central database. This is done in IODEF, a XML based format containing no binary data.
The computers on the LAN that maintain a sensor of the computer immune system have a maybe changing set of connections between them. It is not even necessary that a computer is connected to the LAN, when it is part of the computer immune system. A computer is representing an autonomous detection node.

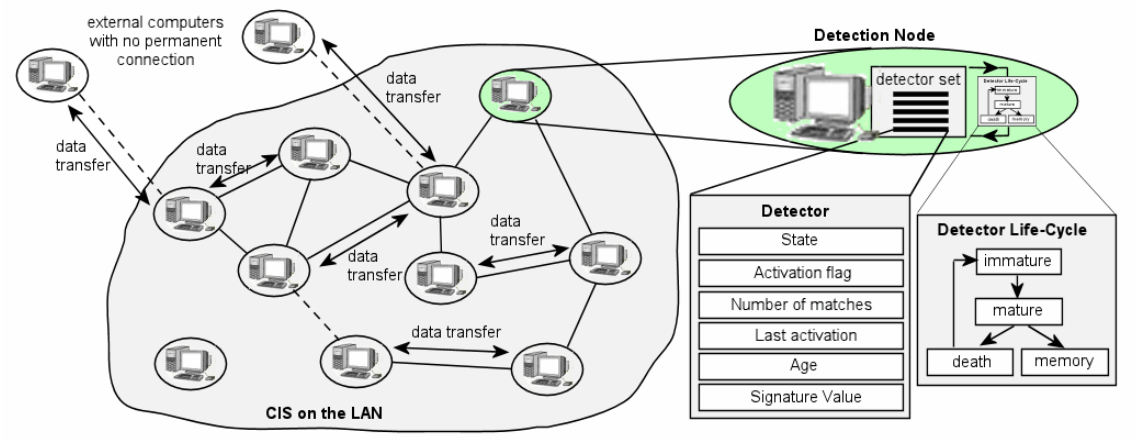

Figure 5.4: Example network constellation on which the computer immune system is applied. The computer immune system (CIS) is distributed over computers that are not necessary connected to the LAN. Each computer of the CIS represents a detection node which holds a set of detectors. The set of detectors is permanent changing with the life-cycle of the detectors. Each detector contains a randomly generated symbol sequence (Signature Value), and a specific information to the detector: Its state, which can be immature, mature or memory; an activation flag, which indicates the wait state for costimulation; the number of matches it has accumulated; the time of the last activation, indicating the timeout for the costimulation and finally the age of the detector.

5.4.1 Configuration of the preprocessor

The preprocessor maintains two files for its configuration. The first file provides the parameters for adjusting the preprocessor. The second file saves the actual training status of the detector set.

5.4.1.1 The preprocessor configuration file

Preprocessors are turned on through Snort's main configuration file snort.conf. When the computer immune system is activated, following options can be set by the user in the Snort configuration file:

activation threshold: how often a detector must match with packets of the network traffic before it gets activated. Higher values will reduce false positives, but on the other hand suspect packets will be missed. Therefore, the threshold is set by default

to the value 1. The user should pay attention to the consequences, if it is changed from default.

tolerance period: the length of the training period of the detector set. It is most important for reducing false positives. The detectors set should learn to adapt to packets being transferred over the network which are assumed to represent normal traffic. Detectors which are matching anything during this period are replaced, ensuring the tolerance of the detector set to normal traffic. A higher value for the tolerance period results in a lower chance in receiving false positives. But as the tolerance period increases also possible infrequent nonself packets are tolerated. Furthermore, they will not be detected from mature detectors released from this longer tolerance period. The longer the period the higher the chance for creating detection holes in the system and maybe it fails in detecting anomalies. Detectors in the tolerance period have an immature status. Detectors released from the tolerance period have mature status. The tolerance period is set by default to 4 days.

life expectancy: the time span a detector is given to prove its success. If the detector was not activated during this time it is assumed that it is less useful and needs to be replaced. This parameter ensures a dynamical changing detection environment and possible detection deficits will not be permanent. Detectors exceeding the life expectancy are replaced by new randomly generated ones. A finite lifetime provides a detector set which can adapt dynamically to a slowly changing self set. It is important to define the right proportions between the lifetime and the tolerance period. Otherwise maybe the computer immune system faces a high number of false positives or maybe even the loss of the detection capability. The default value for the maximum age of the detectors is set to 14 days.

match decay period: this parameter is only valid when an activation threshold greater than the default value of 1 is used. It reduces the value of accumulated matches over time.

sensitivity decay period: this parameter is only valid for activation thresholds greater than the default value of 1. The sensitivity level reduces the activation threshold. It is decremented again within the time period given by this parameter. The activation threshold is therefore an adaptive threshold, which enables the preprocessor to adjust the sensitivity of the detector set in a dynamic way.

costimulation delay: gives the time that the user has to decide when a detector has been activated if indeed nonself has been detected. If no action is taken the detector is replaced by a new random one after the delay. The default value for the delay is 24 hours.

number detectors: can be adjusted to set the optimal number of detectors.

5.4.1.2 The preprocessor data file

The preprocessor saves the learned values to a data file periodically. When the preprocessor is initialized it accesses this file and reads the last saved values to reinstall the states of the data structures. The file holds the following values:

a) The general configuration of the detector set

Detector number: the count of the local detector set, for example 1500 detectors at this node.

Detector length: sets the length of the byte vector, for example 30.

b) Some statistic values which document the learning progress of the detector set

Tolerance replacement: the total count of the detectors which were replaced during the tolerance period since the preprozessor was initialized the first time.

Activation replacement: the total count of the detectors which were replaced due to lack of receiving the costimulation signal.

Lifetime replacement: the total count of the detectors which were replaced because of their age.

Engine running time: the total time the engine was running.

Number of processed packets: the total number of packets which were processed by the preprocessor so far.

Number of rejected packets: the total number of packets which were filtered out of the network traffic. The preprocessor discards packages with presumably clean content like text-based contents, such as html files. This is done to reduce computing load on the detectors. This number is only set if the packet filters are activated.

c) Values which are stored separately for each detector of the set

Detector state: immature, mature, memory

Detector activated: Boolean value.

Number of matches: the number of the accumulated matches of the detector.

Last activation: holds the relative time when the detector was last activated. This value is zero when the detector was never activated, but also when the costimulation signal was received. The relative time is measured in terms of the running time of the engine since the detector was activated.

Time created: the relative time which gives the age of the detector.

Value of the detector: it is given by the array of symbols (e.g. a character sequence) which are random created sequences with the length given by the parameter "Detector length".

5.4.2 The logfile of the preprocessor

The preprocessor periodically writes the progress in teaching the detector population to a logfile. It documents all important changes of the detectors and useful additional information like statistics. The logfile is intended for the use by the monitor application of the computer immune system. The contents of the logfile and their effect on the monitor application are described in more detail in section 5.5.

5.4.3 Applying the string match algorithms

After a detector is released from the tolerance period it is treated as mature detector. When it becomes activated and receives the costimulation signal it will be a memory detector. These successful detectors have the feature that they can detect structurally related nonself strings. This feature is supported by all match algorithms.

- The **Pearson correlation** enables the detectors to respond to payloads with related spectrum. Therefore, this algorithm gives for each detector a broad bandwidth of possible detections. By this it is a very general approach. For better discrimination the value of the detector must be of sufficient length and/or the probability of the significance level must be high.
- The **Hamming match rule** is more general than the **r-contiguous byte match rule**. Hamming can detect the longest contiguous sequence plus some possible matching bytes in the surrounding area. For the Hamming match rule contiguous sequences are not required. The match value received from the Hamming rule is at least equal when compared to the r-contiguous byte match rule.
- **Levenshtein distance** and its special case the **Longest Common Subsequence** include operations which alter the size of the strings. For example, the Longest Common Subsequence algorithm searches for the occurrence of a string sequence all over the other string and accepts gaps within the strings. This property makes it possible to search in the payload also for mutated code segments. Almost all of today's successful threats and attacks use some polymorphism [26].

The ranking of the algorithms concerning their specificity (from specific to general) is as follows:

1. r-contiguous bytes,
2. Hamming Distance,
3. Levenshtein distance or edit distance,
4. Longest Common Subsequence,
5. Pearson correlation.

Now following is a detailed description of the match algorithm implementation.

5.4.3.1 Overview about the algorithm implementation

The calculation of the match value by using the Hamming Distance, the r-contiguous symbols and the Levenshtein distance or edit distance string matching algorithm requires equal size of the compared strings. Since the payload is of variable length and the detectors contain strings

with fixed length, the implementation of the algorithms has to take care of that fact. All three implementations therefore use sliding windows with the size of the detectors string length. The window is slided from the beginning of the payload until its end, skipping one byte of the payload each time. The Longest Common Subsequence match algorithm does not need to have strings of equal length. But it turned out that a learning effect in the detector set can only be achieved if also a sliding window is applied.

5.4.3.2 Implementation of the Hamming Distance match algorithm

For Hamming Distance and the r-contiguous symbols algorithms the matching is performed as a one dimensional process. This also can be seen in the code example for both algorithms (see chapter 4). But there is a difference in the alignment of the sliding window. The payload string is aligned in horizontal direction; let's say it is beginning from left with byte zero and ending on the right side with the last byte. Then for the Hamming match rule the alignment of the sliding window is also in horizontal direction.

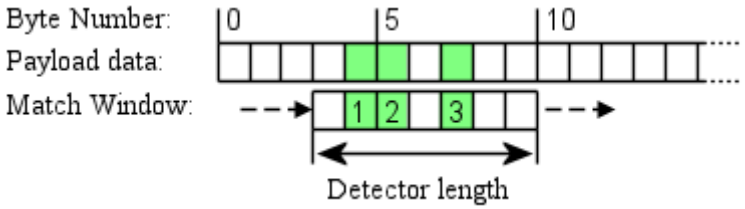

Figure 5.4.3.2: Alignment of the sliding window for the Hamming match rule.

5.4.3.3 Implementation of the r-contiguous symbols match algorithm

The window alignment for the r-contiguous symbols match rule is vertical to the payload data. All symbols of the detector string are compared at once with one symbol of the payload until the end of the payload. This constellation results in a two dimensional matrix of match values.

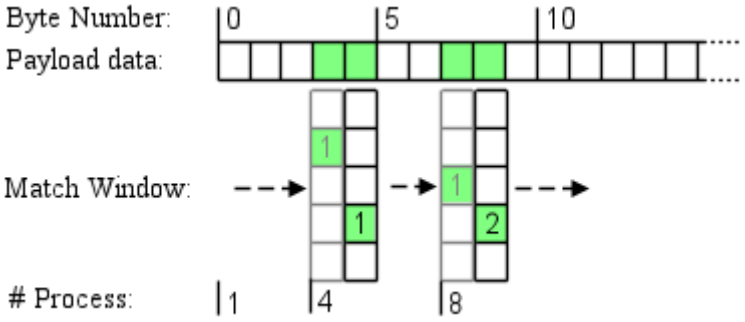

Figure 5.4.3.3: Alignment of the sliding window for the r-contiguous symbols match rule. In the two dimensional matrix all values are initially set to zero. If a match occurs then the value on the upper-left side is incremented. The symbol in the upper-left position must have matched before to get a greater match value for a contiguous symbol sequence. # Process indicates the number of the current match.

5.4.3.4 Implementation of the Levenshtein Distance match algorithm

The implementation of the Levenshtein distance was more complicated. The window alignment here is also vertical, but the implementation is only useful for comparing strings of equal (or almost equal) length in horizontal direction. The calculation needs a two dimensional matrix for each match process. This requires a two dimensional sliding window with the dimension of the detector length in both directions. The window was slided in steps over the payload data. The step size was 1 or 2 bytes, depending on the match value of the previous window. If the distance had a decreasing tendency the step was set to 1 otherwise the step was 2. The purpose of skipping most of the time two bytes was to speed up the match process. But the sliding window was slowed down in positions of the payload with a potential higher match value. The upper asymptotic running time of the implemented algorithm was not as proposed $\mathcal{O}(nm)$, but approximately $\mathcal{O}(n^2m)$. Where n is the window size and m is the length of the payload.

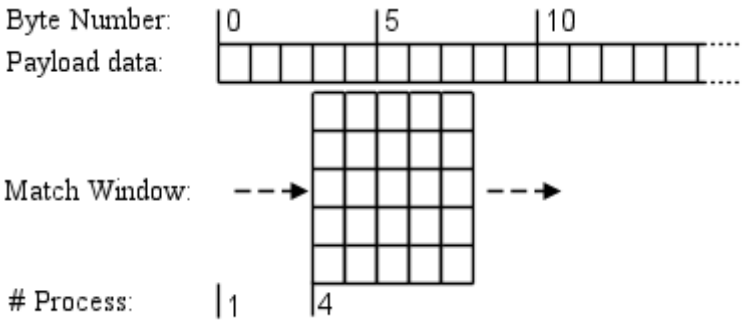

Figure 5.4.3.4: The two dimensional sliding window for the Levenshtein distance or edit distance.

5.4.3.5 Implementation of the Longest Common Subsequence match algorithm

The Longest common subsequence (LCS) algorithm is also applied to the payload with a sliding window. Without a window, longer payloads produce significantly longer common subsequences with the detector string than shorter payloads. It has been verified that LCS with no window will have no learning effect in the detector population. A horizontal window of size 300 is slided over the payload with a step size of 50 bytes. That ensures the detection of those LCSs which are appearing within a smaller range. The window alignment is similar to the r-contiguous symbols match rule. All symbols of the detector string are compared at once with one symbol of the payload until the end of the horizontal window in the payload. The first match run goes from byte 0 to byte 299 in the payload, the next match process is going from byte 50 to byte 349 in the payload, and so forth. The size of the two dimensional matrix is in vertical direction the length of the detector and in horizontal direction it is 300.

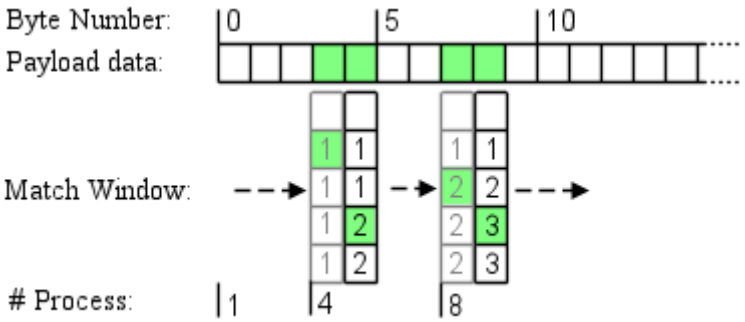

Figure 5.4.3.5: Alignment of the sliding window for the longest common subsequence match rule. It is calculated similar to the r-contiguous symbols match rule, but in here not only the upper-left position is of interest. In addition also if no match occurs in the process before then two further positions are checked. These additional positions are in the matrix the upper position and the left position. The highest value then is filled in the actual position of the matrix. This procedure results in the longest contiguous sequence with allowing gaps within the sequence.

5.4.4 Using the Preprocessing Engine for payload processing

When a packet is captured from the Pcap Library, Snort fills up an internal data structure with the packet. This data structure is passed to the preprocessor, so all parts of the packet can be accessed easily. Primary of interest is the payload data. If there is no significant difference between the detector and the payload data then some action is taken. In the tolerance phase of the detector (immature state) this will be the replacement with a new random detector. In the mature or memory phase it will be the raise of a costimulation request. This chapter presents the results in the training of the detector set for the different match algorithms described above.

5.4.4.1 Simulated network traffic

The network traffic for training the engine was simulated. This allows the comparison of all match algorithms under the same traffic and the same conditions. The network traffic simulated internet file transfers. The TestData set contained two main file types:

	File count	Space amount
Binary files	2937	419,7 MB
Text files	7173	91,8 MB

Table 5.4.4.1: The composition of the TestData set

The binary file type contains executable files (exe, dll, ocx), images (gif, jpg), archives (zip, tar, gz, cab, rar), document formats (doc, xls, pdf, chm) and a few music files (mp3). The text file type contains source code (c, cpp, h,…), hyper-text files (htm, html), scripts, text files and a few base64 coded files (simulating email attachments).
The transfer of all files in the TestData set was repeated without break after the last file was sent. In that way a continuous transfer rate over the network was ensured.

5.4.4.2 Measuring the success of a learning process

The learning progress is measured by two performance values. The first is the number of **detectors that are replaced** in the tolerance period (will be plotted in the chart "Replacements"). The training will be successful when the turnover of detectors is decreasing over time. As time advances, more and more detectors should survive the replacement in the tolerance period. Then the complete detector set is assumed to show a better tolerance to the presented network traffic.
The second performance value is the **age of the oldest detector** in the detector set (will be plotted in the chart "Oldest Detector"). In a successful training some of the detectors must have become resistant to replacement. While the captured packets are processed they are getting older and older. The oldest detectors are assumed to have the greatest tolerance to the presented network traffic. The lifetime of the oldest detector was chosen as a representative performance value.

5.4.4.3 Overview of engine testing

The algorithms are tested whether they have a significant learning effect, which should be seen after a few repeated transfers of the TestData set. Some of the detectors, a fraction not too big, should then be resistant to the continued replacement. That means some detectors will be equipped with a character sequence which is not included in the TestData set. They will be tolerant to the self set (which was presented as the TestData set).
Two different approaches are chosen to train the detectors set. In both of them the minimum payload size for a packet that is intended for processing was set to 60 bytes. This limit will filter out packets involved in the TCP-Handshake procedure, packets with a payload size of zero bytes and other small payloads. It was assumed that packet payloads containing malicious code will have a length of at least 60 bytes.
The first measurement had the purpose to determine Snort's idle packet processing rate. In this measure the implemented algorithms were deactivated. The TestData set was transferred over the network and the number of packets which Snort has received was recorded. With this value the packet drop rate could be calculated in the following runs. When the algorithms were activated in the preprocessor engine Snort was slowed down due to the processing of the packets and dropped a percentage of the packets. The fact that Snort dropped a fraction was not a problem, because the data being transferred was repeating. Because of a rearrangement in the sequence of files Snort was not dropping the same packets on each run. Therefore, it was ensured that the training set was a bit different each time.
The implemented algorithms are applied with different parameter configurations. In a configuration three parameters can be adjusted:

1. the size of the detector set (**detector number**),
2. the **detector length** and
3. the **significance level** (threshold for a match).

Now following both approaches and their results are described in more detail.

5.4.4.3.1 The first approach, without the use of packet filters

In the first approach all transferred packets are passed to the preprocessor engine. It was necessary to limit the speed of the 100Mbit network connection, because the preprocessor could not process the data in realtime. Thus, the speed of the network connection was controlled and set to an average value of 94.4 kB per second. Even in a slowed down file transfer Snort did drop packets, because it could not catch up with processing speed. The percentage depended largely on the used match algorithm.
Many configurations were resulting in a not successful training. The following table gives an overview of the successful configurations.

Algorithm	**# detectors**	**Detector length**	**Significance level**	**Drop rate**
Pearson correlation	3000	220	0.99995	35.4%
r-contiguous symbols	600	8	3 bytes	51.8%

Table 5.4.4.3.1: This table shows the successful configurations of the implemented algorithms in the first approach. Only three of 12 configurations lead to a successful training of the detector set. The third successful training was a slightly changed configuration of the r-contiguous symbols match rule and is therefore not listed in the table.

5.4.4.3.2 The second approach, with the use of packet filters

In the second approach the preprocessor engine was implemented with a packet filter. Overall we have an enormous number of packets that are subject to inspection. It is in the interest of the implementation to use a mechanism for the reduction of the processed data. A huge amount of packets has to be monitored. The algorithm candidates will be tested if they can discriminate between text based packets and binary data packets. Packets with text based content are common in the transfer of html web pages and packets that contain binary data are for example applications. By filtering out text based packets this approach has the advantage of reducing the high data load. When Snort faces a high data load the queue of buffered packets is reduced due to fastly dropping text based contents. According to this the packet drop rate caused by slow packet processing is lower. In addition also the network speed was slowed down even more. This time the speed was set to an average of 40 kB per second which should assist the filters in reducing the packet drop rate. A high portion of packets that were not of interest were rejected. Due to the fact that the training of the detectors is based on a data selection which is possibly more dangerous the training of the detector set will be more efficient.
For implementing the filters some file formats have been analysed. As it can be seen in Appendix A the average spectrum is definitely different between text based file formats, such as html pages, and binary file formats including executables. It is easy to distinguish between payload data containing text and containing binary data. But it is very difficult to distinguish between binary data coming from data format files, like pictures, and executable files.
It turned out that the Pearson correlation coefficient gives a very good discrimination between text based file formats and binary file formats. The correlation coefficient between the spectrum of an incoming packet and the text reference spectrum is highly significant for packets containing text based data. So a major part of the packets which are not of interest can be rejected with the use of a Pearson text filter.
A second filter was implemented with the purpose to reject binary packets with a definitely wrong spectrum. When the traffic was analyzed it turned out those packets containing no file

data fall in this class. These packets have a short payload and are occurring frequently. This filter was implemented in a very conservative way to make sure that no packets of interest are missed. The spectrum of this filter is also presented in Appendix A.
The second approach had a better performance in comparison with the first approach, which was mainly based on the knowledge of not successful configurations.

Algorithm	**# detectors**	**Detector length**	**Significance level**	**Drop rate**
Pearson correlation	3000	220	0.99995	2.1%
Hamming Distance	500	10	4 bytes	19.4%
r-contiguous symbols	250	16	4 bytes	32.4%

Table 5.4.4.3.2: This table shows only the successful configurations of the implemented algorithms in the second approach. There are already three of six configurations which lead to a successful training of the detector set.

5.4.4.3.3 Factors that influence the training

In the second approach the packet drop rate was analysed for a significant effect in influencing the training of the detector set. Therefore, the three successful configurations (see table above) were also chosen for a second training. The training conditions of the first approach were set up again. The filters were turned off and the data was transferred with the faster network speed. It turned out that the training took longer when the drop rate is higher. This was resulting from the presentation of only a fraction of the self set in one run. The size of the fraction depends on the packet drop rate. Because this fraction has in each run different contents it is a very realistic approximation to the real world demands. The training was finished with the same result. There is no significant negative effect caused by different packet drop rates (with the respect to the longer training time). Differences in the training success are only caused by the combination of two parameters. The one is the **number of symbols** with which the detector is equipped and the other is the chosen **significance level** for a match. The number of detectors is responsible for the coverage of patterns belonging to the nonself set and is not influencing the success of the training.

5.4.4.4 Results of the first approach (without filters)

The results are presented in a graphical form, the numerical data can be found in Appendix B.1.1.

5.4.4.4.1 The Pearson correlation coefficient

The first algorithm implemented was the Pearson correlation coefficient. A byte spectrum was generated from the payload of a packet. The number of occurrences of each byte was stored in a 256 byte vector as a frequency distribution. To be able to compare it with a reference it is necessary to divide the vector by the length of the payload. After this normalisation the vector is correlated with each detector. Three different configurations have been tested to determine the difference in the learning process in response to changing parameters. The number of detectors was 5000 in the first configuration and 3000 in both the second and in the third configuration. Each detector was holding 30, 90 or 220 values out of the 256 possible byte values of the

spectrum. The values represent, like the values in the data vectors, relative occurrence frequencies which have been normalized to the packet length.
After a Pearson correlation coefficient is calculated it will be tested if it is significant. For this, three tables with cutoff-scores were added using four different statistical significance levels: 0.95, 0.975, 0.990 and 0.99995.
During the training of the detector set a significance level of 0.975 was used in the first two configurations and 0.99995 in the third configuration. That means that during the tolerance period all detectors were replaced, which correlate to an occurring data vector with a significance level of 0.975 or 0.99995, respectively.

The results of the implemented Pearson correlation coefficient are presented in figure 5.4.4.4.1a and figure 5.4.4.4.1b. The replacement rate in the first and second configuration was much too high. Furthermore, the age of the oldest detector stays pretty much the same. Not one detector from the set survived a run. Both performance indicators do not show any tendency in the direction towards a useful detector. This leads to the decision that the training will not be successful within a reasonable time period. The training was interrupted after two respectively three runs. The third configuration was chosen properly. Both performance values indicate a successful configuration. The number of replaced detectors is dropping with each run and also the age of the oldest detector has a higher value after each run. This configuration had also the lowest packet drop rate.

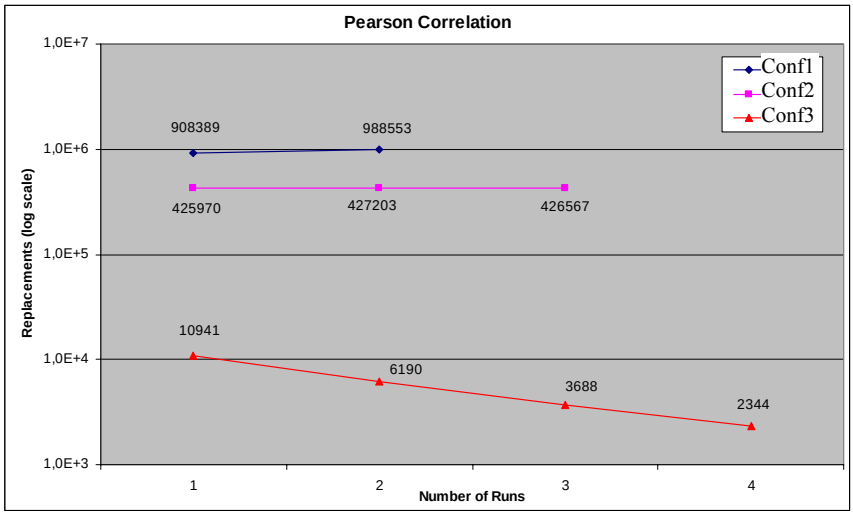

Figure 5.4.4.4.1a: The Replacement rate of detectors using the Pearson correlation in three different configurations.

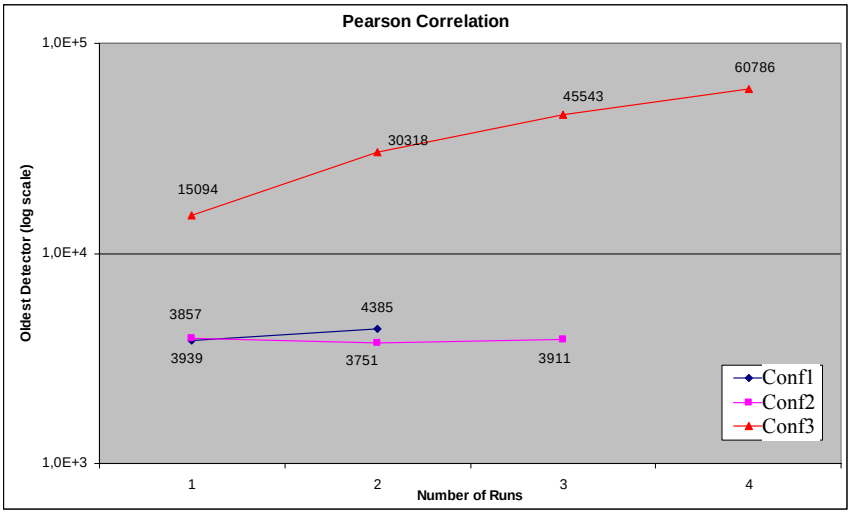

Figure 5.4.4.4.1b: The age of the oldest detector when using the Pearson correlation at three different configurations.

5.4.4.4.2 Hamming Distance

The Hamming distance match algorithm was slightly modified, because it should return the similarity value of two sequences and not their distance. With this it was easier to compare the results with the other algorithms.
The Hamming distance was also implemented in three different configurations. The size of the detector set was 600 in the first configuration and 300 in both the second and the third configuration. Each detector was equipped with 8 bytes in the first configuration, with 20 bytes in the second and 18 bytes in the third one. The threshold for a match was set to 3 bytes in the first configuration and 4 in the second and the third one.
The results of the implemented Hamming Distance algorithm are presented in figure 5.4.4.4.2a and figure 5.4.4.4.2b. The tendencies of the replacement rate of all three configurations indicate not successful runs. The rate stays approximately the same over a long time. The first configuration was the fastest, because of a much lower detector length. For the second and the third configuration the length was increased because of the higher match threshold. This was necessary to ensure that a fraction of the detectors are still replaced. Otherwise a detector length of 8 in combination with a match threshold of 4 will lead to a replacement rate near zero. Then the detector set will not be trained at all. For the second and the third configuration it was initially difficult to see if the training will be successful in further runs. This was because the age of the oldest detector in the set was increasing. After the sixth run in both configurations it turned out that the training was instable and not one detector was immune to the continually replacement. It was observed that a detector length of 16 will lead to a more stable training, but at the same time also the replacement rate was shrinking. This result is not published here because the impression of a “stable training” was caused more by the low replacement rate than by a tolerant detector set.

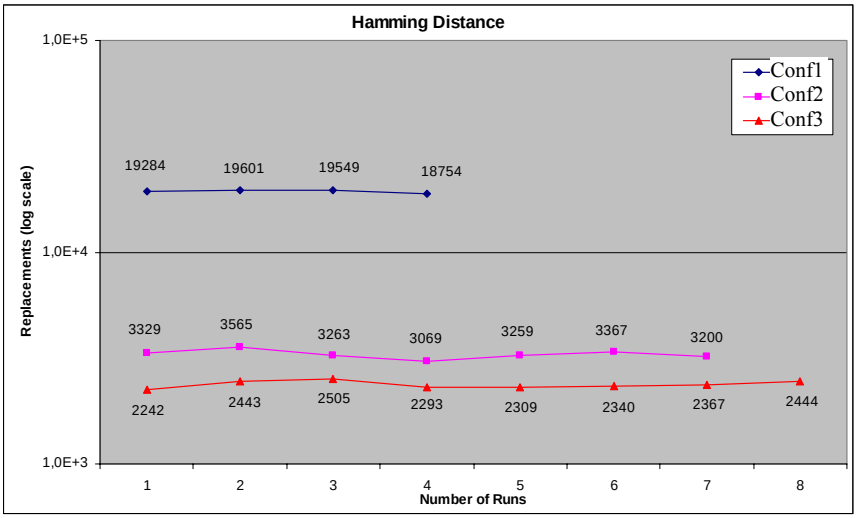

Figure 5.4.4.4.2a: The Replacement rate of detectors using the Hamming distance in three different configurations.

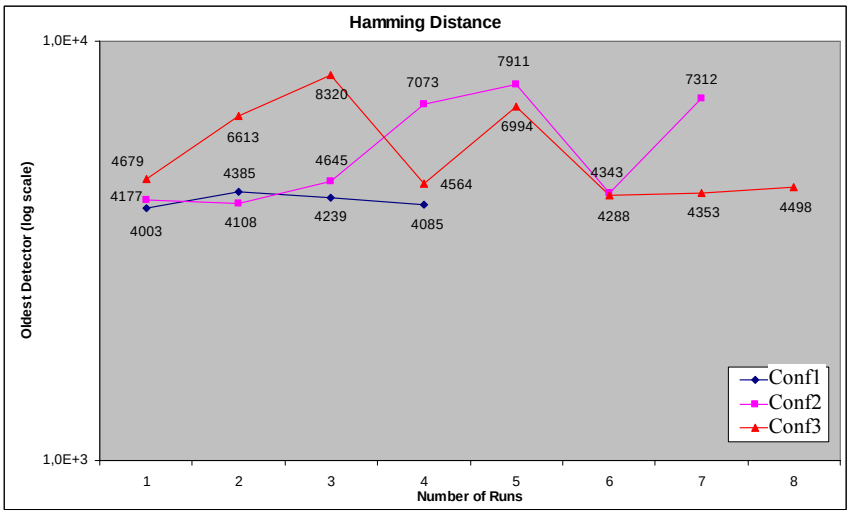

Figure 5.4.4.4.2b: The age of the oldest detector when using the Hamming distance at three different configurations.

5.4.4.4.3 r-contiguous bytes

This algorithm was tested in two configurations and both appeared to be successful. The only difference was the size of the detector set. In the first configuration 600 detectors are trained and in the second 800 detectors. Each detector was equipped with 8 bytes and they were exposed to the traffic over 10 runs. The threshold for a match was set to three contiguous bytes. Both sets resulted in very similar training values.
This training shows two important facts:

- The training success of algorithms is independent from the number of used detectors.
- The training success of algorithms is independent form their packet drop rate (see Appendix B.1.1.3).

The replacement rate was decreasing in each further run and at the same time the age of the oldest detector was increasing. Both are indicating that some of the detectors become immune to

replacement and are therefore tolerant to the presented TestData traffic. In figure 5.4.4.4.3a and figure 5.4.4.4.3b the results of the implemented r-contiguous bytes algorithm can be seen.

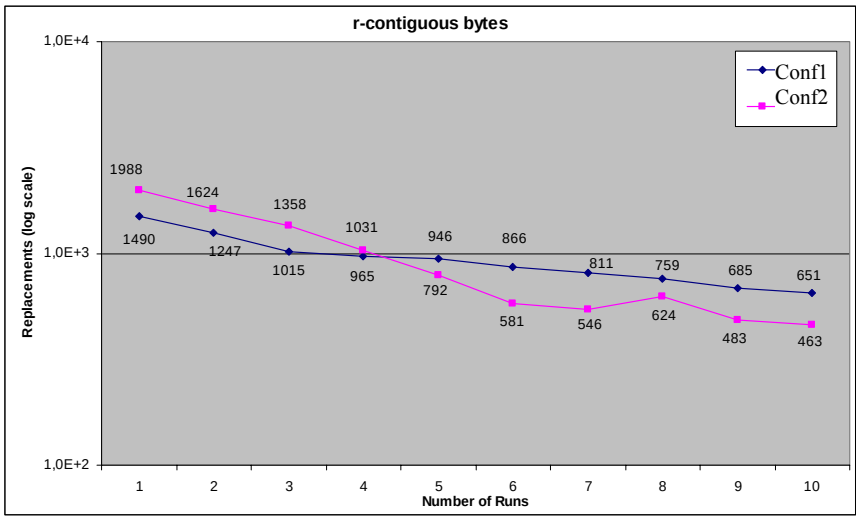

Figure 5.4.4.4.3a: The Replacement rate of detectors using the r-contiguous bytes match rule.

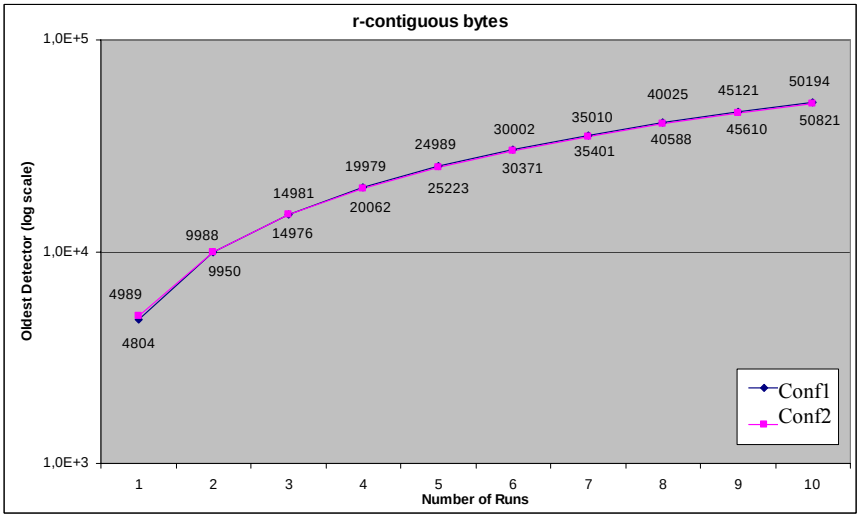

Figure 5.4.4.4.3b: The age of the oldest detector when using the r-contiguous bytes match rule.

5.4.4.4.4 Levenshtein distance or edit distance

This algorithm was tested with only one configuration. The size of the detector set was 100, each equipped with 8 bytes. This algorithm is measuring the distance between two sequences, but it was modified like the Hamming distance to return the similarity. The threshold for a match was set to three equal bytes.

The algorithm had a low packet processing rate, because of the need for two dimensional matching windows. Snort's packet drop rate lies in each run on the average at 84%, although a low number of detectors have been chosen. A high drop rate is responsible for a slow adaptation to the TestData set. But beside this low processing speed the training seems to be not successful.

The Levenshtein distance or edit distance was tested in only one configuration. The replacement rate of the detectors in the set stayed almost the same over all runs (figure 5.4.4.4.4a) and also not one detector survived a run (figure 5.4.4.4.4b). The result of this configuration shows the training of detectors which were not specific enough to adapt to the presented TestData set.

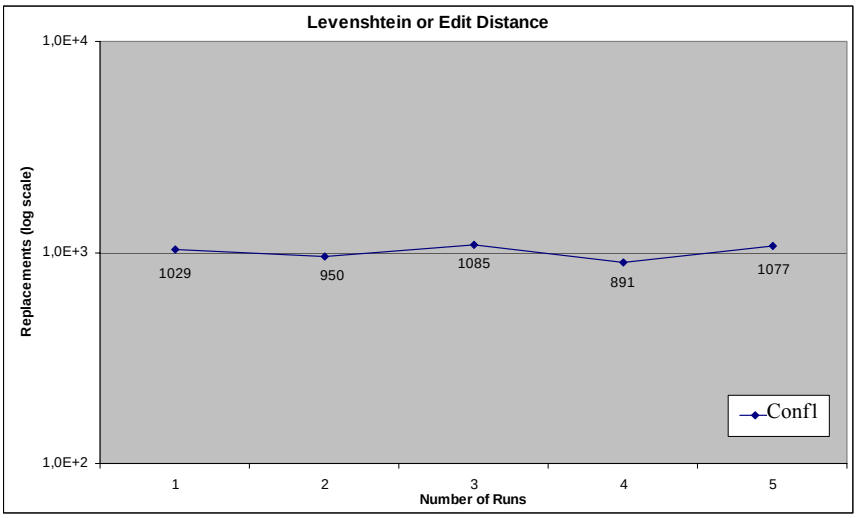

Figure 5.4.4.4.4a: The Replacement rate of detectors using the Levenshtein or Edit distance.

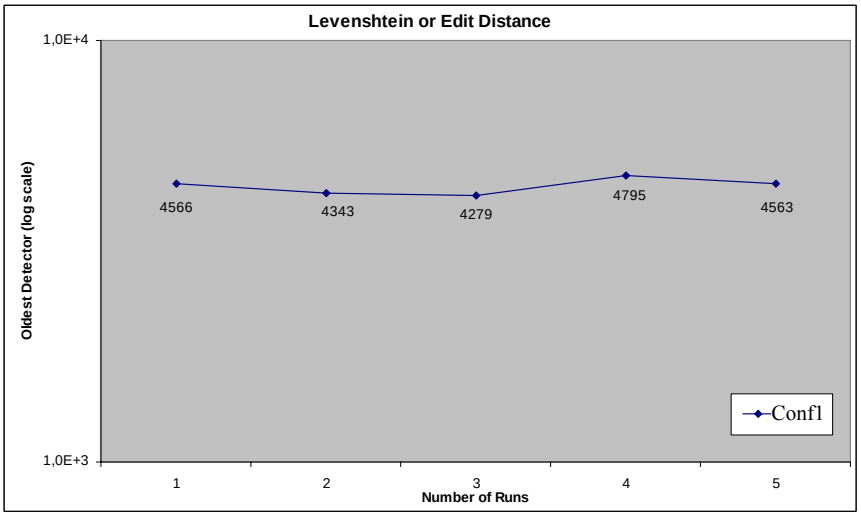

Figure 5.4.4.4.4b: The age of the oldest detector when using the Levenshtein or Edit distance.

5.4.4.4.5 Longest common subsequence (LCS)

The LCS was tested in two different configurations. The first was using 130 detectors equipped with 18 bytes. The second was using 100 detectors of 17 bytes length. In both configurations the threshold for a match was set to 11 bytes. This algorithm had a poor packet processing speed with a packet drop rate near 90% so it was necessary to watch the training in more runs. The first configuration appeared to result in a detector set which is getting older each further run. Although the age of the detectors stays between the third and the fifth run almost the same. Also the replacement rate remains at the same level. The second configuration was applied with a slightly lower number of detectors, which therefore had a higher packet processing rate. The lowered length of the detectors was intended to get a maybe clearer tendency in the training result. The detectors are again getting older and this time until the seventh run in monotone upward tendency. But the eighth run in combination with the low replacement rate of the detectors lead to the conclusion that this partial success is more up the very low packet processing rate than to a real success of the training. This training was done before the implementation of the Levenshtein algorithm.

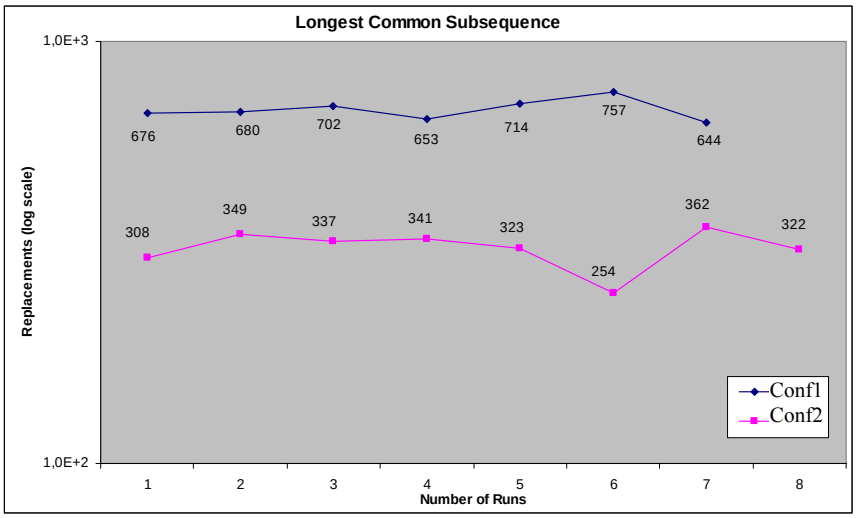

Figure 5.4.4.4.5a: The Replacement rate of detectors using the Longest Common Subsequence in two different configurations.

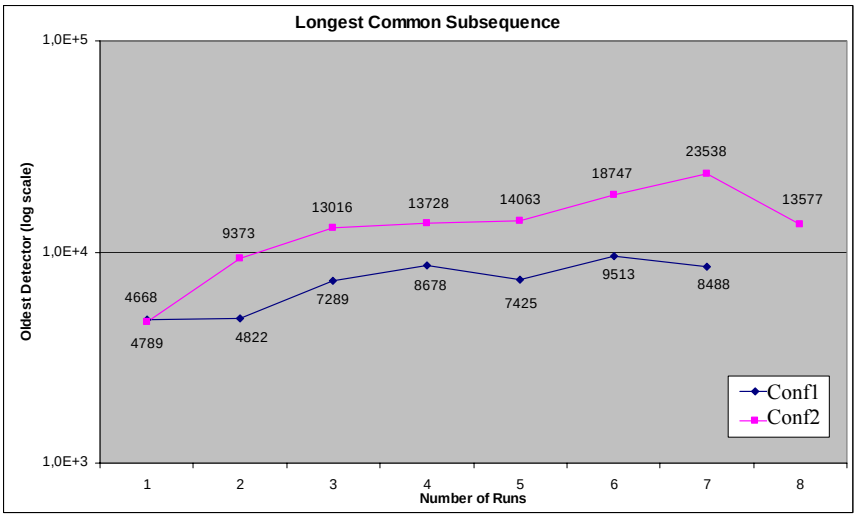

Figure 5.4.4.4.5b: The age of the oldest detector when using the Longest Common Subsequence in two different configurations.

5.4.4.5 Results of the second approach (using filters)

In the second approach four algorithms were tested if they are able to train the detector set when the engine faces a higher load of binary packets. The high rate of binary packets was achieved through the activation of the two implemented packet filters which drop all text data. Now the engine only processes packets which potentially contain malicious code. In comparison to the first approach a slowed down network speed had to be used. This was up to the low performance of some algorithms. Due to the lower network speed also a lower packet drop rate was achieved. All measured data are listed in Appendix B.1.2.

5.4.4.5.1 The Pearson correlation coefficient

The successful Pearson correlation of the third configuration in the first approach was reused to see how it is performing at high loads of binary packets. The algorithm trained the detector set successfully. The values for the replacement rate as well as those for the oldest detector were almost the same as they have been before. This algorithm is able to adapt to different data constellations and is able to ensure that the detector set is after some time tolerant to the presented network traffic.

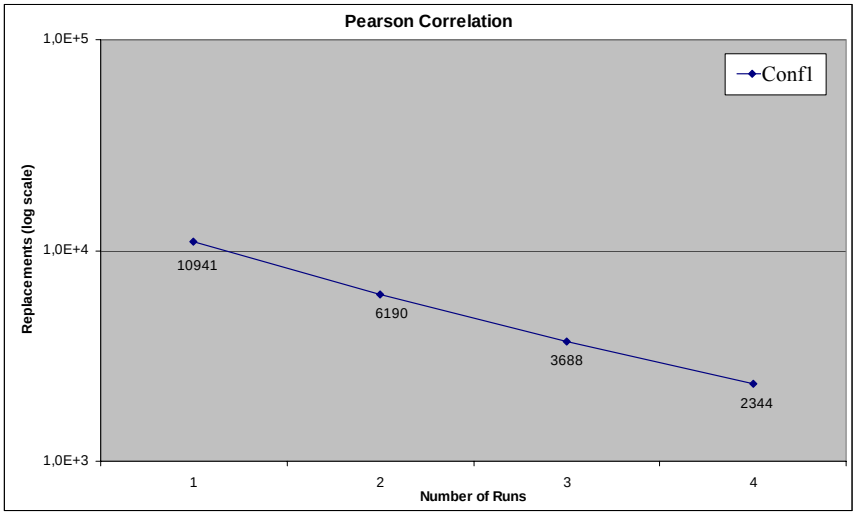

Figure 5.4.4.5.1a: The Replacement rate when using the Pearson correlation in the second approach.

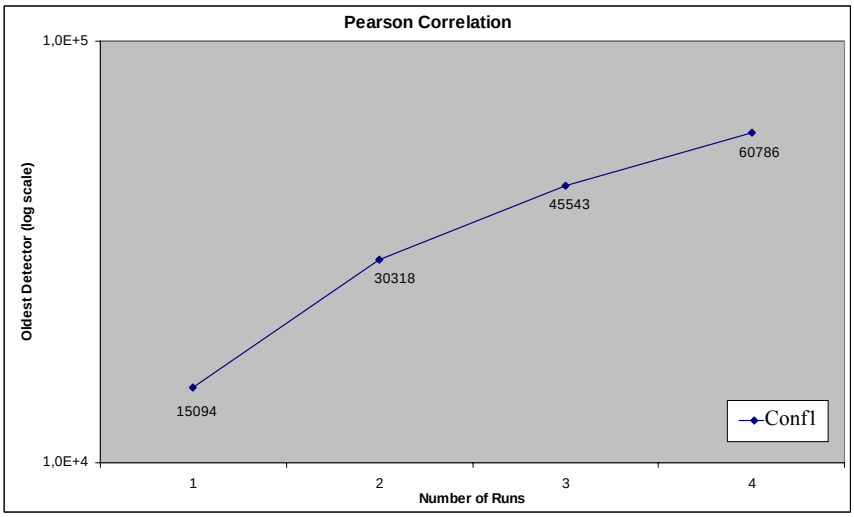

Figure 5.4.4.5.1b: The age of the oldest detector when using the Pearson correlation in the second approach.

5.4.4.5.2 Hamming Distance

The training in the first approach was resulting in an instable detector set. Due to the lower network speed a higher packet processing rate was achieved. Therefore, the number of detectors was increased to 500, each equipped with 10 bytes. The threshold of a match was again set to 4 bytes. This training of the detector set will be successful in further runs, although the training

process is very slowly. Figure 5.4.4.5.2a shows the replacement rate which is slightly decreasing towards the seventh run. After the seventh run the detector set contained 35 detectors out of the 500 which are with a high probability tolerant to the presented self set. Their age was continually rising between the fourth and the seventh run.
This training configuration was something special. In all the other ones the replacement rate decreases fast for successful runs, resulting in a fast growing population of detectors which are tolerant to the presented network traffic. This training configuration is also working. The algorithm had a high packet processing rate. But in contrast to the other successful trainings it takes much more time to adapt to the presented TestData set.

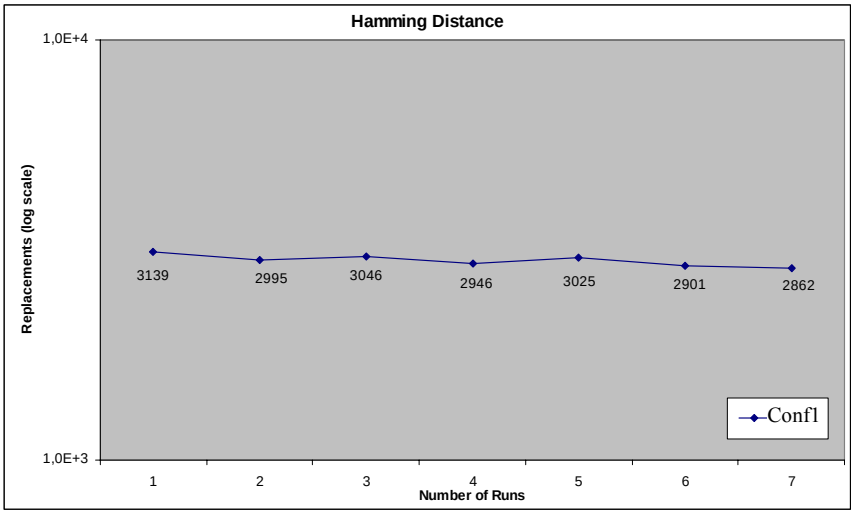

Figure 5.4.4.5.2a: The Replacement rate when using the Hamming Distance in the second approach.

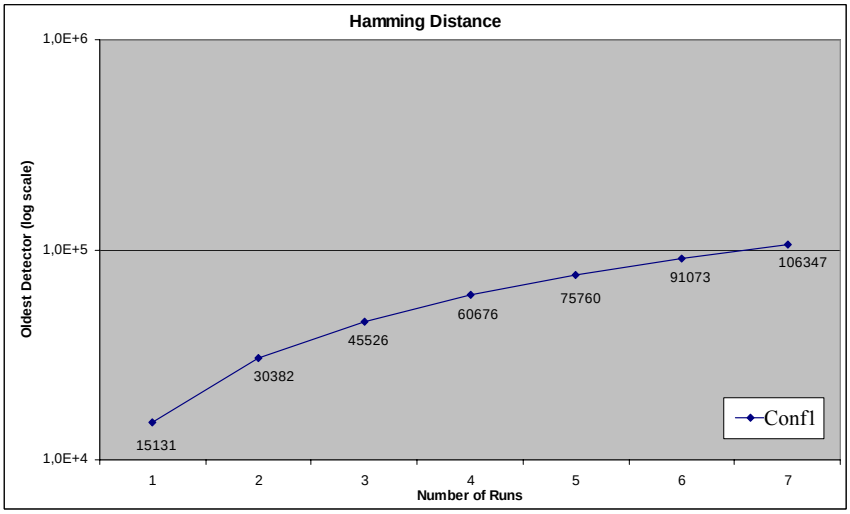

Figure 5.4.4.5.2b: The age of the oldest detector when using the Hamming Distance in the second approach.

5.4.4.5.3 r-contiguous bytes

The training in the first approach was successful in both configurations. In the new training situation the second configuration of the first approach was reused. 800 detectors were selected, each with the length of 8 bytes. The match threshold was set to 3 bytes. This configuration now

failed in the training of the detector set. This was possibly up to the higher packet processing rate and the high load of binary packets. The high number of binary packets was leading to a definitely higher number of replacements. Two additional configurations were tested on their success. The first of them used a set of 2000 detectors, each equipped with 4 bytes. The match threshold remained at the value 3. As it can be seen in the following figures the replacement rate stayed at almost the same level. All detectors were continually replaced in sooner or later runs. Finally in the ninth run not one detector was immune to replacement, therefore also the configuration was assumed to be not successful.
The other configuration was using 250 detectors, each of 16 bytes length. This time the match threshold was increased to 4 bytes. The packet processing speed in this configuration was also acceptable with a drop rate of 32.4%. The replacement rate was very low, because a defined string of 4 contiguous bytes is contained in a very low fraction of the payloads. That's why the training was successful.

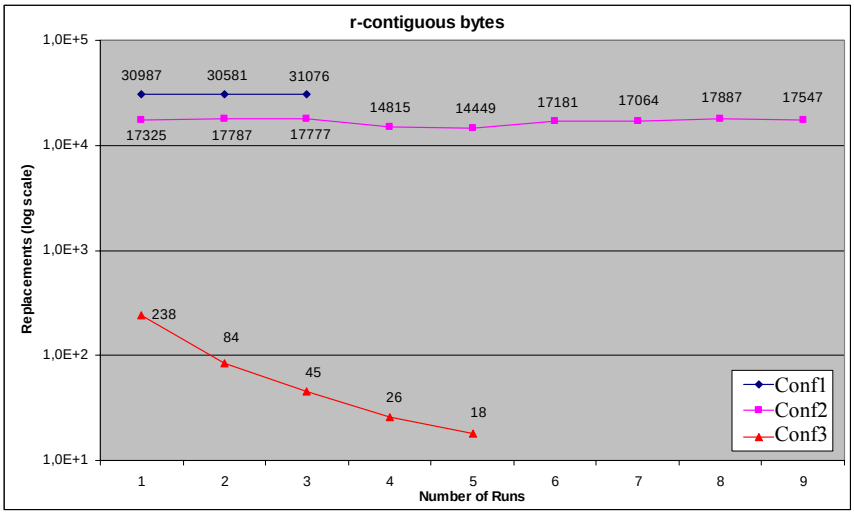

Figure 5.4.4.5.3a: The Replacement rate when using the r-contiguous bytes in the second approach.

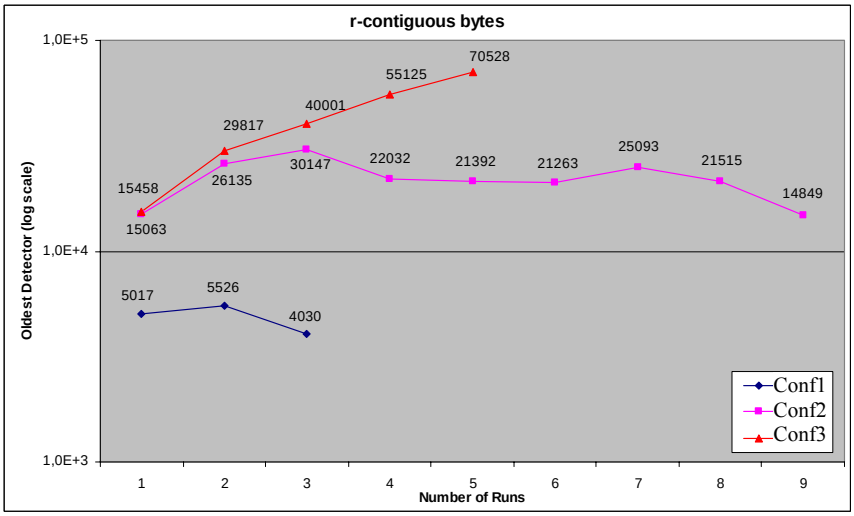

Figure 5.4.4.5.3b: The age of the oldest detector when using the r-contiguous bytes in the second approach.

5.4.4.5.4 Longest Common Subsequence (LCS)

The second configuration of the LCS algorithm in the first approach was reused. Here 100 detectors equipped with 17 bytes were trained on the network traffic. The match threshold was kept at 11 bytes.
The training of the detector set was not successful. The replacement rate was not changing during the runs and remained at almost the same level. But compared to the first approach the rate was definitely higher which was causing the low value for the age of the oldest detector. After the fourth run not one detector was immune to replacement. The lower speed of the network traffic had unfortunately no effect on the rate of the processed packets.
Afterwards a second configuration was using a higher specificity of the detectors. The number and the length of the detectors remained the same, but the match threshold was increased to 12 bytes. This training was developing in the same way as before the Hamming Distance (see 5.4.4.5.2). The number of replaced detectors remains over the runs at almost the same level. But age of the oldest detector is from the sixth run on continually increasing. This result came from one detector that is with a high probability tolerant to the presented TestData set. The 99 other detectors are in each run subject to replacement – and therefore the replacement rate does not significantly change.

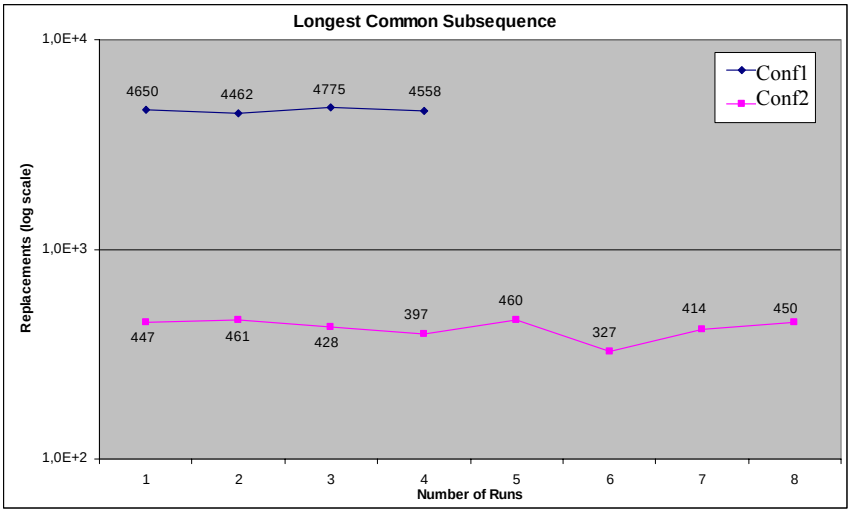

Figure 5.4.4.5.4a: The Replacement rate when using the longest common subsequence in the second approach.

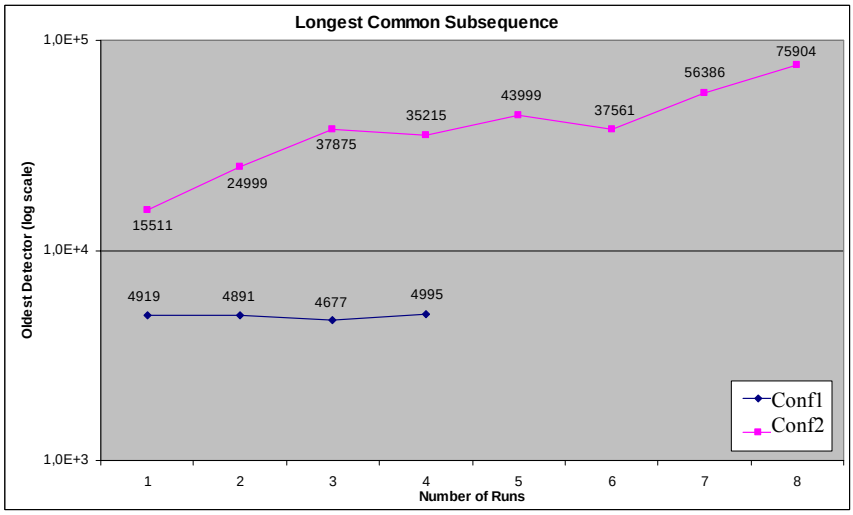

Figure 5.4.4.5.4b: The age of the oldest detector when using the longest common subsequence in the second approach.

5.4.4.6 Conclusion on the training results of both approaches

Overall, three of the five implemented algorithms were able to train a detector set successful in terms that it would be tolerant to the presented network traffic. The remaining two algorithms (LCS and Levenshtein distance) unfortunately are processing the packets much to slow.
After the training period two of the successful algorithms from the second approach have been tested on their success on detecting nonself elements. The first algorithm was the Pearson correlation and the second the r-contiguous bytes string matching method (third configuration). For this purpose 9064 virus files have been collected which includes potential threats to each version of the Windows operating system. After the training of the detector set these 9064 files were transferred over the network. They should be detected by the computer immune system as elements which were not observed previously. The system should declare them as anomalies in the network traffic. These 9064 files were transferred by using 36972 packets. The following table shows how many of them are declared as suspect packets.

Algorithm	**Detection Rate**
r-contiguous bytes	17 suspect packets
Pearson correlation	6433 suspect packets

Table 5.4.4.6: The detection rate of transferred virus files by two algorithms.

The very low detection rate of the r-contiguous bytes algorithm came from the fact that only 250 detectors are used in combination with a match threshold of four contiguous bytes. The possibility for observing a predefined sequence of four contiguous bytes is low. In addition, when using 250 detectors only a very small fraction of possible patterns contained in the nonself set are covered. Therefore for better results it will be necessary to increase the number of used detectors in a large extent. But now this is not practicable because of a too low performance of current computer systems.
The Pearson correlation used a set of 3000 detectors. It is able to detect previously not presented packets in the network traffic with an acceptable result. For an even higher detection rate it is also required to increase the number of detectors.

5.4.5 Modification of the data representation

We are interested in detecting malicious executable code in the payload of the packets. The goal in training the detector set is to enable the detectors finally to detect malicious runable machine instructions. For this the comparison of binary byte strings is not the best approach. In the methods used so far the payload is represented as binary string where the symbol range is from 0 to 255. A big fraction of the packets being transferred is not executable at all. Therefore the representation of the data in the payload can be converted to disassembled machine instructions. Then the unambiguous number of the machine instructions is the new basis for the match algorithms. At the low level observation, the traffic stream has no indicators for the actual position in the files, if the packets are treated as being independent. A detailed description of the modification of the data representation is provided in chapter 6. There also the implementation of the disassembler engine and the results of the applied algorithm are presented.

5.5 Using the Preprocessing Engine for packet header processing

The inspection of the payload content described in chapter 5.4 had a low success rate. This was shown by the result of the practical tests in chapter 5.4.4.6. The complexity of the payload data on which the training is based is too high. The assumption behind this modification was: As soon as the data complexity is reduced the results of the computer immune system will lead to the detection of attacks.
The detector set used in LISYS was trained on the headers of packets and the system had a high detection rate with a low number of false positives. Therefore, the implementation in this thesis was modified. By this, the system is trained on a similar data set, as it was used for the training of LISYS. Now the training of the detector set is not based on the payload data, but on some fields of the packet header. The training of the detector set should verify the results of LISYS implemented by Hofmeyr and Forrest. Further it should show that the presented architecture of the computer immune system in chapter 3 is performing good job in anomaly detection.

5.5.1 Description of the data representation

In LISYS the datapath information is extracted from TCP/SYN packets and compressed to a string of 49 bit length. In this thesis a string of 72 bit length is used to represent the source IP address, the source port, the destination port, one bit to indicate if the payload is of zero length, one bit that defines the protocol and if the protocol is TCP then also the TCP-Flags of the packet are included. It was assumed that the underlying topology of the network is switched ethernet and not like in LISYS a broadcast topology. Therefore, the destination IP address was not of interest, because for the incoming packets it is always the same. After the training the detectors would be not specific to a single computer in the LAN. They can be spreaded over the network in case of a successful detection.
The network traffic contains two protocols of interest: TCP and UDP packets. Here both protocols are inspected. In case of TCP traffic only the packets with a SYN Flag were selected for further processing. But in addition also UDP packets are processed.
Our representation of the extracted 72 bit string uses no compression. The 72 bit were split up in two bit sequences which were distributed over an array with 36 bytes. The format of the byte array is as follows:

- **Index 0 to 15:** stores the 32 bit SourceIP Address
- **Index 16 to 20:** store the lower 10 bits of the Source Port

  (Portnumbers ≤ 1024)
- **Index 21 to 25:** stores the lower 10 bits of the Destination Port (Portnumbers ≤ 1024)
- **Index 26 to 28:** stores the remaining 6 bits of the Source Port (Portnumbers > 1024), which bits are often Zero
- **Index 29 to 31:** stores the remaining 6 bits of the Destination Port (Portnumbers > 1024), which bits are often Zero
- **Index 32 to 34:** stores the TCP Flags. Those indices are zero for the UDP protocol
- **Finally the Index 35:** stores in the first bit a boolean value indicating if the packet length is equal to zero, and in the second bit if the current packet was TCP or UDP.

5.5.2 Training of the detector set

The detector set was trained on the above described constellation of the TestData set. This time a small LAN consisting of 15 workstations was used for the test environment. The sensor was installed one of them and all network nodes were transferring the data set to the one equipped with the sensor. Therefore, the self set consists of 15 IP addresses including the ports used for the data transfer.
It had been observed that under two conditions the increment in the training time for tolerating a few additional network nodes is low. First, this applies to nodes which are part of the same IP-Subnet (Class C Network Addresses). Second, the detector set should already be tolerant to the actual number of network nodes.
The used algorithms were the Hamming Distance, the Levenshtein Distance and the Longest Common Subsequence. We do not use the r-contiguous bit match rule.

5.5.3 Results of the applied algorithms

All three algorithms have been trained successful on the file transfer using 15 IP addresses out of the Class C Network range from 192.168.123.0 to 192.168.123.255. The Hamming Distance was tested in more detail if it detects portscans from different source IP addresses. The portscan attack ranges from port number 1 to 1000 and is producing 1566 packets. The source of the first portscan was an IP Address from one of the 15 network nodes, which are part of the local network. This is simulating an attack from a "trusted" computer from the network neighbourhood. The second attack was driven from a random chosen IP Address belonging to the same Class C IP addresses as the 15 network nodes. This was simulating an attack from a computer which is not normally connected to the local network. And finally the third tested attack came from a completely different IP Address.

Portscan number	Source	Detection Rate
1	192.168.123.50	517 suspect packets
2	192.168.123.99	751 suspect packets
3	169.254.244.19	1496 suspect packets

Table 5.4.6.3: The detection rate of a portscan attack from different IP addresses, by using the Hamming Distance algorithm.

This result shows that the architecture behind the computer immune system is capable for detecting anomalies very well. The training of the computer immune system is not restricted to IP addresses belonging to the local network; it is also possible to include external connections. The training will be successful if these connections are occurring frequently and use static IP's. A static IP Address set can not be defined for machines which are frequently connecting to various WWW or FTP Servers on the web. Therefore the anomaly detection with this approach will not work if it's based on the extraction of some header fields on these machines. This limitation is also mentioned by Hofmeyr and Forrest.
It works very well for protecting the own network from external connections. Precisely that are IP addresses which are not belonging to the previously defined self set. Here in this test 95.5% of the packets from an external driven attack are declared as being suspect, and that's definitive the indication for a correctly detected portscan. Internal attacks are also detected by a high rate of suspect packets: 33% for the first portscan and 48% for the second portscan.

5.6 The monitor of the computer immune system

The monitor application is the interface of the sensor to the user. It periodically reads the logfile of the sensor and displays the ongoing changes to the user in a visual comfortable manner. In addition the monitor informs the user about the actions requested by the sensor and the status of some autonomic tasks. Changes of interest are for example:

- Changes in the state of a detector. After the tolerance period detectors become automatically mature detectors. If they have successful detected malicious activity they receive the costimulation and become memory detectors.
- The activation of a detector. In the logfile also the dump of the packet(s) is included which is responsible for the activation.
- The replacement of detectors caused by the end of their lifetime, or by an absent costimulation signal.
- The replacement of detectors in the tolerance period. This information is displayed per time interval.
- The appending of a successful detector to the detector set which is trained by another network node.

The user should get an overview about the update actions during the learning process of the detector set since the preprocessor was initialized. The information is displayed as a preprocessor event log. Additional statistical information about the learning process of the sensor is updated periodically. That statistical information includes the following values:

- Number of detectors in tolerance phase
- Number of mature detectors
- Number of activated detectors
- Number of memory detectors
- Number of replaced detectors in tolerance phase since the beginning of the learning process
- Number of replaced activated detectors since the beginning of the learning process (false positives)
- Number of replaced mature detectors because of their age since the beginning of the learning process
- The overall running time of the preprocessor, this is the time since it was initialized
- The age of the oldest detector
- Number of packets being processed since the beginning of the learning process
- Number of packets being rejected (if the filter is used) since the beginning of the learning process

5.6.1 Periodically produced events

The sensor periodically logs a summary of the number of detectors which were replaced due to a match in their tolerance period. This is the only message that is displayed by the monitor until some of the detectors reached the mature state.
As mentioned before also statistical information about the training process is frequently updated. This information is decoded and displayed by the monitor in corresponding fields to provide a clear overview about the status of the detector set.

5.6.2 Events produced by detectors in the mature and memory state

Mature detectors need a number of match events (according to the activation threshold parameter) then they get activated. This event will be handled by the monitor as a costimulation request and therefore it is displayed to the user in a separate window. This ensures that the user does not lose track of all user requests. While a detector is activated it can discover further sequences in the payload data. Every time a detector matches a symbol sequence in the payload it will increment its match count. This is logged in combination with detailed information of the discovered packet. If the sensor requests a costimulation of one detector the user gets also a dump of the packets which were causing the activation. All the packets can be disassembled with a free selectable entry point in the payloads. The user now can decide if the detector had observed some malicious code. If this is the case, the monitor application sends the costimulation signal back to the sensor and the detector will become a memory detector. Then the sensor delivers the successful detector to the monitor, which is building up an IODEF (Incident Object Description and Exchange Format) document. The monitor then sends this document to a special table in the central MYSQL-Database (Memory Detector table, see chapter 5.7). The communication of the monitor with the database for the exchange of successful detectors is done with IODEF, which structure is XML based. The content of the document is described in chapter 5.9. The monitor frequently requests updates of successful detectors from the database. If a detector was added recently then the monitor receives the detector and sends it to the sensor, where it can be appended to the local detector set. If a new detector is added to the database it can be spread over the whole network. The thesis provides the framework for the implementation of such a distribution. All tests of the system did not use such a function, because the work focused more on the recognition of malicious code than on the distribution of information about them.
If an activated detector has detected no malicious activity the sensor logs its replacement after the costimulation timeout. Then it is also deleted from the user request window. If mature detectors did not match any strings during their lifetime their replacement is also logged. Finally the sensor produces some more events when the activation threshold is changed from the default value. This will activate the local sensitivity level of the sensor and the belonging messages.

5.6.3 Updates of the MYSQL-Database

The monitor application performs the update of three tables in the computer immune system database.

- The monitor updates the *Statistic Table*. It sends the statistical information about the current training status to the database. The monitor receives this information from the logfile of the sensor.

- It updates the *Chart Data Table*. This includes the calculation of the data needed for displaying the charts on the webpage.
- It updates the *Memory Detector Table*. When a detector finally has been upgraded to a memory detector then the monitor produces an IODEF document. The document is appended to this table in the database.

5.7 The computer immune system database

The database is designed as a MYSQL database [64] on a web server. This database must be accessible by the monitor applications from all computers which belong to the computer immune system. It consists of three tables:

- the Statistic Table,
- the Memory Detector Table
- and the Chart Data Table.

Each computer system is registered in the database through its IP-Address. This is necessary to ensure that data updates are associated correctly with the corresponding monitor application.
The data of the Statistic Table and the Chart Table are used to keep the Web-Frontend up to date. The Memory Detector Table is used to spread successful detectors within the immune system. It is also open for access from other frameworks using a similar kind of anomaly detection environment. These foreign systems can use the IODEF document to improve their own detection system. The incident information report is available for public access.

5.8 Web-Frontend

The purpose of the Web-Frontend [65, 66] is the presentation of the current status of a computer immune system which is running on a network. The Web-Frontend allows a public view. The statistical data of all network nodes is presented on this web page. This allows to track the actual progress in the training of the system.
The data presentation is divided into two sections:

- The first section is an overview about the training success of the detector set in the computer immune system, shown as two system performance charts. One chart shows the number of replaced detectors in the tolerance period per time interval. The other chart shows the average age of the oldest detector in the computer immune system.
- In the second section the detailed learning progress of the independent network nodes is presented in two tables. All values of the Statistic Table from the database are viewed.

5.9 Incident Object Description and Exchange Format (IODEF)

The decision for using IODEF in this system was based on the advantage that IODEF defines a common data format [27]. It is used for documenting the actions that have been taken to repair contaminated systems and to prevent a repeated incident occurrence. The incident itself is also reported through statistical information, logfile entries, special behaviours, code fragments, and so forth. This document can be exchanged between CSIRTs (Computer Security Incident Response Teams). In this way successful detectors and their incident data are accessible in a common data format. This makes it possible to exchange information with other frameworks. The IODEF reports also contain the patterns of the detected anomalies and by this they will enable other groups to eliminate those anomalies.

5.9.1 IODEF Overview

This is a data format that enables security working groups to exchange computer security information between them. The intention of the IODEF is to provide a message format that can extend the capability of CSIRTs. It satisfies the required format for exchanging incident reports. By using IODEF it is easier for people working in the computer security field to give or receive support when security incidents occur. Also the cooperation between them is easier because they can share specific data in a defined format. In the design of the IODEF format it was important to remain compatible with IDMEF (Intrusion Detection Message Exchange Format), which was designed for intrusion detection systems. Therefore, it was necessary to implement a data model which was based on IDMEF, but that represents, handles and exchanges incident data. The format allows to append also other important security information like the information about a security hole or the link to a virus database. The focus of the format design is centered at the documentation of sources and targets of malicious activities in the system. Also the behavioural analysis of the malicious activities is a central issue. The defined format model offers a common way to document the analysis and the investigation of the incident. This security report can finally be shared among working groups. The implementation of IODEF is done in XML.

5.9.2 The contents of the report in the computer immune system

If a detector was successful (which requires both activation and costimulation) it is sent by the sensor to the monitor application. There the incident report is generated and delivered to the memory detector table in the database. The IODEF incident report contains the information about all captured packets which caused the system to raise an alarm on this incident. Further it includes information about the environment of the system.
The contents of the report can be divided into four parts:

a) Information that is specific to the system. This part describes where the attack has been discovered:

- A value defines the purpose of a generated incident report which is here the description of the detected malicious code.
- On which network the system is running on. This is given by the name of the current network.

- The algorithm which was used for the training of the detector set.
- A unique number of the incident report defines the IncidentID.
- Time the report was generated.

b) Information to the owner of the report:

- A contact information record contains also the address and the function of the contact person.

c) The incident description:

- The classification and description of the attack method. It defines how the attack works, including the analysis of the incident. If the attack was already known then its reference name can be provided. This includes also an eventual available URL to further details.
- The expected impact on the network if the discovered attack would have been successful.
- A detailed description of the incident event, including the information about the source and the destination of the attack. The source is given by the IP-Address and the used port, whereas the information about the target computer system includes the attacked service. The complete event data is supplied, which includes the protocol, the time stamp and the hex dump of the payload data. Finally it reports the significance level of the detector which has detected the occurrence.

d) Additional Data:

- In the additional data section the detector and its length is stored. It can be loaded into the set of detectors of other computer systems which then are capable of detecting this particular attack.

6. Changing the data representation

The decision to disassemble the payload data was made because the byte stream of the payload data has a very big spectrum of possibilities. There is a huge number of possible data arrangements. It will be very difficult to detect something useful. Disassembling the payload has two advantages:

- The space of possibilities in the payload data is reduced by splitting up the opcode in machine instructions and their arguments. For the training, only the instructions are of interest and therefore their arguments are dropped. The arguments vary on a very large extent.
- The detectors are trained on real instructions and not on the byte code like before.

The Netwide Disassembler [53] (NDISASM) is included in the Netwide Assembler package as an additional application. NDISASM produces disassembled code from binary source input data. Netwide Disassembler is just disassembling and does not understand any of the input file formats. The disassembling starts at a free adjustable offset from the beginning of the input data. The produced output uses 32-bit instructions.
NDISASM produces machine instructions independently from the format of the input data. Packets, which contain some data that is not machine code will be disassembled as well as if the data contains some machine code. When arbitrary data is disassembled the resulting instructions will often look different to correctly executable code. This is because some of the assembler instructions will be combined with unusual prefixes and also Define-Byte (DB) instructions are generated more often.
The decision for using NDISASM was based on the fact that it does not need to understand the executable headers. The whole payload of the packets can be disassembled, where other disassemblers will produce errors due to missing header information. An advantage is further the configurable data offset. The Netwide Assembler package provides also the source code. The files needed for disassembling were added to the Snort preprocessor after some modifications.

6.1 Disassembling the payload data

The output of NDISASM was modified that it is returning only the instruction number without its opcode or parameters, but if needed, they can be printed in a code listing. For the analysis of polymorph code it is very useful to return only the instruction number. For example, the XOR instructions can be built with several different opcode byte sequences. These sequences are indicating different parameter constellations, but are still only XOR instructions. The opcode is often so different that it cannot be detected by training the detector set only on byte sequences. Because the XOR operation is often used for polymorphic decryption engines this disassembling method allows to detect such decryption engines more easily.
Different opcodes which are finally disassembled to the same instruction apply to many instructions on the Intel x86 processor. In the following example, five MOV instructions are included in a code listing. All MOV instructions have the number 292 but a completely different opcode. As it can be seen all bytes, except the first byte in a fraction, are varying. Also the instruction's opcode is of different length.

```
01 241  CC                int3
02 292  8BFF              mov edi,edi
03 456  56                push esi
04 292  8BF1              mov esi,ecx
05 456  68F4A10001        push dword 0x100a1f4
06 292  C70608130001      mov dword [esi],0x1001308
07 292  C7460401000000    mov dword [esi+0x4],0x1
08 026  FF15B8100001      call near [0x10010b8]
09 292  8BCE              mov ecx,esi
10 409  5E                pop esi
11 250  E94FFCFFFF        jmp 0x11cc
```

Figure 6.1: Opcode example, showing the large range of opcode and arguments for the instruction MOV.

When the arguments are skipped the new representation of the data will be a sequence of instruction numbers. The Intel x86 processor supports 563 instructions in total [53]. One instruction can have from one to eight bytes of opcode. Some instructions have a fixed instruction length, like some one byte instructions. But others, like MOV or ADD, are supporting different opcode lengths within the instruction. The average opcode length per instruction in an executable file is 2.437 bytes. The average reduction in possibilities of an executable is $563:256^{2.437}$ representing a symbol factor of 1:2.437.
In the following table the average opcode length per instruction is listed for some file formats. The value for each file type is based on the analysis of the number of files listed in the Filecount column. The number of files is high enough that the result can be seen as representative for each file type. It is interesting that the symbol factor for not executable file formats is significant different (p=0.995) from executable file formats. The training of the detector set on a practical data representation maybe leads to better training results.

Filetype	Filecount	Symbol Factor	Variance
pdf	1993	2,39326886	1,09E-02
jpg	5541	2,37042501	1,36E-02
txt	3955	2,53194582	4,64E-02
cpp	2235	2,32021505	8,35E-02
dll	4541	2,44399021	5,61E-02
exe	2752	2,43717452	4,99E-02

Table 6.1: Symbol factors of some file types

6.2 The instruction spectrum of different file formats

The relative frequency spectrum of the instruction set has been calculated for some file formats. The analysis is including also a lot of virus files of the file type com, exe, pif and cpl. The tables and diagrams are presented in Appendix C.

The spectrum of all text based file formats is very similar. In all of the three analysed formats the same instructions were generated. The instruction set is concentrated in some main frequencies with almost nothing in between. Of the possible 563 instructions only 19 appear in the chart with a relative occurrence rate of 1 percent or higher. By this narrow spectrum text files can be distinguished very easy from binary file formats. That result was expected from the analysis done in chapter 5, where the byte spectrum of text files (see Appendix A) showed already a concentration on fewer bytes.

Binary data files also have a similar spectrum. Compared with executable files they have a broader instruction range, including more instructions with a relative appearance rate around 0.04. This result came from an almost continual byte spectrum of the binary files as demonstrated in Appendix A. In binary files the byte values are nearly equal distributed, that's why a broader instruction range is produced.

The byte spectrum of executable files shows one value which is by far higher than all others: The zero byte. This is because zeros are used to fill sections of the file until their size reaches a multiple of the file alignment value (see [47]). The higher occurrence of values in the range of byte 65 to 122 results from data sections in the executables. These data sections include some text i.e. for dialogue windows.

In the instruction spectrum of executable files a high occurrence value is located at the instruction with the number 5 (ADD instruction). The high number of ADD instructions results from the disassembled sequence of zero bytes in the section padding.

6.3 The effect of the entry point on disassembled instructions

6.3.1 Overview of the problem

In Windows operating systems the Win32 loader decodes the header information of the Portable Executable (PE) Files and executes the sections which are marked as executable [47, 49, 50]. It is obvious where the execution begins. In the same way it is no problem to disassemble a PE file with a disassembler application which understands the header information. All sections are decoded correctly. For further information about the PE file format the reader is referred to [45, 46, 48].

But that information can not be applied to the payloads of packets transferred over networks. On this low level of observation the header information can not be used, because each packet is analysed independently. It would not be a big problem to search packets for the MS-DOS header, which is included in every valid application. But, besides the synchronisation problems with following packets, it is not useful to do that, because malicious code may not follow these predefined structures. The goal of the computer immune system is not only to search for complete malicious applications, but also for malicious code fragments, like exploits of some operating system vulnerabilities. Code fragments do not contain any header information, they are inserted into the memory space of the exploited application and are executed straight forward.

Now the question is: Where is the right entry point for disassembling the payload data? This is not only a question of defining one out of eight bytes (because of the maximal instruction length) in the payload as being the correct one. We do not even know where to start disassembling eight bytes.
In that context a further question arises: What effect does it have if the entry point is not chosen correctly? In the following the answer to that question is presented. For that purpose an intensive investigation of PE executables and other file formats has been carried out.

6.3.2 Investigation of PE executables and other file formats

There are zero or more correct positions in the payload where disassembled code could be started. The number of possible starting points in an executable section is equal to the count of instructions. This can be explained as follows: If all opcode bytes of the first instruction are skipped then the second instruction will establish the next starting point. Then again the assembly sequence is correctly decoded. The number of starting points is zero for data sections and will be more than one if the packet contains an executable section. Wrong starting points are only those offsets in the section for which the disassembling is started somewhere within the opcode of an instruction.
The correct entry point is defined here as the pointer to the first executable instruction in the payload in terms of the original instruction sequence. When a payload is disassembled, we unfortunately do not know what the original instruction sequence is. We will not know the correct entry point. A possible way to defeat this problem is to choose every offset from the start of the payload until its end as an entry point for the disassembler. Then we know that the right one must be included in the analysis. In fact, we only need to choose 8 different offsets, because we know that the longest opcode is 8 bytes. That's why the correct sequence must be one of those 8 disassambled sequences. When we choose larger offsets, e.g. the next 8 bytes or a random window of 8 bytes somewhere in the payload, then we will get again 8 instruction sequences. But at least one of them must be equal to a sequence we found from the first 8 bytes. If there is more than one match, we repeat the method with a selection of another 8 byte window, until only one sequence is left. This must then be the sequence with the correct entry point. It has now to be investigated whether this method leads always to the selection of a single sequence or if more than one sequence may be left in some cases.

6.3.2.1 Finding the synchronisation point

There are several ways how the basic facts mentioned above might be exploited in algorithms for a computer immune system. The following analysis focuses on the search of the synchronisation point in the instruction sequence. This information is essential for the reconstruction of the original instruction sequence by an algorithm. When a wrong entry point for the disassembler is chosen, then it is of interest to determine the number of instructions which have to be disassembled before the new sequence gets synchronized with the true (original) instruction sequence. With which probability can we be sure that the rest of instruction sequence is identical to the original one? This is a question which is independent from the implemented algorithms and it will come up in all cases.
Initially, an executable file was disassembled with a few random chosen entry points. This test showed that the synchronisation of the instruction sequence with the original one happens very soon. Therefore, the consequence of choosing a wrong entry point will now be investigated more accurately.
We define the reference instruction sequence (RIS) as the one which we get when disassembling is started at offset zero of a file. This sequence serves as reference for any other disassembled sequence which is created by using some offset in the data.
Most disassembled sequences will start with different instructions compared to the RIS, because most disassemblies will start within an opcode. Synchronisation is found if from a certain instruction number on all following instructions are equal to the RIS.

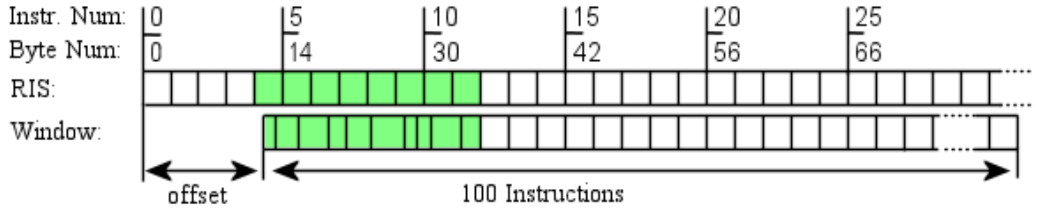

Figure 6.3.2.1: The reference instruction sequence (RIS) is compared with a window of 100 disassembled instructions. The window is generated with a different entry point in the file data. The offset of the entry point is increased byte by byte until the end of the file data. Each time this results in a different sequence of disassembled instructions. It was observed that after a few instructions the window got synchronized with the RIS. The unsynchronized part of the window is marked green in the figure.

6.3.2.2 Details of the investigation

The reference instruction sequence (RIS) is created by the disassembler by starting at file offset zero. It contains a sequence of pure instruction numbers without their parameters. The actual file offsets of these instructions are stored with each instruction.
A two dimensional array contains:

- First, it contains the disassembly of the whole opcode in the file. This sequence of instruction numbers is defined as the RIS. All instruction numbers are stored.
- Second, it contains the offset from the start of the file to each instruction number.

A window of 100 disassembled instructions is slided from the beginning of the file until its end by increasing the byte offset in the file data one by one. The offset gives the entry point for the disassembler. Every time, when the match process with the RIS has been finished, the offset is

increased and the array is filled again with 100 instructions of the file data. The actual byte offset refers to the position of the corresponding instruction in the RIS. As shown in figure 6.3.2.2a the offset may point somewhere between two instructions of the RIS. If the offset is not equal to the one stored beside the RIS then the next lower offset is chosen to select the RIS instruction. From this point on the reference will be compared with the array of the new disassembled instructions. The synchronisation and match process of the window with the RIS is shown in figure 6.3.2.2b. A practical explanation of the match process is given in Appendix D through a code example.

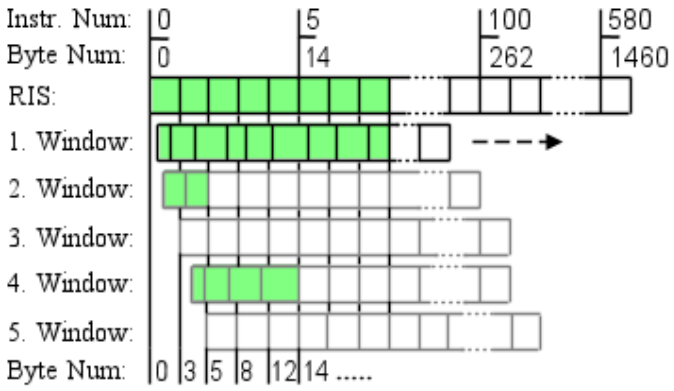

Figure 6.3.2.2a: The offset of the disassembly window is increased byte by byte. The first window had an offset of 1 byte from the start of the file data. The green colour shows that the disassembler generated nine instructions which are different from the RIS. From the 10th instruction on the sequence was synchronized with the 9th instruction of the RIS. The second window used an offset of 2 bytes from the start. The disassembler generated two different instructions. The remaining 98 instructions of the window were the same as in the RIS. The third window got an immediate match, because the first instruction in the RIS was three byte long. It was just skipped in the third window. The same process is applied to the fourth and the fifth window.

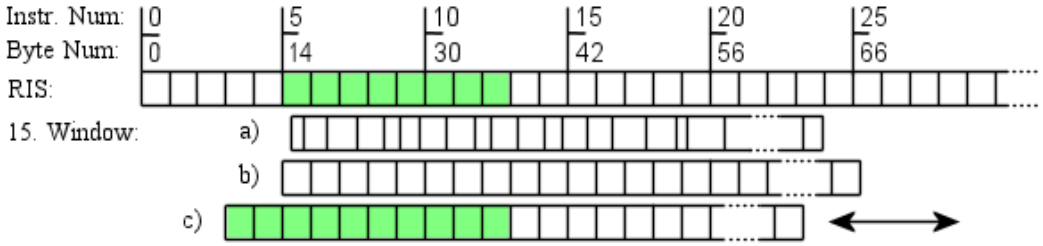

Figure 6.3.2.2b: This figure shows how new disassembly sequences are matched and synchronised with the RIS. In a) a new sequence of disassembled instructions is generated. In the RIS there is at byte number 14 an instruction of three byte length. Since the actual decoded window starts at byte offset 15, we have started decoding within an instruction of the RIS. Therefore, the new sequence first is aligned to the corresponding instruction in the RIS, which is here to the instruction at byte 14. In b) the length of the instructions in the window is ignored. This is needed to compare the instructions independently from their offsets with the RIS. One instruction of the window is then compared with one instruction of the RIS. And finally in c) the whole window is slided over the RIS to discover the maximum match count between them. This has to be done, because the disassembly was started somewhere within an instruction of the RIS. This could lead to a different instruction or even a different number of instructions. When

the window is not slided the match count is zero. In this example the relative position of the RIS (compared to the window) which receives the highest match count is +2. The relative position is calculated from the length of the instruction sequences before the point of synchronization. The difference to the RIS gives the relative position.

6.4 Results of the investigation

This measure determines the number of instructions before the point of synchronisation in the investigated files. The dual value of this number is presented in the following charts. The window contains 100 disassembled instructions and they are compared with the RIS. The resulting value from the match operation of both sequences is the **number of matching instructions after the point of synchronisation**.
At the same time the number of matching instructions is determined by the algorithm, also a second match value is calculated. The second match value gives the **absolute number of matching instructions** in the window regardless to the point of synchronisation. This second value was of interest because of the observation that in some cases also many instructions of both sequences are matching before the point of synchronisation. The reason for this is that different opcode constellations can result in the same instruction number (see chapter 6.1, opcode example of the MOV instruction). Therefore, the second value is at least equal (often higher) than the first match value.
The matching algorithm has to synchronize the window with the RIS. Synchronisation is the process of finding the highest match value for the current window. After both sequences are synchronized, the **relative position of the RIS to the sliding window** is important. This value is an indicator of how the number of generated instructions is influenced by a wrong entry point. Sometimes the window contains more instructions before the point of synchronisation, sometimes it contains fewer instructions. By this the value shows the direction of the distribution with the maximum match value. The direction is given by the positive or negative difference in the number of generated instructions when the RIS is compared with the window.
Two classes of files were analysed: executable files and not executable data files. 51 executable files were analysed resulting in over 58 million sliding windows, what approximately corresponds with the number of bytes in the files. The analysed of not executable data files include 74 files in the following formats: Text, Music, Pictures, Archives, MS-Office documents and binary data files. More than 56 million sliding windows were compared with the RIS. The offset in the files is incremented as long as 100 instructions can be disassembled from the remaining data.

6.4.1 The position of the instruction synchronisation

Figure 6.4.1a shows the result of comparing the RIS with the sliding windows of 100 disassembled instructions using executable files as inputs. The match count indicates the length of the synchronized instruction sequence. Therefore, it gives the number of equal instructions in both sequences from the point of synchronisation to the end of the sliding window. The x-axis gives the count of instructions which match with the RIS. The y-axis gives in logarithmic scale the relative number of the occurrence of each synchronisation length.
A clear flattened curve with a relative high level can be recognized until point 80 of the x-axis. This results from the section padding with zeros in executable files. Long sequences of equal bytes in the data are resulting in long sequences of equal instructions. By increasing the byte offset the window will slide over such an area. If the end of the padding area is reached and the

offset is increased once more then the sliding window includes on its end already instructions of the next file section. This first step into the next section will make the first 99 instructions equal to the RIS. But in this case the last, the 100th instruction, will differ. Such an instruction sequence will receive the match value 0, because a synchronisation did not happen within the window.
In figure 6.4.1a can be seen that the frequencies raise steep near the match count 100. This shows that in most disassembled sequences the synchronisation is reached very soon, although the disassembling is started often within an instruction of the RIS. Especially the high value for a complete identical sequence of 100 instructions is noticeable. When the offset is increased it had been observed that the sequence often remains the same. This applies if a one byte instruction was skipped, if an entry point lays within some two byte instructions or in minor cases longer ones if other special opcode constellations are met. As it can be seen in the examples in Appendix D some different opcode sequences are disassembled to the same instruction number. That's the reason for the high value at match count 100.
Figure 6.4.1b shows the same analysis using not executable data files as inputs. The data usually contains a low number of equal byte sequences. That's why the synchronisation appears faster and clearer than before.

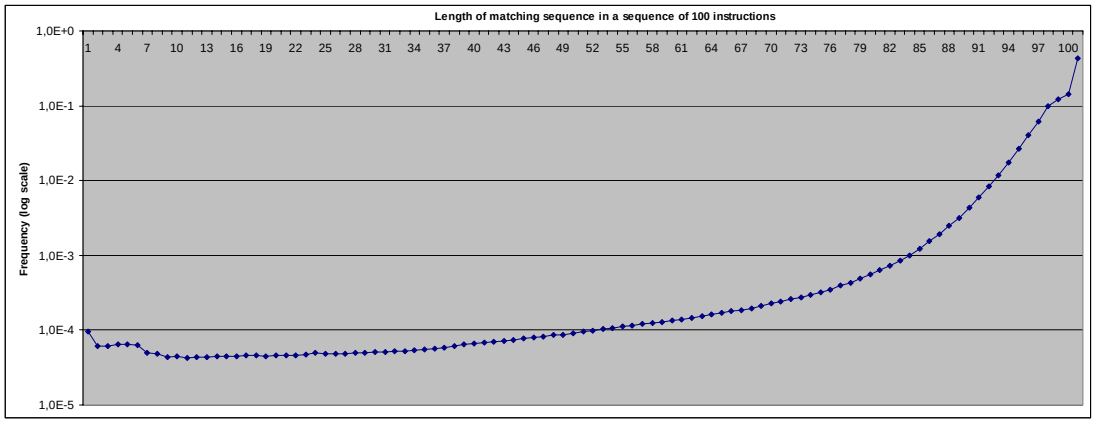

Figure 6.4.1a: The position of the instruction synchronisation in executable files.

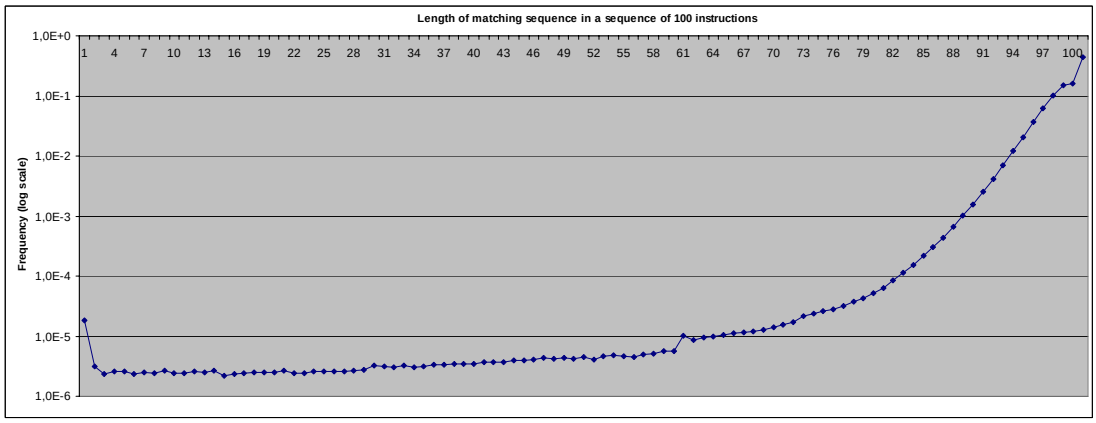

Figure 6.4.1b: The position of the instruction synchronisation in not executable data files.

6.4.2 Absolute equal instructions

In figure 6.4.2a the absolute matching value of the instruction sequence is presented. Here the overall count of matching instructions is calculated, independent from the synchronisation point.

In the padding example given above, where the window steps over section boarders in executable files and gets 99 equal instructions and one different, this function will receive not a zero match like before, but a match value of 99 for the total count of matches. In this curve a clear decrease from a match value 100 to about 60 can be noticed. This decrease is even clearer in figure 6.4.2b which presents the analysis of the not executable data files.

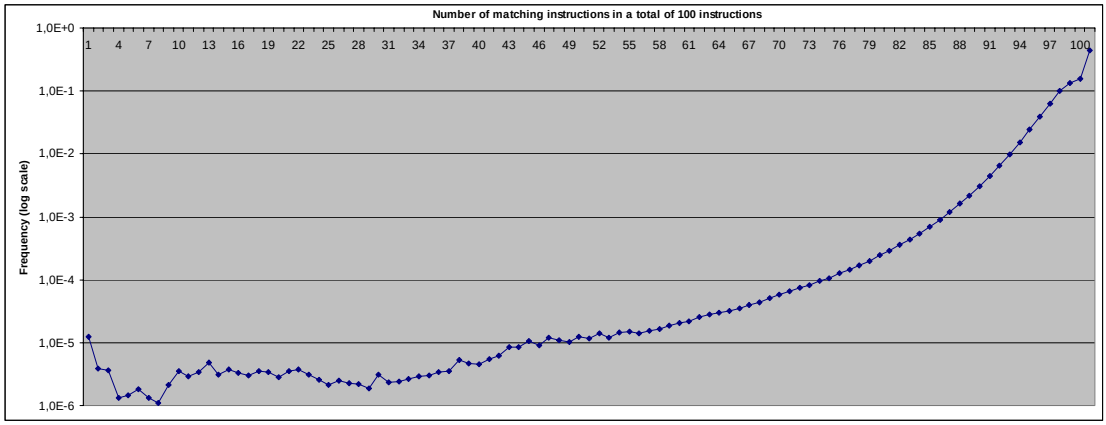

Figure 6.4.2a: Absolute equal instructions, ignoring the position of synchronisation in executable files.

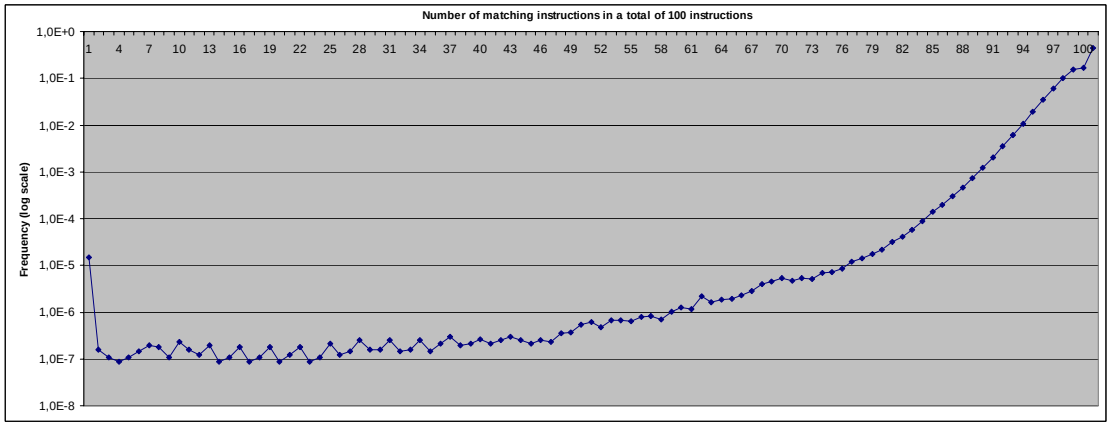

Figure 6.4.2b: Absolute equal instructions, ignoring the position of synchronisation in not executable data files.

6.4.3 Relative position of the RIS to the sliding window

When the offset of the window is increased, the disassembler sometimes generates a different number of instructions compared to the RIS. For example, when there is an instruction in the RIS with a length of eight bytes, then skipping one byte the disassembler may generate seven new one byte instructions. That is not limited to only one instruction. The following instructions of the RIS maybe are influenced as well. Maybe the eighths byte indicates a two byte instruction. Then the next instruction in the RIS will also be affected and maybe split up or combined, and so forth. The difference in the number of instructions between RIS and the window sequence can also be negative. On the one side from fewer instructions in the RIS can result a longer sequence, and on the other side also from more instructions in the RIS can result a shorter sequence.
In the search for synchronisation the matching algorithm takes the 100 new disassembled instructions and slides them over the RIS from minus 15 to plus 29 relative to the match

position which was found first. For every slide position it counts the number of matches in the sequence. From these 45 sequence match operations the algorithm chooses the highest match value and records the position of the sliding window.

In figure 6.4.3a the relative frequency of each position of the sliding window is presented. The x-axis gives the position of the RIS relative to the sliding window starting at minus 15. The peak value at position zero shows that the number of instructions usually stays the same, although the entry point of the disassembler is often somewhere within the instructions of the RIS. It can be noticed that the negative range seems to be narrower. The negative range means that from more instructions in the RIS the disassembler will generate fewer instructions in the window before the point of synchronisation (see example (c) in figure 6.3.2.2b or "relative position" in Appendix D).
Figure 6.4.3b shows the same distribution for the not executable data file. It can be recognized that this distribution possibly is symmetrical. Starting from the origin at zero both sides have an almost equal frequency.

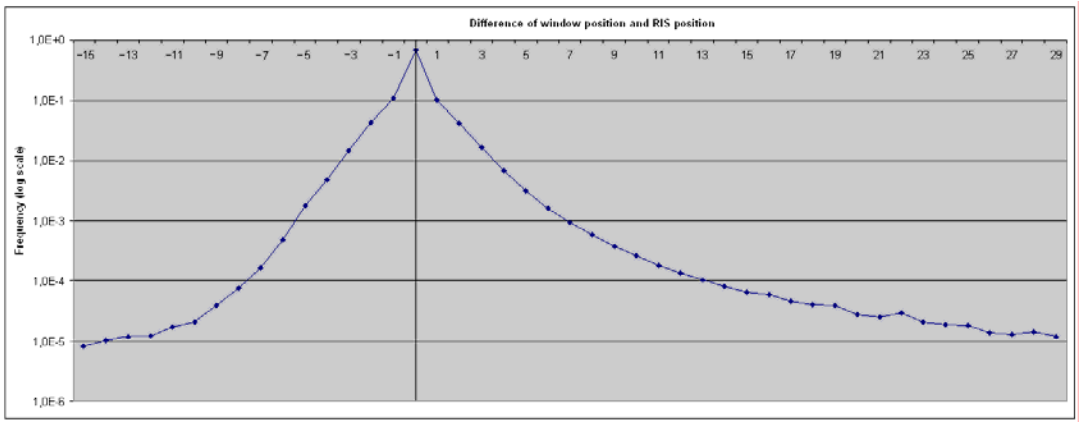

Figure 6.4.3a: Distribution of the relative position of the RIS to the sliding window with the highest match count in executable files.

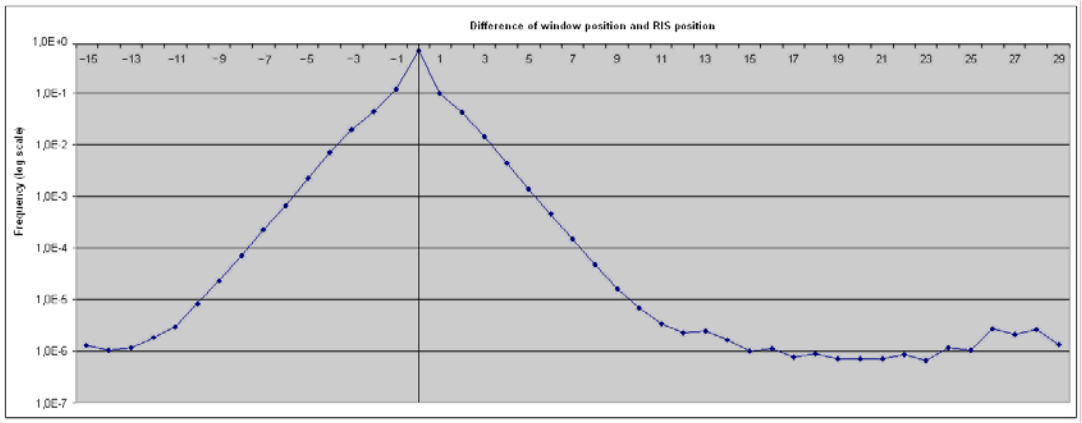

Figure 6.4.3b: Distribution of the relative position of the RIS to the sliding window with the highest match count in not executable data files.

6.5 The reconstruction of the original instruction sequence

The original instruction sequence possibly can be reconstructed as follows. An entry point is chosen in the payload. This method is independent from the selected offset in the data. At the

entry point in the data an 8 byte window is selected. All bytes in the window are selected as an entry point for the disassembler and therefore, 8 instruction sequences are produced. One byte is skipped each time. The window is selected with 8 bytes length, because of the maximum opcode length (as mentioned in chapter 6.3.2). Then we know that the original instruction sequence must occur in at least one of those 8 sequences. All 8 sequences are compared with each other to find the longest common instruction sequence. The found instruction sequence is the original sequence by the probability p:

$$p = \frac{n}{8},$$

where n is the count of sequences in the window which have the sequence in common.
In the analysis of the synchronisation point above, it was observed that a high fraction of the disassembled sequences synchronize with another sequence. This fact can be applied to the 8 sequences in the selected window. Therefore, a high fraction of the sequences should synchronize with the other sequences after a certain number of instructions. But for the reconstruction of the original instruction sequence it is of interest to find the earliest possible synchronisation. In some cases more sequences can be the original one. It is not known if the one that was selected with the probability p is the original sequence.
A problem of this method is that a window of 8 bytes can be selected for which no synchronisation occurs. In this case the original instruction sequence can not be reconstructed. The probability for selecting such an 8 byte window has to be analysed in more detail in future.

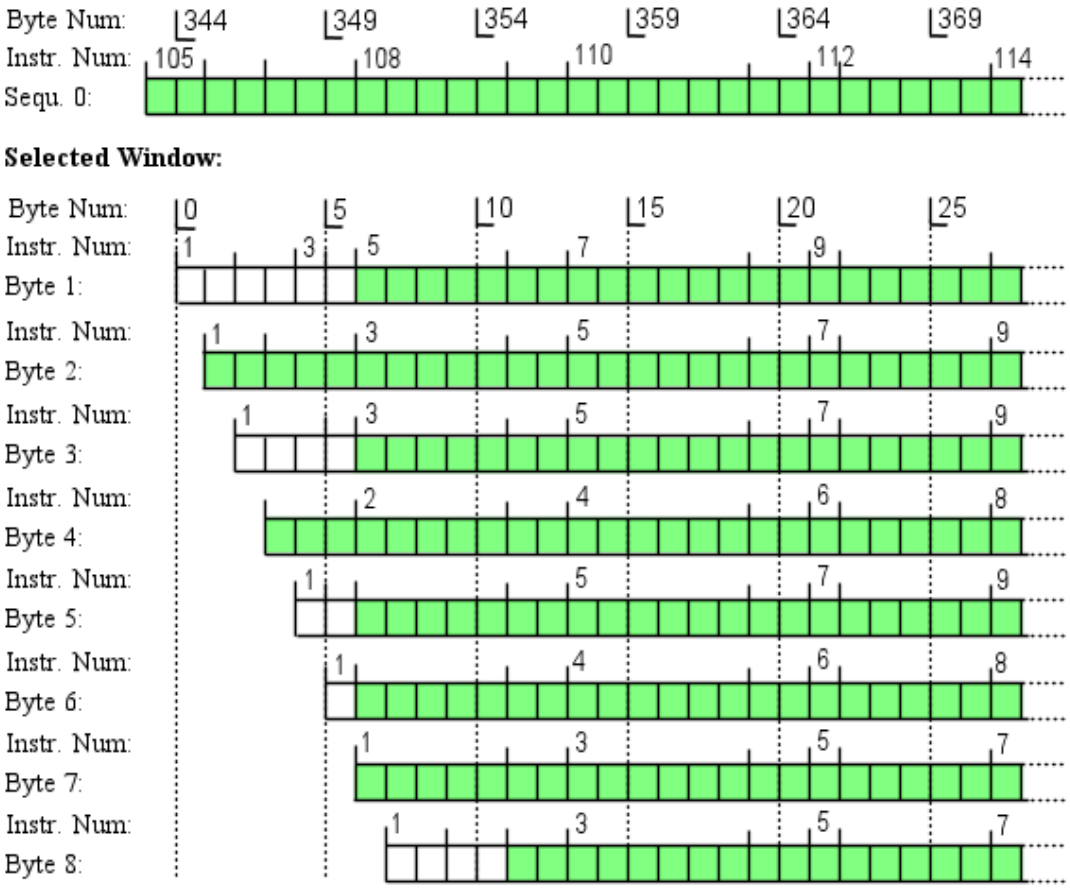

Figure 6.5: An example for the reconstruction of the original instruction sequence. This example is taken from the disassembled code in Appendix D (Entry point +1 until +8). The disassembled sequence at byte 1 is analysed. From instruction number 5 on the sequence is identical to 6 other

sequences in the window. The probability that this sequence is the original one is 7/8. The synchronisation is achieved after skipping 6 bytes of the window which corresponds to 4 skipped instructions. If one further instruction is skipped, then the probability is 100 percent. This is because the sequence occurs in all sequences contained in the window. Sequence 0 in this figure was disassembled with an entry point that is selected far before the selected window. It is assumed that the instructions have synchronized with the original sequence before the selected window. Here this sequence serves only as a possible verifier of the found sequence in the window. The green marked instructions in the window are equal to sequence 0.

6.6 Conclusion and further work

It has been discovered that independent from the entry point there is a synchronisation with the original instruction sequence within a few instructions. This information is needed for the **reconstruction of the original instruction sequence**. It is important to know how many instructions are influenced (with an error rate) by a wrong chosen entry point for the disassembler.
The main intention for this analysis was to show the fact that not knowing the right entry point in the payload possibly is not a big problem. The number of analysed instructions sequences is on the order of 10^7. This number is too low for a clear statement on the results. This investigation shows a tendency to a fast synchronisation, but the error rate is on the order of 10^{-5} (synchronisation point at instruction 30).
But at the same time this analysis raises some questions. It will need some deeper data analysis to interpret the reason for the following observed facts:

- Sometimes the match count for instruction sequences is low (<75). What is the reason for the sometimes highly different instruction sequences? Under those circumstances, what opcode constellations are met?
- For decreasing match count values it seems that the curve becomes more flattened. Are match count values below 50 instructions noise in the distribution?
- There is a difference in not executable data files and executable files. Is this difference only caused by long sequences of equal bytes (section padding in executable files)?
- For a practical application it is primary of interest to have a very low error rate. Analysing a high amount of data, which is needed for a clear statement on the error rate, is not practicable within a reasonable amount of time. Therefore, in some cases, the probability that the reconstructed instruction sequence is the right one possibly is low. For a higher probability more instructions have to be skipped, but this is not useful when detecting malicious executable code.

The future work of this measure:

- It would be interesting to determine the match count for executable code only. Here the whole executable file (all sections) is disassembled. But maybe there is a difference in the match distribution if all not executable sections are skipped. This analysis would reveal also the real error rate for the number of influenced instructions in executable code. Executable code is the target data for the practical application.

- The actual knowledge of the synchronisation point should be implemented in an algorithm. It should try the reconstruction of the original instruction sequence, as proposed in chapter 6.5. Possibly most instruction sequences can be reconstructed, even with the high error rate of the synchronisation point.

When the entry point in the payload is selected then an additional error can occur, if the entry point for the disassembler lies in a data section of the file:
This error can occur if the data section is followed by an executable code section. It is then possible that the disassembled instruction sequence contains again not synchronized instructions. When a data section is disassembled then it is possible that the last instruction from the data section is generated as a more-byte instruction. It then needs some bytes of the following code section. In this situation the disassembler does not decode the first instruction of the code section correctly. Because some bytes are missing in the code section it is treated as if the entry point is chosen incorrectly for a second time. We do not know if the actual entry point lies within a data section or an executable code section. If the entry point error can be corrected at all (with a probability) then it is at the start of the instruction sequence. A section transition is not considered. In that case, as shown above, again a number of instructions are needed to synchronize such a starting error in the code section. This is the error we will have to keep in mind when not knowing the right entry point.

6.7 Implementing the payload disassembling function in the preprocessor

The NDISASM sources have been appended to the Snort Framework. All packets, which were intended for processing, were first passed to the disassembler function. The data in the payloads was disassembled, all parameters of the instructions were skipped and the instruction sequence was stored in an array. This array then provides the data basis for the match algorithm. The symbol sequences of the detectors now need to consist of instruction numbers instead of byte values.
The implemented match rule for this approach was the r-contiguous instructions algorithm. Three different configurations were tested. The first configuration had the packet filter deactivated. The first and the second configuration were using the faster network speed. For the test of the third configuration the speed was slowed down. The network traffic was simulated with the continual transfer of the TestData set defined in chapter 5.
The first and the second configuration were using the same set of detectors: 1000 detectors equipped with 8 instructions. The third one uses 2000 detectors, where each is holding a sequence of 5 instructions. The threshold of a match was set to 2 contiguous instructions. Independently from the packet drop rate all three configurations became tolerant to the presented traffic after a short period of time. Detailed results are presented in Appendix B.2.

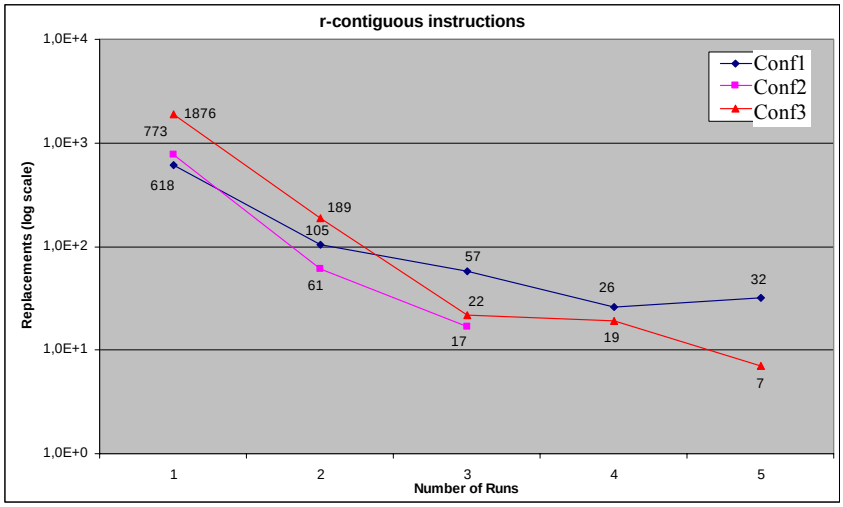

Figure 6.7a: The Replacement rate of detectors using the r-contiguous instructions match rule.

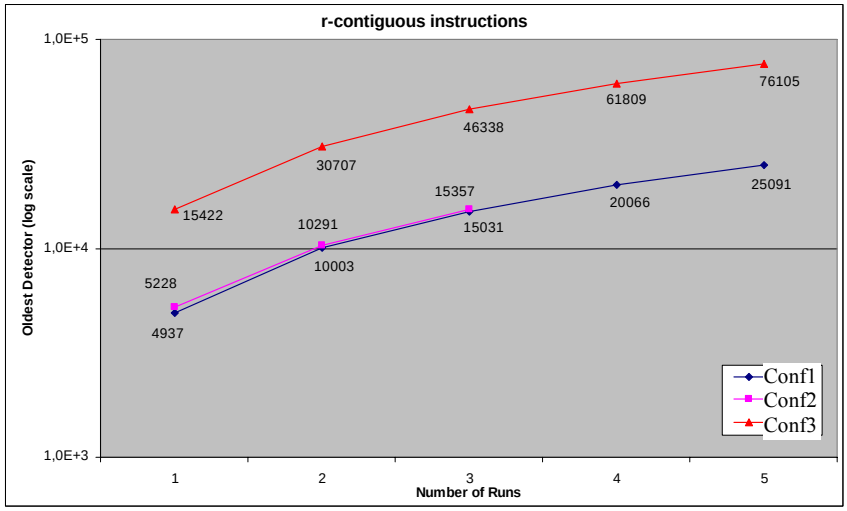

Figure 6.7b: The age of the oldest detector when using the r-contiguous instructions match rule.

6.8 Conclusion on the results of the packet disassembling approach

In Appendix C the distribution of the disassembled instructions is presented. The charts are showing that most of the instructions are occurring with a very low rate somewhere near zero. Instructions with an occurrence rate of 1% and more make only a small fraction of all possible instructions. And not more than 10 instructions out of the 563 are exceeding an occurrence rate of 4%.

When detectors are generated with a random sequence of instructions under these circumstances they will very soon be equipped with instructions which are very rare. And further when rare instructions are compared to an incoming sequence this will hardly ever match, even when the match threshold is set to the smallest possible contiguous sequence of two instructions. This is what the training had demonstrated.

7. Conclusion

The proposed anomaly detection system is performing well if three requirements apply to the training of the detector set:

Requirement 1: *The self set has a stable definition.* This requirement influences the training success.

Requirement 2: *The number of detectors in the set is chosen properly.* This requirement influences the detection rate. The detection capability of the system is influenced by the size of the detector set. In general, the higher the number of detectors in the set, the higher is the detection rate. The selected number depends on the size of the self set (in relation to the size of the nonself set). For a big self set also a high number of detectors have to be selected.

Requirement 3: *The inspected data has the right proportion of self to nonself.* This requirement influences the practical applicability of the detection system. This can be explained as follows:

First example: The system is designed to protect a LAN from nonself connections. If the training of the detectors is based on the source and destination IP address field of the packet header and requirement 1 is met then self is usually much smaller than nonself. There self can be defined as all IP addresses which belong to the LAN (plus possibly some others), and nonself is the complement of the self IP address space. It is easy to detect nonself connections, because nonself contains much more IP addresses.

Second example: The system is designed to protect a LAN from malicious code in the payload of the packets. If the training of the detectors is based on the content of the packet payloads and requirement 1 is met then self is usually much bigger than nonself. Each payload is containing a huge amount of self strings – and beside that – possibly a few nonself strings. These nonself strings are the signatures of the observed malicious code. It is very difficult to detect those signature patterns in the payload.

If all three requirements are met, then the system will detect attacks. This was proven by the modification of the inspected data (see chapter 5.5). If the implementation uses a detector set that was trained on some fields of the packet header then this will lead to the detection of attacks.

The implemented system for the detection of malicious code fragments in the payload of the packets meet only the first requirement. Overall the detection rate of malicious code fragments in the payload of packets is very low. The trained detector set was tested for the capability of detecting virus files when they are transferred over the network (see chapter 5.4.4.6). The detection capability of the computer immune system suffers from the afterwards described problems.
The approach of reconstructing the original instruction sequence was analysed in chapter 6. The data representation was changed from byte code to disassembled machine instructions. For the training of the detector set on the changed data representation a measure was necessary. The measure was done by using a disassembler with a wrong entry point. The question was how many instructions must be disassembled to get synchronized with the original instruction sequence. This information is needed for the reconstruction of the original instruction sequence. The measure shows that disassembling data straight forward without knowing the entry point has to be explored in more detail to get a clear statement. A deeper data analysis will reveal if the error rate in this measure is too high for the reconstruction (see chapter 6.6).

7.1 Problem of the useful detector symbol length

For each algorithm a parameter configuration could be found which results in a successful training. The success of a training is influenced by the adjustment of two parameters: The threshold for a match and the length of the detector symbol sequence.
The relation of the match threshold to the length of the detectors symbol sequence defines the specificity of the detector. By the adjustment of the specificity a good trade off between the size of the detector set and their replacement rate can be achieved. If for example the threshold for a match is set too high then the training will certainly be successful in terms of surviving detectors. But this configuration is less useful, because there is no adaptation to the presented self data set. These detectors are tolerant to the self set as soon as they are generated, but they are also not able to fulfil the discrimination job, because they possibly never match any string during their lifetime. In other words: The specificity is a measure of how successful the trained detector set can discriminate between the self and the nonself set.
The size of the detector set is responsible for identifying a particular nonself element and does not influence the success of the adaptation process to the self set. The size of the detector set should be selected in relation to the size of the nonself set that has to be covered. The covered space of patterns is defined by the combination of the size of the detector set and the specificity of the detectors.
In this thesis it was of interest to find configurations which train the detector set successfully and at the same time are able to fulfil the discrimination job. Therefore many configurations were implemented and tested on their success. Several configurations were not successful, because their detectors have not been specific enough to adapt to the presented self set.
The probability of a match can be calculated (see the formula in chapter 4). The results are presented in the following table for the r-contiguous bytes match rule. The table defines the match probability for two random selected byte strings with length l, under the assumption that the occurrence rate of each character is equal and the match threshold is set to r bytes.

l\r	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	256	1	0	0	0	0	0	0	0	0	0	0	0	0
2	65536	1280	1	0	0	0	0	0	0	0	0	0	0	0
3	16777216	587008	1280	1	0	0	0	0	0	0	0	0	0	0
4	4,29E+09	2,16E+08	589824	1280	1	0	0	0	0	0	0	0	0	0
5	1,10E+12	7,17E+10	2,18E+08	589824	1280	1	0	0	0	0	0	0	0	0
6	2,81E+14	2,25E+13	7,30E+10	2,18E+08	589824	1280	1	0	0	0	0	0	0	0
7	7,21E+16	6,80E+15	2,31E+13	7,30E+10	2,18E+08	589824	1280	1	0	0	0	0	0	0
8	1,84E+19	2,01E+18	7,04E+15	2,31E+13	7,30E+10	2,18E+08	589824	1280	1	0	0	0	0	0
9	4,72E+21	5,80E+20	2,09E+18	7,04E+15	2,31E+13	7,30E+10	2,18E+08	589824	1280	1	0	0	0	0
10	1,21E+24	1,65E+23	6,09E+20	2,09E+18	7,04E+15	2,31E+13	7,30E+10	2,18E+08	589824	1280	1	0	0	0
11	3,09E+26	4,65E+25	1,75E+23	6,09E+20	2,09E+18	7,04E+15	2,31E+13	7,30E+10	2,18E+08	589824	1280	1	0	0
12	7,92E+28	1,30E+28	4,96E+25	1,75E+23	6,09E+20	2,09E+18	7,04E+15	2,31E+13	7,30E+10	2,18E+08	589824	1280	1	0
13	2,03E+31	3,59E+30	1,39E+28	4,96E+25	1,75E+23	6,09E+20	2,09E+18	7,04E+15	2,31E+13	7,30E+10	2,18E+08	589824	1280	1

Table 7.1: This table shows the absolute number of byte strings a detector of length l can match under a defined match threshold r using the r-contiguous bytes match rule. In vertical direction the detector length starts at 1 byte until 13 bytes. For each length all possible threshold values r are presented, where r = 0 defines a completely general detector and r = l a completely specific detector. As an example: The second value in the first column (65536) is the number of possibilities a string with 2 bytes (l=2) can match when the threshold is zero (r=0).

All approximate string matching in this thesis is done with a r-value of at most 4 bytes. When the threshold was increased to a higher value then there was no learning effect in the detector set and all of them were “tolerant” to the presented self set as soon as they were generated. Even a value of 4 bytes is a much too low specificity value to correctly identify malicious agents. To be able to identify malicious executable code correctly the absolute minimum threshold would be 12 bytes [51, 52].
The next step in this thesis was the change of the data representation to disassembled assembler instructions. The threshold in this approach was even worse: At most 2 instructions could be used to achieve a learning effect (training on the self set). This was caused by the irregular spectrum of disassembled files, showing many instructions with a diminishing occurrence rate. During a pure random generation of detectors very soon a combination of two instructions will be found which is occurring in a very low rate – therefore it will survive the training and the training will be successful. But in practice it is useless to try the identification of virus threats with a threshold of only two instructions! To characterize a virus it is assumed that at least 10 instructions will be needed.
As presented in table 7.1 the probability of a match dramatically shrinks with higher thresholds. For example a detector length of 13 bytes and a threshold of 9 bytes will give a probability of 3.59989E-21 for a detector match. Assuming an equal distribution of the bytes in the spectrum of the transferred data then 10^{20} bytes have to be transferred in average to get one match. But this example does not even come near to the needed specificity requiring a threshold of 12 bytes.
The time required for transferring random distributed 10^{20} bytes on a 1 Gigabit/second connection speed is $617 * 10^6$ hours.

7.2 Problem of the nonself coverage

The required symbol length is not the only problem when approximate string matching is applied. It is further necessary to implement a **much greater detector set** to cover a reasonable pattern space in nonself. The size of the detector set must be somehow proportional to the size of the pattern space. Therefore, if inspecting the packet headers the number of detectors can be much lower than if inspecting the payload of packets. Furthermore, its size must be chosen according to the selected specificity of the detectors.
If for example one million detectors are chosen, then a new problem arises. To achieve real time processing on a 1 Gigabit/second connection speed the needed computational power for the approximate string matching would be $12.5*10^6$ times more than the calculation speed which current machines offer. According to Moore's Law ("computational power doubles each 18 months") this is reached in the year 2040. But also from year to year the data transfer speed will increase and the year 2040 is probably too optimistic for a real application.

7.3 Solutions and suggested future work

It is very important to include a deeper data analysis of executable code in a future work. Polymorph shellcodes do not use the complete spectrum of instructions and maybe this applies also to a part of viruses. For shellcode detection it is possible to reduce the instruction set. This should lead to a drastic reduction of possibilities in the watched patterns.
The random creation process of the detectors is an increasingly inefficient method when the detector sequences are getting longer. The detector generation can be improved with some sophisticated mechanisms. The randomly created detectors in this work are based on a white spectrum where all symbols are appearing with equal probability. For better results it would be a good idea to use a weighted random creation process where the weights are calculated from the distribution of the target data. This is especially important when the data is decoded into instructions, because as chapter 6 has shown, the instruction frequency spectrum is distributed in a very irregular way. When a white spectrum is assumed in such a case then the training can only be carried out with a very low number of instructions, which will not be useful – as shown above.
Beside the measures in chapter 6 on reconstructing the original instruction sequence it would further be of interest to verify if the instructions are executable at all. If the payload contains a sequence which is executable then the payload it is selected for a deeper analysis, otherwise the packet is dropped. The suggested approach for virtually executing the instructions is called "Abstract payload execution" which has been published in [67].

Appendix A

This chapter presents the relative occurrence rate of ASCII characters in some file types.

- In the **first section** the spectrum of five text based file types is analysed. This includes a high number of Hypertext Files (HTM) as well as a high number of ordinary Text Files (TXT). Also some source code formats, like the CPP or Ansi C Files as they are used in Visual C and also FRM Files which are Visual Basic Form files.
- The **second section** shows the analysis of four binary based file formats, which are the Adobe portable document format (PDF), a picture file format (JPG), a common archive file format (ZIP) and a music file format (MP3).
- **Section three** presents the spectrum of two Binary Executable File Formats (EXE and DLL).
- Finally, **section four** shows the values which were used for the implementation of the filters with the Pearson match rule.

A.1 Text based file types

They all have a very similar spectrum which is concentrated somewhere between byte 40 and byte 122. The spectrum includes most of the printable characters in the ASCII character table. The exceptions are the bar at byte 9, indicating a tabulator, at byte 10 and 13, representing Line Feed and Carriage Return and the Space character at byte number 32.
Bytes that represent numbers are located at position 48 till 57. Capital Letters start at byte number 65 and range to byte 90 and small letters range between byte number 97 and 122. The probability of the occurrence of small letters is higher than that of capital letters, like expected.

File type	Number of files
HTM	5383
CPP	520
C	448
FRM	322
TXT	12818

Table A.1: Text based file types

A.1.1 Byte spectrum of HTM Pages

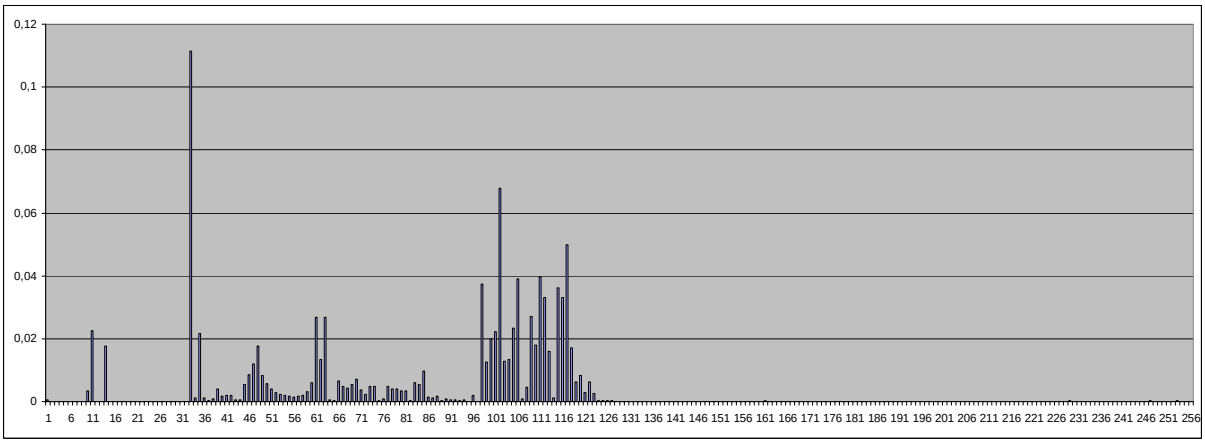

Figure A.1.1 Byte spectrum of HTM pages

A.1.2 Byte spectrum of CPP Source files

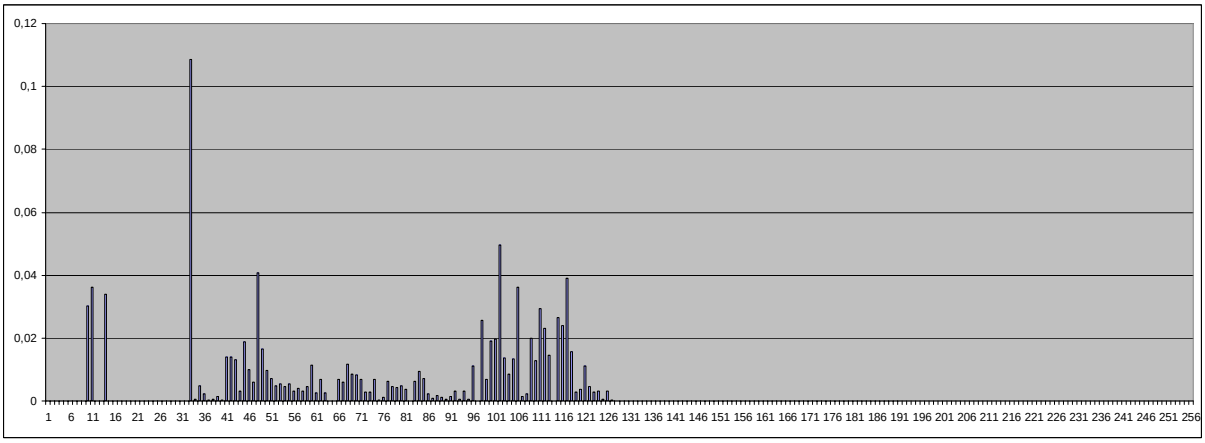

Figure A.1.2 Byte spectrum of CPP Source files

A.1.3 Byte spectrum of C Source files

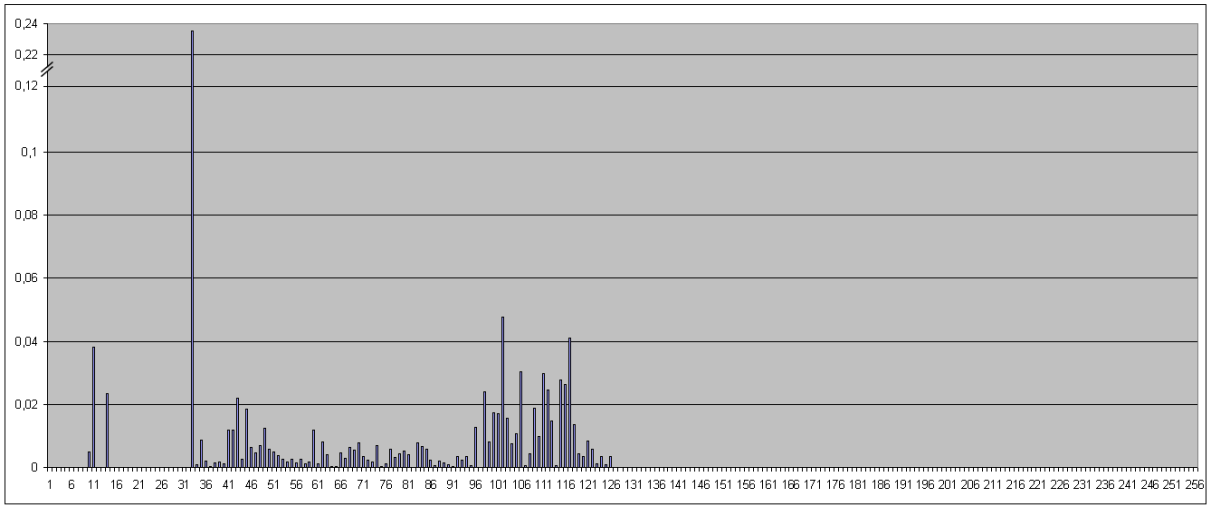

Figure A.1.3 Byte spectrum of C Source files

A.1.4 Byte spectrum of FRM Source files

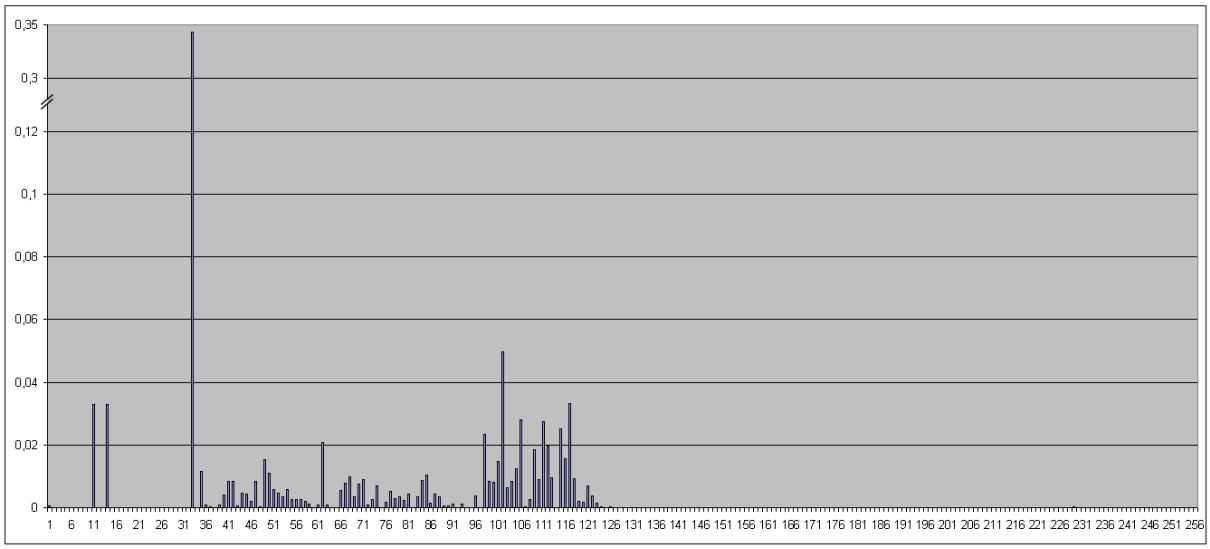

Figure A.1.4 Byte spectrum of FRM Source files

A.1.5 Byte spectrum of TXT Files

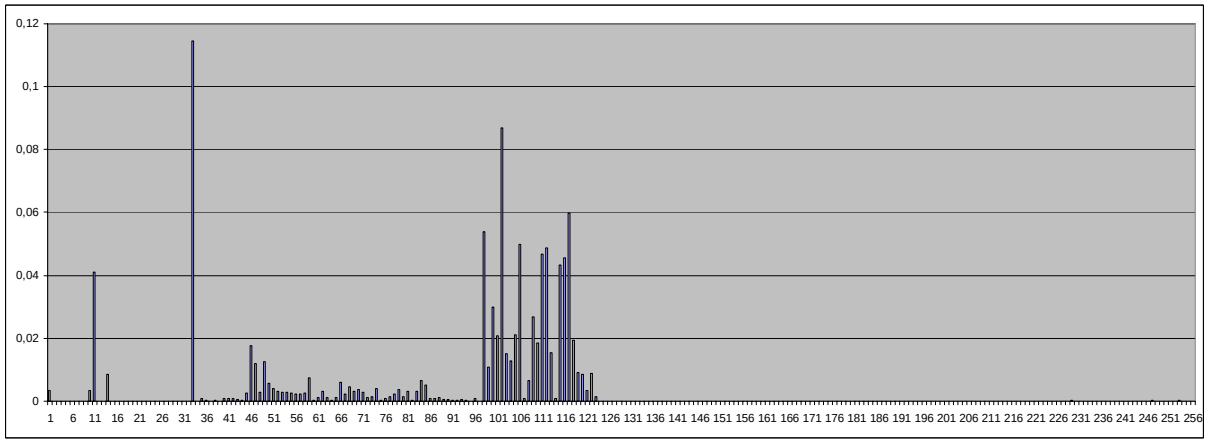

Figure A.1.5 Byte spectrum of TXT Files

A.2 Binary based file types

The spectrum of binary based file types has with some exceptions an almost equal distribution of ASCII characters.
It can be seen that PDF-Files have sections which represent text. All other characters are distributed over the spectrum with almost equal occurrence probability.
The spectrum of the JPG, MP3 and especially the one of the ZIP format is almost rectangular distributed. The ZIP file seems to contain a very small portion of text. The one of the MP3 and JPG Format is a bit more uneven than the spectrum of the ZIP file format. In binary files the characters 0 and 255 have relative to the other characters a higher occurrence rate.

File type	Number of files
pdf	3618
jpg	13686
zip	3696
mp3	4684

Table A.2: Binary based file types

A.2.1 Byte spectrum of PDF files

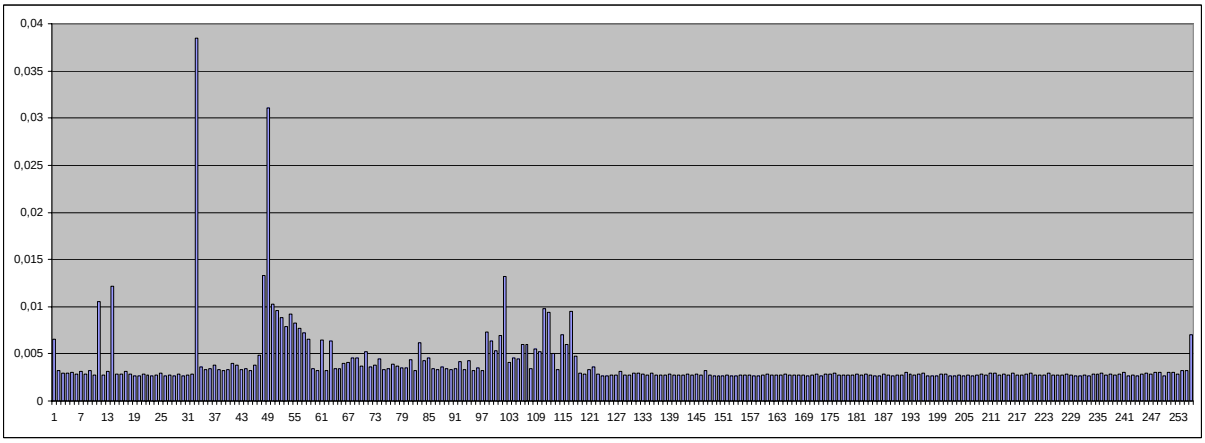

Figure A.2.1 Byte spectrum of PDF files

A.2.2 Byte spectrum of JPG Image files

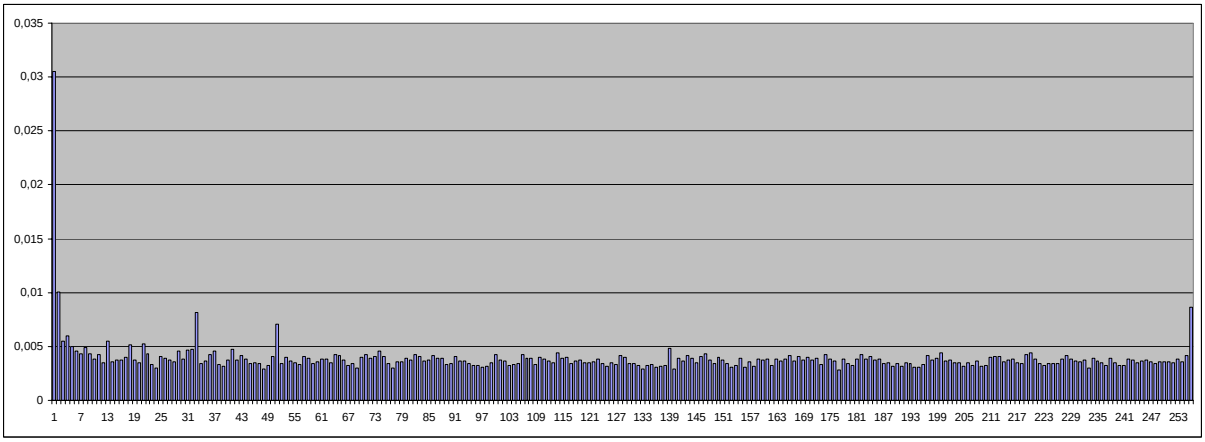

Figure A.2.2 Byte spectrum of JPG Image files

A.2.3 Byte spectrum of ZIP Archive files

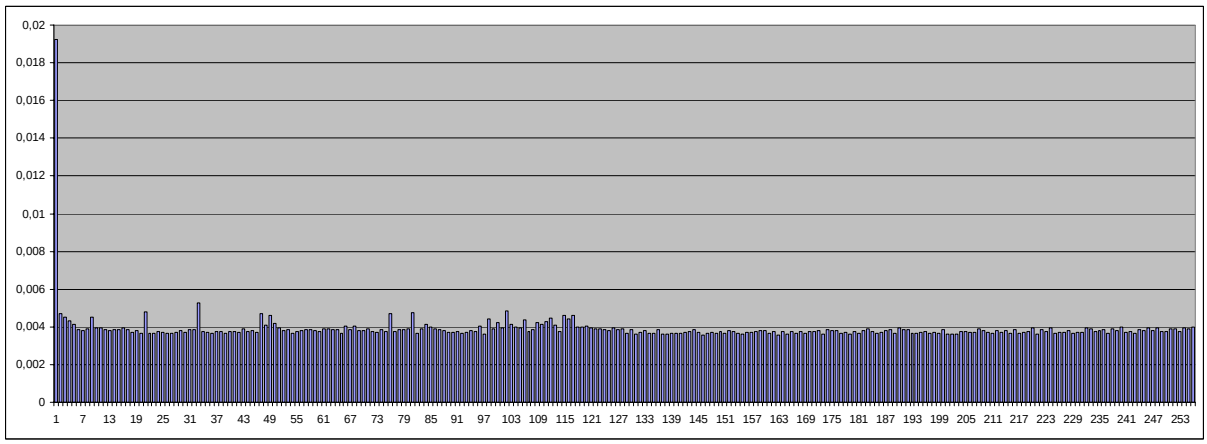

Figure A.2.3 Byte spectrum of ZIP Archive files

A.2.4 Byte spectrum of MP3 Music files

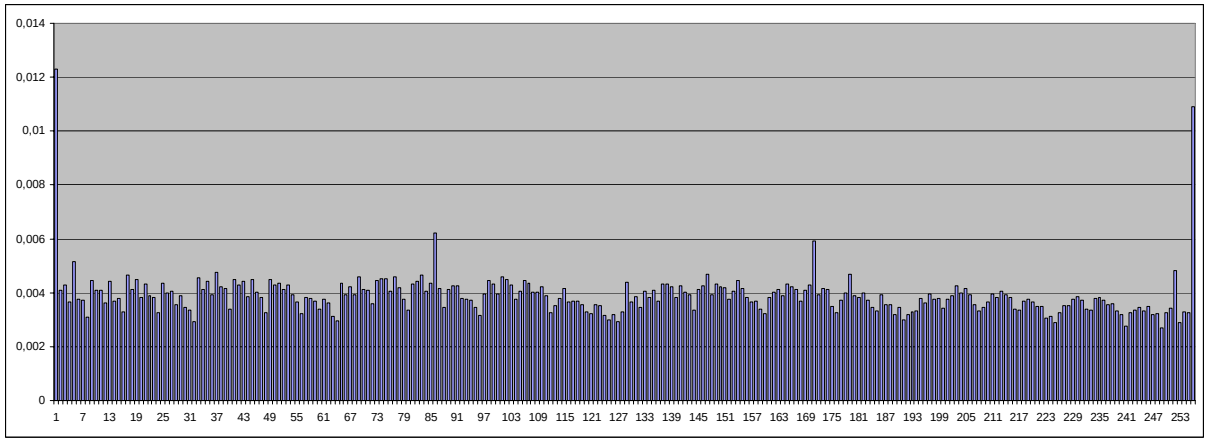

Figure A.2.4 Byte spectrum of MP3 Music files

A.3 Executable file types

The spectrum of both analysed executable file formats can be seen as statistically equivalent. It has a low portion of noise distributed over the whole spectrum with a relative occurrence rate of about 0.001, which is definitely lower than the one observed in the binary file types above.
This feature including the main bars of the spectrum could possibly be used for discriminating between those files. But when it was applied on network packets it resulted in a lot of missed payloads containing executable code, even when the filter was relaxed. The reason was that the different sections of the executable files which have different contents are resulting in a different form of the spectrum.

File type	Number of files
EXE	4110
DLL	6038

Table A.3: Executable file types

A.3.1 Byte spectrum of EXE files

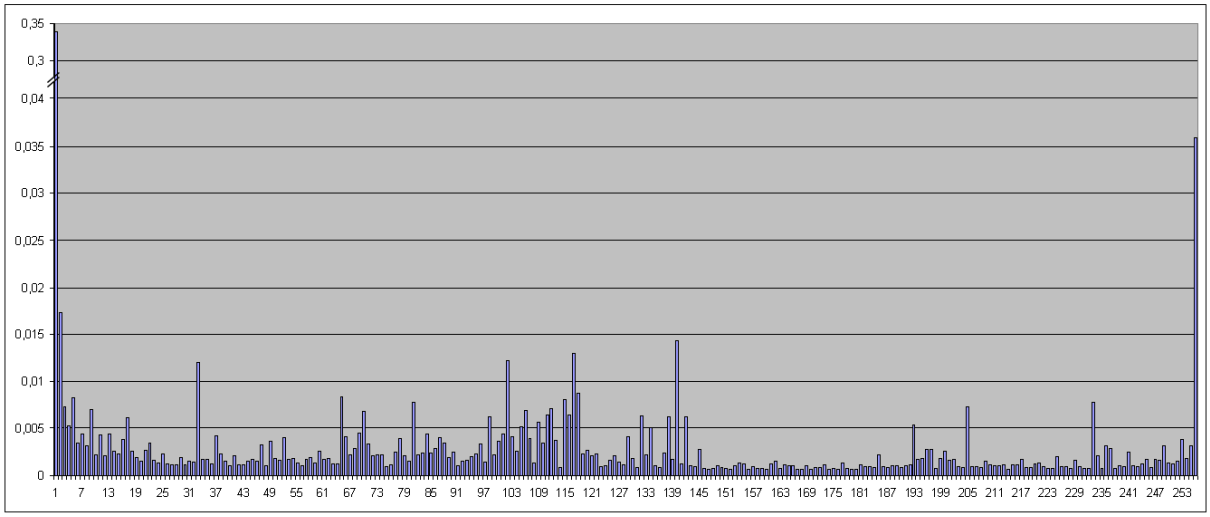

Figure A.3.1 Byte spectrum of EXE files

A.3.2 Byte spectrum of DLL files

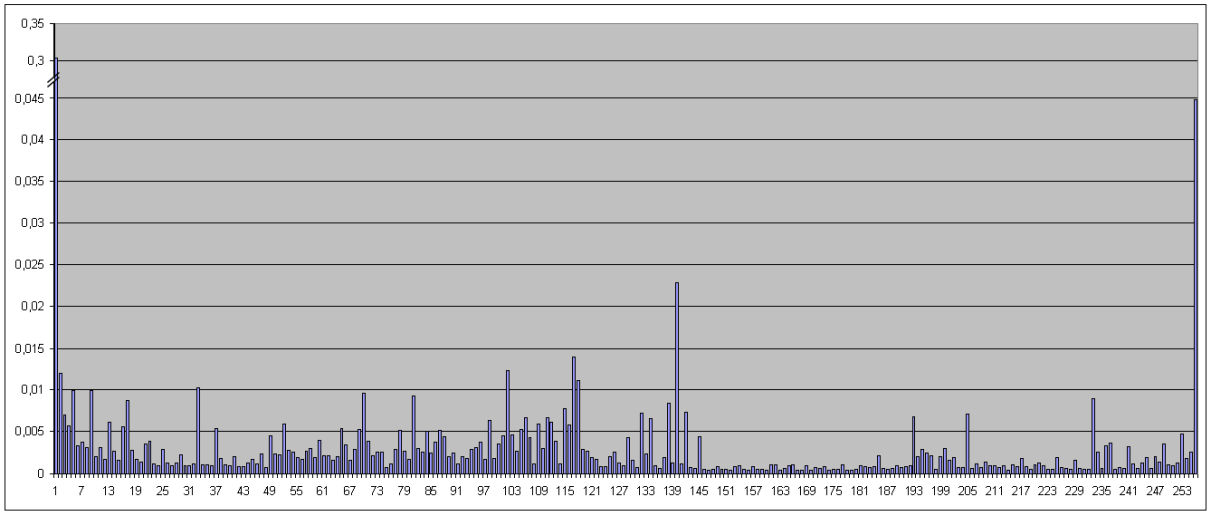

Figure A.3.2 Byte spectrum of DLL files

A.4 Byte Spectrum of the implemented filters

Both implemented filters had a high success rate. They were filtering out most of the desired packets. The purpose of implementing two filters was to reduce the high data load. A high portion of packets that are not of interest should be rejected before the preprocessor starts their inspection. The implementation of the filters was not intended to block for example all incoming text based packets.

The filters were implemented with a conservative threshold in order to miss not any of the possible malicious binary packets, but still catch and drop a lot of packets that are definitely not of interest. Therefore the preprocessor can train its detector set on a high portion of binary packets and the processing load is successfully reduced. The filters are applied by using the Pearson correlation coefficient.

A.4.1 Text filter

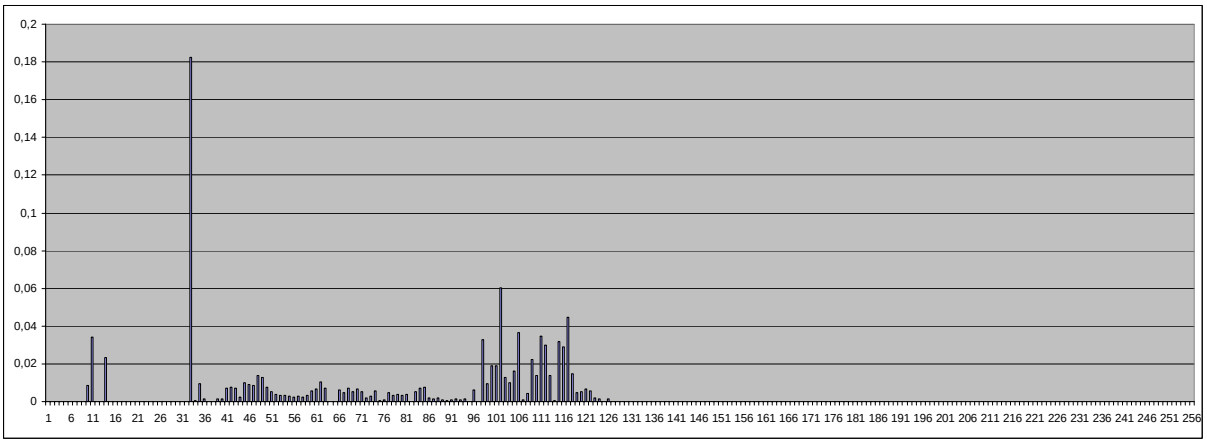

Figure A.4.1 Byte spectrum of the Text filter

A.4.2 Binary filter

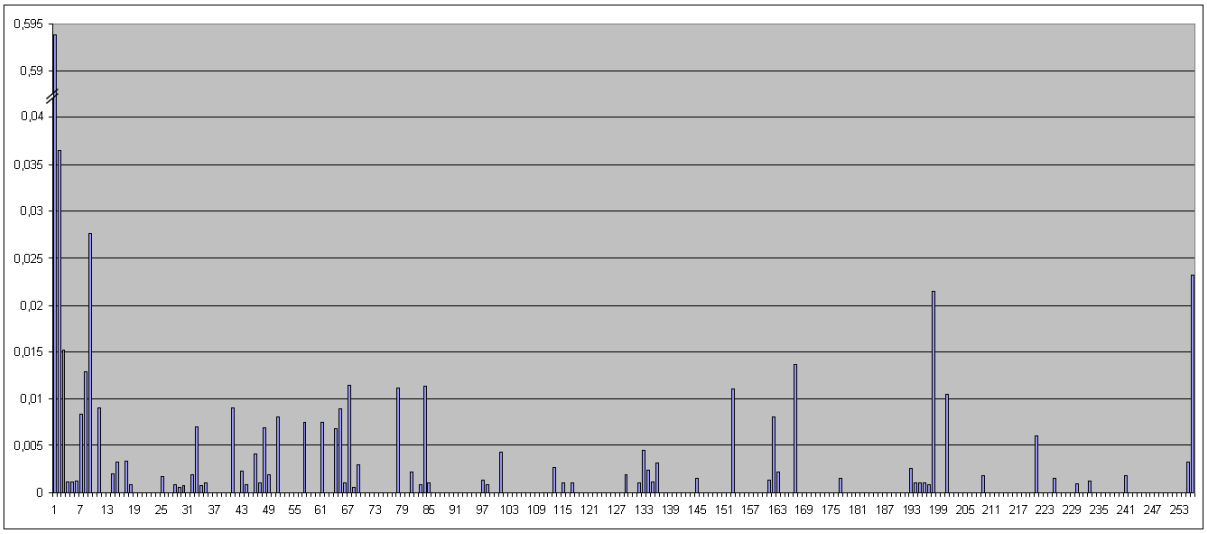

Figure A.4.2 Byte spectrum of the Binary filter

Appendix B

This appendix presents the numerical results of the tests on the match algorithms in chapter 5 and chapter 6. The charts presented in those chapters are based on the numbers in the tables given here. The columns of the tables are defined as follows:

- The **first column** in the following tables is the number of the current run. A run is the whole transfer of the TestData set, from the first file until the last one. The transfer of the TestData set is repeated without break until the training has been interrupted.
- The **second column** in the tables gives the number of replaced detectors during the tolerance phase. If the detectors have matched anything they were replaced.
- The **third column** is the count of packets which were processed by the preprocessor. When filters are used this column is splitted into "rejected" and "processed", because then a fraction of the packets are rejected by the filter. The number of processed packets is an indicator for the performance of the algorithm. The higher that number is the faster is the algorithm.
- The **last column** gives the age of the oldest detector after each run, measured in seconds.

To all configurations of the algorithms the drop rate of packets is calculated. The activated algorithms caused Snort to drop a fraction of the packets. This came from a slowed down packet processing rate. Therefore first the average number of packets is recorded which Snort received when the algorithms are turned off. This value is treated as reference for the calculation of the drop rate.

B.1 Results of the tested algorithms in chapter 5

B.1.1 First approach, without filters

One transfer of the TestData lasts on the average 5292 seconds and Snort received an average of 619684 packets during this time.

B.1.1.1 Pearson correlation coefficient

In the first configuration 5000 detectors were used. All of them have different values assigned to 30 positions of the spectrum. After their creation they were exposed to the simulated network traffic. The detectors were trained on two transfers of the TestData set. During this time Snort dropped 59.5% of the packets transferred. In the first run 908389 detectors were replaced due to significant similarity with the TestData set. In the second run 988553 detectors were replaced. The results in both runs were about the same. Not one of detectors survived a run, because otherwise in the second run the replacement rate of the detectors would have decreasing tendency. This conclusion can also be drawn from the age of the oldest detector which is much lower than the duration of the data transfer for one run.

Run	Replacements	Packets	Oldest Detector
1	908389	248930	3857
2	988553	253259	4385

Table B.1.1.1a: Pearson Correlation in the first configuration

The purpose of the next configuration was to check if the training of the detector set develops differently for a changed initial configuration. Therefore, 3000 detectors with 90 values were chosen and exposed to the simulated network traffic. The number of values of the spectrum per detectors was increased from 30 to 90, which may result in a better discrimination of the data. This time the detectors were trained on three transfers of the TestData set over the network. The drop rate of Snort was 62.6% of the packets. In each run at average 426580 detectors were replaced. The result after the third run was the same as in the first configuration shown above: Not one of the detectors lived long enough to expect a success of the training in further runs.

Run	Replacements	Packets	Oldest Detector
1	425970	224980	3939
2	427203	237264	3751
3	426567	231526	3911

Table B.1.1.1b: Pearson Correlation in the second configuration

In the third configuration the significance level was set to the definitely higher value of 0.99995 compared to a value of 0.975 in the previous two configurations. Therefore the detectors are now more specific than in the configurations before. In addition more values of the spectrum are used. Instead of 30 in the first configuration respectively 90 in the second now 220 values are generated. To improve the drop rate, the network speed was slowed down from 94.4 kB on the average to 40 kB per second. Snort's packet drop rate was now 35.4%. This configuration shows a nice learning effect on the detector set.

Run	Replacements	Packets	Oldest Detector
1	10864	655692	15016
2	6064	667390	30375
3	3523	670504	45415
4	2263	678016	60698
5	1524	682026	76091

Table B.1.1.1c: Pearson Correlation in the third configuration

B.1.1.2 Hamming Distance

The Hamming distance algorithm was tested for success by using three configurations. In the first the number of detectors was set to 600 with a length of 8 bytes. The training took place on four transfers of the TestData with a drop rate of 42.5% of the transferred packets. The threshold for a match was set to 3 bytes. The result shows that the training was not successful. The details are presented in the following table.

Run	Replacements	Packets	Oldest Detector
1	19284	342826	4003
2	19601	358175	4385
3	19549	353876	4239
4	18754	369555	4085

Table B.1.1.2a: Results of the Hamming distance algorithm using the first configuration.

In the second configuration the number of detectors was set to 300 with a length of 20 bytes. The training took place on seven transfers of the TestData. The drop rate comes to 66.9% of the transferred packets. Here, because of the longer byte sequences of the detector, the threshold for a match was increased to 4 bytes. The result of this configuration shows an instable progress in the training, with no indication for a lower replacement rate in future.

Run	Replacements	Packets	Oldest Detector
1	3329	196469	4177
2	3565	205747	4108
3	3263	206539	4645
4	3069	215767	7073
5	3259	210221	7911
6	3367	195359	4343
7	3200	195840	7312

Table B.1.1.2b: Results of the Hamming distance algorithm using the second configuration.

The third configuration was the same as the second except of the length of the detectors. The length was slightly decreased to 18 bytes. The purpose of decreasing the length by only two bytes was to check if the results are changing towards a stable training. The training took place on eight transfers of the TestData with a drop rate of 61.0% of the transferred packets. Unfortunately this configuration produces the same result as it was seen in the configuration before.

Run	Replacements	Packets	Oldest Detector
1	2242	207864	4679
2	2443	244061	6613
3	2505	248428	8320
4	2293	244861	4564
5	2309	242075	6994
6	2340	244161	4288
7	2367	249324	4353
8	2444	250868	4498

Table B.1.1.2c: Results of the Hamming distance algorithm using the third configuration.

B.1.1.3 r-contiguous bytes

The configuration of this match rule was equal to the first configuration of the hamming distance. The number of detectors was also set to 600 with 8 bytes length. The detectors were trained on 10 transfers of the TestData set over the network. During this time Snort dropped 51.8% of the transferred packets. The discriminator value r was set to 3. The detectors therefore have to match at least 3 contiguous bytes to become activated.
In the following table the number of replaced detectors is continual decreasing. At the same time the number of the processed packets increases slightly. That's because Snort needs to replace less detectors and can process more packets in the same time. The age of the oldest detector is increasing in all runs.

Run	Replacements	Packets	Oldest Detector
1	1490	286251	4804
2	1247	294480	9950
3	1015	296846	14976
4	965	299349	20062
5	946	304152	25223
6	866	303056	30371
7	811	295992	35401
8	759	305047	40588
9	685	294990	45610
10	651	304601	50821

Table B.1.1.3a: Results of the r-contiguous bytes algorithm using the first configuration.

The second configuration was using a set of 800 detectors. All other parameters are left unchanged. The detectors were trained on 10 transfers of the TestData set over the network. During this time Snort dropped 64.9% of the transferred packets. The results of this algorithm shows that algorithms are performing well independently form their drop rate.

Run	Replacements	Packets	Oldest Detector
1	1988	217091	4989
2	1624	217127	9988
3	1358	215998	14981
4	1031	217955	19979
5	792	218328	24989
6	581	218236	30002
7	546	217599	35010
8	624	218381	40025
9	483	216813	45121
10	463	218807	50194

Table B.1.1.3b: Results of the r-contiguous bytes algorithm using the second configuration.

B.1.1.4 Levenshtein distance or edit distance

The implementation of this algorithm was quite slow. The number of detectors was set to 100 with a length of 8 bytes. The threshold for a match was set to a distance of 5 operations, which

corresponds to a similarity of 3 bytes. The detectors were trained on five transfers of the TestData set over the network. During this time Snort dropped 84.0% of the transferred packets.

Run	Replacements	Packets	Oldest Detector
1	1029	98597	4566
2	950	98427	4343
3	1085	98202	4279
4	891	99706	4795
5	1077	98459	4563

Table B.1.1.4a: Results of the Levenshtein distance or edit distance algorithm.

B.1.1.5 Longest common subsequence (LCS)

Two slightly different configurations have been implemented. The first uses a set of 130 detectors with a length 18 bytes. The threshold for a match was set to a value of 11 bytes. The detectors were trained on seven transfers of the TestData set over the network. During this time Snort dropped 89.3% of the transferred packets.

Run	Replacements	Packets	Oldest Detector
1	676	66324	4789
2	680	66802	4822
3	702	65223	7289
4	653	67138	8678
5	714	65602	7425
6	757	65869	9513
7	644	66155	8488

Table B.1.1.5a: Results of the Longest common subsequence algorithm using the first configuration.

The second configuration uses a smaller detector set (100) to increase speed and a detector length of 17 bytes. The algorithm was tested during eight TestData transfers. The drop rate of Snort was 86.2%. This configuration is a more stable version of the one before.

Run	Replacements	Packets	Oldest Detector
1	308	85630	4668
2	349	83689	9373
3	337	86344	13016
4	341	83457	13728
5	323	84346	14063
6	254	83043	18747
7	362	87434	23538
8	322	87674	13577

Table B.1.1.5b: Results of the Longest common subsequence algorithm using the second configuration.

B.1.2 Second approach (using filters)

When Snort was used with the filters it captured 1038957 packets. 421203 were forwarded for processing and 617754 were rejected. This ratio can not directly be applied to the transferred amount of data, because the average packet size of the rejected packets is significantly smaller (see chapter 5, some packets do not contain file data). An approximate value for the processed packets is around 420 MB compared to the effective transferred 511 MB. The time for one transfer of the TestData set was on the average 15474 seconds.

B.1.2.1 The Pearson correlation coefficient

The parameters of the Pearson correlation were set to the same values as they have been in the third configuration of the first approach. Here the significance level was set to the highest value (0.99995) of the four implemented tables. The size of the detector set was 3000, with an individual length of 220 values. Snort trained the detector set during four transfers of the TestData set and it can be seen that it has almost the same learning effect compared with the results of the first approach. Snorts drop rate was definitely low with a value of 2.1%.

Run	Replacements	Packets		Oldest Detector
		processed	rejected	
1	10941	409529	612004	15094
2	6190	414580	610997	30318
3	3688	411421	615097	45543
4	2344	412980	617796	60786

Table B.1.2.1: Results of the Pearson Correlation with packet filters.

B.1.2.2 Hamming Distance

This time the Hamming distance algorithm was training a set of 500 detectors, each equipped with 10 bytes. The threshold for a match remained unchanged with 4 bytes. Snorts drop packet rate was 19.4%. This training process on the seven runs was interesting, because the detectors are getting continually older, but the replacement rate stayed almost the same. It can be seen that it is slightly decreasing towards the seventh run. This result came from a continually increasing number of detectors that are probably tolerant to the presented TestData set. But after the seventh run this number is still low.

Run	Replacements	Packets		Oldest Detector
		processed	rejected	
1	3139	339624	478365	15131
2	2995	334369	512381	30382
3	3046	337424	511949	45526
4	2946	343014	513033	60676
5	3025	343991	519782	75760
6	2901	339083	473294	91073
7	2862	339372	487247	106347

Table B.1.2.2: Results of the Hamming distance algorithm with packet filters.

B.1.2.3 r-contiguous bytes

The third tested match rule was the r-contiguous bytes algorithm. It worked in the first approach and therefore it will be interesting to see if it also trains the detector set on the new conditions. Three configurations have been implemented. The first is using a set of 800 detectors with length 8 bytes. The discriminator value r was set to 3. The detectors were trained on three transfers of the TestData set. During this time snort dropped 37.5% of for the processing intended packets. Not one detector lives longer than a third part of the training time. Therefore the training can not be assumed to be successful in further runs.

Run	Replacements	Packets		Oldest Detector
		processed	rejected	
1	30987	266840	390284	5017
2	30581	267009	396000	5526
3	31076	255809	361169	4030

Table B.1.2.3a: Results of the r-contiguous bytes algorithm with packet filters, using the first configuration.

The second configuration uses a set of 2000 detectors with length 4 bytes. The discriminator value r remains at the value 3. The detectors were trained on 9 transfers of the TestData set. During this time Snort dropped 57.3% of for the processing intended packets. The progress of the training was instable.

Run	Replacements	Packets		Oldest Detector
		processed	rejected	
1	17325	186269	347966	15063
2	17787	189940	359788	26135
3	17777	189401	370579	30147
4	14815	143452	330645	22032
5	14449	167286	339332	21392
6	17181	187339	371473	21263
7	17064	184055	356877	25093
8	17887	189791	362712	21515
9	17547	184724	346102	14849

Table B.1.2.3b: Results of the r-contiguous bytes algorithm with packet filters, using the second configuration.

Finally the third configuration uses a set of 250 detectors with a length of 16 bytes. The discriminator value r was set to 4. The detectors were trained on 2 transfers of the TestData set. During this time Snort dropped 32.4% of the packets. As it can be seen in the following table, the training was successful.

Run	Replacements	Packets		Oldest Detector
		processed	rejected	
1	238	296284	276743	15458
2	84	273113	241395	29817

Table B.1.2.3c: Results of the r-contiguous bytes algorithm with packet filters, using the third configuration.

B.1.2.4 Longest common subsequence (LCS)

The parameters for the LCS algorithm were set to the same values as they have been in the second configuration of the first approach. The size of the detector set was 100, with a length of 17 bytes. The threshold for a match was again 11 bytes. Unfortunately although the network speed was reduced it had again a low performance with a packet drop rate of 84.5%. This time the training was stable but not successful, because the age of the oldest detector was less than a third of the training duration. This was up to a definitely higher number of replaced immature detectors.

Run	Replacements	Packets		Oldest Detector
		processed	rejected	
1	4650	64391	185455	4919
2	4462	65920	224724	4891
3	4775	65892	222770	4677
4	4558	64600	229027	4995

Table B.1.2.4a: Results of the Longest Common Subsequence algorithm with packet filters, using the first configuration.

The second configuration was using a detector set with the same size and length as in the first configuration. The threshold for a match was increased to 12 bytes. This time the packet drop rate was 79.8%. This training success is developing similar to the result of the Hamming Distance presented in B.1.2.2. The detectors are getting older but the replacement rate stays almost the same and has no decreasing tendency. This result came from a low number of detectors that seem to be already tolerant the presented TestData set. The training of the detectors is based on 8 transfers of the TestData set.

Run	Replacements	Packets		Oldest Detector
		processed	rejected	
1	447	84641	61773	15511
2	461	84124	61898	24999
3	428	85555	61353	37875
4	397	84397	60000	35215
5	460	85932	61526	43999
6	327	85324	106812	37561
7	414	83613	62948	56386
8	450	86253	60611	75904

Table B.1.2.4b: Results of the Longest Common Subsequence algorithm with packet filters, using the second configuration.

B.2 Results of the tested algorithm in chapter 6

In chapter 5 the data basis for the match algorithms were byte sequences in the payload. In chapter 6 the new representation is the numbers of disassembled instructions. Three configurations have been tested on their success in training a detector set. The detector set was trained with the r-contiguous instructions algorithm.
The first configuration uses 1000 detectors equipped with 8 instructions. The discriminator value r in all three configurations was set to 2. In the first configuration the detectors were trained on five transfers of the TestData set and the packet filter was turned off. During this time Snort dropped 77.6% of for the processing intended packets. The training was successful.

Run	Replacements	Packets	Oldest Detector
1	618	135889	4937
2	105	136926	10003
3	57	138596	15031
4	26	140140	20066
5	32	140986	25091

Table B.2a: Results of the r-contiguous instructions algorithm using the first configuration without packet filters.

The second configuration again uses 1000 detectors holding 8 instructions. The only difference in the first and the second configuration was that in the second the packet filter was turned on. The detectors were trained on three transfers of the TestData set. During this time Snort dropped 72.0% of for the processing intended packets. This training was successful.

		Packets		
Run	Replacements	processed	rejected	Oldest Detector
1	773	74443	279526	5228
2	61	76558	281534	10291
3	17	76863	283453	15357

Table B.2b: Results of the r-contiguous instructions algorithm using the second configuration with packet filters.

In the third configuration the network speed was slowed down again to the 40 kB per second. The packet filter was also turned on. Here, the number of detectors was increased to 2000 with a length of 5 instructions. The detectors were trained on five transfers of the TestData set. During this time Snort dropped 31.3% of for the processing intended packets. Also the third configuration has been successful.

		Packets		
Run	Replacements	processed	rejected	Oldest Detector
1	1876	285157	422310	15422
2	189	293154	456374	30707
3	22	296867	450068	46338
4	19	296379	469710	61809
5	7	275208	423116	76105

Table B.2c: Results of the r-contiguous instructions algorithm using the third configuration with packet filters.

Appendix C

This appendix presents the assembler instruction frequency spectrum which is produced when different file types are being disassembled.

- In the **first section** three text based file types are analysed, which are CPP Source Code Files, HTM Web pages and TXT Files.
- The **second section** contains the analysis of five binary based file formats. Included are two picture formats commonly used for Web pages (GIF and JPG). This section includes also the Adobe portable document format (PDF), a common archive file format (ZIP) and a music file format (MP3).
- **Section three** shows the instruction distribution of two Binary Executable File Formats (EXE and DLL).
- The **last section** presents the spectrum used by a high number of computer viruses. These files have been collected from past threats which are reaching back until the beginning of the Microsoft Windows area.

C.1 Text based file types

The text based files all are using the same instruction set. This can be seen if the instruction distribution in the spectrum is compared. When these files are disassembled the resulting instruction range is very limited. This result was expected from the byte spectrum analysis in Appendix A. There the observed range of the characters was also limited.

File type	Number of files	Disassembled instructions
CPP	2235	11637493
TXT	3955	149139095
HTM	2763	21462699

Table C.1: Text based file types

C.1.1 Instruction spectrum of CPP Source Files

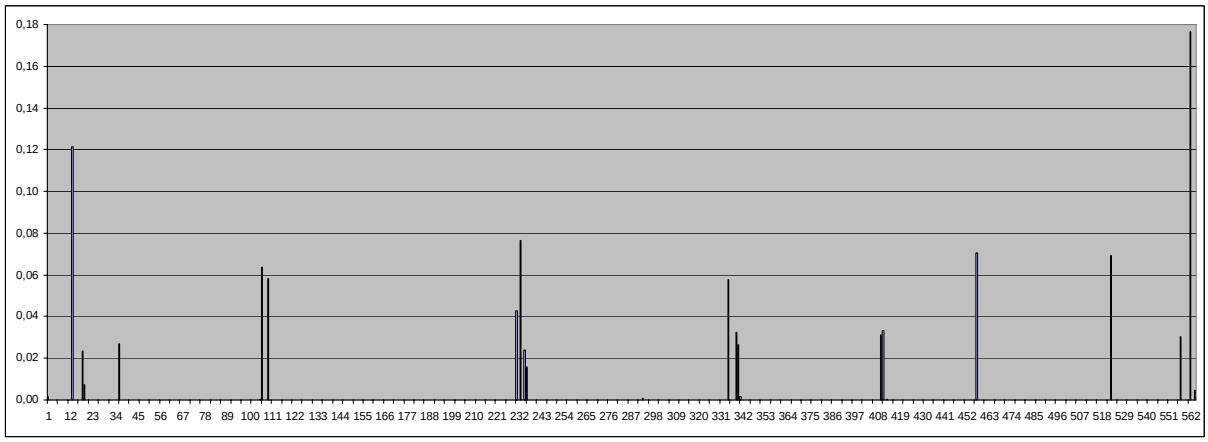

Figure C.1.1 Instruction spectrum of CPP Source files

C.1.2 Instruction spectrum of TXT Files

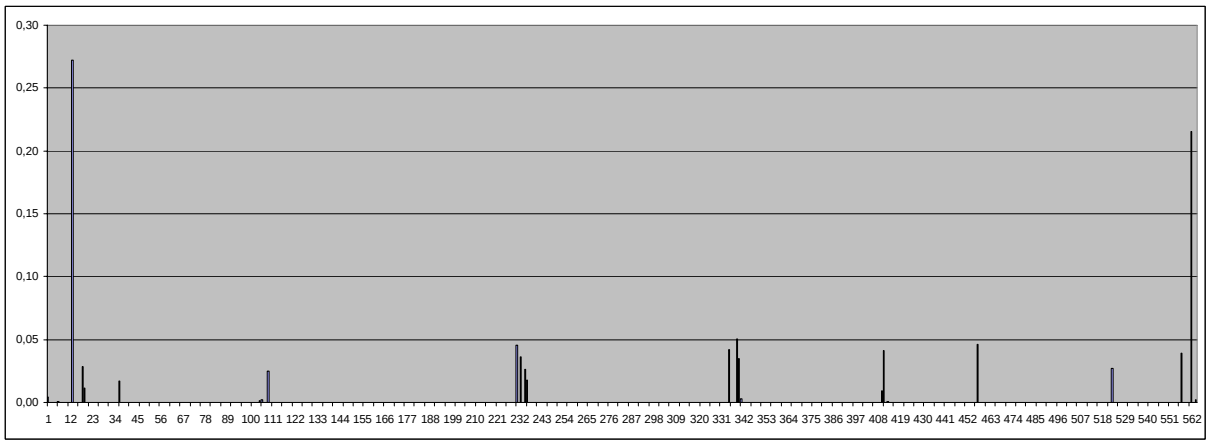

Figure C.1.2 Instruction spectrum of TXT files

C.1.3 Instruction spectrum of HTM Pages

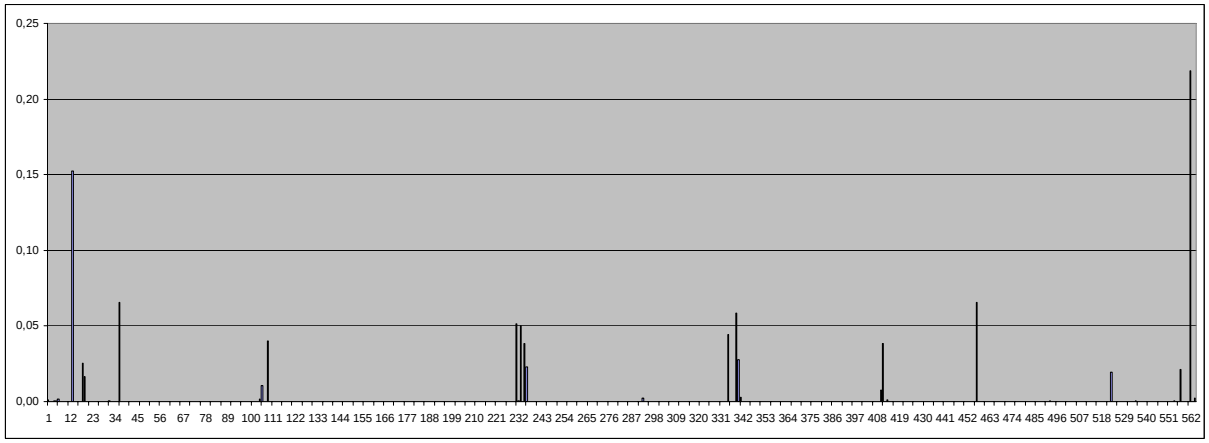

Figure C.1.3 Instruction spectrum of HTM Pages

C.2 Binary based file types

The disassembly of binary files resulted in a broader spectrum of instructions. The Intel x86 processor supports 563 instructions. Most of them are produced with a very low occurrence rate.

File type	Number of files	Disassembled instructions
GIF	2554	8561282
JPG	5541	287398930
MP3	272	678829903
PDF	1993	356274169
ZIP	3657	427752106

Table C.2: Binary based file types

C.2.1 Instruction spectrum of GIF Image files

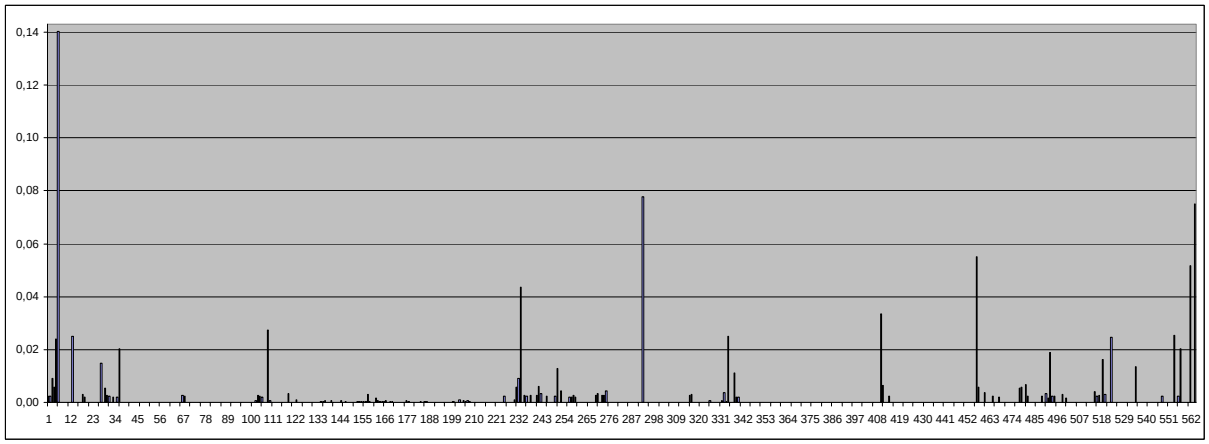

Figure C.2.1 Instruction spectrum of GIF Image files

C.2.2 Instruction spectrum of JPG Image files

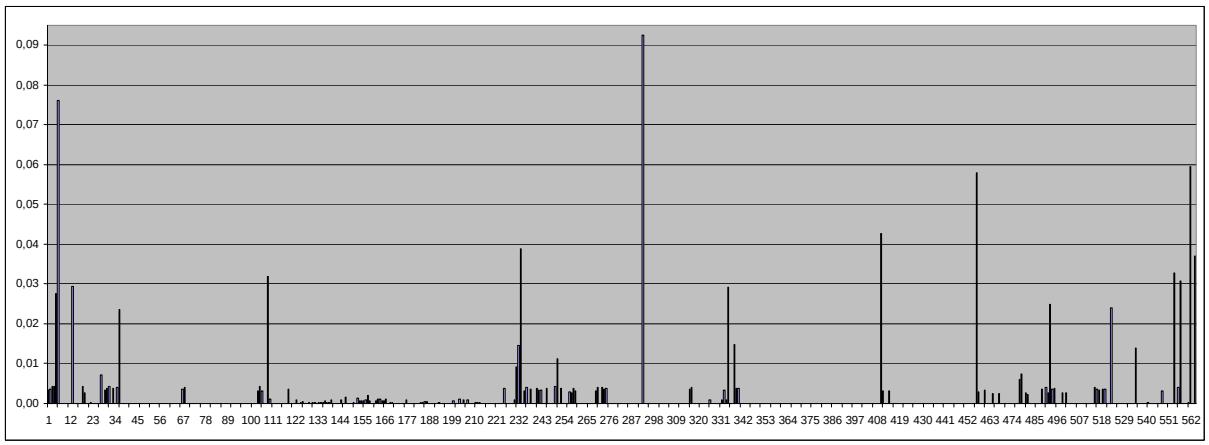

Figure C.2.2 Instruction spectrum of JPG Image files

C.2.3 Spectrum of MP3 Music files

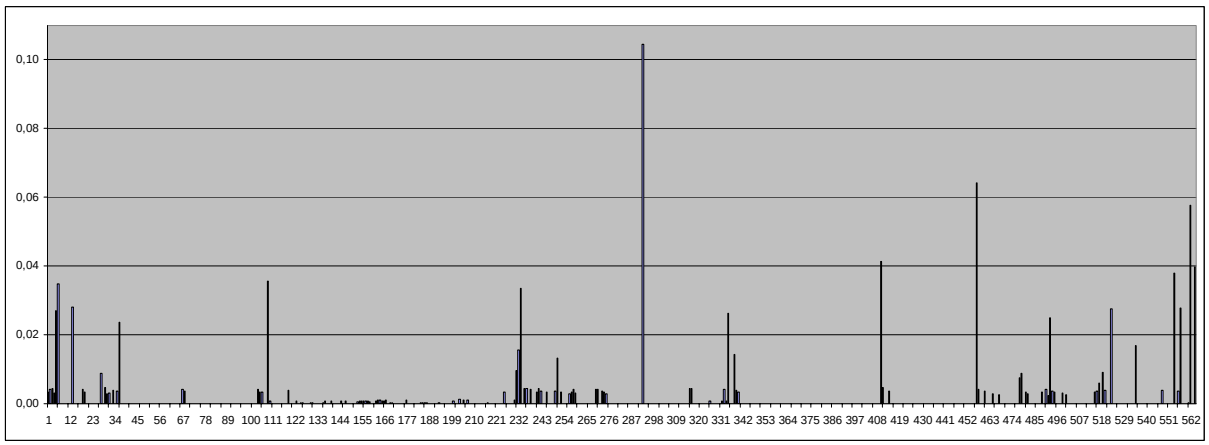

Figure C.2.3 Instruction spectrum of MP3 Music files

C.2.4 Instruction spectrum of PDF files

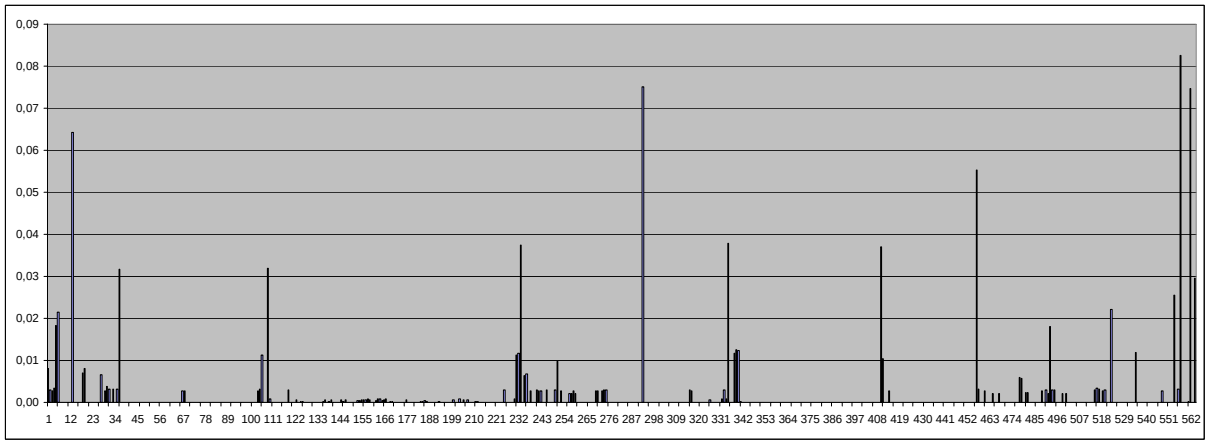

Figure C.2.4 Instruction spectrum of PDF files

C.2.5 Instruction spectrum of ZIP Archive files

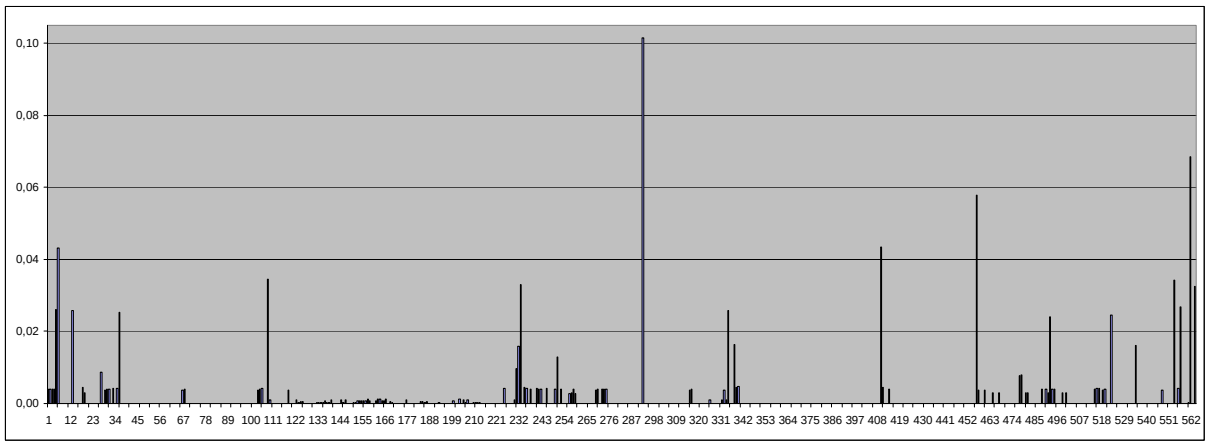

Figure C.2.5 Instruction spectrum of ZIP Archive files

C.3 Executable file types

If the disassembled instruction spectrum between binary file types and executable files is compared, it can be noticed that some noise in the spectrum is removed. The individual bars appear clearer. The ADD instruction (Instruction number 5) in executable file types is observed with a much higher probability than in non-executable file types. The ADD operation was produced in 32.8% (EXE) and 31% (DLL) of all instructions. This results from the section padding in PE Files.

File type	Number of files	Disassembled instructions
EXE	2752	376588208
DLL	4541	492301150

Table C.3: Executable file types

C.3.1 Instruction spectrum of EXE files

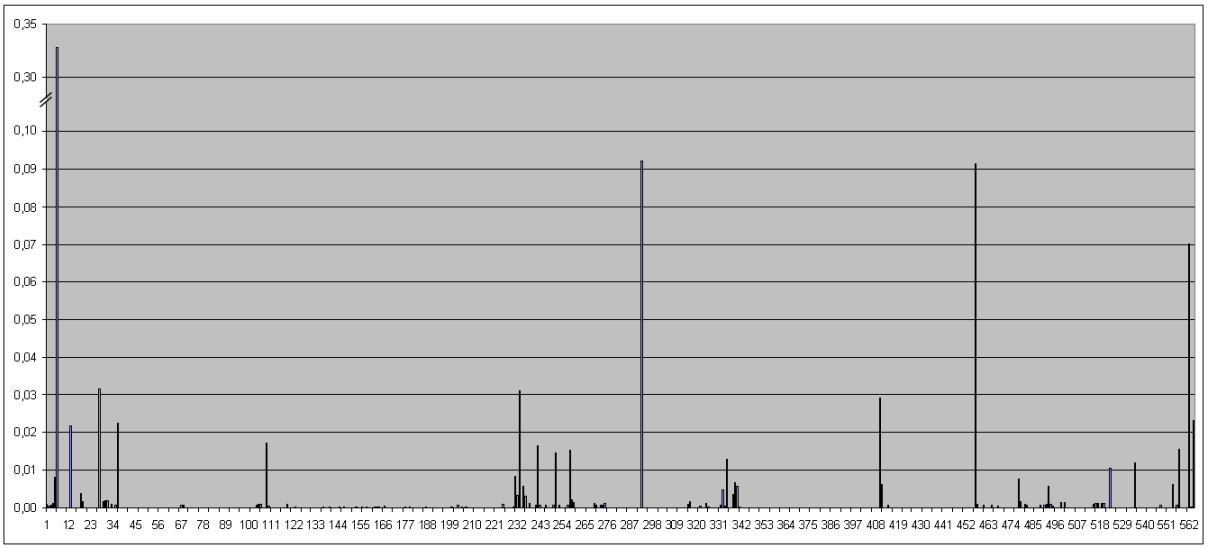

Figure C.3.1 Instruction spectrum of EXE files

C.3.2 Instruction spectrum of DLL files

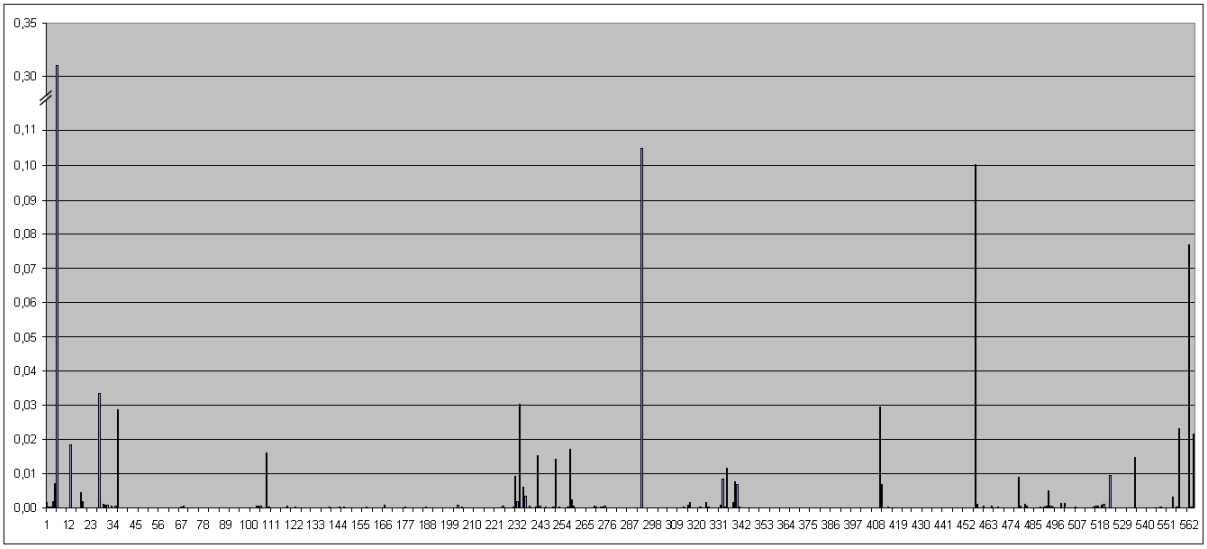

Figure C.3.2 Instruction spectrum of DLL files

C.4 Virus files

The spectrum of the virus files does not significantly differ from the executable files presented in section 3. The exception is the occurrence of the instruction with the number 333 in EXE Virus files. This instruction is the NOP instruction and it had been observed that in early Windows viruses some quite large NOP zones are present.

File type	Number of files	Disassembled instructions
COM	6337	8581825
EXE	1470	7714879
PIF	44	773372
CPL	8	83657

Table C.4: Virus files

C.4.1 Instruction spectrum of COM Virus files

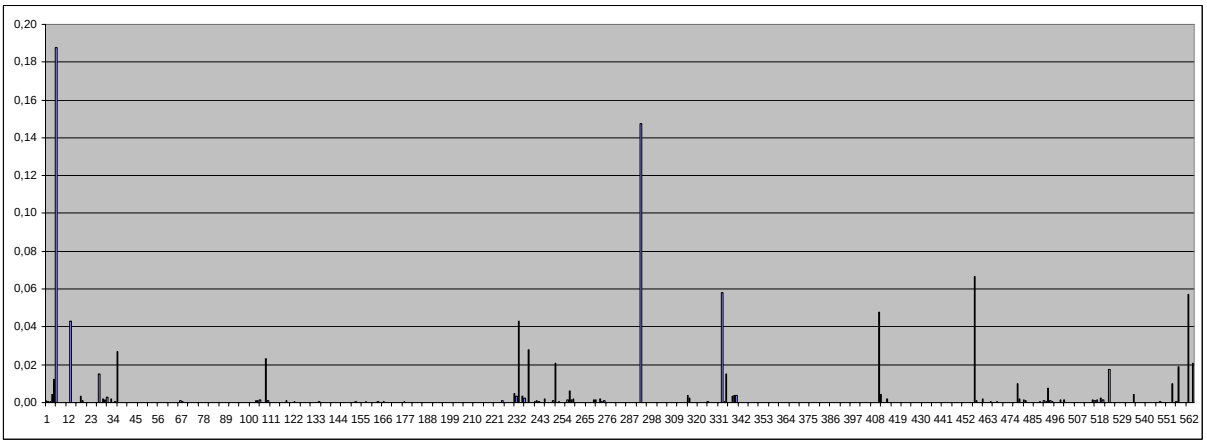

Figure C.4.1 Instruction spectrum of COM Virus files

C.4.2 Instruction spectrum of EXE Virus files

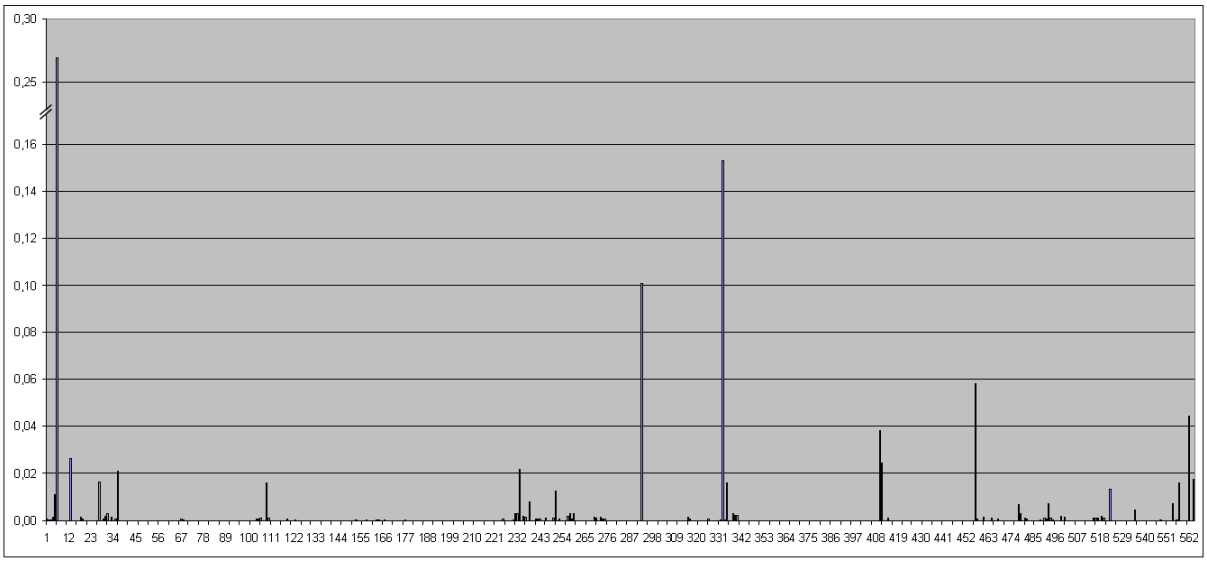

Figure C.4.2 Instruction spectrum of EXE Virus files

C.4.3 Instruction spectrum of PIF Virus files

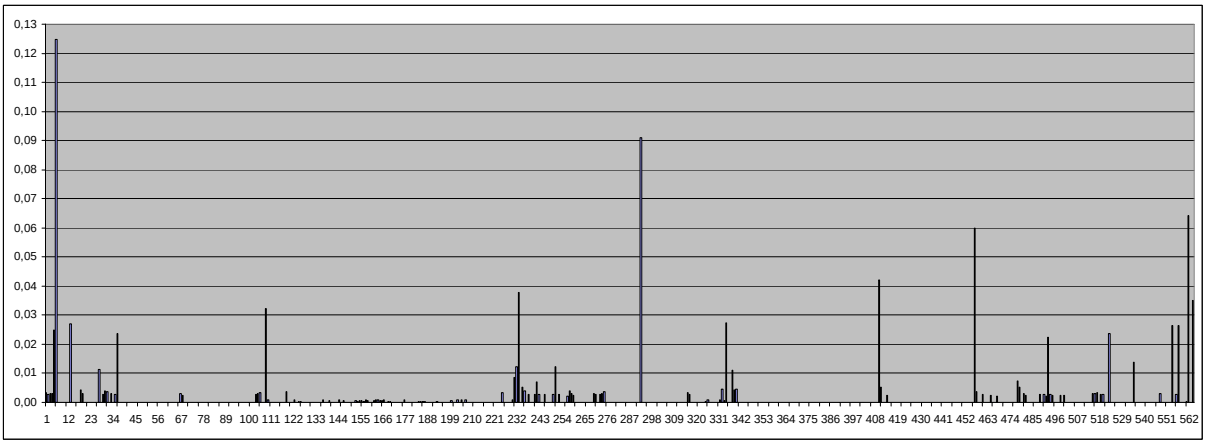

Figure C.4.3 Instruction spectrum of PIF Virus files

C.4.4 Instruction spectrum of CPL Virus files

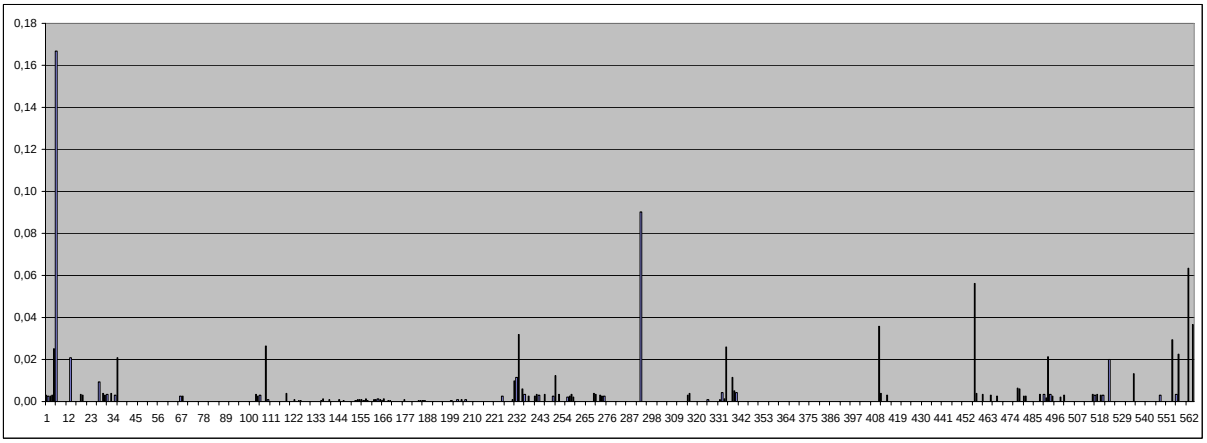

Figure C.4.4 Instruction spectrum of CPL Virus files

Appendix D

This appendix shows examples of the entry point effect, which occurs when code is being disassembled without knowing where the code really starts. The hypothesis is that even if the exact entry point is missed, a correct instruction sequence will be reproduced after a few wrongly disassembled instructions.
Here a code example is presented, where the disassembly process was started at a random offset within an executable file. From there the offset of the entry point was increased between 1 and 17 bytes with a step of one byte. The consequences in the produced instruction sequence are shown.

The code fragment on the left side is always the same. It is the disassembled reference instruction sequence (started at the entry point with offset 0). The code on the right side was produced with an offset in the start of the disassembly process. It has a length of 100 instructions.
For each example a table with the results of the analysis is included.

- The first row gives the **data offset** (entry point) for the disassembled instruction sequence on the right side.
- The second row shows the **number of equal instructions**. This is the number of instructions after the sequences are completely identical again.
- The **absolute equal instructions** in the third row gives the total number of identical instructions. At the start of the assembly process it could happen that a different opcode results in the same instruction, but the next byte is disassembled wrong (see example #11). This row counts the number of matching instructions in both sequences regardless of holes in it. The value therefore is always equal or higher than the number of equal instructions.
- The last row reports the **relative position** in the new disassembled instruction sequence where the sequences start to be equal (without holes). It counts the number of how many more instructions are generated in the window compared to the RIS before the point of synchronisation.

In the following examples a part of the disassembled sequence is marked with grey background. This indicates the areas where both sequences are equal. The only difference which remains in the marked sequence is the data of the JMP instruction (Number 561). This difference in the disassembled instruction comes from the relative address coding.

Entry point: +1:

```
LN INS OPCODE            INSTRUCTION
 0 561 7600              jna 0x441
 1 005 0000              add [eax],al
 2 005 00501E            add [eax+0x1e],dl
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 005 0000              add [eax],al
 1 005 0000              add [eax],al
 2 456 50                push eax
 3 456 1E                push ds
 4 035 807C29B580        cmp byte [ecx
 5 561 7C2F              jl 0x3c
 6 334 08817C8A2B86      or [ecx+0x862
 7 561 7C16              jl 0x2b
 8 456 1E                push ds
 9 035 807CE51780        cmp byte [ebp
10 561 7C11              jl 0x2e
11 409 59                pop ecx
12 563 82                db 0x82
13 561 7C0E              jl 0x2f
14 492 18807C4E9980      sbb [eax+0x80
15 561 7C9F              jl 0xffffffc8
16 561 0F817CCC4083      jno near 0x83
17 561 7C98              jl 0xffffffc9
```

Data offset	1
# equal instructions	96
absolute equal instructions	96
relative position	1

Table D1: Entry point +1

Entry point: +2:

```
LN INS OPCODE            INSTRUCTION
 1 005 0000              add [eax],al
 2 005 00501E            add [eax+0x1e],dl
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 005 0000              add [eax],al
 1 005 00501E            add [eax+0x1e
 2 035 807C29B580        cmp byte [ecx
 3 561 7C2F              jl 0x3b
 4 334 08817C8A2B86      or [ecx+0x862
 5 561 7C16              jl 0x2a
 6 456 1E                push ds
 7 035 807CE51780        cmp byte [ebp
 8 561 7C11              jl 0x2d
 9 409 59                pop ecx
10 563 82                db 0x82
11 561 7C0E              jl 0x2e
12 492 18807C4E9980      sbb [eax+0x80
13 561 7C9F              jl 0xffffffc7
14 561 0F817CCC4083      jno near 0x83
15 561 7C98              jl 0xffffffc8
```

Data offset	2
# equal instructions	100
absolute equal instructions	100
relative position	0

Table D2: Entry point +2

Entry point: +3:

```
LN INS OPCODE            INSTRUCTION
 1 005 0000              add [eax],al
 2 005 00501E            add [eax+0x1e],dl
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 005 0000              add [eax],al
 1 456 50                push eax
 2 456 1E                push ds
 3 035 807C29B580        cmp byte [ecx
 4 561 7C2F              jl 0x3a
 5 334 08817C8A2B86      or [ecx+0x862
 6 561 7C16              jl 0x29
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp
 9 561 7C11              jl 0x2c
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x2d
13 492 18807C4E9980      sbb [eax+0x80
14 561 7C9F              jl 0xffffffc6
15 561 0F817CCC4083      jno near 0x83
16 561 7C98              jl 0xffffffc7
```

Data offset	3
# equal instructions	97
absolute equal instructions	97
relative position	1

Table D3: Entry point +3

Entry point: +4:

```
LN INS OPCODE            INSTRUCTION
 2 005 00501E            add [eax+0x1e],dl
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 005 00501E            add [eax+0x1e
 1 035 807C29B580        cmp byte [ecx
 2 561 7C2F              jl 0x39
 3 334 08817C8A2B86      or [ecx+0x862
 4 561 7C16              jl 0x28
 5 456 1E                push ds
 6 035 807CE51780        cmp byte [ebp
 7 561 7C11              jl 0x2b
 8 409 59                pop ecx
 9 563 82                db 0x82
10 561 7C0E              jl 0x2c
11 492 18807C4E9980      sbb [eax+0x80
12 561 7C9F              jl 0xffffffc5
13 561 0F817CCC4083      jno near 0x83
14 561 7C98              jl 0xffffffc6
```

Data offset	4
# equal instructions	100
absolute equal instructions	100
relative position	0

Table D4: Entry point +4

Entry point: +5:

```
LN INS OPCODE            INSTRUCTION
 2 005 00501E            add [eax+0x1e],dl
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 456 50                push eax
 1 456 1E                push ds
 2 035 807C29B580        cmp byte [ecx
 3 561 7C2F              jl 0x38
 4 334 08817C8A2B86      or [ecx+0x862
 5 561 7C16              jl 0x27
 6 456 1E                push ds
 7 035 807CE51780        cmp byte [ebp
 8 561 7C11              jl 0x2a
 9 409 59                pop ecx
10 563 82                db 0x82
11 561 7C0E              jl 0x2b
12 492 18807C4E9980      sbb [eax+0x80
13 561 7C9F              jl 0xffffffc4
14 561 0F817CCC4083      jno near 0x83
15 561 7C98              jl 0xffffffc5
```

Data offset	5
# equal instructions	98
absolute equal instructions	98
relative position	1

Table D5: Entry point +5

Entry point: +6:

```
LN INS OPCODE            INSTRUCTION
 2 005 00501E            add [eax+0x1e],dl
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 456 1E                push ds
 1 035 807C29B580        cmp byte [ecx
 2 561 7C2F              jl 0x37
 3 334 08817C8A2B86      or [ecx+0x862
 4 561 7C16              jl 0x26
 5 456 1E                push ds
 6 035 807CE51780        cmp byte [ebp
 7 561 7C11              jl 0x29
 8 409 59                pop ecx
 9 563 82                db 0x82
10 561 7C0E              jl 0x2a
11 492 18807C4E9980      sbb [eax+0x80
12 561 7C9F              jl 0xffffffc3
13 561 0F817CCC4083      jno near 0x83
14 561 7C98              jl 0xffffffc4
```

Data offset	6
# equal instructions	99
absolute equal instructions	99
relative position	0

Table D6: Entry point +6

Entry point: +7:

```
LN INS OPCODE            INSTRUCTION
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 035 807C29B580        cmp byte [ecx
 1 561 7C2F              jl 0x36
 2 334 08817C8A2B86      or [ecx+0x862
 3 561 7C16              jl 0x25
 4 456 1E                push ds
 5 035 807CE51780        cmp byte [ebp
 6 561 7C11              jl 0x28
 7 409 59                pop ecx
 8 563 82                db 0x82
 9 561 7C0E              jl 0x29
10 492 18807C4E9980      sbb [eax+0x80
11 561 7C9F              jl 0xffffffc2
12 561 0F817CCC4083      jno near 0x83
13 561 7C98              jl 0xffffffc3
```

Data offset	7
# equal instructions	100
absolute equal instructions	100
relative position	0

Table D7: Entry point +7

Entry point: +8:

```
LN INS OPCODE            INSTRUCTION
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 561 7C29              jl 0x2b
 1 292 B580              mov ch,0x80
 2 561 7C2F              jl 0x35
 3 334 08817C8A2B86      or [ecx+0x862
 4 561 7C16              jl 0x24
 5 456 1E                push ds
 6 035 807CE51780        cmp byte [ebp
 7 561 7C11              jl 0x27
 8 409 59                pop ecx
 9 563 82                db 0x82
10 561 7C0E              jl 0x28
11 492 18807C4E9980      sbb [eax+0x80
12 561 7C9F              jl 0xffffffc1
13 561 0F817CCC4083      jno near 0x83
14 561 7C98              jl 0xffffffc2
```

Data offset	8
# equal instructions	98
absolute equal instructions	98
relative position	1

Table D8: Entry point +8

Entry point: +9:

```
LN INS OPCODE            INSTRUCTION
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 522 29B5807C2F08      sub [ebp+0x82
 1 035 817C8A2B867C161E  cmp dword [ed
 2 035 807CE51780        cmp byte [ebp
 3 561 7C11              jl 0x26
 4 409 59                pop ecx
 5 563 82                db 0x82
 6 561 7C0E              jl 0x27
 7 492 18807C4E9980      sbb [eax+0x80
 8 561 7C9F              jl 0xffffffc0
 9 561 0F817CCC4083      jno near 0x83
10 561 7C98              jl 0xffffffc1
```

Data offset	9
# equal instructions	95
absolute equal instructions	95
relative position	-3

Table D9: Entry point +9

Entry point: +10:

```
LN INS OPCODE            INSTRUCTION
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 292 B580              mov ch,0x80
 1 561 7C2F              jl 0x33
 2 334 08817C8A2B86      or [ecx+0x862
 3 561 7C16              jl 0x22
 4 456 1E                push ds
 5 035 807CE51780        cmp byte [ebp
 6 561 7C11              jl 0x25
 7 409 59                pop ecx
 8 563 82                db 0x82
 9 561 7C0E              jl 0x26
10 492 18807C4E9980      sbb [eax+0x80
11 561 7C9F              jl 0xffffffbf
12 561 0F817CCC4083      jno near 0x83
13 561 7C98              jl 0xffffffc0
```

Data offset	10
# equal instructions	99
absolute equal instructions	99
relative position	0

Table D10: Entry point +10

Entry point: +11:

```
LN INS OPCODE            INSTRUCTION
 3 035 807C29B580        cmp byte [ecx+ebp-0x4b],0x80
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 035 807C2F0881        cmp byte [edi
 1 561 7C8A              jl 0xffffff91
 2 522 2B867C161E80      sub eax,[esi+
 3 561 7CE5              jl 0xfffffff4
 4 409 17                pop ss
 5 035 807C115982        cmp byte [ecx
 6 561 7C0E              jl 0x25
 7 492 18807C4E9980      sbb [eax+0x80
 8 561 7C9F              jl 0xffffffbe
 9 561 0F817CCC4083      jno near 0x83
10 561 7C98              jl 0xffffffbf
```

Data offset	11
# equal instructions	91
absolute equal instructions	93
relative position	1

Table D11: Entry point +11

Entry point: +12:

```
LN INS OPCODE            INSTRUCTION
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 561 7C2F              jl 0x31
 1 334 08817C8A2B86      or [ecx+0x862
 2 561 7C16              jl 0x20
 3 456 1E                push ds
 4 035 807CE51780        cmp byte [ebp
 5 561 7C11              jl 0x23
 6 409 59                pop ecx
 7 563 82                db 0x82
 8 561 7C0E              jl 0x24
 9 492 18807C4E9980      sbb [eax+0x80
10 561 7C9F              jl 0xffffffbd
11 561 0F817CCC4083      jno near 0x83
12 561 7C98              jl 0xffffffbe
```

Data offset	12
# equal instructions	100
absolute equal instructions	100
relative position	0

Table D12: Entry point +12

Entry point: +13:

```
LN INS OPCODE            INSTRUCTION
 4 561 7C2F              jl 0x47c
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 105 2F                das
 1 334 08817C8A2B86      or [ecx+0x862
 2 561 7C16              jl 0x1f
 3 456 1E                push ds
 4 035 807CE51780        cmp byte [ebp
 5 561 7C11              jl 0x22
 6 409 59                pop ecx
 7 563 82                db 0x82
 8 561 7C0E              jl 0x23
 9 492 18807C4E9980      sbb [eax+0x80
10 561 7C9F              jl 0xffffffbc
11 561 0F817CCC4083      jno near 0x83
12 561 7C98              jl 0xffffffbd
```

Data offset	13
# equal instructions	99
absolute equal instructions	99
relative position	0

Table D13: Entry point +13

Entry point: +14:

```
LN INS OPCODE            INSTRUCTION
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 334 08817C8A2B86      or [ecx+0x862
 1 561 7C16              jl 0x1e
 2 456 1E                push ds
 3 035 807CE51780        cmp byte [ebp
 4 561 7C11              jl 0x21
 5 409 59                pop ecx
 6 563 82                db 0x82
 7 561 7C0E              jl 0x22
 8 492 18807C4E9980      sbb [eax+0x80
 9 561 7C9F              jl 0xffffffbb
10 561 0F817CCC4083      jno near 0x83
11 561 7C98              jl 0xffffffbc
```

Data offset	14
# equal instructions	100
absolute equal instructions	100
relative position	0

Table D14: Entry point +14

Entry point: +15:

```
LN INS OPCODE            INSTRUCTION
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 035 817C8A2B867C161E  cmp dword [ed
 1 035 807CE51780        cmp byte [ebp
 2 561 7C11              jl 0x20
 3 409 59                pop ecx
 4 563 82                db 0x82
 5 561 7C0E              jl 0x21
 6 492 18807C4E9980      sbb [eax+0x80
 7 561 7C9F              jl 0xffffffba
 8 561 0F817CCC4083      jno near 0x83
 9 561 7C98              jl 0xffffffbb
```

Data offset	15
# equal instructions	97
absolute equal instructions	97
relative position	-2

Table D15: Entry point +15

Entry point: +16:

```
LN INS OPCODE            INSTRUCTION
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 561 7C8A              jl 0xffffff8c
 1 522 2B867C161E80      sub eax,[esi+
 2 561 7CE5              jl 0xffffffef
 3 409 17                pop ss
 4 035 807C115982        cmp byte [ecx
 5 561 7C0E              jl 0x20
 6 492 18807C4E9980      sbb [eax+0x80
 7 561 7C9F              jl 0xffffffb9
 8 561 0F817CCC4083      jno near 0x83
 9 561 7C98              jl 0xffffffba
```

Data offset	16
# equal instructions	93
absolute equal instructions	95
relative position	-2

Table D16: Entry point +16

Entry point: +17:

```
LN INS OPCODE            INSTRUCTION
 5 334 08817C8A2B86      or [ecx+0x862b8a7c],al
 6 561 7C16              jl 0x46b
 7 456 1E                push ds
 8 035 807CE51780        cmp byte [ebp+0x17],0x80
 9 561 7C11              jl 0x46e
10 409 59                pop ecx
11 563 82                db 0x82
12 561 7C0E              jl 0x46f
13 492 18807C4E9980      sbb [eax+0x80994e7c],al
14 561 7C9F              jl 0x408
15 561 0F817CCC4083      jno near 0x8340d0eb
16 561 7C98              jl 0x409
```

```
LN INS OPCODE            INSTRUCTION
 0 292 8A2B              mov ch,[ebx]
 1 553 867C161E          xchg bh,[esi+
 2 035 807CE51780        cmp byte [ebp
 3 561 7C11              jl 0x1e
 4 409 59                pop ecx
 5 563 82                db 0x82
 6 561 7C0E              jl 0x1f
 7 492 18807C4E9980      sbb [eax+0x80
 8 561 7C9F              jl 0xffffffb8
 9 561 0F817CCC4083      jno near 0x83
10 561 7C98              jl 0xffffffb9
```

Data offset	17
# equal instructions	97
absolute equal instructions	97
relative position	-1

Table D17: Entry point +17

References

[1] S. Hofmeyr and S. Forrest. *Architecture for an Artificial Immune System*, Evolutionary Computation, 8,(4), 443-473, 2000.

[2] J. Balthrop, S. Forrest, and M. Glickman. *Revisiting lisys: Parameters and normal behavior*. In CEC-2002: Proceedings of the Congress on Evolutionary Computing, 2002.

[3] J. K. Percus, O. E. Percus, and A. S. Perelson. *Predicting the size of the antibody-combining region from consideration of efficient self/nonself discrimination.* In Procedings of the National Academy of Science 90, pages 1691-1695, 1993.

[4] A. Somayaji, S. A. Hofmeyr, and S. Forrest. *Principles of a Computer Immune System*. In Proceedings of the Second New Security Paradigms Workshop, 1997.

[5] S. Forrest, A.S. Perelson, L. Allen, and R. Cherukuri. *Self-nonself discrimination in a computer*. In Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy. IEEE Press, 1994.

[6] F. Gonzalez and D. Dasgupta. *Anomaly detection using real-valued negative selection*. Journal of Genetic Programming and Evolvable Machines, 4:383-403, 2003.

[7] Uwe Aickelin, Julie Greensmith and Jamie Twycross. *Immune System Approaches to Intrusion Detection - A Review*, In Proceedings of ICARIS '04, the 3rd International Conference on Artificial Immune Systems, LNCS 3239, pp316-329, Springer-Verlag, 2004.

[8] Rune Schmidt Jensen. *Immune System for Virus Detection and Elimination*. Informatics and Mathematical Modelling, Technical University of Denmark, 2002.

[9] Stephanie Forrest, Justin Balthrop, Matthew Glickman, David Ackley. *Computation in the Wild*. Dept. of Computer Science University of New Mexico, 2002.

[10] Ranang, Martin Thorsen. *An Artificial Immune System Approach to Preserving Security in Computer Networks*. Sivilingeniør's thesis, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, 2002.

[11] Marcelo Reis and Fabricio Paula and Diego Fernandes and Paulo Geus. *A Hybrid IDS Architecture Based on the Immune System*, 2002.

[12] Stephen Northcutt and Judy Novak. *Network Intrusion Detection*, Third Edition. New Riders Publishing, 2002.

[13] Jack Koziol. *Intrusion Detection with Snort*. Sams Publishing, 2003.

[14] Kerry J. Cox and Christopher Gerg. *Managing Security with Snort and IDS Tools*. O'Reilly, 2004.

[15] Charlie Scott, Paul Wolfe and Bert Hayes. *Snort For Dummies*. Wiley Publishing, 2004.

[16] Martin Roesch and Chris Green. *Snort Users Manual 2.3.2*. Sourcefire, Inc., 2005.

[17] Richard O. Duda, Peter E. Hart and David G. Stork. *Pattern Classification*, Second Edition. Wiley Publishing, 2000.

[18] Andrew R. Webb. *Statistical Pattern Recognition*, Second Edition. Wiley Publishing, 2002.

[19] William E. Paul. *Fundamental Immunology*, 5th Edition. Lippincott Williams & Wilkins Publishers, 2003.

[20] K. Lee Lerner and Brenda Wilmoth Lerner. *World of Microbiology and Immunology*. Thomson Gale, 2003.

[21] Richard A. Goldsby, T. J. Kindt, B.A. Osborne, Janis Kuby. *Immunology*, 5th Edition. W. H. Freeman and Company, New York, 2003.

[22] S. A. Hofmeyr. *An Interpretative Introduction to the Immune System*. To Appear in Design Principles for the Immune System and other Distributed Autonomous Systems. Oxford University Press, Eds, I. Cohen and L. Segel. 2000

[23] Abul K. Abbas et al. *T cell tolerance and autoimmunity*. Department of Pathology, University of California San Francisco School of Medicine. Autoimmunity Reviews 3, 2004.

[24] J. R. Kalden, M. Herrmann. *Apoptosis and autoimmunity: from mechanisms to treatment*, Weinheim, Wiley-VCH, 2003.

[25] Sanjay Goel and Stephen F. Bush. *Kolmogorov complexity estimates for detection of viruses in biologically inspired security systems: a comparison with traditional approaches*. To appear in the Complexity Journal, vol. 9, issue 2, 2003.

[26] Oleg Kolesnikov and Wenke Lee. *Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic*. 2004

[27] J. Meijer, R. Danyliw and Y. Demchenko. *The Incident Object Description Exchange Format Data Model and XML Implementation*. INCH Working Group, 2004. http://www3.ietf.org/proceedings/05mar/IDs/draft-ietf-inch-iodef-03.txt retrieved on 25.05.2005

[28] Steven A. Hofmeyr. *A Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, Department of Computer Sciences, University of New Mexico, 1999.

[29] Fernando Esponda and Stephanie Forrest. *Detector coverage under the r-contiguous bits matching rule*. University of New Mexico Technical Report TR-CS-2002-03. 2002

[30] Longest common subsequence algorithm. CEOI 2003. http://ceoi.inf.elte.hu/probarch/03/trip-solution.pdf retrieved on 06.06.2005

[31] Steven S. Skiena. Longest Common Substring from The Stony Brook Algorithm Repository. Department of Computer Science, State University NY, 2001. http://www.cs.sunysb.edu/~algorith/files/longest-common-substring.shtml retrieved on 18.06.2005

[32] Robert Giegerich and David Wheeler. Pairwise Sequence Alignment in BioComputing Hypertext Coursebook. 1996. http://www.techfak.uni-bielefeld.de/bcd/Curric/PrwAli/prwali.html retrieved on 30.07.2005

[33] Levenshtein distance. http://www.answers.com/topic/levenshtein-distance retrieved on 30.07.2005

[34] Hamming Distance. http://en.wikipedia.org/wiki/Hamming_distance retrieved on 17.06.2005

[35] Pearson Product Moment Correlation Coefficient. http://www.mnstate.edu/wasson/ed602pearsoncorr.htm retrieved on 22.05.2005

[36] Pearson's Product-Moment Correlation http://www.mhhe.com/socscience/intro/cafe/common/stat/cstats03.mhtml retrieved on 22.05.2005

[37] Agnitum Outpost Firewall – Documentation. http://www.agnitum.com/products/outpost/docs.php retrieved on 27.05.2005

[38] Andre Yee. *Demystifying Intrusion Detection*. NFR Security, 2002. http://www.ebizq.net/topics/security/features/1674.html retrieved on 13.12.2004

[39] Process Guard Documentation. Diamond CS. http://www.diamondcs.com.au/processguard/ retrieved on 27.05.2005

[40] Kris Kaspersky. *Kaspersky Anti-Virus 5.0 for Windows Workstations – Administrator's manual*. Kaspersky Labs, 2005.

[41] Microsoft Windows XP Service Pack 2 Resources. http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/winxpsp2.mspx retrieved on 16.06.2005

[42] Steven S. Skiena. *The Algorithm Design Manual*. Telos/Springer-Verlag, 1997.

[43] Steven A. Hofmeyr and Stephanie Forrest. *Immunity by Design: An Artificial Immune System*. Dept. of Computer Science, University of New Mexico, 2000.

[44] Donald E. Knuth. *Dynamic Storage Allocation*. Fundamental Algorithms, Volume 1 of The Art of Computer Programming, 3rd edition, Addison-Wesley Publishing Company. 1997.

[45] Prasad Dabak, Milind Borate and Sandeep Phadke. *Portable Executable File Format in Undocumented Windows NT*. M&T Books, 1999.

[46] PE File Structure. http://www.madchat.org/vxdevl/papers/winsys/pefile/pefile.htm retrieved on 16.06.2005

[47] *Microsoft Portable Executable and Common Object File Format Specification*. MSDN 6.0 1999. http://elfz.laacz.lv/ms_exe_spec.html retrieved on 16.06.2005

[48] *PE file format, size optimizing and standard files*. http://deinmeister.de/w32asm5e.htm retrieved on 16.06.2005

[49] Matt Pietrek. *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*. MSDN 1994.

[50] Matt Pietrek. *An In-Depth Look into the Win32 Portable Executable File Format*. MSDN 2002.

[51] System User's Guide: Operating System and Devices: Testing Files for Viruses. http://www.unet.univie.ac.at/aix/aixuser/usrosdev/test_virus.htm retrieved on 06.08.2005

[52] John W. Lockwood. *Application of Hardware Accelerated Extensible Network Nodes for Internet Worm and Virus Protection*. International Working Conference on Active Networks (IWAN), Kyoto, Japan, December, 2003. Lecture Notes in Computer Science (LCIS)

[53] Netwide Disassembler: NDISASM. http://nasm.sourceforge.net/ retrieved on 20.04.2005

[54] D. Dagupta and F. Gonzalez. *An Immunity-Based Technique to Characterize Intrusions in Computer Networks*. IEEE Transactions on Evolutionary Computation, 6(3), pages 1081-1088 June 2002.

[55] Jeffrey O. Kephart, Gregory B. Sorkin, Morton Swimmer, and Steve R. White. *Blueprint for a Computer Immune System*. IBM Thomas J. Watson Research Center, Presented at the Virus Bulletin International Conference in San Francisco, California, October 1-3, 1997

[56] Justin Balthrop, Fernando Esponda, Stephanie Forrest, and Matthew Glickman. *Coverage and generalization in an artificial immune system*. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), 2002.

[57] S. Goel and S. F. Bush. *Biological Models of Security for Virus Propagation in Computer Networks*. ;login:, 29(6), 49-56. 2004

[58] Jungwon Kim and Peter J. Bentley. *An Evaluation of Negative Selection in an Artificial Immune System for Network Intrusion Detection*. Proceedings of the Genetic and Evolutionary Computation Conference, 2001.

[59] J. Kim and P. Bentley. *The Human Immune System and Network Intrusion Detection*. 7th European Conference on Intelligent Techniques and Soft Computing (EUFIT '99), Aachen, Germany, 1999a.

[60] J. Kim and P. Bentley. *The Artificial Immune Model for Network Intrusion Detection*. 7th European Conference on Intelligent Techniques and Soft Computing (EUFIT'99), Aachen, Germany, 1999b

[61] P. D'haeseleer, S. Forrest, and P. Helman. *An immunological approach to change detection: algorithms, analysis and implications*. In Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy. IEEE Press, 1996.

[62] Greg Hoglund and Gary McGraw. *Exploiting Software - How to Break Code*. Addison Wesley, 2004.

[63] Larry L. Peterson and Bruce S. Davie. *Computer Networks – A Systems Approach*, Third Edition. Morgan Kaufmann, 2003.

[64] Matthew Norman. *Database Design Manual: using MySQL for Windows*. Springer-Verlag, 2004

[65] A. Dubey, P. Sharma, S. Dasgupta, V. Gupta and L. Kapoor. *Integrating PHP and XML*. SkillSoft Press, 2004.

[66] Stig Sæther Bakken and Egon Schmid. *PHP Manual*. PHP Documentation Group, 2004.

[67] T. Toth and C. Kruegel. *Accurate Buffer Overflow Detection via Abstract Payload Execution*. In Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID), October 2002.