Detecting malicious network activity using flow data and learning automata

By

Christian Auby
Torbjørn Skagestad
Kristian Tveiten

Thesis submitted in Partial Fulfillment of the requirements for the Degree Master of Technology in Information and Communication Technology

Faculty of Engineering and Science University of Agder

Grimstad

May 2009

Table of Contents

Table of Figures

Table of Tables

Abstract

Malicious software has become an increasing problem for both businesses and home users. The traditional antivirus solutions are not always enough to detect an infection. As a result of this a lot of businesses are deploying Intrusion Detection Systems, so that they may have an extra level of protection by analyzing the network traffic.

Intrusion Detection Systems are resource hungry, and may in some cases require more resources than what is available. This means that some of the traffic will not be analyzed, and malicious software may be able to avoid detection.

In some cases, laws and regulations may prevent you from inspecting the content of the network traffic, making it difficult to detect infected clients. In these types of scenarios a solution not dependent on traffic content is a viable alternative.

In this paper we will propose a solution to detect malicious software in a network with less resource demands than a traditional Intrusion Detection System. The method will only use flow data when determining whether a client is infected or not. The decision will be made using both learning automata and stochastic weak estimators.

We have shown that it is possible to detect malicious software in a network without inspecting the content of the packets, and that it is achievable by the use of both learning automata and stochastic weak estimators.

Preface

This thesis is submitted to University of Agder, Faculty of Engineering and Science in partial fulfillment of the degree of Master of Science in Information and Communication Technology.

This work has been carried out in collaboration with Telenor Security Operation Centre in Arendal, under the supervision of associate professor Ole-Christoffer Granmo at Agder University, and Morten Kråkvik at Telenor Security Operation Centre.

We would like to thank our supervisors for valuable suggestions and insight in the problem at hand. Their help has been most appreciated during the research. Morten Kråkvik has provided help and suggestions regarding implementation and testing, and was essential in the malware selection and analysis. Ole-Christoffer Granmo has been of great help in the planning of the research and in the selection of the algorithms used. His feedback regarding our research has been invaluable.

Grimstad, May 2009

_____

Christian Auby, Torbjørn Skagestad and Kristian Tveiten

1 Introduction

An Intrusion detection system (IDS) is a way to combat the increasing dangers related to computer security. Some IDSes can prevent malicious activities by blocking or shutting down a connection, but in many cases this will be too intrusive. If the IDS is wrong and blocks important traffic pure detection might be a better choice.

Detection can be done by inspecting the content of the network traffic, called Deep Packet Inspection (DPI). This can be expensive in terms of computing resources. Our solution will combat this by looking at traffic patterns instead of content.

The problem at hand is to detect computers infected by malicious software at decreased resource requirements. This has been done using machine learning algorithms to analyze the traffic patterns and give feedback regarding suspicious clients.

Malicious software is used by criminals to control computers connected to the internet, usually without the owner's knowledge. These applications are often used to collect sensitive information or perform illegal activities. Having such an application installed is a risk to home users as well as businesses.

Our solution only uses information regarding the traffic, making detection difficult compared to existing DPI systems. We will take one such existing solution and compare it to the solution we create. Comparison will be made on detection and performance.

This thesis documents one approach to this problem and presents our research and results regarding the implementation and testing of one such solution.

1.1 Background and motivation

There are several forms of IDS, but some of the common ones are deep packet inspection systems (DPI) and firewalls. They can also be configured to work together. A DPI system works by monitoring a network connection, usually a mirror port on a switch to be able to monitor an entire network, and analyzing all the traffic that passes on this port.

This allows for complex rules that define malicious behavior, and allows detection of many types of malware, but it comes at a price; computing resources. More bandwidth requires more computing resources, and computing resources requires faster hardware through upgrades or load balancing using more than one IDS.

Finding some solution to this ever increasing problem would save both investments and energy. A lot of malicious software developers are aware of the IDSes detection capabilities and can program their software to be harder to detect. This leads up to a never ending race between the authors of malicious software and IDS researchers.

In computer security the word malware is used to describe a computer program containing functionalities to harm you either by stealing sensitive information or by using your computer to perform tasks without your consent. The word is created by blending the words malicious and software.

Malware exhibits a wild variety of behaviors such as spreading through network connections, “rootkit” functionality to hide the code in memory, heuristic self modifying code, ability to disable local antivirus and firewalls and “connect-back” functionality giving the author an ability to remotely control the computer.

Connect back functionality usually implies that the malware connects to a Command and Control Center (C&C). A C&C is a computer or network node connecting infected computers to a botnet. A C&C can tell the botnet it is controlling to spread spam, install rogue applications or to perform Denial of Service attacks (DDoS).

Certain malware always connect the infected client to a botnet. Some use peer to peer connectivity instead of a central C&C server. Others only report sensitive information, displays ads or similar. There are many variations.

According to the security firm Trend Micro's annual threat report approximately 700.000 new malware were identified each month comprising a total number of 16.495.000 unique threats in 2008. [1].

The number of new malware is not only increasing, the malware itself is also becoming more sophisticated by adding information stealing techniques. One such functionality are keyloggers.

Keyloggers are programs used to pick up keystrokes input by the user and sending these to a predetermined destination. Logging keystrokes would make it easy to pick up all usernames and passwords typed by the user. Getting login information to these types of services may in many cases expose the users social security number, bank account number or other sensitive information. Information stealing impose a new threat to today's computer users and can harm people in ways that they did not think were possible. [2][3]

It is common knowledge that keyloggers are frequently used, and most internet banks have therefore applied countermeasures like a requirement to authenticate by using security certificates and single use passwords. A username and static password will in this case not be enough to successfully log in as the password expires after use.

To avoid these additional encryption and authentication techniques several keyloggers have lately been developed as specialized browser plugins used to extract both login information and security certificates. Such information could then be sold by the underground businesses to competitors giving them an unfair, and illegal, advantage over other legal businesses. [4][5]

In order to notify of a successful installation the trojans often send some type of data back to the creator. This is usually done by connecting to an IRC server or by sending a HTTP request. The latter is called a check in request.

Some malware sends these types of requests at regular intervals, in order to inform that the malware still has its victim under control. In most cases it is hard for a user to notice when first infected, giving the malware a possibility to secretly exchange private information to its control server.

Larger networks can often have a lot of variations in traffic load, giving Snort trouble at peak hours. When Snort (an existing IDS) exhausts the resources available it usually starts to drop packets, ignoring a lot of the traffic that would otherwise have been analyzed. In these situations a more resource friendly solution would be preferred.

We believe that a solution that analyzes traffic flows could work better since it does not open and inspect any packets. Traffic flow, or flow data, is information regarding

connections and the data that passes through them, but not about content. A real life analogy would be the postal system, where flow data would be the addresses on the mail and the size of the packet, etc. This makes it irrelevant whether the traffic is encrypted or not since the content is not inspected anyway.

In many countries internet service providers (ISP) are not allowed to inspect the content of their customers traffic. Even so they wish to have control of infected clients in their network. In this type of situation an effective flow based detection technique could be a solution.

1.2 Proposed solution

Our proposed solution is a flow based analysis tool, implemented as a component to Telenor Security Operation Centre's (TSOC) existing IDS system. These systems are passive sensors located in customers' networks, giving TSOC the ability to analyze the network traffic that passes through. The solution we create will be added to one of these sensors, giving it new analysis functionality in addition to DPI with Snort.

We wish to develop this sensor addon as a set of learning automata that detects malware using flow data. Learning automata will allow us to do detection over time, developing the threat value based on traffic information that spans hours or days.

Flow data is smaller than traffic captures because it does not contain the packet contents. It does contain information such as IP and port information, number of bytes and number of packets transmitted in each direction. This information will be collected using the Argus tool. Argus can also be configured to store small amounts of packet content in addition to the flow data. We have chosen not to use this information since this would violate one of our requirements.

Snort is dependent on the packet contents to be able to function, while our solution does not need this at all. Argus has been configured to not capture any packet content to save valuable storage space on the sensor. This will give our solution two main advantages. Saved storage space can be used to store data over a longer period of time. The time spent on disk writes will be shorter as the amount of data written is less than a full packet capture.

This implies that we will have less data to work with, and force us to use an out of the ordinary approach to detect malware; behavioral analysis. We believe that it should be possible to detect at least some sorts of malware by using this type of approach. We believe that we can find a recognizable pattern by studying the external communication the malware creates.

Another possibility could be to use behavioral analysis to detect patterns that are common to many malware. Such an approach could possibly detect new malware that traditional IDSes does not yet have signatures for. Snort must have signatures to detect suspicious traffic, which can make it overlook unknown threats. We believe that a general rule used for behavioral detection in our automata can be useful in these types of situations.

1.3 Limitations

No deep packet inspection

The automaton will not look at what the packets going on the network contains. The process of deciding whether malicious activities have occurred or not will be based only on information from data gathered by the flow data collection tool Argus. This tool will be more thoroughly explained under the *Related work* chapter.

Even if the flow data contains small amounts of packet contents these will not be used.

No DDoS or brute force detection

The automaton will not detect attacks such as distributed denial of service or brute force attacks. These types of attacks are easy to detect using existing solutions. It might also be possible to detect them using our automaton, but we will not focus on exploring this possibility in our research.

Limited set of rules

We will only write detection rules for the malware samples we will select. Writing a rule will require analysis that takes some time, and it is not achievable to replace a Snort signature set. Spending time on this is not a priority, nor is this important for the results of our research. This means that our proof of concept will not be a replacement for existing solutions.

No traffic interference

The automaton will only listen to traffic. It will not actively block traffic it considers malicious. All suspect events detected by the automaton will only be reported. It is up to an administrator to decide whether action is needed.

1.4 Assumptions

Linear reward / penalty inaction will work with network traffic

We wish to use the linear reward / penalty inaction algorithm used by Morten Kråkvik in his thesis on *DDoS detection based on traffic profiles*. This thesis indicates that this is a good algorithm to use for analysis of DDoS traffic, and we assume that it should work in an acceptable manner on other types of network traffic as well.

Noise limited to a per connection basis

The design and implementation of the automaton will not use any non-malicious traffic. Non-malicious traffic is legal traffic generated by clients and servers on a given network. Malicious traffic is what we call traffic generated by malware, or clients that have malicious intentions.

Real traffic has a high amount of legal traffic and small amounts of malicious traffic. Our automata will be designed to listen on a per connection basis, establishing a new automaton when a host opens a new connection.

We therefore assume that this will limit the amount of noise to each separate connection. Each automaton will be defined to detect a specific behavior related to an application with malicious intentions. Snort does this also by using different signatures to define the traffic.

Malware samples suitable for detection exists

Since our research relies on traffic generated from malware we must assume that we can be able to get hold of samples that produce enough traffic to generate a distinguishable pattern in a reasonable amount of time.

2 Related work

There are several types of related work relevant for our research. Existing solutions such as Snort provides ready to use intrusion detection, while papers touch upon the learning aspects. In addition to this there are several tools suitable for certain parts of our task.

In this chapter we will give an overview of previous work we found useful in our research. A brief description of software relevant to our research will also be added. This will consist of both external software used in our proposed solution, and software we have chosen as a reference point for our comparisons.

2.1 DDoS detection based on traffic profiles

By Morten Kråkvik. Agder University College, Faculty of Engineering and Science, Grimstad, Norway, May 2006.

2.1.1 Goal and results

In this paper Morten Kråkvik propose a method for detecting and identifying the sources of DDoS attacks. The detection is based on research in the field of network traffic and IP address monitoring. A learning automaton is responsible for the tracking of source-IPs and subnets.

Kråkvik manages to show that by using a specific reinforcement algorithm for the automaton, it is possible to identify participants in a DDoS attack. The solution is also robust against attacks which incorporates IP spoofing.

2.1.2 Relevance

Morten Kråkvik uses a learning automaton with the linear reward penalty algorithm. This is relevant, as we intend to use the same algorithm in our research. The research done by Morten Kråkvik shows that this type of automaton is able to handle network traffic in a suitable manner.

2.1.3 Differences

In this paper Morten Kråkvik only focuses on DDoS traffic. The main goal of performing a DDoS attack is to stop or slow down a service or host in another network. To be able to do this an attacker would in most cases have to generate a lot of traffic. Or in other words, more traffic than the victim can handle. This can be done by using multiple machines and configure them to for instance send multiple SYN packets towards one target at the same time.

From an IDSes point of view, if properly configured, such an approach will generate a lot of SYN flooding alarms and would eventually distinguish themselves from the other alarms generated by other network traffic. A properly configured IDS should be able to detect a two minute ping sweep from one attacker just by logging the numbers of the packets sent from this host.

The IDS should, when keeping this in mind, be even easier to detect and log a thirty minute SYN flood DDoS attack from multiple host by using this approach. A DDoS

attack can last for hours and even days, or in other cases until the people responsible for the target machines are able to block the attackers.

In our research we will not go into DDoS at all. Our focus will be on the detection of malicious traffic generated by malware. Since the main objective for malware authors in most cases are to earn money they often configure it to avoid detection, either by using encryption techniques or by only generate traffic that is required for operation. Or even both. While a typical DDoS attack can be detected in seconds, properly configured malware can take several minutes or hours, or completely avoid detection.

Morten created a network traffic collector to generate the flow data and present it for the automaton. We will use the network collection tool Argus, which is already integrated into Telenors systems, for this job.

2.2 An intelligent IDS for anomaly and misuse detection in computer networks

By Ozgur Depren, Murat Topallar, Emin Anarim, M. Kemal Ciliz Bogazici University, Electrical and Electronics Engineering Department, Information and Communications Security (BUICS) Lab, Bebek, Istanbul, Turkey.

In this paper the authors propose an Intrusion Detection System (IDS) utilizing anomaly, and misuse detection. Their solution consists of two modules, one for anomaly detection, and one for detecting misuse. A decision support system (DSS) is used to combine the results of the two modules.

2.2.1 Goal and results

The main goal for this paper was to benchmark the performance of the proposed hybrid IDS. This was done by using KDD Cup 99 Data Set. A rule based DSS then compares the results of the two modules, to see how they perform. The results shows that the hybrid IDS have a higher detection rate than individual approaches.

2.2.2 Relevance

The anomaly detection module consist of a pre-processing module, several anomaly analyzer modules and a communication module. The analyzer modules utilizes self organizing maps (SOM). SOM is a neural network. This means that the modules uses machine learning, and thus needs to be trained. This is interesting as we also will be using machine learning, but not neural nets. Other methods of machine learning may yield different results.

The data used to train and test the IDS in this paper is from the KDD Cup challenge of 1999. As malicious traffic changes rapidly this dataset is completely uninteresting today. The process of training software to detect malicious activities from network traffic is however relevant. The solution proposed in this paper detects both misuse and anomalies.

2.2.3 Differences

The proposed solution in this paper uses deep packet inspection and signatures to detect misuse. We however will use information about the flow data, and machine learning to accomplish similar detection. By only using flow data, the way our

automaton analyzes will be a little different than for the proposed neural net, as it was only detecting anomalies.

The dataset used in this paper was a known set. The same set was used for both training and benchmarking. We will use data collected from a live network for our research. The benchmarking will be carried out using data from the this same network. This way we will have a more interesting view on how our solution performs, as it is tested in a real world environment, with real traffic.

In the paper the solution proposed were compared to the individual parts of the solution. We will compare our solution and results to the performance of Snort. Snort is a signature based IDS. The comparison will give interesting information on whether a flow data based solution could eventually replace a signature based IDS, or should be considered an addon to the existing solutions.

2.3 Visualizing netflow data

By Zhihua Jin, Department of Telematics, The Norwegian University of

Science and Technology (NTNU), Trondheim, Norway

2.3.1 Goal and results

This thesis was written by Zhihua Jin at NTNU. The primary goal was to create a different method of analyzing large amounts of network traffic. By using netflow data gathered from the traffic seen in a live network, the author has developed a system that represents this data in a visual manner.

An analyst will then look at the produced graphs and can from that detect anomalies in the network. There are many different views, and the analyst can dig deeper and look closer at specific things of interest. The end product is a set of tools that lets you explore data visually.

2.3.2 Relevance

This thesis is very interesting to us, as it is, from what we can find, the only one in the field of intrusion detection that uses netflow data. It touches upon detecting infected clients, botnet traffic and Distributed Denial of Service.

It is also very recent (July 2008), and is thus in no way outdated, with problems that are seen today.

2.3.3 Differences

The main focus of this thesis is how to visually represent the data. Our focus will be analyzing the data automatically, and then present the end results to an analyst. We will do this using machine learning techniques, which the NTNU thesis only mentions briefly.

Areas such as distributed denial of service and port scanning will not be a part of our research, as it is in the thesis from NTNU.

2.4 An efficient intrusion detection model based on fast inductive learning

By Wu Yang, Wei Wan , Lin Guo, Le-Jun Zhang, Information Security Research Center, Harbin Engineering University, Harbin 150001, China

2.4.1 Goal and results

The paper describes how fast inductive learning could be used to detect malicious traffic in an Intrusion Detection System. The goal of this research is to use the four different learning methods clustering, C4.5Rules, RIPPER and their own optimized algorithm called FILMID to be able to categorize the traffic into several classes. These classes consists of normal traffic, Denial of Service (DOS) attacks, Remote to Local (R2L) attacks, User to Root (U2R) attacks, and Probing attacks (Probe).

The authors use a dataset from the 1998 Intrusion Detection Evaluation Program created by DARPA which consists of a training set and a test set. When the traffic is categorized using the different methods above, the results are measured up against each other by their detection rate (DR) and false positive rate (FPR). The results are good, providing a DR of around 90 percent and a FPR around 2 percent for all of the methods except the clustering.

2.4.2 Relevance

This thesis is directly related to our work since it uses different methods of machine learning to detect and categorize network traffic. The dataset analyzed consists of different connections which is fairly related to flow data analysis in Argus. And this also confirms that our goals for this project should be feasible.

2.4.3 Differences

The test data used in this thesis is taken from a dataset created by DARPA in 1998, which is considered very old. We will use datasets created by Argus from a network consisting of thousands of clients, using data that reflect today's threats. Having such a large test network available will also give us the opportunity to use our learning algorithms on live traffic.

Another difference is the fact that this thesis focused on testing the accuracy of different learning algorithms. We find their results very interesting since it shows that they actually does not differ that much. This gives us the opportunity to focus more on

optimizing our code for use with Argus and on creating more rules based on different kinds of malware. A larger set of rules will give us a better foundation for testing our solution up against snort.

2.5 Argus

Argus is a flow monitor tool package for network traffic. [6] It can be used to collect flow data on a live network. Argus comes with a set of separate tools that each perform a specific function, such as capture, grouping and splitting of data.

2.5.1 argus

The tool named after the package is used to capture flow data on a network connection, or from a file with existing network captures, such as pcap files from Wireshark or tcpdump. The output is an Argus specific binary format that can be written to disk or sent to stdout. Working with this binary format directly is not recommended, but rather use other Argus tools to extract the data you want.

2.5.2 ra

Ra stands for "read argus", and is used to extract information from data captured by Argus. It can read from file, or directly from Argus using stdin. Ra allows you to select what fields to extract, such as addresses and ports, and print them out in a human readable way. You can also ask for other output methods that are easy to use in applications.

2.5.3 rasplit

rasplit is used to split Argus-data based on time, such as individual files containing five minutes of data. This is very useful for live analysis, as you can process data in chunks of five minutes.

2.5.4 racluster

racluster is suitable for statistics, by allowing you to group information based on certain fields, e.g. total source bytes and destination bytes count between two IP-addresses regardless of ports.

2.6 Snort

Snort is an intrusion detection and prevention system that uses a signature based approach. [7] It allows you to monitor a network interface and detect certain types of traffic, defined by the Snort rule set. Snort rules are often referred to as signatures. We will be using this naming, as we have chosen rules to describe our own specialized automata.

If the network environment has specific detection needs the set of signatures can be modified to accommodate this. New signatures may be added when needed. As a result of this snort loads all the selected signatures before it starts analyzing. The number of signatures, and the total size of the set is therefore an easy way to predict how much memory Snort will be using when doing live analysis.

Signatures can be written to detect many different types of activities, such as syn-floods, which are unusually many connection attempts, but can also be used to inspect the actual content of data.

Deep packet inspection can be very powerful and in many cases makes detection easy, depending on how predictable the data is. Signatures can define source IP/net and port, as well as destination ip/net and port.

When a rule is triggered Snort has several actions available, such as sending an alert, or depending on configuration, dropping / blocking the traffic. When using Snort as a pure detection mechanism, which is less disruptive as there's no chance of erroneously blocking legit traffic, an administrator or operator must analyze the alerts from Snort and take action from there.

3 Experimental setup

In this chapter we will describe how our work was organized. We will give a detailed description of the experimental setup, and what methods we used to get our results. The first part will give reasons for our malware selection, and how the testing of it took place. This includes information about our test network, and its parts.

There were a total of three test sets used in our research. Each of these will be described here.

3.1 Malware selection

There are many types of malware and trojans, and not all might be suitable for detection by a learning automaton. Because of this we will initially select a set of samples to study further.

It is important that the sample has some sort of network activity that is recognizable to us, so we can build an automaton that also knows of this behavior. Most malware has some way of reporting to a command and control center, and we will be focusing on this.

The samples will be provided to us by Telenor, acquired through automatic gathering systems at Telenor SOC, or from private research related websites. The selection of samples that are to be used is however decided by us. The set of samples we have selected represents a small amount of actual malware that is currently active threats seen in the wild by Telenor SOC.

Some malware are auto generated with random values making for thousands and millions of different permutations (versions), however their underlying idea remains the same. In addition to randomize data that isn't used for anything inside the binary to make sure it can't be detected by simply comparing it to known versions, different configurations also make for different binaries.

To give an example the BlackEnergy sample we selected was created by us using a botnet kit. These types of kits are sold at underground websites and used by many different people. All these users configure their botnet differently, resulting in many versions of essentially the exact same malware. C&C control server adresses would be different, reporting intervals, what binaries to download after first infection, and many more.

A packer is often used to reduce code size and make it difficult to detect what type of malware, if any, a binary contains. Using a different packer would result in a different file, but with the exact same malware contained inside.

The samples are thus representative for malware not in the sample set.

The malware we have selected are:

- BlackEnergy
- Conficker
- Danmec
- Waledac

Details for each of these malware samples will be documented in the *malware chapter*.

3.2 Malware analysis

To get a foundation to design the automata around, we have to do an analysis of the selected malware. This will be done in a closed environment which we control entirely. Preventing outside factors such as additional network traffic not related to the sample meddling with the test will ensure that the data we extract is valid for this particular sample and make the job analyzing this data easier.

The malware lab consists of three main parts. A client where we run malware, a bridge and firewall, which act as the sniffing point, and a server running virtual clients to simulate a local network.

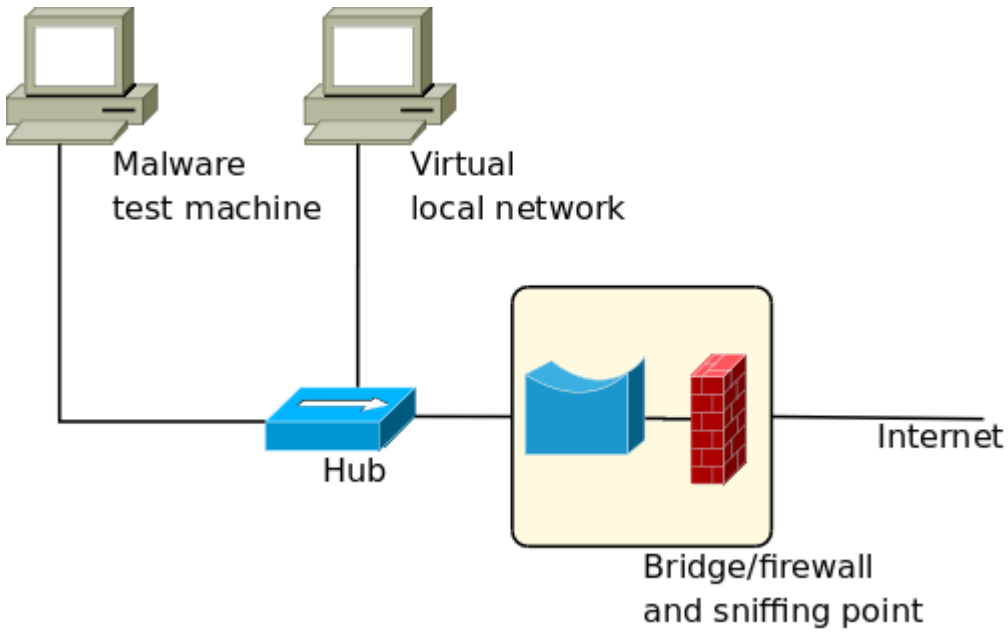

Figure 1: Malware lab

Figure 1 displays the design of our malware lab. The network is isolated from other networks including the internet in the bridge/firewall, preventing unwanted incoming connections. Note that the firewall has rules to allow some outgoing traffic since the samples require this connectivity.

The client used to run malware is essential, as some malware act differently on virtual clients. Among our selected samples only Conficker is aware of virtualization.

The virtual clients will have different roles, depending on the test we are doing. In some scenarios they may act as an infected host, while they in other only represent a target for malware.

There will be a total of eight virtual clients available. A complete list of the clients, and their roles are described in *Table 1*. The hardware used in addition to the virtual clients are also included.

IP address	Type/ Purpose	Special property	comment
192.168.1.1	Router	DNS server and Gateway	Connects the clients to the internet using the Network Address Translation protocol
192.168.1.80	Firewall/ bridge	Collects network data	Works as a firewall and a network traffic collection point (sniffer)
192.168.1.90	Native client	Conficker.B client on native hardware	Conficker is VMware aware. Exits on virtual machines upon execution
192.168.1.70	VMware server	Controls the virtual clients	Server running VMware server 2
192.168.1.71	Virtual client/ Fully patched	Danmec client	Will be infected with the Danmec trojan
192.168.1.72	Virtual client/ Fully patched	Conficker.B client	Used for test of Conficker.B VMware awareness
192.168.1.73	Virtual client/ Fully patched	BlackEnergy client	Will be infected with the BlackEnergy DDoS bot
192.168.1.74	Virtual client/ Fully patched	Waledac client	Will be infected with the Waledac trojan
192.168.1.75	Virtual client/ Fully patched	Target	Target for Conficker
192.168.1.76	Virtual client/ Fully patched	Target	Target for Conficker
192.168.1.77	Virtual client/ Fully patched	Target	Target for Conficker
192.168.1.78	Virtual client/ Patched up to SP2	Special target	Special target for Conficker. Unpatched to test if the trojan behaves differently

Table 1: Lab network setup

3.2.1 Router

The router is used as a DNS server and gateway that provides us with a connection between our local network and the internet. Network address translation is used for this purpose. A DHCP service, which is a service that automatically provides clients in a local network with IP addresses, is unnecessary here since all the clients are statically configured.

3.2.2 Firewall/ bridge

The firewall/ bridge is the central sniffing point we will use to collect network data generated by the malware.

The firewall/ bridge consists of a standard desktop computer with two ethernet interfaces. The interfaces are bridged. This means that all traffic received on one interface will be transmitted on the other. A firewall is also in place so that we are able to restrict traffic going through the bridge. This is important as we do not want the infected client to attack, and infect clients on the Internet.

Initially when testing malware the firewall will always be set at a very strict level. If the infected client seems to generate interesting traffic, but is limited by the firewall we can easily adjust the firewall and have another try.

The Bridge/firewall is running Ubuntu 8.10 desktop i386. The only changes made to the standard install is that network manager have been removed, and Wireshark, tcpflow and ngrep have been installed. The removal of network manager is due to the configuration of the network bridge.

3.2.3 Client network

Native client

The client that is running the malware is set up to represent a typical home computer. The hardware itself is not of much importance except that it is a x86-compatible CPU, which is a requirement to be able to run most malware.

It is running Windows XP with Service Pack 3 installed, as well as all hot patches released by February 2009. The reason we need a physical computer is simple; make sure that VMware aware malware like Conficker can be tested in our network. Certain malware checks the memory for VMware related processes upon execution and exits

if these are detected. This is a feature used to confuse malware analysts and delay the analysis process.

To be able to quickly restore the client after running a test we use an application called Moonscape. It allows you to set a harddrive restore point and return to this later. This saves us a lot of time as we do not need to reinstall the operating system to get a clean system.

Virtual local network

The virtual local network consist of a server running VMware Server 2 on top of a clean Debian Lenny install. There are seven virtual clients, all running Windows XP. The clients will have different levels of patching depending on the tests we are performing. For each test the patch level of the virtual local network will be described separately. After a test have been performed, the clients will be reverted to their initial state.

3.2.4 Operation

Each malware sample will be tested in the lab under the same conditions. This means that the infected host(s) will be reset for each sample, and thus provide grounds for comparison. As we want to make sure we have enough data, each sample will be executed and the network traffic captured for a minimum of three hours. Samples that do not behave regularly or have small amounts of network traffic will require more. This will be taken on a case by case basis.

Although the automata will use flow data only, we still wish to capture the contents as this will help in the manual analysis. Because of this we execute tcpdump / Wireshark (same output data) before the samples are run, and the traffic dumped in it's entirety. A fresh capture will be started for each sample.

Flow data will later on be generated from these data captures using Argus for use in the automata.

3.3 Test sets

In many papers regarding IDS, you will see the use of KDD Cup 1998 Data set. This is a data set released by DARPA to challenge, and test how well an IDS works. [8] This set was a good measurement for your IDS when the set was released, but not so much today. The way people use the Internet, and what types of attack you might observe is profoundly different.

Most types of attacks you might have seen in the late nineties that you still see today is fairly easy to detect. Attacks like brute forcing a service is avoidable in most software, and easily detected.

Because of this we have chosen to generate our own test sets. They will be based on both lab environments, and live traffic from a network consisting of thousands of clients. This means that our results will be comparable to other client networks within a typical work environment. It also means that our tests are relevant in today's network environment, and not a relic from the nineties.

Each of our test sets are files consisting of the Argus data for the network traffic that was captured during execution. Test set 1 and test set 2 will also contain the actual traffic data in the form of a pcap file, but will only be used in manual analysis, not by the automata. Content will not be captured for test set 3 as it would require far too much disk space, as well as needing far too much time to transfer back to Telenor. Test set 3 will thus consist purely of Argus data.

3.3.1 Test set 1 - Initial research

This test set will be created in the *malware lab*, by running each sample separately. There will only be one sample running at once, which results in a test set comprised of one smaller test for each malware. Each sample will be executed long enough to allow us to capture enough data, typically a few hours, up to twelve hours.

The data collected in this set will only be used for research and development of the automata and not in any of the tests that will be run once development is finished. There is no sensitive data in this test set.

3.3.2 Test set 2 - Mass infection

Test set 2 will be created using the entire lab network as described in *Table 1*. All the infected clients will be running at the same time, simulating a network with infected clients. The capture will run for approximately 24 hours.

This set will be used in tests that require infected clients to be present, such as detection. It will also be used to see if there are any overlapping detection between the different samples. Snort events from this test run will also be collected.

There is no sensitive data in this test set.

3.3.3 Test set 3 - Large client network

Set 3 will by far be the largest test set, and for this reason we will only store Argus data. This is also the only test set that won't be created in our lab at Telenor. Argus data will be collected on an undisclosed network with thousands of workstations, and the data transferred back to Telenor for analysis. Due to the sensitivity of the data we are not allowed to bring this set outside of the secure zone at Telenor, Arendal.

Argus data from this set will be used in performance analysis and documentation of false positives.

3.4 Automata implementation

The automata implementation will be written using information gathered during malware research, using *test set 1*. Once we see what type of activity we need to focus on to detect the malware samples, it will be clear what type of functionality the automata needs.

Overall, there will be an automata controller that receives Argus data, which then parses this, and sends the parsed data to every automata it controls. Each of these automata will detect one type of malware each.

The base functionality will be written in a base class, and each specific automaton will inherit this base class. All of this will be written in Python to fulfill the requirement set by Telenor.

3.5 Automata testing

The testing of the automata will be done in two parts. The first part consists of a detection test. Here we will see if the automata are able to detect the malware samples we have selected.

In the second test we will gather data regarding performance. The interesting parts here will be the memory demands, and how much CPU time the proposed solution needs.

To ensure that all test results are valid and comparable there will be no further development once the testing is started. If there were any development done after testing, all test would have to be redone.

The tests will be run on the i386 platform, ensuring that the platform requirement is fulfilled.

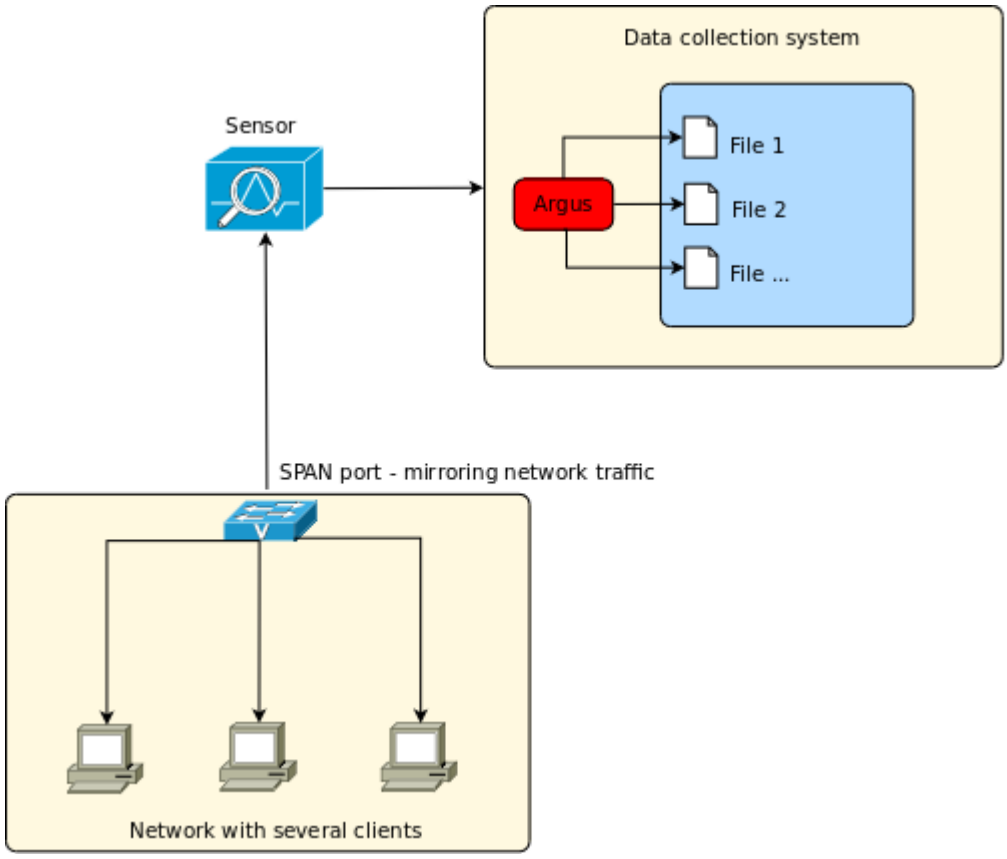

Figure 2: Collection of flow data

Traffic will be gathered in real time on a live network, and it is therefore important that data is gathered simultaneously, otherwise there's a risk of comparing high

bandwidth to low bandwidth and so on. Statistics on resource usage will be stored during execution. The test will give us data to fill in the test matrix, as described in *Table 2*

	Detection	**CPU**	**Memory**
Snort	n/a	n/a	n/a
Automata	n/a	n/a	n/a

Table 2: Test matrix

The results of each of these tests will be presented and discussed in later chapters.

3.5.1 Detection

Detection results will be used to judge effectiveness and decide usage scenarios. Snort and our automata have different modes of operation thus require separate methods of gathering statistics.

All detection tests will use *test set 2*.

Snort

Snort detection results will be gathered by looking at the feedback from existing Telenor systems which make use of Snort. This will give us the Snort signature that triggered, the IP-addresses of the clients that are involved and the time of the incident.

Automata

The Automata will print detection to the console once the threat level of a specific automaton is higher than the set threshold. Information includes IP-addresses, which automaton type that triggered and how many pieces of Argus data it took to reach the threat threshold, in addition to the run time since the first data arrived.

3.5.2 Performance

Performance analysis will tell us how the two methods compare in regards to CPU and memory usage.

All performance tests will use *test set 3*.

To log resource usage the processes for the automata and for Snort will be monitored. Once a second the following will be extracted and stored:

- CPU Usage
- RAM Usage

The time spent by the automata will also be measured.

Snort

Snort runs continuously and the CPU usage might vary depending on current used bandwidth. Statistics will be collected using standard Linux tools using the following commands:

```
while true; do DATE=$(date "+%Y-%m-%d %H:%M:%S");
LINE=$(top -b -n1 -p $(pidof snort) -p $(pidof argus) | egrep "snort|argus" | awk '{print $9 " " $10}' | tr '\n' ' ');
echo "$DATE: $LINE";
sleep 5;
done > performance.snort.log
```

Figure 3: Commands for gathering snort statistics

Our automata uses the scripting language python and the tool radump for file processing. This implies that these two processes will run together when we run our automata. We collected this data and combined them using the following two commands:

```
while true; do DATE=$(date "+%Y-%m-%d-%H:%M:%S");
PID=$(pidof radump); if [ -z $PID ];
then echo "$DATE: 0 0";
else LINE=$(top -b -n1 -p $(pidof radump) | egrep "radump" | awk '{print $9 " " $5}' | tr '\n' ' ');
echo "$DATE: $LINE";
fi;
sleep 1;
done > performance.radump.log
```

Figure 4: Commands for gathering radump statistics

```
while true;
do DATE=$(date "+%Y-%m-%d-%H:%M:%S");
LINE=$(top -b -n1 -p $(pidof python) | egrep "python" | awk '{print $9 " " $5}' | tr '\n' ' ');
echo "$DATE: $LINE";
sleep 1;
done > performance.python.log
```

Figure 5: Commands for gathering python statistics

It is important to note that we did not run the snort and the automata performance tests at the same time since this would make them interfere with each other. Sensor statistics show that Snort has a constant CPU usage of 100% and would be forced to share the CPU with our automata, forcing it and our automata to analyze the dataset at a slower pace.

From personal experience we have seen that when Snort is under heavy load it has a tendency to drop traffic, which means that it does not analyze these packets at all. When sharing CPU time Snort would eventually be forced to do this, resulting in it not giving us a complete analysis of the dataset.

Automata

Since the automata processes Argus data in five minute chunks it will use 100% CPU when it is active. To get numbers that are comparable to Snort the execution of the automata will be timed:

```
time python app.py
```

Figure 6: Timing of the automata execution

This will give us the total amount of time spent, running at 100% CPU. The automata processes faster than real time, so the average CPU usage will be calculated like this:

$$\frac{100}{\left(\frac{Time\ span\ of\ Argus\ data}{Time\ spent\ in\ automata}\right)}$$

Figure 7: CPU usage calculation

This number will then be compared to the average CPU usage from the Snort data.

4 Malware

There are many terms related to computer security, many of which change over time. We have selected the term malware as the basic term for a computer application that have bad intentions, or rather, is used by a person or a group of people with bad intentions.

Infected computers can be used for criminal activities such as the collection of sensitive information and attacks on infrastructure, both of which can be sold to third parties. [9] Two of the more prevalent types of malware are trojans and worms. Some also exhibit behavior of both. The samples we have selected fit in these categories.

In this chapter we will present the samples we have selected and the information we have found regarding them. Behavior will be described in detail, but also areas such as purpose and method of infection.

All of the malware selected target the Windows platform, and although there are malware and trojan threats for other platforms as well [10], they more or less drown in the amount of Windows malware that exists. [11]

The techniques in this paper only use network traffic for detection and could thus be used to detect malware on other platforms as well. It would also be possible to detect other behavioral traits, such as legal services, or specific applications, as long as they generate network traffic.

The following chapters contain a table with file specs for each malware type. These specs can be used to identify them using file size and checksum, and be of interest to other researchers working on the same samples.

The tables include results from VirusTotal. VirusTotal is a free web service that accepts files and scans these with a wide range of antivirus software. [12] Numbers indicate the number of software that considered the particular sample as suspicious and the total number of software used.

4.1 BlackEnergy

BlackEnergy is a HTTP based DDoS botnet from 2007, which consist of clients mainly in Russia and Malaysia. [13] It was used to perform DDoS attacks, mostly against Russian web servers.

This botnet was configured and set up by us, and will serve as the control sample. The botnet consists of one client that reports to an HTTP location we decide on set intervals. This will prevent us from relying on outside factors such as existing command and control servers.

File name	crypted__bot.exe
File size	12382 bytes
MD5	9bf2f13b3d21dd4dc00f4ea0bdef503a
Alternate names	kbot
Virus total detection	26/ 39 (66.67%)

Table 3: BlackEnergy description

BlackEnergy uses a packer called "Stalin" to hide its program file. It was the packed file that was submitted to virustotal.com.

4.1.1 Network behavior

Clients report to C&C servers on regular intervals, such as every 10 minutes. This is done using a HTTP post:

```
POST /stat.php HTTP/1.1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)
Host: www.somehost.net
Content-Length: 43
Cache-Control: no-cache
```

Figure 8: The default BlackEnergy HTTP POST. Can be modified by the user

If the client is asked to participate in a DDoS attack a wide range of DNS floods can be seen, depending on the command. Supported floods are: [14]

- ICMP
- SYN
- UDP
- HTTP
- DATA
- DNS

The bot will only perform these if asked to by the C&C server.

4.1.2 Client behavior

Clients can be asked to download and execute a file [14], but what is actually downloaded depends on the person running the C&C server. If no extra malware (such as fake antivirus) is installed, there will be no visual indication that the client is part of a botnet.

4.1.3 Spreading

The kit does not provide a method of spreading, instead it relies on the user to do this. It's up to the user to spread the exe file that is generated, e.g. by using publicly available exploit code for various vulnerabilities.

4.1.4 Lab testing

Our client was configured to connect externally every ten minutes. This was reflected in the traffic that we observed. We did not ask our single client to perform any DDoS attacks, which would have been illegal even if it's only one bot, and thus observed no such traffic.

4.2 Conficker

Conficker / Downadup is a worm that was first discovered in late 2008. [15] It is currently very active with over 8 million infections based on f-secure's estimations. [16] It is however impossible to say how accurate this number is; if the method is wrong the chances are there are less infected clients, if it's right the number of infections is probably higher. The botnet and the malware associated with it are still evolving and are likely to change.

There are currently several versions of Conficker, among those Conficker.A, Conficker.B and Conficker.C.

File name	downadup.dll
File size	165397 bytes
MD5	62c6c217e7980e53aa3b234e19a5a25e
Alternate names	downadup, kido
Virus total detection	38/ 40 (95%)

Table 4: Conficker description

The sample we have selected is of the Conficker.B type.

4.2.1 Conficker.A

Rather easy to detect, Conficker.A contacts trafficconverter.biz, from which it got it's name. Can easily be detected using existing solutions, such as DNS blacklisting. Information later in this thesis regarding domain generation and spreading is not valid to Conficker.A.

4.2.2 Conficker.B

Conficker.B adds more methods of spreading, as well as a few other changes. [17] Instead of a static domain it automatically generates domain names based on time, 250 each day. [18] Conficker.B uses whatismyip.com to retrieve its external IP address. This is a characteristic that has been used in some degree to detect possible infected clients.

4.2.3 Conficker.C

This variant increases the amount of domain names generated each day from 1st of April, selecting 250 domains out of a pool of 50000. [19] There have also been reports of this version installing fake antivirus and other malware, such as the Waledac trojan. [20]

4.2.4 Network behavior

In addition to the spreading behavior, the worm can also connect externally to download additional files. To be able to do this the Conficker worm uses dynamically created domain names. In the first step of this process the worm contacts either Yahoo or Google to get the current system date, which again is used to randomly create domain names using a secret algorithm.

This algorithm have already been cracked, and a group consisting of several security firms are using the algorithm to pre-register these domains, thus being able to halt the worms update functionality. This task becomes virtually impossible as the amount of domains generated increases. [21]

4.2.5 Client behavior

Currently Conficker.A and Conficker.B does not do anything that gives a direct visual result such as installing rogue software, but they do make changes to the system, such as prevent access to a list of security related websites, create scheduled tasks, add registry keys and add files to USB drives.

Conficker.C might exhibit visual behavior, but only if the optional fake antivirus products are installed.

4.2.6 Spreading

The worm currently spreads through various ways, but gets it's worm nature by exploiting a bug in Windows Server Service. [22] This bug was patched by Microsoft in the MS08-067 patch, but as the infections show there are still many that have not updated.

It also spreads by attacking network shares, trying different passwords to get access. If successful it copies itself to the administrator share and starts itself on the remote computer as a service. [23]

USB Drives that are plugged into an infected system can also be used as carriers by adding a trojan as an autorun file. This also makes it possible for hosts without internet to be infected.

4.2.7 Protection features

When installed it disables several security related applications and services to protect itself. [24]

4.2.8 Lab testing

Conficker.B and .C was seen scanning for other clients, using layer two ARP broadcasts. If a client was found a login request was sent to the client using the local username and passwords from a dictionary. Several other usernames are also used and we have also seen the Windows Server Service exploit being executed towards the client. It was also seen connecting to randomly connected domains and sending information to them.

4.3 Danmec

Danmec is a trojan that was first introduced in 2005. Since then it has evolved and added more components, and it is still active today. It consists of several different modules that each serve a specific purpose. [25]. One of its most commonly known modules are called Asprox.

After installation of this module the client becomes a member of the Asprox botnet. The infected client also sends information about itself to the controller. Currently the Asprox botnet has somewhere around 300 000 nodes. [25]

File name	load.exe
File size	73216 bytes
MD5	1311f650aa1209a3ec962b6a9a38fc98
Alternate names	asprox, aspxor
Virus total detection	37/ 40 (92.5%)

Table 5: Danmec description

Asprox is a module that Danmec uses to add functions like spreading via SQL injection attacks towards ASP sites. This functionality has given Danmec a lot of publicity, and in many cases characterized its behavior.

Many antivirus companies choose to name the Danmec trojan after the asprox module, likely because Danmec is almost always instructed to download this module. They are however separate and should not be seen as the same malware.

4.3.1 Network behavior

Infected hosts can be seen performing a number of actions, of which phoning home to get updates and performing SQL injection attacks are easiest to see. These two behaviors are network behaviors and thus of interest to us.

4.3.2 Client behavior

The infected client is often loaded with a fake antivirus program, which informs the computer user that the computer is infected with some virus, but that you are required to purchase an upgrade/full version to get rid of it.

This scam technique is also in use by many other trojans and is of interest to users because it makes it easy to spot an infected computer as long as they know what sort of software they have installed themselves.

4.3.3 Spreading

The SQL Injection component is used to spread to new hosts. It uses Google to find ASP pages, and tries to SQL inject a JavaScript that uses various exploits on visitors to that page. [26] If successful the visitor will then in turn be a part of the Danmec botnet.

4.3.4 Lab testing

The Danmec trojan was tested in our malware lab using the procedures specified in the *experimental setup*. Upon execution the trojan contacted www.yahoo.com and www.web.de by sending SYN packets. When the sites answered the SYN, the client (trojan) closed the connection. This was probably done to check if the client had internet access, and by contacting "legal sites" this procedure would not raise any suspicion.

After successfully being able to verify its internet access the trojan started reporting to the C&C server. The control server IP address is 203.174.83.75 which is in an address pool belonging to NewMedia Express located in Singapore. The trojan contacts this server by sending an HTTP POST packet containing 1810 bytes of data.

```
POST /forum.php HTTP/1.1
Host: 203.174.83.75:80
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)
Accept: */*
Accept-Language: en-gb
Accept-Encoding: deflate
Cache-Control: no-cache
Content-Type: multipart/form-data; boundary=1BEF0A57BE110FD467A
Content-Length: 1085
```

Figure 9: The HTTP header of the packet sent to the control server

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS ( msg:"HTTP POST
request boundary matching to Trojan.Asprox detected"; flow:established,from_client;
content:"POST|20 2F|forum|2E|php"; nocase; offset:0; depth:15; content:"boundary|3D|
1BEF0A57BE110FD467A"; nocase; distance:0; sid:2003179; rev:1; )
```

Figure 10: Snort detects Danmec by looking at the line "POST /forum.php"

The interesting thing about this HTTP POST is that it is sent with fairly equal time intervals with some slight derogations. The time interval is ten minutes, with a few skipped updates averaging to twelve minutes between updates.

4.4 Waledac

File name	free.exe
File size	420864 bytes
MD5	144dd3f6aebf07048addcd7ca874d2ab
Alternate names	waledec, iksmas
Virus total detection	15/ 40 (37.50%)

Table 6: Waledac description

4.4.1 Network behavior

This worm communicates with other nodes over port 80 using standard HTTP traffic with an encrypted payload. These nodes forward the traffic to other nodes which in turn forward to other nodes generating a P2P like botnet over HTTP. The IP-addresses of these nodes are hardcoded into the trojan binary. [27]

4.4.2 Client behavior

Upon installation Waledac scans through all of the local systems files in search for email addresses for use as spam targets. The data gathered is also sent to one of its control servers. [28]

4.4.3 Spreading

The Waledac trojan mainly spreads through spammed e-mail. It also utilizes social engineering by tricking people to execute the malicious file. This was seen during the last US president election. Fake Obama web sites tempted the visitors with spectacular news. To access the news you would need to execute a program. This was of course the Waledac trojan. [29]

In march 2009 Waledac spammed users with reports of terror attacks in the receivers local area. The e-mail contained a link to a site posing as a Reuters news article. The site used the visitors IP-address to find a locations nearby, and then use that name as the target of the terror bombing. A video of the bombing was available if you downloaded and installed an executable file. Again this was the Waledac trojan. [30]

A typical strategy for Waledac is to spam e-cards during holidays. The e-mail promises that the e-card contains a greeting of some kind related to the holiday. To

see the e-card you have to follow a link. The link points you to the Waledac trojan. [31]

4.4.4 Lab testing

The client communicates with the other servers over HTTP. The payload of the post is not very readable and seems to be encrypted with some sort of strong encryption technique. Please take note of the HTTP parameters "a=" and "b=" in the payload, since these parameters are used by several other trojans.

```
POST /dawww.htm HTTP/1.1
Referer: Mozilla
Accept: */*
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla
Host: 24.175.141.95
Content-Length: 957
Cache-Control: no-cache

a=_wAAArmlulT2zuigy47e8KufXAkDvcFRujUV4KdeizQlggnAEtChW9jZP5qt0Bq8[...]&
b=AAAAAA
```

Figure 11: An HTTP POST sent from a Waledac infected client to one of its peers

The trojan was also seen contacting one of its fast flux domains which in this case was usabreakingnews.com. The TTL value of this domain is 0 which implies that each time someone queries it a new IP address is returned.

5 Proposed flow based LRPI and SWE IDS

All the code written by us is written in Python. It's designed to work independently of what operating system it runs on, and although the target hardware runs Linux only thus there's no actual need for this it's useful during development as not all the developers are using Linux on their workstations. It also opens up for better code reuse. Python looks in itself like a pseudo code language and should be easy to understand.

The target platform is an existing platform created by Telenor, which dictates what sort of software and hardware is available:

Debian Linux with:

- Python 2.4
- Argus 3.0

This will also serve as the test platform.

The hardware is irrelevant as comparison data will be collected on the same system.

5.1 Class structure

Logical parts of the implementation are split in classes. There are two main classes, AutomataController and Automaton, as well as a helper class called ArgusData which is used to pass data around.

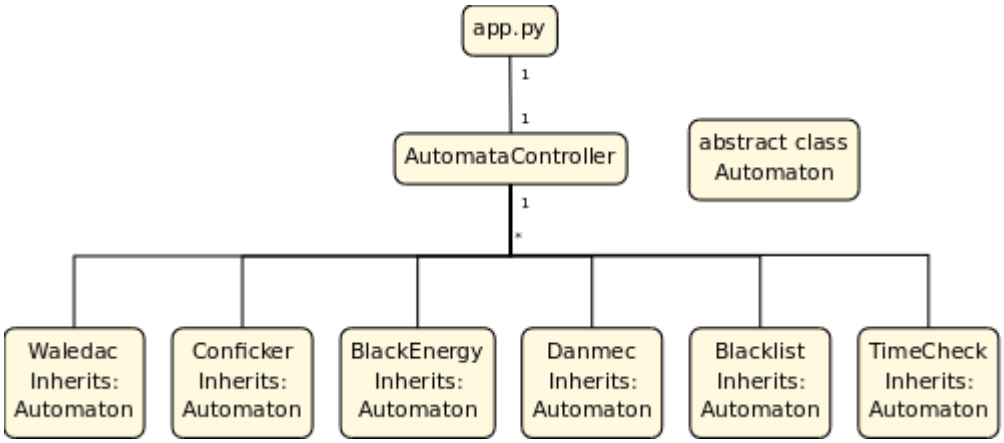

Figure 12: Automata class hierarchy

This chapter will describe the various parts in more detail. If you feel lost refer back to this chart to get an overview.

5.1.1 Main application

The main app is not a class but the application entry point, implemented in app.py. It has the following responsibilities:

- Create an instance of the AutomataController class
- Execute radump to retrieve Argus data
- Pass this data to the AutomataController

To understand how the data is analyzed it is important to understand what the data looks like and what it contains. We have mentioned previously that Argus stores flow data in an Argus specific file format. This file format is of little interest to us, as it is much easier to call Argus tools from Python to extract the data we need. This is done in app.py, like this:

```
def getArgusData(argusFile):
    command = "radump -r " + argusFile + " " + "-n -s saddr daddr sport dport sbytes dbytes stime -c ,"
    process = os.popen(command)
    lines = process.readlines()
    process.close()
```

Figure 13: Argus data import code

This function is called with an Argus data file as argument, and returns the Argus data as comma separated values, in a list of strings. It parses the Argus file by calling radump, specifying which Argus fields to retrieve, using the -s option. The -c option tells radump that we want the data comma separated. Python automatically waits for this process to finish, and reads all the lines in the buffer to an array. This array is then returned.

```
lines = getArgusData(watchFolder + "/" + fileName)

for line in lines:
    ac.parse(line.replace("\n", ""))
```

Figure 14: Application passing data to the automata controller

Each of the lines in the returned array will be passed to the automata controller, as seen in *Figure 14*. The controller then processes these values further into the final data format.

5.1.2 AutomataController

The app only has one AutomataController, and this controller is responsible for all the automata that that are running, if any. When the application is first launched there will be no automata, as the controller hasn't received any data yet. Once data is received, the controller will determine what source-IP / destination-IP pair this data belongs to. If automata for this pair does not exist, create it, then pass the data it received to these automata. For every IP pair there will be a set of automata, as there is one specific automaton for each malware we have selected.

Passing each string that is received from app.py to the automata is not practical, as changes such as adding a field or changing the delimiter would require extensive code changes in many places. Instead, the automata controller further processes this string into a Python data type. This Python class is the data type that the code in Automaton and the inherited classes use.

Every time the automata controller receives a string of data an ArgusData object is constructed, using the Argus data string as an argument:

```
class ArgusData:

    def __init__(self, data):

        parts = data.split(",")
        adata = ArgusData(parts)

        self.srcAddr = data[0]
        self.dstAddr = data[1]
        self.srcPort = data[2]
        self.dstPort = data[2]

        if not self.srcPort == '':
            self.srcPort = int(data[2])

        if not self.dstPort == '':
            self.dstPort = int(data[3])

        self.srcBytes = int(data [4])
        self.dstBytes = int(data[5])
        self.time = data[6]
```

Figure 15: Argus data class

First the string is split into an array with comma as the delimiter. Each value is then assigned to class members with a descriptive name. Once this is done the data from the binary Argus file is in a format that is easy to use and native to our Python application, and in the format that is expected by the Automaton class, which is next in line to receive the data.

```
    srcip_dstip = adata.srcAddr + " " + adata.dstAddr
    dstip_srcip = adata.dstAddr + " " + adata.srcAddr

    check1 = srcip_dstip
    check2 = dstip_srcip

    if(check1 in automata):
        automata[check1].analyze(adata)
    elif(check2 in automata):
        automata[check2].analyze(adata)
    else:
        automata[check1] = Automaton(adata.srcAddr, adata.dstAddr, adata.time, check1)
        automata[check1].analyze(adata)
```

Figure 16: Automata controller passing data to Automaton

First we check if an automaton of this IP pair already exist. *automata* is a hashmap with automata, the index being a string with the source and destination IP. Since we don't know in which order the two IPs first occured we try both. If the automaton already exists the data is passed to it. If it does not exist a new automaton is constructed, and the data passed to the new automaton. *Figure 16* contains the important parts of this code. The implementation contains additional logic to deal with different automata types.

After the automata controller is done sending data to the correct automata (if there are more than one specific automata type running there will be more than one automaton for each pair) the code flow returns to app.py which parses the next file, if it exists.

5.1.3 Automaton discarding

The AutomataController class has the ability to discard automata. There are two strategies to select the automata that are candidates for removal. The first approach selects automata based on how long an automaton has existed without getting any new data. If an automaton have been in memory for a certain amount of time without any updates it is prone for deletion. This could allow the deletion of all automata that have not received any updates the last e.g. hour.

The other approach removes automata if the traffic is considered not to be of malicious nature. If an automaton has reached a threat level considered safe, and also has a sufficient amount of updates it may be removed from memory. An automaton might need to reach a threat level of 30% or lower, and at the same time have at least 10 updates. The threat level always starts at 50%.

Both of these are implemented in the same place, as they are not particularly complex.

```
remove = []

for k, a in automata.iteritems():
    # Schedule old inactive automata for deletion
    if time - a.getUpdateTime() > timedelta(seconds=1800):
        remove.append(k)
    # Schedule safe automata for deletion
    elif a.getUpdates() > 10 and a.getThreatLevel() < 0.3:
        remove.append(k)

for k in remove:
    del automata[k]
```

Figure 17: Automata removal code

first an empty list called *remove* is created, to keep track of automata that are to be removed. The time difference since last update is then calculated, and if this delta is larger than half an hour (1800 seconds) the automaton is added to the removal list.

If the automaton wasn't too old the number of updates and the current threat level is checked. If both number of updates is larger than ten and the threat level is smaller than 0.3 it is added to the removal list.

Lastly the removal list is iterated, removing all the automata in it from the automata controller's *automata* list.

Discarding automata is not required for operation but will reduce the memory usage.

5.1.4 Automaton

Automaton is the base class of all the specialist malware classes. It sets the template from which you start when implementing a new type of malware. It defines methods for receiving data, getting feedback, guessing and analyzing. Children of this class can override functions to suit that particular type of malware, however not all functions are necessarily required to be replaced. In those cases the default implementations in the Automaton class will be used instead.

The member functions of the Automaton class are:

Function name	Arguments	Return type	Description
getGuess	self	boolean	Uses previous experience to guess if the data is malicious
analyze	self, data	void	Analysis the argus data and updates threat level
isOld	self, time	boolean	Returns if the current Automaton is old, based on the time distance from the time argument
useData	self, data	boolean	Decides if the argus data is to be used. Can be replaced in inherited class.
getFeedback	self, data	boolean	Looks at the Argus data and decides if it is malicious. Dummy function that does nothing, replace in inherited class.
getThreatLevel	self	float	Returns the current threat level
getUpdates	self	int	Returns the number of updates this Automaton has received
getUpdateTime	self	string	Gets the time of the last used piece of data
generateGraph	self, type	void	Generates a graph for the current Automaton

Table 7: Member functions of Automaton class

Refer to *Table 7* when creating a specific malware automaton. Replace functions as needed.

The ability to generate graphs is an entirely practical part of the class that is not related to detection, but rather allows us to visualize data over time. These graphs will be used later in this thesis, and they were also used during development to understand what was happening.

Let's take a look at how the Argus data flows inside a single automaton. The data an automaton receives is guaranteed to be for the IP pair it is assigned to, courtesy of the *automata controller*.

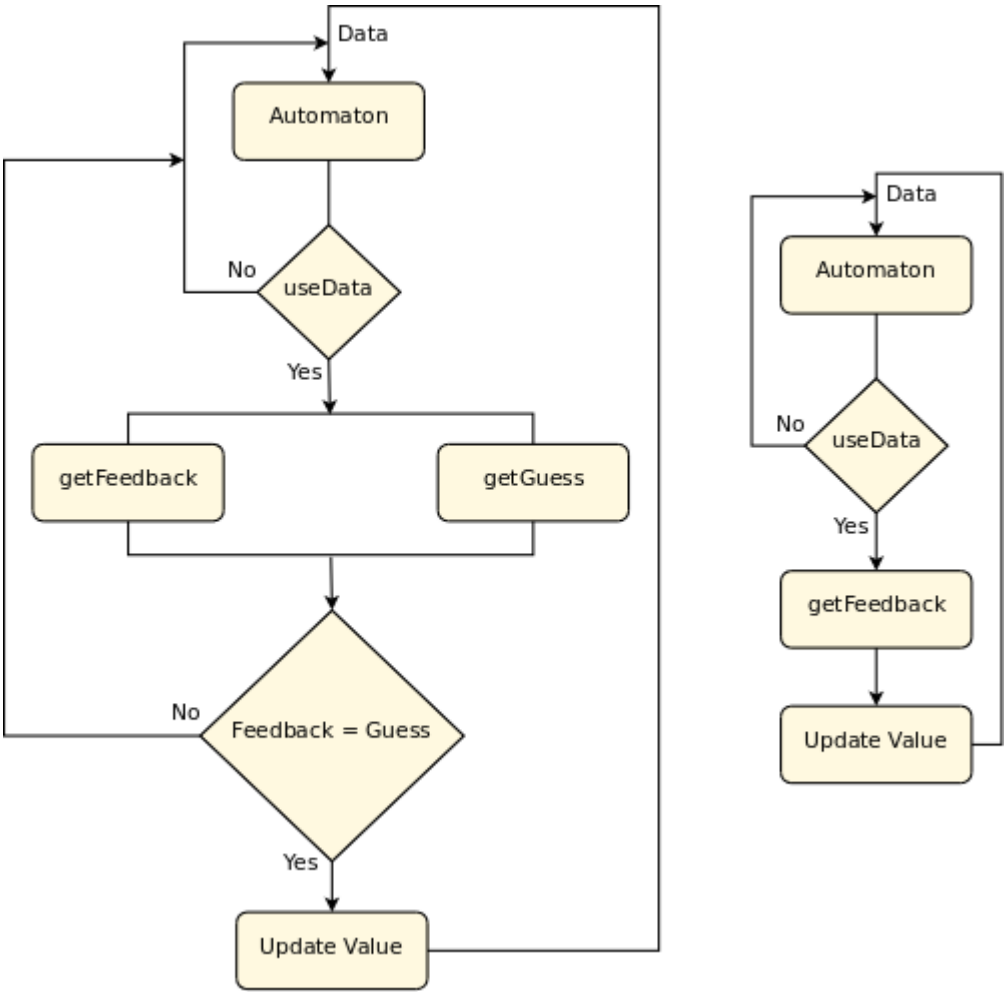

Figure 18: Automata flowchart

There are two paths the data can take, depending on which algorithms is used. The left chart displays the linear reward penalty inaction data flow, and the right the stochastic weak estimator data flow. These will be detailed in the *algorithm chapter*. The arrival of the data is common for both paths, where it first is checked for usability. If the data does not fit the specific criteria, such as being port 80, it will be discarded. This is determined by the useData() function.

Once the data is valid for use the two algorithms take different paths. The basic principle is to get feedback from the feedback function and use this value to determine if the data is malicious or not, and modify the current threat value based on this. Once the threat value has been updated the automaton again does nothing until new data arrives.

5.1.5 Implement based on rules

The specific automaton classes are created to detect a specific type of malware or pattern. The classes will typically replace two of the functions inherited from Automaton, getFeedback() and useData().

The useData() function determines whether input sent to the automaton is going to be used to give a feedback. To avoid many false negatives, and positives it is necessary to filter out some traffic. Typically we do not want to analyze traffic on ports that the malware we are looking for is not using. Most malware use port 80 to post to their C&C servers, so a common rule here will be to filter all other ports. Filters like this may be based on all the data fields we have chosen to use in our analysis.

The getFeedback() function is responsible for the analysis of the data. When data passes the initial filter getFeedback() start its analysis. If the data feed to the automaton fits the description given here it will return true, thus giving the automaton the possibility to increase its threat level. If the data does not match the description, it will return false, and the automaton may decrease its threat level.

Each of the specific automaton define rules for what is considered malicious traffic, which on the surface is comparable to Snort signatures. The difference is that the automaton rules are actual Python code and can thus do complex tasks. This is seen clearly in the *Timecheck automaton* where data is collected over time. We have decided to use the term *rule* for automata and *signature* for Snort to distinguish them. This will be used throughout the thesis.

To give a concrete example on how to write the specific automaton class based on a rule we will invent a type of malicious traffic, define a rule for it, and implement a specific class based on this rule. This will also allow us to present the rules in the next chapters that cover specific automata without the need to insert large amounts of code. The made up scenario is as follows:

After testing Malware X in the lab we see it is a worm type of malware that can spread to other hosts using a fictive Linux SSH exploit towards port 22. It also reports

to a C&C server over HTTP on port 80. SSH is encrypted and makes analysis difficult. The the data that is sent to port 22 seems random in nature and it was not possible to find a pattern to the traffic. The HTTP posts were done at random intervals but the source size only varied from 1000 to 1010 bytes and the return sizes from 150 to 180 bytes.

The return size should only be used for detection if excluding it results in many false positives, which would happen if the source size range was very large. The return size will certainly change if the C&C server goes offline or changes, as there might be no return data at all, or a HTTP 404 page if it's still up. With this in mind we create the following rule with a little bit of leeway:

Source size	Destination size	Port
995 - 1015	n/a	80

Table 8: Malware X detection rules

This type of activity is common for malware, where one activity is quite random, such as scanning for vulnerable hosts, but with a predictable traffic pattern as well. The implementation will then use this observation to exclude the traffic that is random and use the traffic that is predictable. Using the Automaton class as the base class we get the following implementation:

```
from automaton import Automaton

class MalwareX(Automaton):

    def getFeedback(self, data):

        if data.srcBytes >= 995 and data.srcBytes <= 1015:
            return True

        return False

    def useData(self, data):

        if data.dstPort == 80:
            return True

        return False
```

Figure 19: Malware X automaton implementation

The class MalwareX is created using Automaton as the base class. Two of the functions inherited from Automaton is then replaced to use the rule we created. The SSH traffic was not predictable, so it and everything else except HTTP is filtered in the useData function.

The size interval we saw on the posts is then checked in getFeedback. If the size fits between the border values the traffic is considered malicious. If it does not it is considered non-malicious.

Looking at this simple implementation it might be easy to think that it would trigger a lot of false positives. After all, it doesn't care what the content of the HTTP post is. Port 80 traffic is web traffic, and the massive amount of traffic that usually passes on this port works to our advantage.

The traffic seen by an instance of this class that analyzes data sent to and from an online newspaper would vary greatly in size as each article has different content. Few data transfers would match the rule, resulting in a low threat value. A low threat value would indicate that this traffic isn't malicious.

We have implemented four different types of specific automata based on this technique: Conficker, Waledac, BlackEnergy and Danmec. Blacklist and TimeCheck use the same base class but different approaches.

5.2 Specific automaton

For each of the malware samples we developed a detection class that inherits the Automaton class. These use the data that was found during malware research to detect and distinguish them. For the most part they use a pair of source sizes and destination sizes, filtering on port number. It is also possible to filter using the size values. Values in this chapter have been created using *test set 1*.

Initially we wanted to develop a general automaton that detects several types of malware. This is reflected in the categorization requirement. The slight change to a set of specific automata completes this requirement.

5.2.1 BlackEnergy

BlackEnergy is fairly easy to detect. When reporting to a control server, it will always post the same information. This means that the rule used to detect it can be very specific, matching few characteristics of regular traffic.

The length of the post may however differentiate between different botnets. This is because servers may be set up with different folder structures, and thus different post size. An example could be "http://server.com/postdir/" while another botnet may use something like "http://server.com/blackenergy/host/postings/" the difference between these examples are 18 bytes, thus forcing a much wider rule to cover both.

When detecting BlackEnergy it is therefore most efficient to make rules per BlackEnergy based botnet, and not for BlackEnergy in general. Our implementation is tuned for a BlackEnergy botnet we set up as a test. It only contained one client, and reported back to a server we controlled.

Source size	Destination size	Port
565 - 575	n/a	80

Table 9: BlackEnergy detection rules

The post done by BlackEnergy bots are, once known, very specific, so only one size range was necessary to detect it. The rule is listed in *Table 9*. The data sent to the automata were first filtered so that only data over port 80 was analyzed. All sessions where data only went one way was also ignored.

5.2.2 Conficker

The Conficker class is designed to detect the Conficker trojan. All traffic generated by Conficker is directed to port 80 at the command and control servers. This means that we may discard all other ports when analyzing data, reducing the number of false positives. We also skip all observations where data is only going one way. This is typical for a client trying to contact an unresponsive server.

Source size	Destination size	Port
240 - 245	n/a	80
300 - 306	n/a	80
360 - 370	n/a	80
420 - 440	n/a	80

Table 10: Conficker detection rules

In *Table 10* a complete list of the rules used in Conficker detection is available. The source size represent how many bytes the source have sent to the destination during a session. Destination bytes are the number of bytes returned. In this implementation we ignore the return size, as it varies too much.

5.2.3 Danmec

When detecting Danmec there are two possible strategies. Danmec is very consistent in its behavior, and reports to a command and control server with a regular interval of approximately ten minutes. Sometimes one or two consecutive check ins may be skipped, creating intervals of twenty or thirty minutes. This complicates detection based on intervals alone.

Source size	Destination size	Port
1808 - 1812	n/a	80
1940 - 1960	n/a	80

Table 11: Danmec detection rules

The other strategy, which we use in our class, is detection based on number of bytes passed to the server during a session. A list of the selected byte intervals used in detection can be seen in *Table 11*. To limit the number of false positives, we only consider data on port 80. All sessions where data only goes one way is also ignored.

5.2.4 Waledac

How to detect Waledac turned out to be a bit different than we had expected before we created and analyzed *test set 1*. The connections we saw there were similar to the other malware that reports to a control center, mainly consisting on information uploaded to a web server using the HTTP protocol on port 80. After creating the rules seen in *Table 12* and running it on the same Argus that that was used to deduct them it detected nothing.

Further investigation showed us that there were several Waledac automata that had threat levels above 50%, but that none of them had enough observations to reach the threshold at 70%. This happens due to the randomness of the reporting; only a few check ins to each destination, usually from two to four. Waledac uses fast flux to get information about what destination to contact. As Waledac only has a limited number of sessions with each server, there will never be enough data to detect it.

To circumvent this behavior a single automaton is started for each source instead of having one Waledac automaton per source/destination pair. In our analysis of Waledac we found that the three pairs of data specified in *Table 12* yielded good results.

Source size	Destination size	Port
186	162	80
740 - 780	850 - 920	80
4450 - 4750	4300 - 4400	80

Table 12: Waledac detection rules

As the table also shows, all data is going over port 80. The data received from radump is first filtered based on the source size. This means that a lot of false positives are avoided. If the data passes the first filtering, and destination size matches the given source size, the automaton increases its threat level.

5.2.5 Blacklist

Our Blacklist automaton works in much the same way as the malware specific automata, as it inherits Automaton, and have the same data available when it does its analysis. However, where the malware specific automata analyzes the amount of data sent back and forth in a session, the Blacklist automaton compares the destination address to an existing list.

By marking known bad IP addresses you may get an early warning when an infected client contacts its control server. As this is a well known strategy we will not be using it in our test. The automaton was implemented simply to demonstrate that our solution have the ability to mark IP addresses as bad.

To put this automaton into production some changes are required. First of all, the learning rate would have to be increased to ensure that an alert would be issued fast or even instantaneously. A slow learning rate for a blacklist automaton would make it less useful as other automata could trigger before it, making it unnecessary.

If the automaton was configured to trigger at the first update, meaning the destination IP was in the blacklist and the learning rate is sufficiently high, it would allow us to remove automata after only one observation. If the destination IP is not in the blacklist, there is no need to keep it in memory which is true for all learning rates.

Because we only have a few specific IPs in our blacklist we decided against having the Blacklist automaton in our tests. It is provided here as an example.

```
def getFeedback(self, data):
    return True

def useData(self, data):
    for ip in blacklist:
        if data.dstAddr == ip:
            return True
    return False
```

Figure 20: Blacklist class implementation

The Blacklist class only uses data if the IP address is found in the blacklist, as seen in useData(). If the IP is found, getFeedback() will always return True, thus increasing the threat value.

The blacklist is read from an external file when the first Blacklist automaton is created, and saved in a global list accessible to all new Blacklist automata.

5.2.6 Timecheck

The previous automata have been specific to one of our samples, which doesn't make them very flexible. We also wanted to develop a non-specific automaton that look for a common trait to many malware. This is implemented in the TimeCheck class and fulfills one of the requirements.

$$Total\ time = Average\ time \cdot Number\ of\ updates \quad (1)$$

$$Total\ time = Total\ time + (Data\ time - Previous\ time) \quad (2)$$

$$Average\ time = \frac{Total\ time}{(Number\ of\ updates + 1)} \quad (3)$$

Figure 21: TimeCheck formulas

This automaton works by calculating a running average of time between each update. If the number of updates is high and the time since the previous update is close to the current running average it is perceived as a threat. This code is implemented in the getFeedback function, again using Automaton as the base class.

```
def getFeedback(self, data):

    diff = data.time - self.updateTime

    totalTime = self.avgTime * self.updates
    totalTime += diff
    self.avgTime = totalTime / (self.updates + 1)

    if self.updates < 5:
        return False

    diff = self.avgTime - diff

    if diff >= timedelta(seconds=-5) and diff <= timedelta(seconds=5):
        return True

    return False

def useData(self, data):

    diff = data.time - self.updateTime

    if diff < timedelta(seconds=30):
        return False

    if data.dstPort == 80:
        return True

    return False
```

Figure 22: TimeCheck class implementation

This implementation is not based on a specific malware sample and is thus not based on rules. Instead it only filters the data on port 80, since we have rarely seen malware that does not report over this port.

The learning algorithm is not touched and is either LRPI or SWE like the other automata. Because of this the threat value will go up or down over time depending on how many of the posts are close to the running average. This is done by comparing the time since the previous update to the running average.

If the delta since the last update is from 5 seconds smaller to 5 seconds larger than the running average it is considered malicious. This difference will be referred to as the TimeCheck window, or just window, from now on.

The first five reports are reported as non-malicious by default as the running average hasn't stabilized yet. useData() also drops data that is posted in less than 30 seconds since the last post, as some malware post a few times before idling a constant amount of time. This was implemented because we saw this in the Danmec data in *test set 2*.

5.3 Learning algorithms

Originally we had decided to use the learning algorithm that Morten Kråkvik used in his thesis, as described in the related works chapter. His algorithm used data similar but not identical to netflow data, and thus should serve us nicely. Even so, a comparison with a different algorithm makes sense, as it can showcase how much the algorithm matters, and if one is more suited than the other.

If the term *learning rate* is used outside the context of a specific algorithm it is meant as a general term for algorithm configuration. This will be seen throughout the thesis.

5.3.1 Linear reward / penalty inaction (LRPI)

LRPI works by modifying the current value depending on the result of the guess and feedback functions. Inaction means that nothing happens if the Automaton is wrong.

```
if feedback and guess == feedback:
    self.value = self.value + LEARNING_RATE * REWARD * (1.0 - self.value)
elif feedback == False and guess == feedback:
    self.value = self.value - LEARNING_RATE * REWARD * self.value
```

Figure 23: Linear reward/ penalty inaction

Learning rate defines how fast the value changes, and in our test cases this was set to 0.05. A higher number results in quicker detection, but increases the chance of false positives. The best value to set depends on the network that is being analyzed and should be decided by tests.

Reward is set to 1 and is included for reference. It is only needed when using penalties to allow penalties to have a higher or lower effect than rewards. As we do not use penalties and the value is 1 it can be removed.

The LRPI automaton guesses if the received data is malicious. To be sure previous experience is taken into consideration the guess is done like this:

```
def getGuess(self):
    guess = random.random() < self.value
```

Figure 24: LRPI guess function

First a random number between 0 and 1 is selected, then compared to the current threat value. If the current threat value is high, the likelihood of a malicious guess is

high. In contrast, if the current threat level is low the guess has a bigger chance of being non-malicious. This ensures that previous experience is included in the guess.

5.3.2 Stochastic weak estimator (SWE)

This thesis does not focus on the math behind the learning algorithms but on their use. the SWE algorithm is documented in great detail in Pattern Recognition, vol. 39, no. 1. [32]

SWE does not rely on a guess function but uses the feedback value directly to change the current threat value.

```
self.value = (((SWE_RATE - 1) / SWE_RATE) * self.value) + feedback / SWE_RATE
```

Figure 25: Stochastic weak estimator

As seen by the formula, if the feedback function is malicious, thus one, the threat value is first lowered by the first fraction, then increased by the 1 / SWE_RATE fraction. If the feedback is non malicious, thus zero, the threat value is lowered a fraction of it's current value and the second part equals zero.

5.4 Operating mode

The automata can be executed in two ways, each with a specific purpose. This is needed because we do not do actual development directly on a live network with a lot of traffic, nor would this be feasible. The current mode is selected in app.py and can be changed with a single line of code.

5.4.1 Directory mode

This mode reads all the files in a specified directory, executes them, and prints out the results. After this the application is simply terminated, as it does not have any more work to do. This mode was used during development to feed the automata with the previously captured test data. When adding to or fixing certain aspects of the automata the test data was parsed again, to provide feedback on the change.

The directory that is parsed contains a set of Argus files written by rasplit, the same as on the live network sensor.

5.4.2 Live mode

As the name implies this mode is suitable for live network monitoring. The automata executes and watches for new files, parsing them as they appear. The last parsed file is logged and compared to the second to last newest file. The reason for this is that Argus is currently writing the last file, and thus this file is not ready yet.

If a new file is detected, it will be read by radump and parsed in the automata. Once this is done the automata goes back to looking for new files. The execution never ends. This mode completes one of the requirements.

5.5 Data analysis

The netflow data that Argus captures passes through many layers before arriving at individual automaton. The data flow shown here is the same used for testing, as well as what we propose for actual deployment.

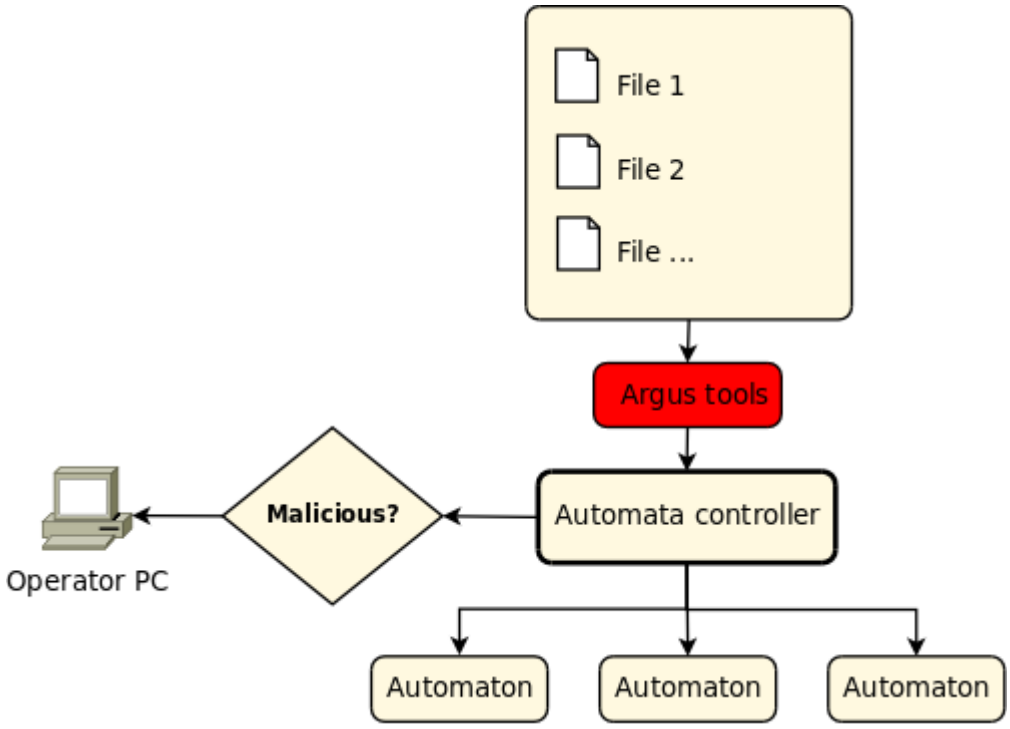

Figure 26: Automata data flow during analysis

The data format between each layer, as seen by *Figure 26*, is quite different. Files on disk is stored in a binary Argus format, and when this data is read from the app using Argus it is converted into comma separated plain text. This plain text is then converted into a Python object that is sent to the running automata from the controller.

The reason for this abstraction is flexibility; if we decide later on to add another field that is available in the flow data, adding this only requires a single change in the data chain. If the binary format or the text format was passed through unchanged every place that uses this data would have to be changed, ending in a code nightmare.

The controller has the job of alerting a human of what is going on. By reading the states of all the automata it is controlling it can reach a decision using a limit criteria. If such an event occurs the controller can send an administrative alert to a human, who can then take appropriate action. The actual method of this alert depends on the

particular environment, but in our case sends an alert to stdout, which is usually the console, when a threshold is reached.

6 Research results

The results are split in two specific categories; detection and performance. Both are highly dependent on the type of network that is being analyzed and what kind of traffic is on that network. Because of this importance we will start with a description of the network that the performance tests were run on.

Snort is used as a reference for detection and performance and the data for Snort will be presented along side the data for our own implementation. The performance tests contain CPU load and memory usage gathering.

Each graph contains the algorithm configuration in parenthesis. For LRPI this value is the value of LEARNING_RATE, for SWE it's the SWE_RATE. The default values are 0.05 and 20 respectively, but some tests use different values for the reasons stated in that test.

6.1 Client network

The client network used in the automata and snort testing consists of thousands of workstations. This makes it ideal for our test, compared to known test sets. This client network was used to create *test set 3*.

The actual count is unknown unless you analyze the data, as not all machines are turned on and connected. A total count of active source IPs in the data set we are working on can be gathered like this:

```
ra -r ./data/* -s saddr | sort | uniq | wc -l
```

Figure 27: Command for counting source IPs

ra (read Argus) will output all the source addresses, which are then sorted, duplicates removed, and lastly counted. On our test set this yields a total of 11090 source addresses. As this is a closed network not accessible from the internet this number is a fair approximation.

The number of work stations combined with the type of traffic determines the amount of connection pairs we will have to analyze. Random web traffic contains many more connections than a backup job between two hosts, even if this connection transfers more data.

6.1.1 Traffic type

The client network bandwidth usage averages close to 100 Mbit/s during work hours. The highest peak is at 140 Mbit/s. The graph *Figure 28* represents the network load during a 24 hour span.

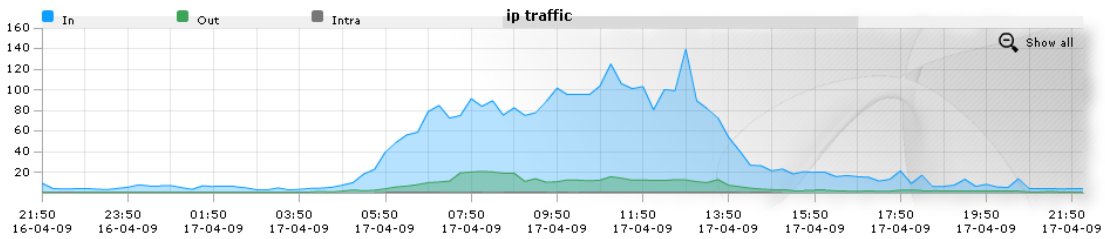

Figure 28: Network traffic 24h work day

Traffic on the network is mostly compromised of http traffic on port 80, which is the same method and port of communication that the malware samples use. It also reflects your average home network (except for size), or corporate workstation network.

6.1.2 Selected period

It is important to select a time period that reflects the top average of traffic, both in quantity and type. This ensures the validity of the test data and of the comparison that follows.

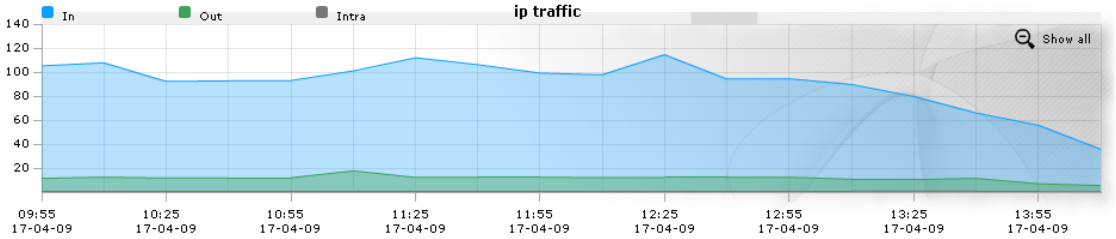

Figure 29: Network traffic 4h work day

In our test, we use data from 10:00 UTC to 14:00 UTC on a work day. It is easy to see that the 10:00 UTC - 14:00 UTC period is the peak hours of the selected network. *Figure 29* describes the selected period in more details. 10:00 UTC to 14:00 UTC represents 12:00 to 16:00 local time. Once time approaches 16:00 people are heading home and the network traffic is reduced. *Test set 3* consists of the Argus data for this period.

6.2 Detection results

Here we will present the results from our detection tests. All tests regarding detection were run using *test set 2*. This set had traffic from all the selected malware samples.

In order to do tests regarding false positives we needed a test set consisting of mostly legal traffic, in addition to the set consisting of malware traffic. *Test set 3* is ideal for these tests, as it is mainly legal traffic, and consists of regular workstation clients.

First we will present the results from Snort detection, and thereafter give the results achieved by our proposed solution.

6.2.1 Snort detection

Type	Number of signatures	Percent
Detection rate	7	100
False positive rate	4	44

Table 13: Detection and false positive rate in *test set 2*

We used 4 different types of malware in our test, and this triggered a total of seven Snort signatures, giving Snort a success rate of 100 percent. It is also worth mentioning that Snort also triggered on a lot of legal traffic like openVPN connections, MSN messenger and online email services.

In total 11 unique signatures triggered where four of these were false positives, or signatures that report legal traffic as malicious. The different malware types and how they were detected are discussed below.

BlackEnergy

The BlackEnergy DDoS bot was detected with two signatures, one by the string stat.php in its post and one by content recognition in the packet. Here the latter is the signature that most likely is the safest one, since the users of this bot is not able to change the content of the packet. But they can, however, change the GET string thus rendering one of the signatures useless.

Conficker.B

A lot of work have been put down to detect this worm, including a special working group. [33] The B variant, which we tested, triggered two signatures. Upon initialization, as described in the *malware section*, Conficker contacts whatismyip.com to retrieve its external IP. A snort signature detecting connections to this address triggered, stating a possible infection, since this type of behavior in most cases are completely legal.

The other signature triggered on Conficker's famous GET request: "search?q=0", where "q=0" indicates how many other clients it has infected, which in this case was zero. One of the major drawbacks with this signature is that it may trigger false positives when users do a search on various web sites. An example is the online store cdon.no. This is because the search GET string posted by users to cdon.no starts with exactly the same string as the Conficker check in request uses, as seen in *Figure 39*.

The signature is limited to only trigger when the first to third symbol in the string is a number, limiting the false positives to trigger only on searches with numbers at the beginning of the string. This is most likely a design feature implemented by the authors of this worm with the purpose of forming confusions.

Danmec

The Danmec worm was also detected with two signatures, both on the POST string that it sends to its control server upon check in. One of the signatures also added some rules for detecting it by the content of the packet, this approach was also successful. These signatures are not known to trigger too many false positives, and they will most likely continue to detect Danmec without making to much noise as long as it does not change its behavior completely.

Waledac

The Waledac worm is trickier to detect on the string of its POST since it deliberately changes this randomly to avoid detection. Snort detected it anyhow using content recognition. This will most likely not produce too many false positives due to the uniqueness of this POST's structure.

False positives

It is impossible to accurately calculate the amount of false positives Snort gave on *test set 3*, as we didn't have control over the actual clients on this network. However, we can look at the count and type of events that were logged. The total number of logged events was 490, including those automatically logged by the system. These were mostly taken up by spyware and malware applications.

None of the malware samples we selected were seen on the day of the test. The total number of snort events was 2624, giving an estimated false positive rate of 81%. This number will not be used for any conclusions in this thesis as not all of the 2624 events were manually analyzed. It also varies a lot depending on what Snort signatures are enabled. Some Snort signatures might also be considered informative but not necessarily indicating malicious activity.

The one conclusion that can be drawn is that Snort generates a large amount of events making it difficult to manually analyze. The sheer amount of signatures also makes it a tedious task to enable or disable them on a per signature basis.

6.2.2 Automata detection

Automata detection tests were run using *test set 2*. The numbers in this introduction are for the LRPI algorithm with a learning rate of 0.05. With the proper configuration SWE would give similar results.

BlackEnergy detection was achieved after 14 updates. The observations started at 2009.04.19-18:29:33, and was marked as malicious at 2009.04.19-20:39:35, giving a detection time of approximately 130 minutes. BlackEnergy was also detected by the TimeCheck automaton. TimeCheck needed a total of 25 observations before marking the traffic as malicious. The 25th observation was seen 2009.04.19-22:29:37, giving it a detection time of approximately 240 minutes.

Conficker was detected with 18 updates. The observations started at 2009.04.19-18:55:07, and was marked as malicious at 2009.04.19-18:57:25. This gives an approximate detection time of three minutes, after the first observation.

The Danmec automata needed 19 updates to reach its trigger value of 0.7. This was achieved at 2009.04.19-20:08:55. The test started at 2009.04.19-18:33:28, so total time before marked as malicious was approximately 90 minutes.

Waledac was detected after 15 updates. The observations started at 2009.04.19-18:28:38, and was marked malicious at 2009.04.19.18-50:15. This gives a approximate detection time of 20 minutes after the first observation.

It was important that the automata reached a decision since this will fulfill one of our requirements.

No. of updates	Source IP	Destination IP	Automata type	Threat level	Runtime
248	192.168.1.71	203.174.83.75	danmec	90 %	1 day, 4:52:36
1155	192.168.1.74	0.0.0.0	waledac_combined	93 %	1 day, 4:57:51
184	192.168.1.73	194.63.248.34	blackenergy	96 %	1 day, 4:50:46
173	192.168.1.73	194.63.248.34	timecheck	100 %	1 day, 4:50:46
138	192.168.1.71	203.174.83.75	timecheck	78 %	1 day, 4:52:30
44	192.168.1.90	221.7.91.31	conficker	95 %	3:05:51

Table 14: Automata detection results, *test set 2*

Table 14 shows the results after a complete run. The total number of updates is vastly different, as it is dependent on the activity rate of the sample.

BlackEnergy

The BlackEnergy bot was detected by the automata by a threat level of 96 percent by the specified automata and 100 percent by the timecheck approach. BlackEnergy makes it possible to adjust both intervals between each check in and the size of the packets sent. By adjusting the latter one could most likely avoid the specified automata. Our bot checked in once every ten minutes, which is the default setting.

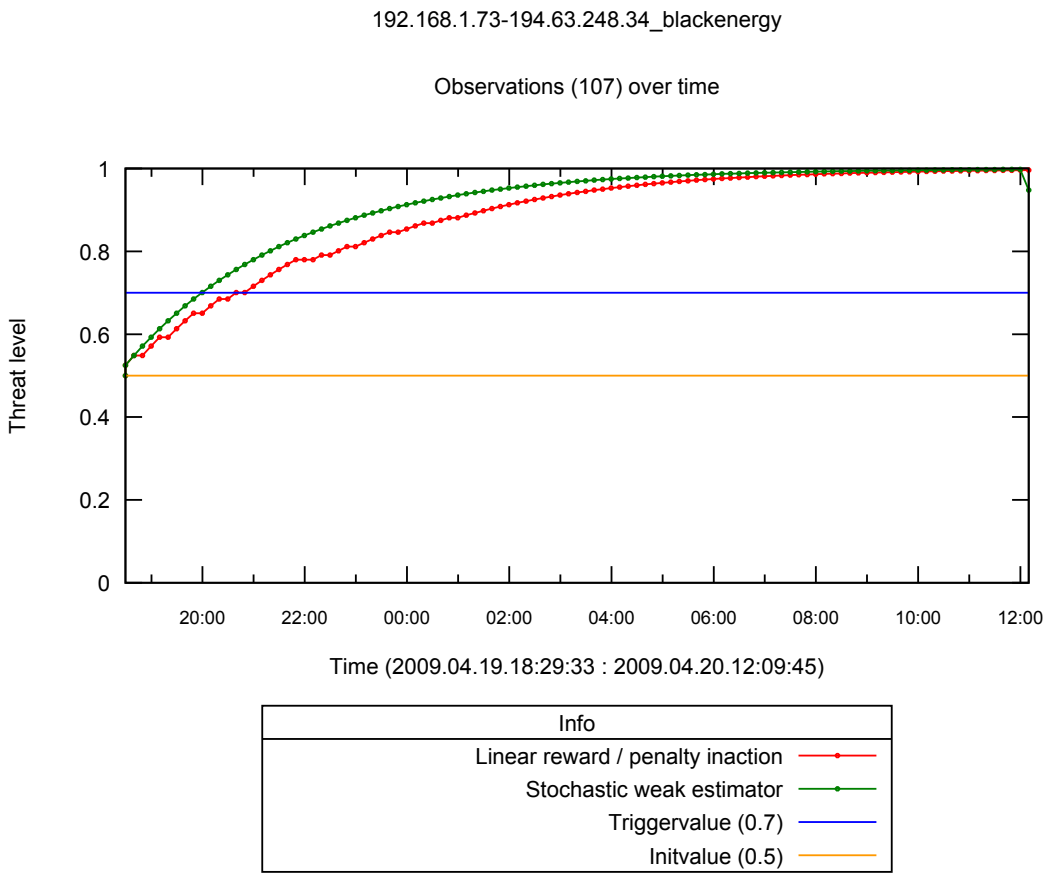

Figure 30: BlackEnergy detection over time, LRPI (0.05) and SWE (20), *test set 2*

In *Figure 30* the regularity of the BlackEnergy post can be seen. In our case the client reported to the control server every ten minutes. This interval is likely to be different for other botnets.

The regular size of the post also makes the graph climb steadily. Most of the observations is well within the interval that is specified in the rule that we defined during development.

Conficker.B

This automata is based on the packets sent between the client and its control server. The communication is initiated at random intervals and cannot be detected by the TimeCheck automata. The packet sizes are regular, so detection using time interval is not required.

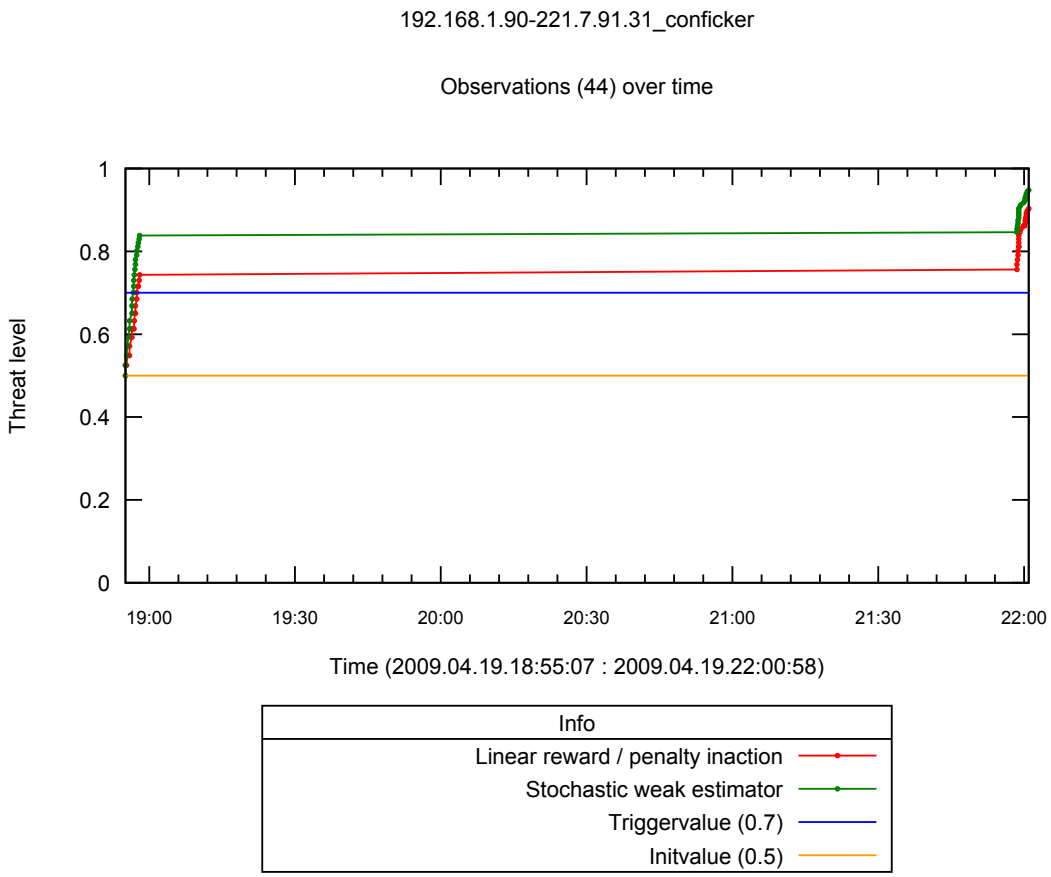

Figure 31: Conficker detection, LRPI (0.05) and SWE (20), *test set 2*

In *Figure 31* the detection of a Conficker trojan is visualized. As the graph shows, there are two update sessions. The first update is at the very beginning of the captured data, where the automaton gets a lot of updates within ten minutes. This is enough to reach our chosen trigger value, and thus give an alert.

Another update session is observed at the end of the period. Here we see a similar increase in the threat level. As the algorithms give a higher increase or decrease when closer to the initial value of 0.5, the second update session increase the threat level at a slower pace.

Danmec

Danmec does, as BlackEnergy, send data at regular intervals. Our test sample did this once every ten minutes, making it possible to detect by the TimeCheck automaton. It is also detected by the specified automata, indicating that this most likely is malicious traffic.

Due to the slight variation in Danmec posting the TimeCheck automaton requires a large window. This makes TimeCheck less ideal for Danmec detection.

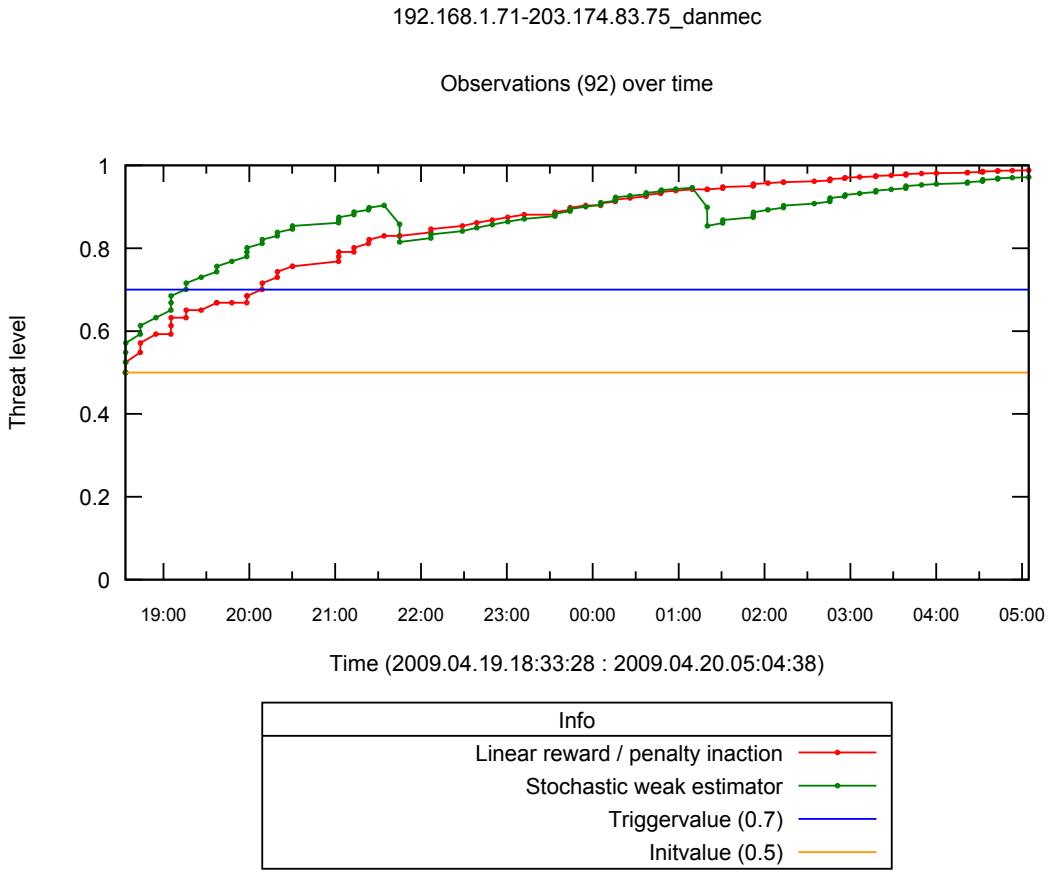

Figure 32: Danmec detection, LRPI (0.05) and SWE (20), *test set 2*

In *Figure 32* Danmec's update behavior over time is illustrated. Our automata is optimized to detect the update sessions by analyzing its size. Each dot in the illustration represents one Argus observation, or in most cases one session, the report from the infected client to its update server.

By looking at the time represented by the x-axis in the illustration one can see that the updates are done with a regular interval, which is approximately every ten minutes.

Waledac

The Waledac worm uses a p2p based approach when initiating a connection to its control server, forcing the automata to look at several destination addresses. This is the reason why the destination IP in *Table 14* is 0.0.0.0. As shown by this table the automata also detects this trojan at an acceptable rate.

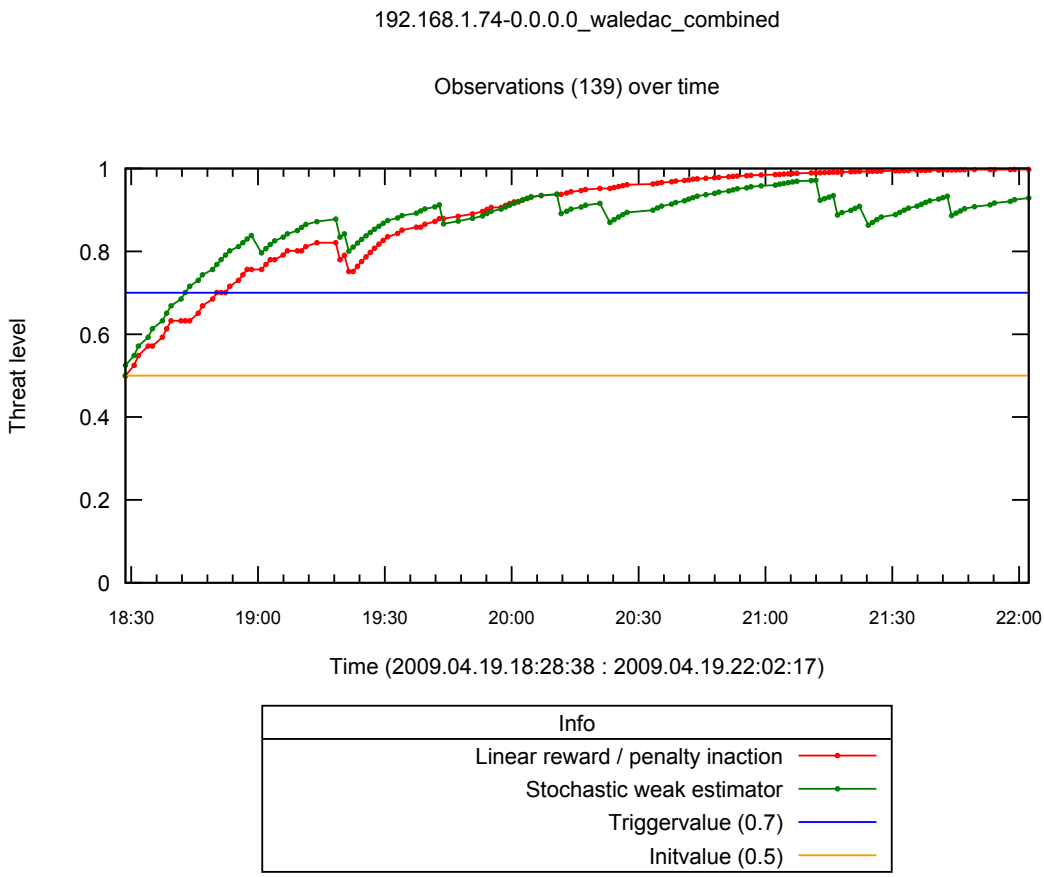

Figure 33: Waledac detection, LRPI (0.05) and SWE (20), *test set 2*

Figure 33 shows that the automaton is able to detect a Waledec infected client in less than thirty minutes. After about 45 minutes the threat level drops, as the data has correct source size but the destination size does not match.

TimeCheck

The timeCheck automata is designed to check for connections that appear at regular intervals. This allows it to detect malware such as the BlackEnergy DDoS bot and Danmec.

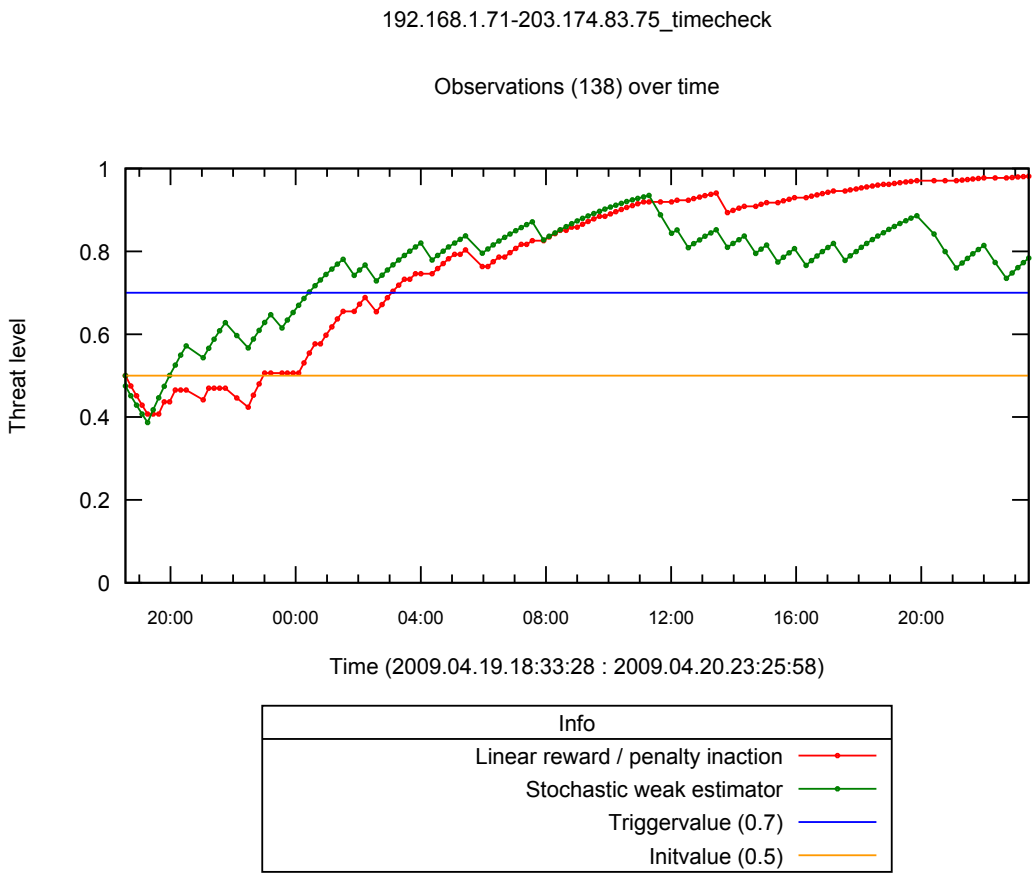

Figure 34: TimeCheck Danmec detection, LRPI (0.05) and SWE (20), *test set 2*

Figure 34 shows the same infected client as detected by the Danmec automaton in *Figure 32*. To begin with the activity seems random as the running average changes often, but as time goes by it gets more consistent and reported as a threat.

The reporting patterns for Danmec is documented in the *Danmec chapter*, where we saw an average of 12 minutes between posts. As most of the posts are every 10 minutes a window of 3 minutes in each direction is required for detection.

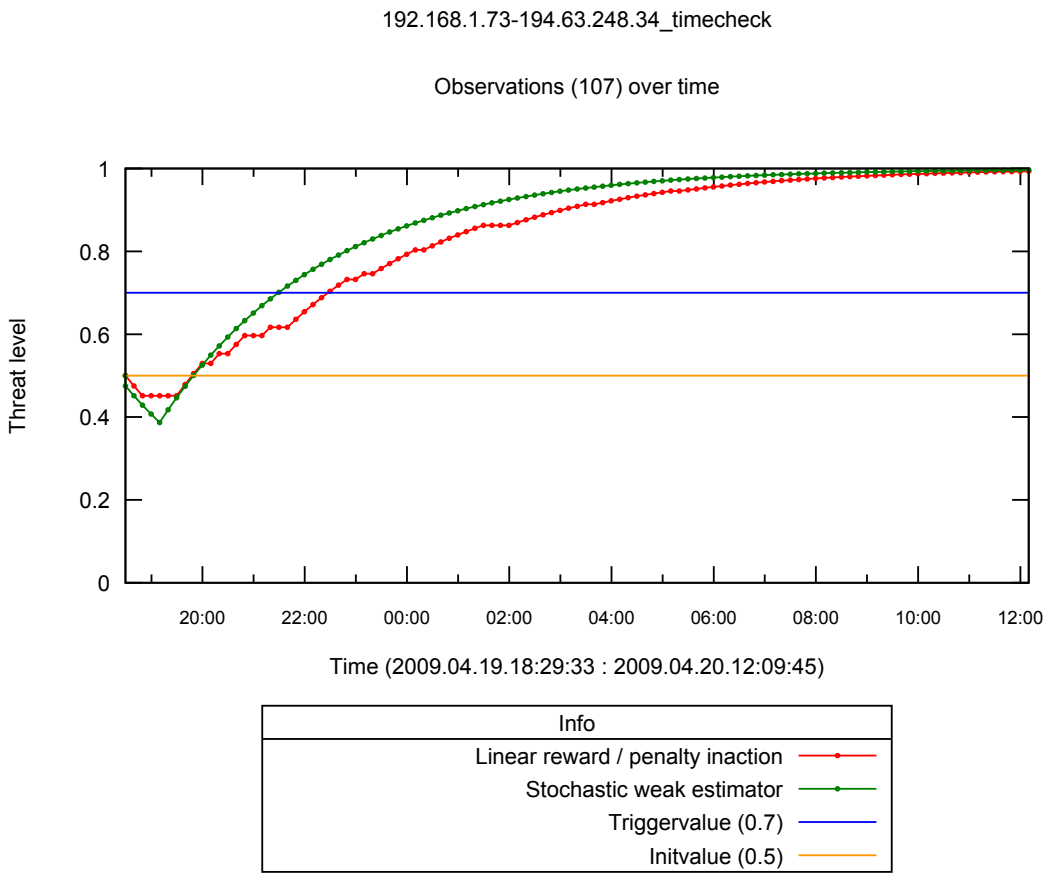

Figure 35: TimeCheck BlackEnergy detection, LRPI (0.05) and SWE (20), *test set 2*

In *Figure 35* the TimeCheck automaton is analyzing the network traffic generated by the BlackEnergy bot. This is the same traffic analyzed in *Figure 30*. The results are similar to what we saw when analyzing the traffic generated by Danmec. First the automaton needs to get some data before it can make a decision whether the traffic has regular timespans or not. BlackEnergy has an exceptionally regular update schedule and only requires a window of 5 seconds, reducing the amount of false positives.

False positives

Testing automata for false positives using *test set 2* turned out to not give any valid data, as all the detections were correct. This wasn't entirely unexpected as the set contains a small amount of clients which are infected with malware that the automata should be able to detect. To do a fair false positive test we used *test set 3*.

Snort recorded no infected clients for the selected samples during the entire time that spans *test set 3*, meaning any reported client in our automata is likely to be a false positive.

Automata type	Detected clients
BlackEnergy	17
Conficker	14
Danmec	3
Waledac	7
Total	41
TimeCheck 5 sec	797

Table 15: LRPI (0.05) false positives overview, *test set 3*

As seen in *Table 15* the TimeCheck automata is what really stands out. This is related to the size of this network and high connection count. The other automata produces an acceptable amount of false positives when taking this into account.

Automata type	Detected clients
BlackEnergy	21
Conficker	12
Danmec	4
Waledac	10
Total	47
TimeCheck 5 sec	986

Table 16: SWE (20) false positives overview, *test set 3*

Table 15 and *Table 16* show the number of detected infections for each algorithm with default configuration for each automaton type. These infections were seen some time during the four hours that the test set covers.

The difference between TimeCheck and the other automata is noteworthy. This automaton might not be suitable for a network this large, at least not without some tweaking.

We investigated some of the detections, and especially the Danmec and Waledac clients were suspicious. Unfortunately we did not have pcap data available for this period because of the sheer amount of storage that would require and were not able to rule out or prove an infected client.

6.2.3 Algorithm configuration

Both LRPI and SWE algorithms have a single value that defines how quickly a connection pair (in the case of Danmec, a source address) reaches the malicious threshold. For LRPI this is the LEARNING_RATE variable, and for SWE the SWE_RATE variable. It is important to understand the effect of changing these two values.

Each of these variables only affect the algorithm it is tied to, so it makes sense to look at them on a per-algorithm basis. *Test set 2* was used to investigate how these variables change the amount of updates that are required, as well as the time.

The detection time for each sample varies greatly, depending on how much of the traffic is ignored and how much traffic fits in the malicious and non-malicious groups.

The TimeCheck automaton was not used in the following tests as it uses the first updates to calibrate the running average. This would make it difficult to draw any conclusion from the numbers. Because it uses the same algorithms the results are still applicable to the TimeCheck automaton.

In the following tables *u* signifies number of updates. Time is measured in hours (h), minutes (m) and seconds (s).

SWE

The default SWE_RATE value set by us is 20, and is the value used in all other tests in this thesis unless otherwise stated. For the comparison the values 10, 15 and 25 will be used.

Rate	BlackEnergy	Conficker	Danmec	Waledac
25	13u, 2h	13u, 110s	13u, 64m	13u, 18m
20	10u, 1h 30m	10u, 95s	10u, 42m	10u, 14m
15	8u, 1h 10m	8u, 80s	8u, 31m	8u, 10m
10	5u, 40m	5u, 51s	5u, 10m	5u, 6m

Table 17: SWE detection comparison, *test set 2*

If we look at the number of updates we can see that all of the automata scale linearly with rate as the x axis. The fact that all samples also require the same amount of updates isn’t guaranteed, but does happen in this test case. With a larger amount of

traffic that is judged non-malicious this number would have been increased, as per the formula in *Figure 25*.

The time is not linear for all the samples, but looking at the samples we know have entirely regular updates, such as BlackEnergy, it is.

LRPI

The default LEARNING_RATE value set by us is 0.05, and is the value used in all other tests in this thesis unless otherwise stated. For the comparison the values 0.025, 0.05, 0.1 and 0.2 will be used.

Rate	BlackEnergy	Conficker	Danmec	Waledac
0.025	42u, 6h 50m	34u, 3h 4m	44u, 5h 10m	43u, 59m
0.05	14u, 2h 10m	18u, 138s	19u, 95m	15u, 21m
0.1	7u, 1h 20m	8u, 80s	7u, 31m	7u, 10m
0.2	3u, 20m	4u, 36s	5u, 10m	4u, 5m

Table 18: LRPI detection comparison, *test set 2*

The number of update requires for LRPI scale linearly to rate as well, as seen by *Table 18*. In contrast to SWE, higher learning rate equals less updates as the value is used directly and not part of a fraction. This can be seen in *Figure 23*.

The time required for lower rates are interesting, especially those seen by Conficker. The large difference between 0.05 and 0.025 is due to the long idle time between Conficker activity, which can also be seen in *Figure 31*. Requiring more than one update period for detection greatly increases the time to detection because of this large idle period.

False positives

To see the effect of the algorithm settings on false positives we ran *test set 3* with different configurations.

Automata type	0.025	0.05	0.1	0.2
BlackEnergy	9	17	32	62
Conficker	12	14	19	58
Danmec	1	3	8	20
Waledac	3	7	12	14
Total	25	41	71	154
TimeCheck 5 sec	613	797	1101	1383

Table 19: LRPI false positives comparison, *test set 3*

A higher LRPI learning rate equals more false positives. The numbers tell us that the total amount is close to linear as a function of the learning rate.

Automata type	25	20	15	10
BlackEnergy	18	21	30	46
Conficker	10	12	16	24
Danmec	3	4	5	11
Waledac	6	10	11	12
Total	37	47	62	93
TimeCheck 5 sec	897	986	1104	1244

Table 20: SWE false positives comparison, *test set 3*

Again SWE has a reverse table, with higher SWE rates equaling smaller amount of detections. The numbers are not quite as linear as LRPI, but the difference is still visible.

6.3 Resource results

When analyzing resource usage, by any type, it is important to first decide which resources are interesting to measure. In our view the only really important measurements regarding our solution are how much memory is used, and how much load is put on the CPU. The power consumption is highly dependent on the CPU architecture, and the hardware around it, so this will not be measured.

Neither Snort nor our proposed solution have huge demands on storage. Snort is able to analyze network traffic on the fly, and does therefore not need more storage than it occupies itself, and that used by the logs. Our solution only needs the space occupied by the temporary argus data feed to the automata by radump. When a file has been read, and fed to the automata, it may be deleted.

The size of this temporary file is depending on the network load, and the timespan of the file. Storage is only an issue if you want to keep the files for post-analysis. In this case a simple solution is to delete the oldest files if you run out of storage. In our test the files was split into five minutes of data each.

Here we will present the results from our memory and CPU load measurements. All resource measurements have been performed using *test set 3* unless otherwise stated.

6.3.1 Memory Usage

The release of 64-bit processors into mainstream computing makes it possible to equip a reasonably priced computer with an amount of memory that used to be available only in high-end systems. When using a 32-bit system you effectively limit the amount of byte-addressable memory to 2^32 bit or 4 GB. Most computers released after 2004 have 64-bit CPUs, and are thus able to have 2^64 or 16 TB of byte-addressable memory. Earlier models might support physical address extensions, PAE.

The amount of memory is therefore mainly limited by other factors than the CPU architecture. In our test the hardware where test were performed had a total of 2 GB of memory. It is therefore still important to keep the memory load as low as possible.

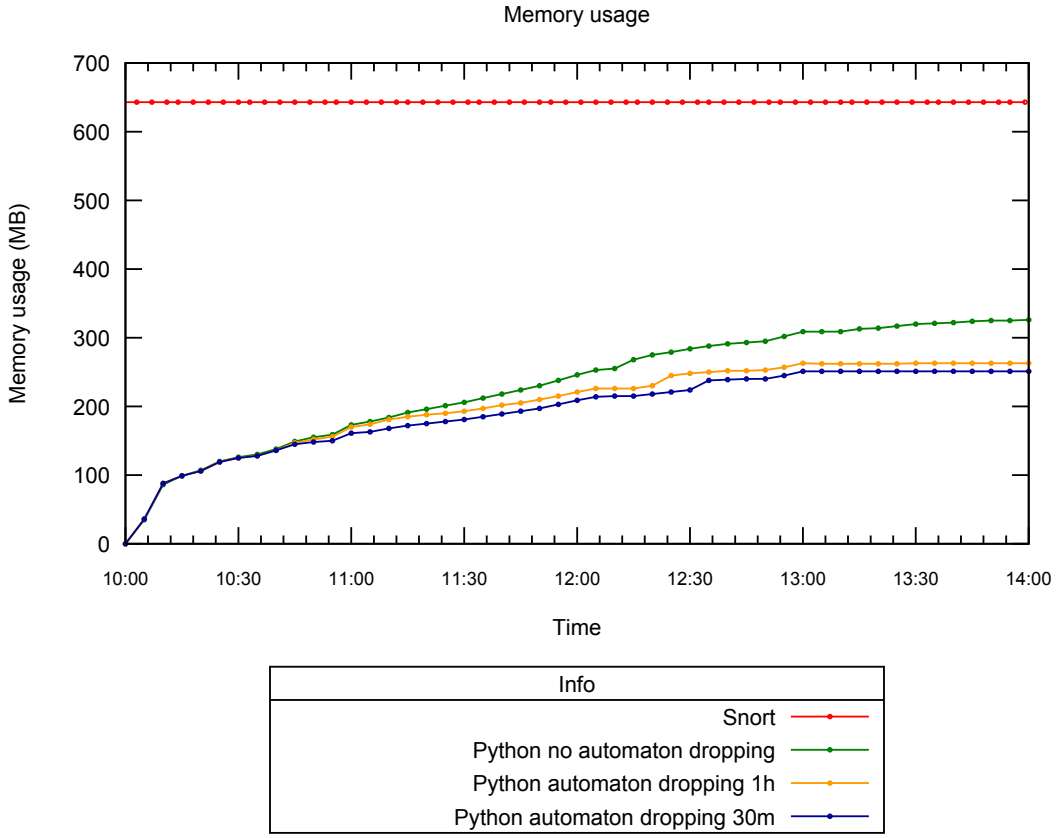

Figure 36: Comparison of memory usage, *test set 3*

Snort

As described in the *related works chapter* Snort will have a steady memory usage. This is due to the initial parsing of the rules used in detection. This was also the case when testing with our four hours of data. As seen in the graph *Figure 36* Snort keeps its memory usage steady at 643 MB.

This is due to the fact that Snort reads the entire signature set and builds everything it needs in memory upon execution, completely unrelated to how much traffic is actually being analyzed. The advantage of this is predictability; even without knowing the maximum bandwidth usage by executing Snort you know how much ram is required.

Automata

The automata memory usage is a combination of python and radump. This is because radump is used to retrieve the data used by the automata, and the automata themselves are implemented in python. In our test we found that the memory usage of radump is

insignificant compared to python. When parsing our four hour data set, radump was using between 6.2 MB and 6.3 MB of memory.

Radump memory usage will not be discussed further, as it is an external application that isn't easily improved, nor would it affect the total amount in a noticeable way.

Python memory usage depend heavily on the number of connection pairs, as this is what determines the total amount of automata. Each source / destination pair creates an instance of every automaton type. Even in an extreme network such as this, with no removal of automaton, 2GB of memory is more than enough for a single day, as seen by the graph below. With optimization techniques the memory requirements are even lower, as seen by *Figure 36*.

One disadvantage compared to Snort is the potential max amount of memory used. It's next to impossible to predict how many connection pairs that will appear without analyzing existing data. This can be avoided by doing analysis of the network before deployment.

Another possibility is to check whether you are about to run out of RAM, and apply a more aggressive automata discarding algorithm until a reasonable amount of memory is available. This would of course give a greater chance of dropping a automata that is observing a malicious source-destination pair. This is not implemented in our prototype.

Different automaton types might use different amounts of memory if they add variables in addition to what is inherited from Automaton. Memory usage value is total memory excluding application. Average is calculated from this total value.

The two algorithms do not have a different variable count and use exactly the same memory. These results are thus general, as there is no need to do twice the amount of tests with the same results.

Automaton type	Memory usage (MB)	Per automaton (KB)
None	21	-
BlackEnergy	120	1.23
Conficker	120	1.23
Danmec	117	1.20
Waledac	119	1.22
Timecheck	124	1.27

Table 21: Memory usage comparison between automata, 100 000 instances

The first four automaton types use roughly the same amount of memory, as they all have the same variables. The difference seen comes from the length of the name as the name is stored inside the object. Timecheck automaton adds a few more variables and uses a bit more memory.

As the difference between automata is negligible, the next table only contains the values for BlackEnergy.

Automata count	Memory usage (MB)	Average (KB)
5 000	7	1.43
10 000	13	1.33
50 000	63	1.29
100 000	120	1.23
100 000 optimized	39	0.4

Table 22: Memory usage comparison for different automata count

The last line of *Table 22* shows the result of a quick memory optimization done on the Automaton class. Variables used for debug and graph generation were removed, thus saving memory in all child classes. The optimized Automaton class is used in all other memory tests. This optimization partly completes the optimization requirement, more on this in the *further work chapter*.

The table also shows that memory usage is close to linear, becoming more linear as the amount increases. If the network have been analyzed and the number of connections is known this can be used to predict the total amount of memory required.

Comparison

Adding removal of old automata and automata with a low threat value flattens the graph compared to no removal. This is to be expected as many connections will not pose a threat. As seen by the graph the advantages increase over time, making it very suitable for long running analysis.

We also tested putting the top ten safe destination addresses in a whitelist and skip creation of automata for these addresses. The top destinations were, among some we can't disclose:

- vg.no
- nrk.no
- nettavisen.no
- finn.no
- nettby.no
- aftenposten.no

These were responsible for a large part of the network traffic, measured in bandwidth usage. Unfortunately whitelisting these did not affect memory usage, thus automata count, nor CPU load in any noticeable degree. Reduction of ~30 000 automata might sound significant, but compared to the total amount of ~one million the effect is negligible. Automata storage is the main memory spender, which results in an average of ~0.3KB per automata and ~10MB saved.

6.3.2 CPU load

Snort

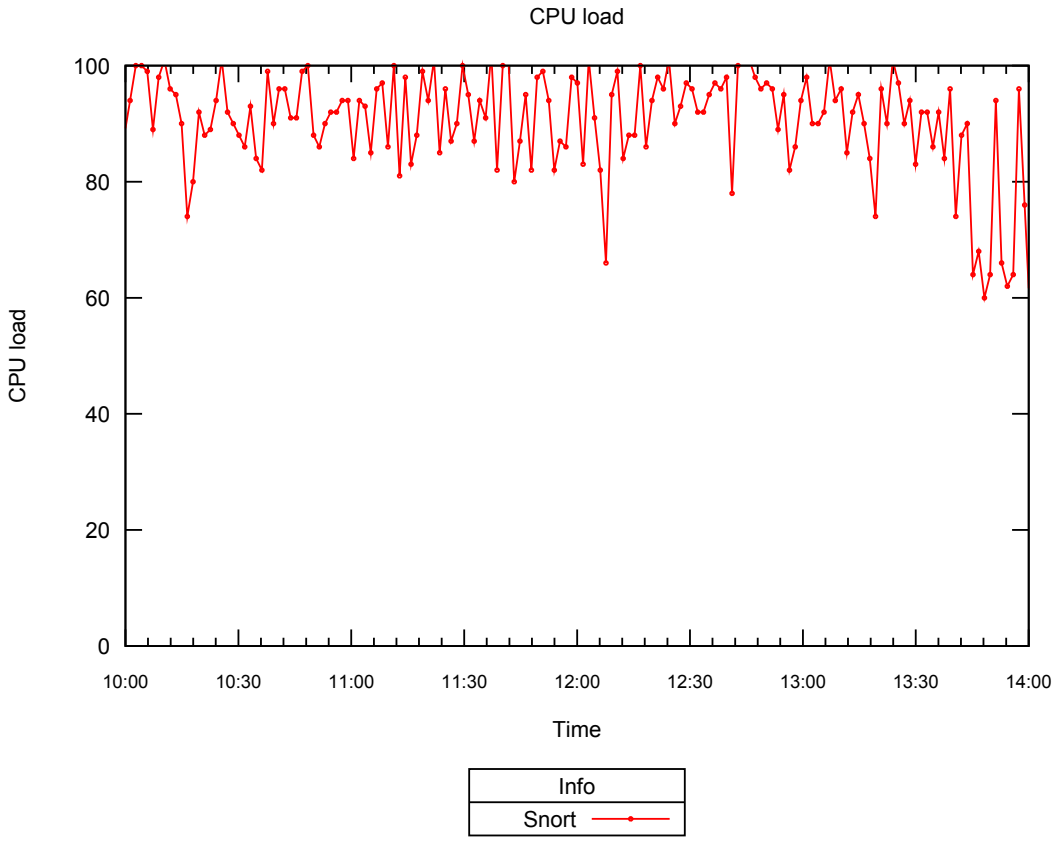

Figure 37: Snort CPU load, *test set 3*

Snort is having issues with this amount of bandwidth during work hours, as the total system CPU load was at 100% during the four hour run. The rest was taken up by misc processes. This poses a problem as it can result in dropped packets. A side effect of this is the possibility of malicious traffic going by undetected, as the data the Snort signatures triggers on isn't analyzed.

Automata

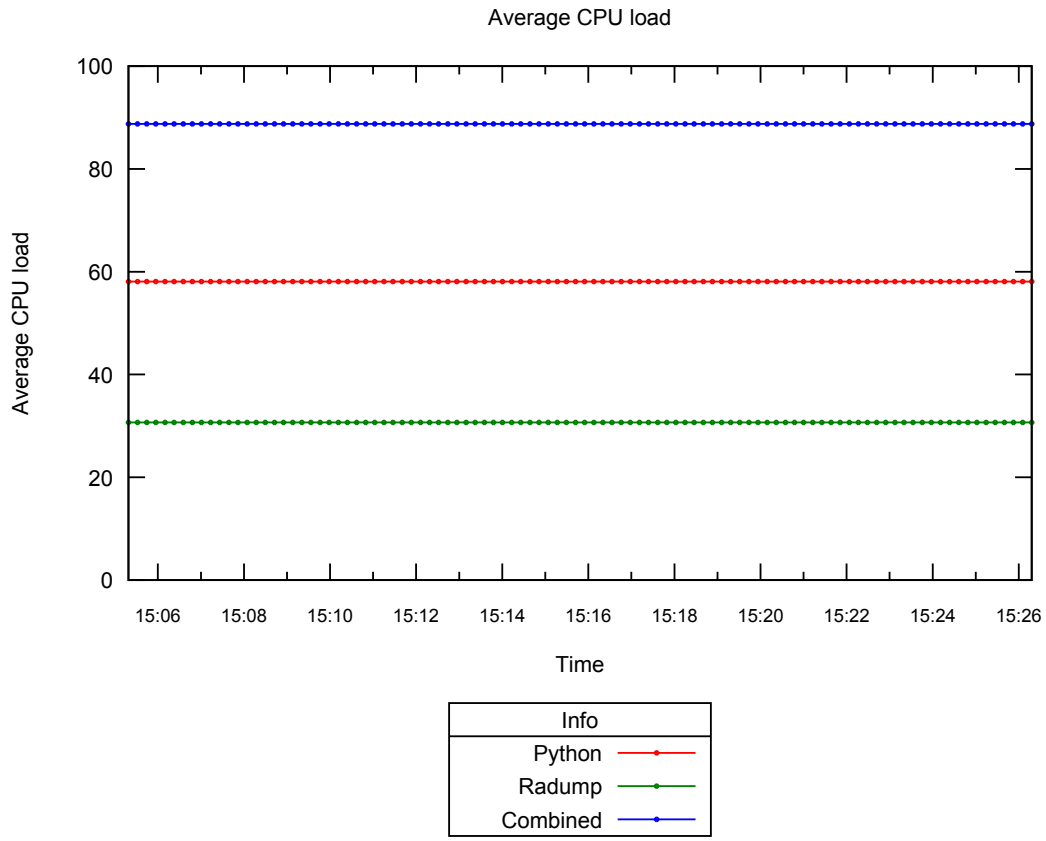

Figure 38: Python and radump avg CPU load, *test set 3*

Our automata analyses all of the data as fast as it can, as it works on pre-existing files. Analyzing four hours of data takes approximately 23 minutes, which results in a 1:10 ratio compared to real time, or a 10% CPU load average. It is important to note that the automata is limited to a small number of malware.

	SWE	LRPI
All	17.3s	17s
Danmec	15.9s	15.8s
Conficker	15.9s	15.5s
BlackEnergy	15.6s	15.4s
Timecheck	16.1s	15.8s
None	15.1s	15.1s

Table 23: Time comparison between different automata, *test set 2*

Table 23 shows the time required to run the specific automata. Running Argus and piping the output to Python takes most of the time, as can be seen by the result from running none. These tests were run on *test set 2* to allow them to complete in a reasonable amount of time.

7 Analysis of results

In this chapter we will present our analysis of the results presented in *research results*. In the first part we will focus on the results from our resource testing, and how they should be interpreted. Results for both CPU and memory usage will be discussed and compared to the results from our testing of Snort. We will also look into the storage demands for our proposed solution and Snort, as they are similar, but not identical.

The second part of this chapter will consist of an analysis of the detection results presented in *research results*. Here we will give a comparison of detection rates for our proposed solution compared to Snort. As all IDS solutions, ours also have false positives. This will be discussed in this chapter. Our proposed solution is currently only detecting the malware we chose. The detection of these will be analyzed, and should give an understanding of how the solution may easily be updated to detect additional samples or other traffic patterns.

7.1 Resources

Resources are often scarce in the computer industry. When analyzing huge amounts of data this becomes painfully clear. Both CPU and memory demands may overcome the available resources in given situations, and cause problems for systems doing analysis. We will here analyze the results presented in *resource results* for both CPU, memory and finally a bit about storage demands for either solution.

7.1.1 CPU

Snort is struggling with the amount of traffic generated by the *client network*, using close to 100% CPU during work hours. *Table 22* show Snort hovering between 80% and 100%, but when running these tests we saw the rest taken up by misc processes.

The total usage only dipping below 100% when work hours were over and people started going home. This can affect detection, as snort will drop packets if it isn't able to process them. The average packet drop rate for the four hour test period was 10%.

The automata always uses 100% CPU as it processes netflow data in chunks split by time, as configured in Argus. In our case this is set to 5 minutes. Most of the time was spent by Argus parsing the netflow files and importing the data into the Python application. This is evident from *Table 23*, as the difference between running none and all the automata is small.

Adding more automata is cheap as it does not increase CPU usage linearly like the memory usage. While waiting for new files the CPU usage is 0%, which is why the calculated usage from *Figure 7* equals 10%.

Looking at both it is clear that Snort requires more processing, as it is processing a much larger amount of data. 5 minutes of Argus data took 70MB, whereas 5 minutes of Snort data would take 3300MB.

The size of data aren't only related to storage, as they also affect how much data must be moved around in memory and how much must be analyzed. It's important to note that the Snort signature set we used contains 12245 rules and is able to detect a large amount of malicious traffic compared to our implementation that only looks for a few specific threats.

7.1.2 Memory

The memory tests showed what we had expected, at least for the automata. The total memory usage of Python is highly dependent on the number of automata as is evident from *Figure 36*. What isn't quite as clear is how the relationship is. The average value of *Table 22* explains this better; increasing amounts of automata flattens the average, coming closer and closer to a linear growth in memory usage.

At first glance this pattern might seem to make memory usage easy to predict, but in fact it is not. If the number of automata is known calculating memory usage is simple, but knowing the number of automata before execution requires the knowledge of how many connection pairs the network has, which on a client network is literally impossible as this value changes depending on user habits. Outside factors, such as banning a website from the corporate network or the arrival of important news, might change these habits abruptly.

The *client network* we tested on is almost a worst case scenario; a large network with a lot of workstations. It isn't hard to imagine a much larger network though, such as the hub of an ISP. At the current memory usage rate such a network might prove a challenge.

One important thing to point out is that the number of automata thus memory usage flattens out after time, as all the connection pairs have already appeared. This can be seen by *Table 22*. Memory optimizations flattened the graph further, reducing the memory requirements from 315MB to 255MB.

Snort on the other hand turned out to be very predictable, again visible on *Figure 36*. Snort builds the internal data it needs from the signature set at startup. Memory usage does not change noticeably during analysis, showing a constant use of 643 MB. This is a big advantage as it allows the sensor builder to know how much memory is required without knowing anything about the network except for the desired signature set.

Finding out how much memory Snort uses as a function of the signature count is not possible, as the signatures can have very different sizes depending on what they do. Because of this we do not have any data on this particular subject.

7.1.3 Storage

Looking at storage requirements there are a few things to consider. At bare minimum, either solution requires very little. Snort must have space for the signatures and the automata must have space for one single Argus file. Signature sets vary in size, but the set we used was 6.8 MB. A single five minute Argus file from *test set 3* was, as mentioned earlier, 70MByte.

Bare minimum is not practical, as it doesn't allow you to go back and see what happened. With Snort you can store events that triggered a certain signature, or you can decide to store all the network traffic to allow full analysis later on.

In the automata it is practical to keep flow data for as long as possible, in case something was missed or the automata was updated. It would also be possible to implement a new automaton and re-analyze old data again, which is also possible with Snort by playing back pcap data.

The total size of all the flow data during the four hour test run was 3.5GB. Pcap data was not captured, but an average of 100Mbit would total 158GB. Storage is thus highly dependent on the amount of traffic that passes through the sensor. Flow data does in all cases take significant less space, as pcap data contains exactly the same data as flow data does, plus the actual content. The advantage of storing flow data only is the possibility to store more historic data than Snort, at the expense of containing less information.

The smaller size of the Argus data files would also make it possible to analyze them somewhere else by capturing the data on modest hardware and transferring them off site. If there are several network points of interest the data from all of them could be collected and stored at a central location. The flow data could then be analyzed by a single computer since analysis is performed much faster than real time as seen by the *automata CPU load test*.

7.2 Detection

Detection is an essential part of any intrusion detection system, whether it is based on deep packet inspection or flow data. Detection and false positives rate differ greatly between solutions. Time until detection and adding support for new threats are also important.

Here we will present the results from our tests regarding detection. The results for both Snort and our proposed solution will be compared. First we give a description of each of the samples. The next part will consist of data regarding false positives, before we move on to detection time, signatures and rules, and behavior based detection.

7.2.1 Samples

As described in the *malware selection chapter*, we selected our malware from samples being observed at Telenor SOC as active. Initially we wanted to use a malware sample that tricked users of MSN to download and install itself, and thereafter control the infected host by an IRC control server.

Unfortunately there were no active botnets of this kind that we could use. Therefore we chose to replace the MSN trojan with the BlackEnergy bot. This way all our samples were active during the test period, and could possibly appear in the traffic from *test set 3*.

BlackEnergy

The BlackEnergy bot was fairly easy to detect. The analysis required for this sample was limited, as we knew some key characteristics beforehand. This was because we had control over the control server for our bot, and defined much of it's characteristics ourselves.

The bot was detected by two of our automata. The specific automaton designed to detect BlackEnergy had a steady curve towards the trigger value, and thus marking the source and destination pair as malicious. The detection graph *Figure 30* clearly shows how the BlackEnergy automaton increases its threat level.

The BlackEnergy bot was also detected by our TimeCheck automaton. The regular contact with the control server, as described in *automata detection*, was ideal for this automaton, as it is designed to find this type of traffic patterns. The TimeCheck

automaton had similar detection results as the BlackEnergy automaton. This can be seen in *Figure 35*

Snort detected the BlackEnergy bot by two signatures, as described in *detection results*. One signature was significantly more robust as it looked for content of the packet used for communication. The other simply reported a specific GET string. The BlackEnergy bot is customizable, so the GET string may be changed. The packet containing the communication is however the same.

It is clear that the detection of BlackEnergy bots are made possible using only flow data. Our proposed solution was able to detect the infected client using two different approaches.

As seen in *detection results, false positives*, the BlackEnergy automaton triggered 21 false positives when tested on *test set 3*. This gives an average of approximately five false positives every hour, or one every eleventh minute. This is few enough to be able to handle manually.

Detection time is also important. In a high bandwidth network where you need an alternative to Snort for BlackEnergy detection, the proposed solution may be an option, as it detected the bot in a fair amount of time.

Conficker

Since there are several different versions of Conficker we had to select one of them. We looked into Conficker.A, Conficker.B and Conficker.C, but as Conficker.C did not change its behavior until April, and Conficker.A is easy to detect using its trafficconverter.biz communication, as described in *the malware section*, our solution is Conficker.B aware only.

As seen in *Table 10* we had to use four pairs of source bytes to detect Conficker. The reason for this is that Conficker has some variety in its behavior, and thus traffic pattern.

As described in *automata detection, conficker*, the communication with the control servers are done at random intervals. It is therefore pointless to do time related detection for Conficker.

The automaton designed to detect Conficker only needed to observe one of these random sessions to consider it as malicious. This can be seen in *Figure 31*. The

automaton reach the trigger value of 0.7 during the first communication with the control server. The graph clearly shows how Conficker then idles for three hours before any further communication is observed.

Both sessions gave a total of 44 observations, as described in *Table 14*. As both update sessions have approximately the same number of observations we conclude that our automaton only needs to observe one session of communication with the control server to detect Conficker.

With more pairs of source bytes marked as malicious, the number of false positives will also increase because more non-malicious traffic will coincidentally fit into the new size ranges. This could be a big issue, as many false positives could result in actual Conficker infections being ignored by the operator who receives these alerts.

In the *test set 3* test we observed a total of 12 Conficker detections, as described in *Table 16*. As the test set had four hours of data this means an average of one false Conficker detection every twenty minutes.

Considering that the network in *test set 3* had an average load of approximately 100Mb/s, as described in *traffic type*, the false positive rate is acceptable. The false positives mainly triggered when users were searching for short strings, using Google, as this gives almost the same amount of data as communication with a control server.

Snort detected two characteristics of Conficker when using *test set 2*. The first signature detected communication with whatismyip.com, as Conficker contacts this site to get its external IP, as described in *the malware section*. This way Conficker can inform the control server where it is located. An external IP resolving to a government facility, or big corporation could be more interesting for an attacker than the computer of a home user. Communication with whatismyip.com is however not malicious traffic in itself, as the site is legit, and people may contact it without being infected with Conficker. As a result of this the signature states that the client is possibly infected, and does only recommend you to investigate.

The other Snort signature triggers on the GET request sent to Conficker control servers. The GET request has a "search?q=0" string, as described in *the malware section*. It is this string that triggers the signature.

The "q=0" is a number representing how many clients have been infected by the client reporting to the control server. The signature only triggers if the GET request contains one to three digits after the "q=" part of the string.

The signature still produces some false positives. An example of where such a false positive can trigger is when doing searches for numbers on the online music and DVD store cdon.com. A search for the popular TV series "24" will send the following GET request:

```
GET /search?q=24 HTTP/1.1
Host: cdon.no
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.0.10)
>Gecko/2009042523 Ubuntu/9.04 (jaunty) Firefox/3.0.10
[...]
```

Figure 39: cdon.no search string - false positive Conficker

This is the same GET request as a Conficker infected client that have infected 24 different hosts will send. It is indistinguishable from the original malicious request and will trigger a false positive.

Danmec

When a computer becomes infected with Danmec it first contacts yahoo.com and www.web.de to confirm that it has a working internet connection. This is described in the *lab experiments*.

After this Danmec reports to its control server with fairly equal time intervals. In our research we found that the standard interval was approximately ten minutes. However, some of the check ins were skipped. This can be seen in the *Figure 32*, where there are no communication with the control server from 20:30 until 21:00. In other parts of the graph you may also see that there are missing communication, giving a timespan of 20 minutes between each check in.

Because of these random missing intervals use of the TimeCheck automaton is not ideal. The TimeCheck automaton required a ± 3 minutes window to detect it properly. This is described in *Figure 34*.

A window of this size, however, gives many false positives as random traffic will match this behavior. *Test set 3*, as described in *Table 16*, shows us that even with a small window a large network will produce too many events to handle. It is therefore preferable to use the specific Danmec automaton over TimeCheck when detecting Danmec traffic.

Most of the communication with the control server falls within the restrictions given in *Table 11*, which results in the automaton regarding it as malicious and increasing its threat level.

When using linear reward penalty inaction our automaton used approximately one hour and thirty minutes to decide that it found a Danmec infected client. With stochastic weak estimator it took 45 minutes. Keep in mind that LRPI has a guess function when it rewards, so some randomness must be expected. If several check ins were skipped initially the time to decide would of course also increase for both algorithms.

When increasing the number of rules used to detect malware, you also increase the chance of false positives. The Danmec automaton only uses two rules, as seen in *Table 11*, to detect the Danmec communication with the control server.

As expected this was noticed when testing with *test set 3*. As seen in *Table 16* the Danmec automaton only detected four clients as infected. Unfortunately the pcap data for *test set 3* were not available during post analysis, so we were unable to determine whether all of these were false positives.

Some of the clients marked as malicious had traffic that seemed more than suspicious. Snort did not trigger on any of the clients, so we can not conclude that the automata actually found infected clients. A false positive rate of one per hour is however a result we are pleased with.

Snort detected the Danmec trojan with two signatures. Both signatures triggered on the initial POST done to the control server. As long as the check in is not altered significantly these signatures should continue to detect Danmec.

These signatures are not known to give a lot of false positives like the Conficker signatures are. As seen in *Figure 9* the post done by Danmec does not include a lot of information. A POST containing only "/forum.php" could in many cases be believed to be legit traffic, and thus avoiding being checked out by an analyst. Any legal traffic containing the same POST would of course trigger the signature as a false positive.

Looking at the limited amount of false positives produces by our automaton, and the relatively short time required to detect infected clients, it would be possible to use our proposed solution as an alternative to Snort for Danmec detection.

Our solution does not detect an infected client during the first POST, but have significantly less demands when it comes to resources.

Waledac

Waledac was not detected to begin with because of the problems regarding communication with its control centers, described in the *Waledac implementation chapter*. Once we had countered those problems by only looking at the source address instead of the connection pair the threat values increased above the threshold, as seen by *Figure 33*. An IP-address of 0.0.0.0 signifies any, as none of the Waledac automata have specific destination addresses.

Detection time varies from half an hour to an hour, depending on algorithm. With the current algorithm settings this was achieved in 10 to 15 posts, which is high enough to prevent a lot of false positives when the automaton is first created.

The Waledac trojan deliberately changes its POST string to avoid detection. This forces snort to use content recognition for detection. In addition to this the trojan uses the same user-agent as the Mozilla Firefox browser uses, as seen in *Figure 11*. This is most likely a disguise used to blend into other normal webtraffic.

The snort signature is configured to look for this user-agent together with the special HTTP parameter "a=" sent by the trojan. The interesting thing here is that by trying to avoid detection, the trojan also managed to limit the potential false positives produced by snort. Only web traffic generated by Mozilla Firefox browsers, or other programs imitating their user-agent, will have the potential to generate false positives.

7.2.2 False positives

When it comes to false positives, Snort has a big disadvantage. First of all, the amount of signatures used in our test was, as described in *analysis of results*, 12245. This gives a lot of possible false positives. The fact that many of the signatures are so called "fail-safe-signatures" does not help.

A lot of tuning is needed before an IDS solution using Snort have an acceptable amount of false positives in any given network. The fail-safe-signatures are often more informative, and does not give information about real malicious traffic.

An example could be a 530 response from a FTP server. This response is given when login is denied because the user or password is incorrect. If someone tries to login as

anonymous first, and then use their provided username and password afterwards, this signature will still give alerts. Only when triggering massive amounts of alarms should this signature be taken serious. There exist several signature in Snort that have similar behavior, and thus triggering when there is no need to.

Our proposed solution did not cause a lot of false positives, if you disregard the TimeCheck automaton. The number of rules we tested with however was insignificant compared to Snort. With only four specific rules, we consider the number of false positives we got as satisfactory. A big increase in false positives could render the proposed solution unpractical.

If we were to increase the number of automata or rules we would most likely get a higher false positive rate. Currently, using *test set 3* with a LRPI rate of 0.05 we got an average of approximately 11 false positive an hour, or one every five minutes, as seen in *Table 15*.

An increase in the number of rules with a factor of 1000 would possibly give us a similar increase in false positives. The results would then be 4000 rules and 10250 false positives every hour, or 171 every minute.

A false positive rate of 171 every minute might sound unmanageable, but such an extensive rule set would have a lot of rules not trigging in all types of networks. The number of false positives, as we see it, would therefore not necessarily be increasing in the same pace as the number of rules. However, as we have not tested such a big rule set, we can not give any conclusion for how the rate of false positives will develop as the number of rules increase.

A rule set of this size would in any case require some tuning before deployed in a network. This is similar to what needs to be done for Snort signatures. By whitelisting certain IP addresses, or IP ranges, a lot of false positives could be avoided. This type of tuning needs to be done after the network have been analyzed for a while, so that the most common false positives can be ignored.

7.2.3 Detection time

Sometimes it is important to detect an infected client as fast as possible. In these scenarios Snort has an clear advantage. Snort may mark a single network packet as malicious right away, while our proposed solution needs several observations before it can decide whether the traffic is illegal or not.

As seen in *automata detection*, the time needed to make a decision about the traffic varies a lot. Conficker only needed three minutes from the first observation, while BlackEnergy managed to remain undetected for as long as 130 minutes.

Even though the time span until detection is very different, the number of observations needed to detect the malware was more similar. Conficker triggered after 18 updates, while BlackEnergy only needed 14. Danmec and Waledac needed 19 and 15 updates before triggering. From this we can see that malware with a lot of activity will be detected faster than those that rarely do anything.

The factor that determines how much time is needed before malware is detected is the rate of communication, and amount of data that the malware sends to its control servers. If the only thing important is immediate detection of malware, your best choice is Snort. If a certain delay is acceptable either solution may prove effective.

Collecting data over time using our solution could also detect some patterns that are difficult to detect with Snort. This is most apparent with the TimeCheck automaton. Detection after several hours is still better than no detection at all.

7.2.4 Algorithm comparison

Both the algorithms perform similarly, as can be seen by the graphs in the *implementation chapter*. The main difference is how adjustments is made to change how quickly the value changes.

The result of changes can be seen very clearly in *Figure 33*; in this case SWE not only increases faster but also changes direction faster.

How much of the previous value that is used can be changed by modifying the fraction in the algorithm, as displayed by *Figure 25*. LRPI on the other hand uses a learning rate variable that defines the rate of change. This can be seen in *Figure 23*.

In the end the choice of algorithm does not matter much; the different values that can be set inside the values for learning rate or previous value fraction is much more important. By experimenting both algorithms can create almost identical graphs.

Since SWE has the disadvantage of using fractions and float casts in the Python code we would recommend LRPI. Both algorithms are available in the source and can be selected in automaton.py.

7.2.5 Algorithm configuration

Both the algorithms seems to scale linearly depending on rate, as mentioned in the *algorithm configuration tests* chapter. This is not entirely true, which is easiest to se in *Figure 30*. Lower LRPI learning rate or higher SWE rate will decrease the slope but not the form. The reason it seems linear is because the graph behaves linear when it is close to 0.5. The graph flattens as the threat value increases or decreases.

With a slow rate the difference between each point will be smaller, and when the value gets close to the threshold more updates are required to tip above. This is only noticeable when the number of updates is really high, as evident by *Table 18*.

Automaton configurations that require this many updates is not practical for actual use as several of the automaton types take hours to reach a decision. Ruling out this slowest rate we can still consider the number of updates required to be linear, making it easier to predict.

It is impossible to predict the time required for detection, unless the malware is entirely regular. It is therefore important to require as few updates as possible in case the malware goes idle after a period of activity.

Table 19 and *Table 20* both display the same result; faster learning rate equals more false positives. For LRPI this means a higher learning rate value, for SWE a lower swe rate value. The highest and lowest value respectively results in a total amount of threats that might be difficult to handle if the person set to handle them is not dedicated entirely to that task. It is also likely that a fast learning rate will make it harder to detect actual infected clients as they will drown in the noise.

All in all there is a race between time to detection and the number of false positives. We do not recommend the fastest or slowest values tested, as neither give satisfactory results. The defaults of 0.05 and 20 and values not too far from them should give a manageable amount of threats.

7.2.6 Signatures and rules

The process of designing Snort signatures is not trivial. There have been several attempts at making solution to automate this process. In the paper NetSpy: Automatic Generation of Spyware Signatures for NIDS [34], the authors propose such a solution. They managed to create signatures for six out of nine samples.

As you can see in the paper Automatic generation and analysis of NIDS attacks [35], the creators of malware have several techniques that they may use to avoid being detected by IDS systems like Snort.

Our proposed solution have an advantage here as it does not inspect separate network packets, but rather a session as a whole. In this way evasion techniques based on splitting information between packets, or adding garbage in between should not have a severe effect.

This also makes the creation of new rules simpler. A Snort signature must match the payload of the packet, while we only need information from flow data. A system for automatic generation of automaton rules should therefore be easier compared to automatic generation of Snort signatures.

7.2.7 Behavior based detection

In many cases it would be preferable to detect malware, not only by a specific signature, but also by its behavior. Snort has the signature approach, while our proposed solution is able to detect behavior patterns over time. The *TimeCheck automaton* in our proposed solution is an excellent example of this functionality. By looking for regular intervals, it was able to detect the BlackEnergy bot easily.

If a new unknown bot was to use regular intervals between its updates, the TimeCheck automaton would most likely mark it as malicious. However, this automaton will produce some false positives if a legal service have regular updates as well. In this case whitelisting is an excellent way of avoiding marking legal services as malicious.

As the rules for our automata are implemented using python, there are a lot of possibilities to make very specific rules, regarding time, which way traffic flows, how much data is transferred, how timely the updates are and so on. The real job is in the malware analysis, as you need some basic knowledge about the traffic patterns you should detect.

8 Summary and conclusions

In this chapter we will present our conclusions based on the results, and our analysis of these results. We will discuss each part of the research, and give reasons for our findings. Discussions, and conclusions about areas of deployment, the algorithms used in our research, resource limitations and the success rate of our solution will be given. As a final step we give our thoughts about further work, pointing to key elements a more robust solution should look into.

8.1 Areas of deployment

We tested our solution in a larger network that was, as *Figure 28* shows us, able to produce over 100Mbit/s of traffic during work hours. The load on the sensor during these hours affected snorts ability to analyze packets, and we believe that it is possible to improve this ability by partly or completely exchanging it with a flow based solution.

Another possibility is to use a flow based solution where DPI is not allowed, as in larger networks owned by ISPs. Privacy issues forces most ISPs to not inspect the network traffic by looking at content. It is often, however, desirable to have some sort of control of the infected clients in such a large network. A flow based detection algorithm could in this case prove useful.

8.2 Algorithms

In our tests we used two different approaches for increasing and decreasing the threat value of the traffic observed by an automaton. Linear reward / penalty inaction (LRPI) was, as described in Morten Kråkviks paper *DDoS detection based on traffic profiles*, already known to work with DDoS traffic.

It was interesting to investigate if it could be applied successfully to malware traffic when only the flow data was available. In our research we found that it performed well, and was able to use the provided data to determine whether it observed malicious traffic or not without too many false positives, as described in *Table 15*.

Stochastic weak estimators (SWE) was also implemented to get a reference point to LRPI. The algorithm, and its applications are described by B. Oommen and L. Rueda. [32] This algorithm yielded similar results as LRPI, as seen in *automata detection*, where all graphs have both the detection rate for LRPI and SWE.

As seen in *Table 23* the difference between the two algorithms when it comes to performance is negligible. The biggest difference between the algorithms when run on *test set 2* was 0.4 seconds when running the Conficker automaton.

In conclusion, one algorithm does not distinguish itself as preferable over the other. The differences in detection is not big enough to give a clear winner whether LRPI or SWE should be preferred as the algorithm over the other. The time needed in processing however puts LRPI slightly ahead. The final verdict is that any of the two may be chosen to detect malware activity based on flow data.

8.3 Resource limitations

The automata solution is not limited by CPU, as shown by the data in the *automata cpu load chapter*. The time spent in the automata is short compared to the time Argus spends parsing the data files and the time Python takes to read the Argus output into the application. Adding another automaton type does not increase the time spent on analysis at the same rate as the increase in total automata.

All the data we have collected points to the same conclusion; the main resource limitation for the automata is memory. It is also the main disadvantage compared to Snort as Snort is predictable and does not increase memory usage during execution.

The almost linear growth in memory use makes it possible to predict the memory requirements if the memory use for a certain amount of automata types is already known. It will however not give any indication to what the initial memory demands are. By listening to the sniffing point beforehand, and counting the number of source - destination pairs, it would be possible to give a fair estimation of the initial memory demands as well.

As our approach reads flow data, based on connections, an increase in bandwidth does not necessarily result in a significant increase in resource demands. In a network where you have some clients or servers with a high bandwidth usage towards few clients, our solution will perform very well, as connections are handled on a per session basis.

Snort on the other hand will have to do a deep packet inspection of every packet it observes. In a network with a lot of connections, like *test set 3*, our solution will require more memory, as the amount of automata increase, as well as more CPU time to process the data. Snort will as always, investigate each packet. From this it is clear that a flow data based solution can yield great performance savings, especially in a high bandwidth environment where you have few connections.

8.4 Success rate

Looking at the *detection results* and the *false positive results* we are quite satisfied with the performance of our solution. The malware samples were detected in a reasonable amount of time, and the amount of false positives is not big enough to make it impossible to see which ones are actually infected.

The actual percentage that makes up false positives is not important, as long as the total amount of threats are small enough for an analyst to look through and clear out. The total of 41 events in four hours is easily managed by a network admin.

The detection tests were done using actual infected clients on real networks, and the performance tests on a large corporate network. This ensures that the tests are both valid and relevant today. We were in no way allowed to infect clients on the corporate network, so a detection test here was not possible unless a pre-infected client existed. This was however not the case.

The results clearly shows that we have found a viable way to detect infected clients using flow data and learning automata. The solution allows for easy deployment in any environment that can capture flow data, and allows the targeting of specific outbreaks, such as those recently seen by Conficker.

The challenge regarding success rate lies in writing proper rules. It is not difficult to detect an infected client, but avoiding a vast amount of false positives is more of a challenge. This was made very clear by the TimeCheck automaton in *Table 16*, where a comparison window of ± 3 minutes gave a whopping 2729 false positives. By reducing the comparison window to ± 5 seconds, the false positive count declined to 986, which still is a very high number. This is of course also true for Snort, as general signatures trigger more false positives than strict ones.

8.5 Further work

Our solution is a proof of concept prototype, and as such there are many changes and features that did not make it into the first implementation. The most pressing change would be an optimization in memory use, as this has already been identified as the most limiting resource. There is still some shared information between objects, so a redesign in those structures would further reduce memory usage.

Automata memory comparison documents that whitelisting certain destinations do not affect memory usage in a noticeable degree, but whitelisting connection pairs from automata that go below a certain threshold might give better results.

Currently we only have automata for a few types of malware. This could be expanded to cover many of the currently active malware and thus increasing the usability. Instead of creating these Python rules manually it might be possible to generate them by automatically analyzing flow data from an infected client.

Different networks might require different settings for algorithms, automata discarding and notification method. These could be drawn out in a separate configuration module to allow easier configuration as well as making it easier to update the automata without overwriting settings. Defaults for typical networks could be provided.

The implementation runs on the Telenor platform, but does currently not insert events directly into the existing interface. We wanted to do this, as seen by the requirement, but it turned out to require work on systems we had no access to. Completing this would make integration even easier.

We only use some of the fields from the flow data that Argus stores. Additional fields such as protocol and flags might also be used to improve accuracy. Argus also has the possibility to store a small part of the content, such as 32 bytes in each direction. Looking at this data would allow for more accurate detection where parts of the content is known. This would make for a hybrid approach.

All of the proposed improvements will bring the implementation closer to a viable solution for deployment in intrusion detection systems. It will never be a replacement for Snort, but a fast, effective and easy to use tool that can offload Snort or be used in places where using Snort is not possible due to traffic amount or privacy laws.

9 Glossary

ACK	A packet message used in the Transmission Control Protocol to acknowledge receipt of a packet. See SYN.
ARP	Address Resolution Protocol. A method of finding a host's hardware address when only the IP address is known.
Botnet	A large number of infected computers that are controlled by criminals. Often used to perform DDoS attacks or spread spam.
C&C	Command and Control. A C&C server is the point where infected clients get updates and tasks.
CPU	Central Processing Unit. The part of a computer that processes information.
DDoS	Distributed Denial of Service. A type of attack where several clients block a service by producing more traffic than the attacked host can handle.
DHCP	Dynamic Host Configuration Protocol. Used to give IP addresses to a group of computers in a network.
DPI	Deep Packet Inspection. The analysis of the content of a network packet.
Flow data	Information regarding size, sender, receiver, port etc. of a network packet.
FTP	File Transfer Protocol. A protocol for transferring files across a computer network.
Hashmap	A data structure that associates keys with values.
HTTP	Hyper Text Transfer Protocol. An application-level protocol for distributed, collaborative, hypermedia information systems.
IDS	Intrusion Detection System. A system capable of analyzing network traffic and detect behavior based on rules or signatures.
IP	Internet protocol. Also used as a short for IP-address.
IP-address	A group of numbers representing the address of a host connected to the internet.
IRC	Internet relay chat, a server based form of communication.
ISP	Internet service provider, a company that is responsible for connecting consumers and businesses to the Internet.
LA	Short form of learning automata.

Learning Automata	A type of algorithms used in computers to give simple learning capabilities to a program.
Learning Automaton	Singular of Learning Automata.
LRPI	Linear reward penalty inaction. A type of learning automata.
PAE	Physical address extensions, used on 32-bit computers to address more than 4GB of RAM.
POST	A HTTP request message resulting in an action on the server.
RAM	Random access memory, the primary type of memory used in computers.
Snort	A signature driven system capable of doing real time packet logging and traffic analysis on IP networks.
SQL	Structured Query Language. A language used when working with data in relational database management systems.
SSH	Secure shell, used to connect to remote hosts over an encrypted channel.
SWE	Stochastic weak estimator. A machine learning technique comparable to learning automata.
SYN	A packet sent to start a connection.
SYN/ACK	Response to SYN acknowledging that the SYN was received.
TSOC	Telenor Security Operation Centre.
VPN	Virtual Private Network, creates a network of hosts independent of their physical location.

10 References

[1] http://us.trendmicro.com/imperia/md/content/us/pdf/threats/securitylibrary/trend_micro_2009_annual_threat_roundup.pdf
[2] http://www.wired.com/techbiz/people/magazine/17-01/ff_max_butler?currentPage=5
[3] http://www.marisolutions.com/pdfs/articles/SME_ChoiceEprint.pdf
[4] http://www.cs.ucsb.edu/~seclab/projects/torpig/torpig.pdf
[5] http://www.f-secure.com/weblog/archives/VB2007_TheTrojanMoneySpinner.pdf
[6] http://qosient.com/argus/faq.htm#1.1
[7] http://www.snort.org/
[8] http://archive.ics.uci.edu/ml/databases/kddcup98/kddcup98.html
[9] http://www.technewsworld.com/story/56378.html
[10] http://news.cnet.com/8301-1009_3-10148359-83.html
[11] http://www.symantec.com/business/security_response/attacksignatures/index.jsp
[12] http://www.virustotal.com/
[13] http://asert.arbornetworks.com/2007/10/blackenergy-ddos-bot-analysis-available/
[14] http://atlas-public.ec2.arbor.net/docs/BlackEnergy+DDoS+Bot+Analysis.pdf
[15] http://vil.mcafeesecurity.com/vil/content/v_153464.htm
[16] http://www.f-secure.com/weblog/archives/00001584.html
[17] http://www.ca.com/securityadvisor/virusinfo/virus.aspx?id=76852
[18] http://www.cert.at/static/conficker/TR_Conficker_Detection.pdf
[19] http://mtc.sri.com/Conficker/addendumC/
[20] http://blog.trendmicro.com/downadconficker-watch-new-variant-in-the-mix/
[21] http://asert.arbornetworks.com/2009/02/the-conficker-cabal-announced/
[22] http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx
[23] http://www.f-secure.com/v-descs/worm_w32_downadup_al.shtml#details
[24] http://www.ca.com/securityadvisor/virusinfo/virus.aspx?id=77976
[25] http://www.toorcon.org/tcx/18_Brown.pdf
[26] http://www.secureworks.com/research/threats/danmecasprox/
[27] http://www.shadowserver.org/wiki/pmwiki.php?n=Calendar.20081231
[28] http://www.f-secure.com/v-descs/email-worm_w32_waledac_a.shtml
[29] http://blogs.technet.com/mmpc/archive/2009/01/19/waledac-trojan-hosted-by-fake-obama-website.aspx
[30] http://blogs.fayobserver.com/techsassy/2009/03/18/security-alert-waledac-trojan-uses-fake-bomb-warning-to-dupe-users/
[31] http://www.mxlogic.com/securitynews/spam/waledac-worm-celebrates-valentines-day-with-spam383.cfm

[32] B. Oommen and L. Rueda - "Stochastic Learning-based Weak Estimation of Multinomial Random Variables and Its Applications to Non-stationary Environments" from Pattern Recognition, vol. 39, no. 1, pp. 328–341, 2006
[33] http://www.confickerworkinggroup.org/wiki/pmwiki.php
[34] http://ieeexplore.ieee.org/iel5/4041138/4041139/04041158.pdf?arnumber=4041158
[35] http://ieeexplore.ieee.org/iel5/9473/30059/01377213.pdf?arnumber=1377213

Appendix A - Requirements

Priority A

The input to the automaton must be Argus generated flowdata

There are much more data in network traffic in addition to values that describe the flow. The automaton will not use this content even if this would have made things easier during design / analysis. Argus is a specific tool that can generate flowdata based on packet captures.

The automaton must be able to reach a decision

It will not be possible to evaluate our solution nor compare it to Snort without a result in the form of a decision. It is therefore crucial that this is implemented.

The automaton must be implemented using Python

To enable integration with existing Telenor systems the automaton must be implemented using the Python scripting language. Even if the integration is not done during our thesis, it is important to have this possibility later on. Python target version is 2.4.

The prototype must be able to run on the i386 platform

This is because the hardware, which will be provided to us by Telenor, is based on the i386 platform.

Priority B

The automaton can work on live traffic

Working on old data is convenient during development as the same data can be tried again and the results compared, but during testing we wish to test on a live network with real traffic.

The automaton must be able to work with traffic over time

To be aware of activities that happen at certain intervals it must be time aware. By processing traffic over time it should be able to detect markers that otherwise would go unnoticed.

Priority C

Integration into Telenor systems

This would be a nice feature for the sake of showing what our implementation can be used for. Integration is not a part of designing the automaton, but a good way to show where it fits into existing solutions.

Categorization / identification after automaton analysis

A step further than just giving an opinion on the traffic, giving a classification could be possible by using additional automata that are tailored to specific behaviors.

Optimizing Python code

As an interpreted language Python does not necessarily provide great performance out of the box. This can be countered by various optimization techniques and tools.

Appendix B - Data CD

A CD should be enclosed with your copy of the thesis.

The CD includes:

- Source code for the proposed LRPI and SWE IDS
- Pcap and argus data for *test set 1*
- Pcap and argus data for *test set 2*
- Snort CPU and memory performance data
- Automata CPU and memory performance data
- PDF version of this thesis

The PDF version of the thesis has interactive references that allow you to jump to the relevant figures, tables and chapters.

Test set 3 is not included on the CD, as we are not allowed to distribute the pcap nor the argus data. Sharing this information would violate our contract with Telenor.

The implementation has been configured to run *test set 2* by default. Open up a console and run the following command:

```
cd <path to cd>
cd "Source code"
python "app.py"
```

Python 2.x and Argus 3 must be available. The output of these commands can be seen in Appendix C.

Appendix C - Sample output

```
 _   ___ ___ ___                 _   _____      _____  ___ ___  ___
| | |  _ \ _ \_ _|  __ _ _ _  __| | / __\ \    / / __|| _ _|   \/ __|
| |_|   /  _/| |  / _` | ' \/ _` | \__ \\ \/\/ /| _|   | || |) \__ \
|___|_|_\_| |___| \__,_|_||_\__,_| |___/ \_/\_/ |___| |___|___/|___/
By Christian Auby, Torbjørn Skagestad and Kristian Tveiten

Automaton name                                Updates Detection time
192.168.1.74 0.0.0.0_waledac_combined         10      0:13:08
192.168.1.90 221.7.91.31_conficker            10      0:01:35
192.168.1.71 203.174.83.75_danmec             10      0:42:22
192.168.1.73 194.63.248.34_blackenergy        10      1:30:02
192.168.1.73 194.63.248.34_timecheck          19      3:00:03
192.168.1.71 203.174.83.75_timecheck          29      5:52:35

Updates  SourceIP       DestinationIP   Automata type     Threat  Runtime
-------------------------------------------------------------------------------
 241     192.168.1.71   203.174.83.75   danmec            100 %   1 day, 4:52:36
1239     192.168.1.74   0.0.0.0         waledac_combined   93 %   1 day, 4:57:51
 173     192.168.1.73   194.63.248.34   blackenergy       100 %   1 day, 4:50:46
 173     192.168.1.73   194.63.248.34   timecheck         100 %   1 day, 4:50:46
 138     192.168.1.71   203.174.83.75   timecheck          78 %   1 day, 4:52:30
  44     192.168.1.90   221.7.91.31     conficker          95 %   3:05:51
```