NTNU
Innovation and Creativity

Intrusion Detection in High-Speed Networks

Martin Riegel
Claes Lyth Walsø

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Svein Johan Knapskog, ITEM
Co-supervisor: André Årnes, Kripos

Norwegian University of Science and Technology
Department of Telematics

Problem Description

In this assignment, methods for high-speed intrusion detection using programmable network monitoring cards will be studied. A distributed high-speed intrusion detection system based on an IDS and DAG monitoring cards is to be deployed in the Uninett backbone network, and experiments and testing will be performed. In addition, strategies for handling high-speed network data, such as the use of programmable network cards, parallel processing and sampling will be considered. Possible applications for a prototype may be real-time security monitoring, as well as the collection and statistical analysis of security relevant data.

Assignment given: 01. February 2007
Supervisor: Svein Johan Knapskog, ITEM

ABSTRACT

This thesis investigates methods for implementing an intrusion detection system (IDS) in a high-speed backbone network. The work presented in this report is run in cooperation with Kripos and Uninett.

The popular IDS software, Snort, is deployed and tested in Uninett's backbone network. In addition, the monitoring API (MAPI) is considered as a possible IDS implementation in the same environment. The experiments conducted in this report make use of the programmable DAG card, which is a passive monitoring card deployed on several monitoring sensors in Uninett's backbone.

As a limitation of the workload, this report only focuses on the detection of botnets. Botnets are networks consisting of infected computers, and are considered to be a significant threat on the Internet as of today.

A total of seven experiments using Snort are presented. These experiments test 1) the impact the number of rules have on Snort, 2) the importance of good configuration, 3)the importance of using well written rules, 4) Snort's ability to run in an environment with minimum external traffic, 5) the impact the size of the processed packets have, 6) the impact the TCP protocol has on packet processing and 7) Snort's ability to run as a botnet detection system for a longer period of time.

Based on the results from these experiments, it is concluded that Snort is able to run as a botnet detection system in a high-speed network.

This report also discusses some strategies for handling high-speed network data and some future aspects. In addition, ideas for further work and research are given in the end of the report.

PREFACE

This thesis serves as a Master of Science thesis pursued in the 10th semester of the masters programme in telematics at the Norwegian University of Science and Technology, NTNU.

The motivation for the research described in this report was given by Uninett and André Årnes from the Computer Crime Division at Kripos.

We would like to thank our main supervisor, André Årnes, for continuous feedback and help throughout the semester, and professor Svein J. Knapskog for his helpful advice and guidance. Also thanks to Erik Hjelmås at Gjøvik University College for providing the IRC server used in the experiments.

A very special thanks goes to Arne Øslebø at Uninett. Without you, this thesis would never have been finished.

Trondheim, June 2007

Martin Riegel Claes Lyth Walsø

ABBREVIATIONS

In this section, abbreviations commonly used in the report are listed in alphabetical order and explained:

AIM AOL Instant Messenger
A freeware for instant messaging

API Application Programming Interface
A set of routines, protocols and tools for creating software applications

C&C Command Control
A C&C server is a server used to control and communicate with bots

CPU Central Processing Unit
The processing component in a computer

CSV Comma Separated Value
A file format which values are separated by a comma

DAG Data Acquisition and Generation
A Gigabit Ethernet monitoring card

DiMAPI Distributed MAPI
A distributed functionality for MAPI (see below)

DoS Denial of Service
A computer attack that attempts to make resources unavailable

FTP File transfer protocol
Protocol used on the Internet for transferring files

GB Giga Byte
One billion bytes

Gpbs Giga bit per second
One billion bits per second. Used in network traffic context

GUI Graphical User Interface
An user interface which let people interact with a computer

ICMP Internet Control Message Protocol
A protocol part of the Internet protocol suite

IDS Intrusion Detection System
A system which detects intrusion attempts in a computer network

IGRP Interior Gateway Routing Protocol
A protocol for exchanging routing data

IP Internet Protocol
A data-oriented protocol used for communicating data across a packet-switched network

IPS Intrusion Prevention System
A system which prevents intrusion attempts in a computer network. Often used with an IDS(see above)

IRC Internet Relay Chat
A communication protocol

ISP Internet Service Provider
An company which sells Internet access to customers

LBNL Lawrence Berkeley National Laboratory
Laboratory in USA which is managed by the University of California

LOBSTER Large-scale Monitoring of Broadband Internet Infrastructures
A pilot European infrastructure for accurate Internet traffic monitoring

MAPI Monitoring API
An API (see above) for creating functionalities for network monitoring

MB Mega Byte
One million bytes

NIC Network Interface Controller
An adapter circuit board installed in a computer to provide a physical connection to a network

OS Operating system
The operating system installed on a computer/server

OSI Open Systems Interconnection
A network standard which provides a layered reference model

Pcap Packet Capture
API (see above) for capturing packets on a computer network

PIDS Privacy IDS
An IDS (see above) that is specifically tailored to intrusion attempts related to privacy issues

PoS Packet over SONET
A protocol for sending packets over a SONET (see below)

SCAMPI Scaleable Monitoring Platform for the Internet
A European project to develop a scaleable monitoring platform for the Internet

SONET Synchronous Optical Networking
A method for computer communication over optical fiber

SSH Secure Shell
A network protocol for securing a communication channel

TB Tera Byte
One trillion bytes

TCP Transmission Control Protocol
A transport protocol that guarantees reliable and in-order delivery of sender to receiver data

ToS Type of Service
A field in the IP header used for specifying a service

UDP User Datagram Protocol
A connectionless network protocol

VM Virtual Machine
A virtual machine achieved by using VMWare

CONTENTS

Abstract **i**

Preface **iii**

Abbreviations **vii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objective . . . 1
1.3 Context . . . 2
1.3.1 Threats on the Internet . . . 2
1.3.2 Botnets . . . 2
1.4 Research methodology . . . 3
1.5 Contributions . . . 3
1.6 Structure . . . 4

2 Background **5**
2.1 Introduction . . . 5
2.2 IDS . . . 5
2.2.1 Signature- and Anomaly-Based IDS . . . 6
2.2.2 Network-Based IDS . . . 6
2.2.3 Host-Based IDS . . . 6
2.2.4 Stack-Based IDS . . . 6
2.2.5 High-Speed IDS . . . 7
2.3 Related Work . . . 7
2.3.1 A Splitter architecture . . . 7
2.3.2 HotBots 07 . . . 8
2.4 Snort . . . 8
2.4.1 The Ruleset . . . 9
2.5 MAPI . . . 9
2.5.1 The Ruleset . . . 10
2.6 Additional Software . . . 10
2.6.1 VMware . . . 10
2.6.2 Wireshark . . . 11
2.6.3 Cerebus . . . 11
2.7 UNINETT . . . 12
2.7.1 Passive sensor . . . 13
2.8 DAG cards . . . 14
2.8.1 Filters . . . 14
2.8.2 Making filters . . . 15
2.8.2.1 Filter syntax . . . 16
2.9 Botnet . . . 18

2.9.1 Botnet Protocols . . . 18
2.9.2 The Botnet Lifecycle . . . 19
2.9.3 Consequences . . . 19
2.9.4 Botnet Families . . . 20
2.10 The IRC protocol . . . 20

3 Approach **23**
3.1 Introduction . . . 23
3.2 Architecture . . . 23
3.2.1 Security Aspects . . . 24
3.2.2 Botmaster . . . 25
3.2.3 Bot . . . 25
3.2.4 IRC Server / C&C Server . . . 26
3.2.5 IDS . . . 27
3.2.6 Pros and Cons . . . 28
3.3 Implementation Preparations . . . 28
3.3.1 Compiling the Bots . . . 28
3.3.2 IDS Rules . . . 31
3.3.3 Malicious Traffic Script . . . 32
3.4 Implementing Snort . . . 32
3.4.1 Experiment 1 - Testing The Number of Rules . . . 34
3.4.2 Experiment 2 - Optimising Snort Configuration . . . 34
3.4.3 Experiment 3 - Testing Impact of Common Strings . . . 35
3.4.4 Experiment 4 - Testing With Minimum External Traffic . . . 35
3.4.5 Experiment 5 - Testing Packet Length Input . . . 35
3.4.6 Experiment 6 - Testing Impact of the TCP Protocol . . . 36
3.4.7 Experiment 7 - IDS Stand-Alone Test . . . 36
3.5 Implementing MAPI . . . 37

4 Results **39**
4.1 Introduction . . . 39
4.2 Results from the Experiments . . . 40
4.2.1 Experiment 1 - Testing Number of Rules . . . 40
4.2.2 Experiment 2 - Optimising Snort Configuration . . . 42
4.2.3 Experiment 3 - Testing Impact of Common Strings . . . 43
4.2.4 Experiment 4 - Testing With Minimum External Traffic . . . 43
4.2.5 Experiment 5 - Testing Packet Length Input . . . 44
4.2.6 Experiment 6 - Testing Impact of the TCP Protocol . . . 45
4.2.7 Experiment 7 - IDS Stand-Alone Test . . . 46

5 Analysis **47**
5.1 Introduction . . . 47
5.2 Analysing the Experiments . . . 47
5.2.1 Experiment 1 - Testing Number of Rules . . . 48
5.2.2 Experiment 2 - Optimising Snort Configuration . . . 51
5.2.3 Experiment 3 - Testing Impact of Common Strings . . . 54
5.2.4 Experiment 4 - Testing With Minimum External Traffic . . . 55
5.2.5 Experiment 5 - Testing Packet Length Input . . . 56
5.2.6 Experiment 6 - Testing Impact of the TCP Protocol . . . 57

5.2.7 Experiment 7 - IDS Stand-Alone Test . . . 58
5.3 Sources of Error . . . 58
5.4 Conclusion . . . 59

6 Discussion **61**
6.1 Introduction . . . 61
6.2 Snort vs. MAPI . . . 61
6.2.1 The Rules . . . 61
6.2.2 Performance . . . 61
6.2.3 Distributed approach . . . 62
6.3 DAG cards and Other Hardware . . . 62
6.4 Circumventing the IDS . . . 63
6.5 Botnet Similarities . . . 63
6.6 Alternative Detection Algorithms . . . 63
6.7 Handling the Detection Data . . . 64
6.8 Future Prospects . . . 64
6.8.1 A distributed Approach . . . 64
6.8.2 IPv4 vs. IPv6 . . . 65

7 Further work **67**
7.1 Introduction . . . 67
7.2 Implementing MAPI as an IDS . . . 67
7.3 Testing Maximum Traffic Load . . . 67
7.4 Botnets Based on Other Protocols . . . 67
7.5 Increased experiment length and frequency . . . 68
7.6 Dynamic Rules . . . 68
7.7 Notification Upon Detection . . . 68
7.8 Detection of Other Types of Intrusion . . . 68
7.9 Sampling . . . 68

References **70**

A Modified Snort configuration file **71**
A.1 snort.noflow.conf . . . 71

B MAPI IDS program **77**
B.1 MAPI IDS program . . . 77

LIST OF FIGURES

1.1 Active bot-infected computers per day . . . 3

2.1 Overview of the splitter architecture . . . 7
2.2 A Snort rule . . . 9
2.3 A MAPI rule . . . 10
2.4 Illustration of a virtual machine . . . 11
2.5 Overview over UNINETT's backbone . . . 12
2.6 DAG 4.3s Networking monitoring card . . . 14
2.7 Options for making filters . . . 15
2.8 DAG Filter syntax . . . 16
2.9 DAG Filter syntax example . . . 17
2.10 How hosts become part of botnets . . . 19
2.11 A small IRC network . . . 21

3.1 Overview of the experimental architecture . . . 24
3.2 UNINETT sensor . . . 27
3.3 Rbot configuration file . . . 30
3.4 An example of a content matching Snort rule . . . 31
3.5 mIRC bot command script . . . 32
3.6 Snort.noflow.conf . . . 33
3.7 Snort.tweak.conf . . . 35
3.8 Mapi cooking function . . . 37

5.1 Detection Ratio in experiment 1 . . . 48
5.2 Snort drop ratio in experiment 1 . . . 49
5.3 Processed packets per second in experiment 1 . . . 50
5.4 Detection Ratio in experiment 2 . . . 51
5.5 Snort dropping ratio in experiment 2 . . . 52
5.6 Packet drop ratio in experiment 3 . . . 54
5.7 The effect of the slen value . . . 56
5.8 The impact of the TCP protocol . . . 57

A.1 snort.noflow.conf part 1 . . . 71
A.2 snort.noflow.conf part 2 . . . 72
A.3 snort.noflow.conf part 3 . . . 73
A.4 snort.noflow.conf part 4 . . . 74
A.5 snort.noflow.conf part 5 . . . 75

B.1 MAPI IDS program part 1 . . . 77
B.2 MAPI IDS program part 2 . . . 78

LIST OF TABLES

4.1 Results when using 5 rules 40
4.2 Results when using 30 rules 40
4.3 Results when using 50 rules 40
4.4 Results when using 100 rules 41
4.5 Results when using 200 rules 41
4.6 Results when using 1000 rules 41
4.7 Results when using 200 rules with optimised configuration file 42
4.8 Results when using 1000 rules with optimised configuration file 42
4.9 Results when common words are inlcuded in the ruleset 43
4.10 Results when using 5 rules and a minimum of external traffic 43
4.11 The effect of the slen value when using the modified configuartion file 44
4.12 The effect of the slen value when using the optimised configuartion file 44
4.13 The impact the TCP protocol has on the ratio of dropped packets 45
4.14 The IDS stand-alone test 46

CHAPTER 1
INTRODUCTION

In this report, methods for high-speed intrusion detection systems using programmable network monitoring cards will be studied. A high-speed intrusion detection system (IDS) based on Snort, MAPI and DAG monitoring cards will be deployed in the Uninett backbone network, and experimental testing will be performed. In addition, strategies for handling high-speed network data will be considered.

1.1 MOTIVATION

With the increasing amount of threats on the Internet there is also an increasing need for systems to detect intrusion attempts. These IDSs need to be able to detect known attacks and to handle new ones as they are discovered.

Modern high-speed networks set requirements to scalability of the IDS, as high-speed IDS need to monitor every IP packet and flow and correlate this with the IDS's database for known attacks.

The research presented in this report is inspired by the LOBSTER[1] and SCAMPI[2] projects. LOBSTER is a pilot European Infrastructure for large-scale monitoring of broadband Internet infrastructure. LOBSTER uses passive monitoring sensors connected to the network at speeds from 2.5 to 10 Gbps. SCAMPI is a predecessor to LOBSTER and is an acronym for a Scalable Monitoring Platform for the Internet. The research presented in this report is conducted in cooperation with Uninett. Uninett is responsible for the national research and educational network, and they are also part of the SCAMPI and LOBSTER projects.

1.2 OBJECTIVE

The primary objective of this project is to implement an IDS in Uninett's backbone. Strengths and weaknesses of the different approaches will be studied using Snort and MAPI. Snort is an open source, de facto standard for intrusion detection and prevention. MAPI is a network monitoring API designed for high-speed networks.

The IDSs presented in this report will, due to workload limitations, only focus on botnets. The overall goal of this report is however to end up with a scalable IDS implemented on a 2.5 Gbps link. Due to privacy issues this report does not include a distributed approach, as access was only granted for one sensor. This is further discussed in Section 6.8.1.

[1] http://www.ist-lobster.org/
[2] http://www.ist-scampi.org/

1.3 CONTEXT

This section will explain some of the current threats on the Internet relevant for this report.

1.3.1 Threats on the Internet

According to the *Symantec*[3] *Internet Security Threat Report* [Cor06], malicious attacks have gone from being network-based to target client side applications. [Cor06] is a report of the security threat level on the Internet based on the observations made in the fist six months of 2006, while [Cor07] is for the last six months of 2006. Symantec's observations are based on the traffic collected on several honeypots and by utilizing over 40.000 sensors in over 180 countries.

The following points sums up some highlights mentioned in [Cor06] and [Cor07]:

- An average of 6110 Denial of Service(DoS) attacks daily in the first six months of 2006.
- In the first six months of 2006, 86% of all targeted attacks were against the home user sector. This increased to 93% during the last six months.
- During the first half of 2006, 18% of all distinct malicious code samples were new.
- Worms made out 38 of the top 50 malicious code samples during the same period.
- During 2006 there was an 81% increase of phishing messages, which again increased another 6% the last six months.
- 84% of the phishing activity had financial gain as intention.

As can be seen from the last point, and also mentioned throughout [Cor06], [Cor07] and [BY07], there has been an increase in attacks with the purpose of financial gain.

Also worth mentioning in this connection is that 38% of DoS attacks were directed against Internet Service Providers (ISP) during the first half of 2006.

1.3.2 Botnets

According to [RZMT06], a botnet is a network consisting of infected computers. These computers, or bots, are under human control.

Symantec mentions in [Cor06] that they have detected an average of 57,717 active bot networks per day during the first six months of 2006. As mentioned in [Cor07], this increased to 63,912 during the last six months. Figure 1.1 shows the number of active bot-infected computers per day during 2006. The *moving average* line is a calculated mean value of the number of active bots at the given period.

Approximately 6.000 of these were command and control servers (C&C). C&C servers are servers which provide communication channels for the botmaster, which allows the botmaster to communicate with the bots in the network [RZMT06].

[3]Symantec is, according to their website, "a global leader in infrastructure software, enabling businesses and consumers to have confidence in a connected world." They sell and manufacture security software.

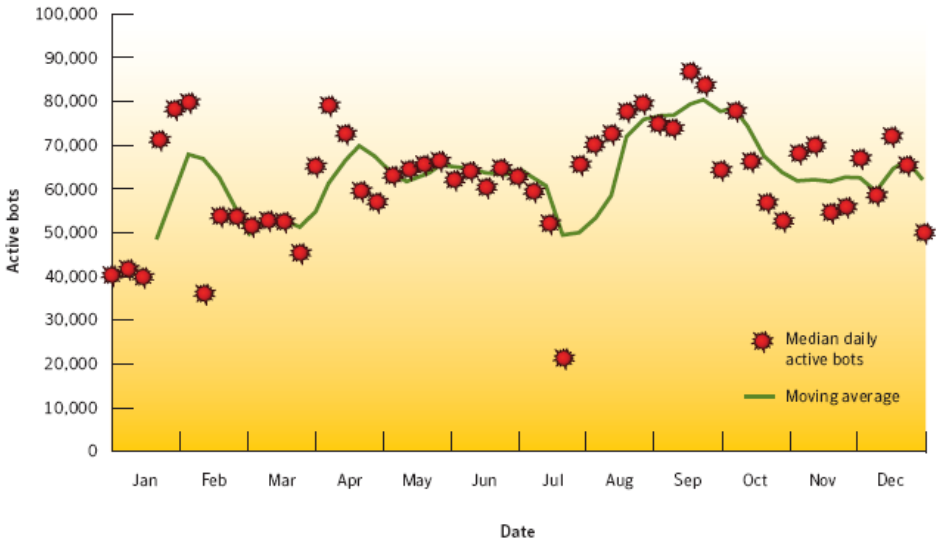

Figure 1.1: Active bot-infected computers per day. The figur is extracted from [Cor07]

Botnets will be further elaborated in Section 2.9 as they are the main focus of the research conducted in this report.

1.4 RESEARCH METHODOLOGY

The following is the scientific steps of the research presented in this report:

1. It is necessary to perform a background and theoretical study.
2. The design of the experimental architecture and intrusion detection system has to be made.
3. The experiments have to be prepared, i.e., locate and install bots in a safe environment. In addition, an IRC server has to be set up in a remote location.
4. Conduct the actual experiments.
5. The results from the experiments will be analysed and discussed.

1.5 CONTRIBUTIONS

This report demonstrates that it is possible to use the popular intrusion detection system Snort as an IDS in a high-speed environment, with the use of specialized network hardware in the DAG cards.

The experiments conducted in this report were run in cooperation with Uninett. On the basis of these experiments several bugs in MAPI were discovered. These bugs have been reported, but they are not included in this report, as they are outside the topic if this research.

1.6 STRUCTURE

This section gives a brief description on the rest of the chapters in the report.

Chapter 2
Chapter 2 provides some background knowledge needed to understand the research presented further in this report.

Chapter 3
Chapter 3 presents the experimental approach.

Chapter 4
Chapter 4 presents the results from the experiments. The results are given in tables.

Chapter 5
Chapter 5 analyses and discusses the results given in Chapter 4.

Chapter 6
Chapter 6 discusses some of the different elements in relations with the experiments presented in this report.

Chapter 7
Chapter 7 suggest some further work related to the research presented here.

The configuration files, bot source code and IDS rules, are included as a digital attachment to this report.

CHAPTER 2
BACKGROUND

This chapter will provide some background material needed for further reading of this report.

The term IDS will be explained and different kinds of IDSs will be discussed. Some related work will be presented next. An overview over the hardware and software used in the research will then given (i.e., Snort, MAPI and some other software applications, and also the DAG-cards). In addition, information about Uninett and some background information about botnets and the IRC protocol will be presented.

2.1 INTRODUCTION

As mentioned in Chapter 1, this report focuses around the process of implementing an IDS in a high-speed environment and the problem and challenges which may arise in that connection.

2.2 IDS

An Intrusion Detection System is continuously inspecting network activity. The IDS is looking for suspicious traffic traversing the network, mainly in the inbound direction. However, there are some IDSs which also analyses the traffic in the outbound direction.

As opposed to other network monitoring systems, an IDS looks for specific traffic patterns which are believed to be malicious or suspicious. When suspicious traffic is detected an IDS may take a number of actions. The most common is to alert the administrators who can take necessary actions.

As mentioned, there are several kinds of IDSs, all of which works in different ways. Some of the most important ones will be explained in the following sections in order to get a full understanding of how an IDS works. Section 2.2.5 describes IDSs working in high-speeds environments and some related work. Most of the explanations below are extracted from NISTs[1] *Guide to Intrusion Detection and Prevention Systems* [SM07]. As the title suggest, [SM07] also covers the field of Intrusion Prevention Systems (IPS). According to [SM07], an IPS is software that has all the capabilities of an IDS, but can also attempt to stop possible incidents. This report will, however, focus on IDS.

[1] NIST - National Institute of Standards and Technology

2.2.1 Signature- and Anomaly-Based IDS

Signature-based detection is the process of looking for known malicious network traffic patterns. This process works very well with known threats, although any new suspicious activity will not be detected.

Anomaly-based detection is the process of comparing the network traffic pattern to what is considered to be known normal traffic. The IDS monitors the traffic over a period of time, known as the training period, and creates a profile of the normal traffic. Statistical methods are then applied to detect deviations. These profiles may be quite extensive and anomaly-based IDS has shown to work well when it comes to detecting unknown threats.

2.2.2 Network-Based IDS

A network-based IDS simply analyses the network traffic. According to [SM07], it is most commonly deployed at the network boundaries. The traffic is analysed in real-time as it traverses the network. Usually a filter is used to decide which traffic is to be let through and which is to be further analysed by the IDS.

Some of the strengths of a network-based IDS are explained in [LA00]. Itis worth mentioning that this type of IDS does not rely on any software or operation system specifications on the hosts, even though they may be configured to do so if necessary. Because it provides real-time packet analysis, a fast and efficient response may be achieved. Using a network-based IDS makes it harder for an attacker to hide her tracks.

2.2.3 Host-Based IDS

A host-based IDS analyses the activity of a single host for suspicious activity (i.e., network traffic specific for the host, system logs, running processes, application activity, file access and modification, and system and application configuration changes) [SM07].

[LA00] list some of the strengths in host-based IDS. This type of detection is less prone to false positives, and may make it possible to see whether an attack was successful. A host-based IDS may also detect system specific activity at the host. Depending on the implementation, host-based IDSs may provide close to real-time analysis. In addition, no additional hardware is needed by using a host-based IDS.

2.2.4 Stack-Based IDS

[LA00] refers to stack-based IDSs as the newest of IDS technology. These IDSs analyse the packets as they traverse the layers of the protocol stack. If the IDS detects malicious traffic, the packets may be rejected before they are processed at the application layer. It is claimed in [LA00] that a *complete* stack-based IDS checks both inbound and outboud traffic.

This kind of IDS will not be discussed any further, as it is considered outside the scope of this report.

2.2.5 High-Speed IDS

A high-speed IDS is, in this report, referred to as an IDS working in an environment with excessive bandwidth, from 2.5 Gbps to 10 Gbps. [SYL03] mentions that the main problem with running an IDS in such environments is the huge amounts of data that needs to be processed.

The next Section describes some work related to high-speed IDS research presented in this report.

2.3 RELATED WORK

In this report, related work is considered to be research on high-speed IDSs and also IDSs specifically designed to detect botnets based on IRC. More specificly, this also includes research related to the LOBSTER and SCAMPI projects, mentioned in section 1.1.

2.3.1 A Splitter architecture

Several solutions for handling the problems concerning high-speed IDS have been published. One solution is described in [XCA+06] by some of the LOBSTER partners. The article presents a traffic splitter architecture, which was also mentioned in [CAM04]. The purpose of this architecture is to split the traffic among several intrusion detector sensors. One important principle is that packets that belong to the same attack will be processed by the same sensor. Figure 2.1 presents an overview of the splitter architecture.

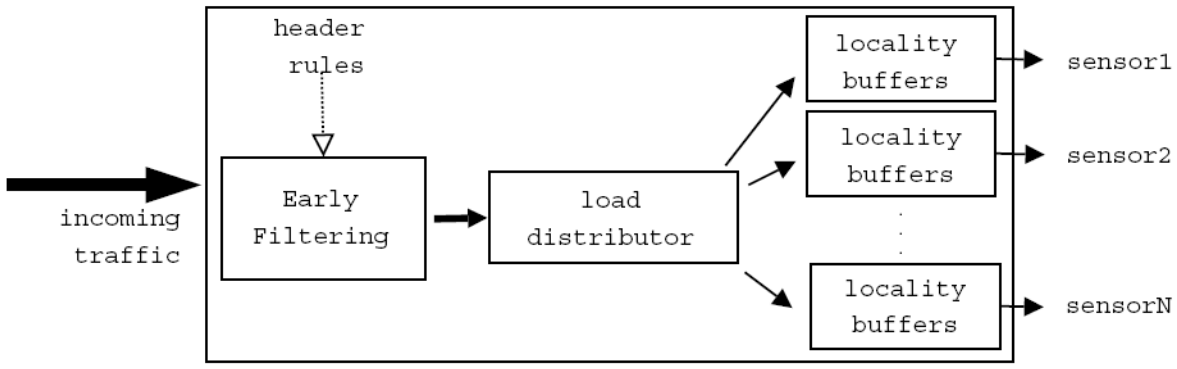

Figure 2.1: Overview of the splitter architecture. Copied from [CAM04].

The splitter architecture consists of the following components:

- The early filtering process only checks the IP packet headers. If the header is not known to be malicious and the packet contains no payload, the packet is filtered out. Otherwise, the packet is forwarded to the load distributor.
- The purpose of the load distributor is to divide the network traffic among the end sensors, and to keep them as evenly loaded as possible. In addition, the packets of the same network flow should be examined by the same sensor. This is achieved by computing

a hash function on selected fields in the packet headers, and then forward the packets with the same hash value to the same sensor.

- The locality buffers attempts to rearrange the interleaving of packets in the traffic so that subsequent packets will trigger the same ruleset as often as possible, and thus decrease the processing time.

This architecture is shown to work well with high speed networks below 1Gbps. However, the architecture based on special designed hardware which will not be used in the research in this report. See [XCA+06] and [CAM04] for further elaboration.

2.3.2 HotBots 07

In April 2007, a conference (HotBots 07) about botnet research was arranged in Cambridge, USA. The article [KRH07] describes a method to detect, track, and characterize botnets on high-speed networks. It is claimed that their method:

1. is entirely passive and therefore invisible to operators,
2. scales to the largest of networks,
3. is based on flow data analysis, which limits privacy issues,
4. has a false positive rate of less than 2%,
5. helps identify botnets that are most affecting real users and customers,
6. can detect botnets that use encrypted communication, and
7. helps quantify size of botnets, identify and characterize their activities without joining the botnet.

The main focus in this thesis report is however to implement a high-speed IDS using MAPI and Snort, eleborated in sections 2.5 and 2.4, respectively. The mentioned approach will, however, not be considered any further.

[GH07] is another article presented at *HotBots 07*. This article is more relevant to the research conducted in connection with this thesis. [GH07] presents a method which relies on the detection of the communication channel between bot and the C&C server. This method is based on the special nicknames that are assigned to the bots in a botnet. By looking at the IRC traffic and inspecting special "unhuman" nicknames, they are able to detect IRC-based botnets.

2.4 SNORT

The research presented in this report, uses the open source IDS called Snort. The following text is cited from Snort's official website (www.snort.org):

Snort is an open source network intrusion prevention and detection system utilizing a rule-driven language, which combines the benefits of signature, protocol and anomaly based inspection methods.

With millions of downloads to date, Snort is the most widely deployed intrusion detection and prevention technology worldwide and has become the de facto standard for the industry.

Snort may be used as a packet sniffer like tcpdump, as a packet logger, or as an IDS. When using Snort as an IDS, it uses a set of rules to detect intrusions. These rules are explained in the next section.

2.4.1 The Ruleset

According to their website, rules are based on detecting the actual vulnerability as opposed to a specific signature. A signature is based on an exploit of a unique piece of data.

Figure 2.2 shows an example Snort rule[2].

A Snort rule

```
alert tcp any any -> 192.168.1.150 111 (content:"|00 01 86 a5|"; msg:"mountd
    access";)
```

Figure 2.2: A Snort rule, as presented in [Pro06].

In short terms, the rule in 2.2 will make Snort alert with the message *mountd access*, when tcp packets with the destination address *192.168.1.150*, port 111 is received and the given payload specified in *content:"|00 01 86 a5|"* is matched.

The rule header is the text up to the first parenthesis in Figure 2.2. As explained in [Pro06], the rule header contains the rules action, protocol, source, and destination IP addresses and netmasks, and the source and destination ports information. Following the header is the rule option section.

The rule option Section is according to [Pro06] the heart of Snorts intrusion detection engine. As can be seen in Figure 2.2, this is where the content matching rule is expressed.

2.5 MAPI

The research presented in this report also uses the Monitoring API called MAPI. The following text is cited from the MAPI webpage (mapi.uninett.no):

MAPI, or Monitoring API, is a multi-user programming interface designed to simplify the development of network monitoring software and allows users to express their monitoring needs in a device-independent way. The main abstraction provided by MAPI is the network flow. Although flows have been used before in network monitoring systems, MAPI gives flows a first-class status.

The experiments described in makes use of *MAPI 2.0Beta1*, which was released in September 2006. MAPI supports normal NICs (network interface controller), SCAMPI adapters, and also DAG cards without a co-processor. MAPI is implemented in C and provides a wide set of

[2]Because of available page width in the report the rule is split over two lines.

monitoring functionalities. MAPI runs as a background daemon. A user typically makes his own programs which uses functions available in MAPI. These programs connect to the MAPI daemon when they are executed. MAPI supports both libpcap and its own DAG API.

The paper [TPP+06] presents a distributed extension to MAPI (DiMAPI). By using DiMAPI, it is possible to create an IDS working with several sensors. [TPP+06] also presents a network IDS. The usage of DiMAPI is discussed in Section 6.8.1.

2.5.1 The Ruleset

It is easy to create rules corresponding to the Snort rules in MAPI. Figure 2.3 presents the code lines representing the same rule as in figure 2.2.

A MAPI rule

```
1   example = mapi_create_flow ("/dev/dag0");
2   mapi_apply_function (example, "BPF_FILTER","tcp AND dst host 192.168.1.150 AND dst port 111");
3   mapi_apply_function (example, "STR_SEARCH", "|00 01 86 a5|", 0, 1500);
4   mapi_apply_function (example, "TO_FILE", MFF_PCAP, "example.pcap", 0);
5
6   if(mapi_connect(example) < 0) {
7      printf("Could not connect to flow %d\n", example);
8      exit(EXIT_FAILURE);
9   }
10
11  mapi_close_flow(example);
```

Figure 2.3: A MAPI rule. Only the relevant lines are included.

First, as shown in Figure 2.3, the correct flow is created for the network interface of interest, `dag0`. Second, the filter, which corresponds to the rule header in the Snort rule, is created. Line 4 applies the search string function. This corresponds to the *content: "|00 01 86 a5|"* part of the Snort rule in Figure 2.2. Line 5 simply logs the detection in a pcap file.

The *if* test is simply to check whether the flow was created in a correct manner. Finally, the flow is closed. Notice that there are some lines missing in order to make this a complete C program.

2.6 ADDITIONAL SOFTWARE

In order to set up the experiments described in this report, some additional software is needed. A short explanation of this additional software is given in the following sections.

2.6.1 VMware

VMware is basically a program that emulates a computer. This makes it possible for multiple operating systems to co-exist as virtual machines (VM) inside the program, as illustrated in Figure 2.4. The main advantage of suing VMware for virtualization is that it is very easy to control the environment, especially when it comes to quickly pausing or shutting down an OS

if needed. These latter qualities are favourable for running the experiments throughout the report in a secure manner, as will be further elaborated in Chapter 3.

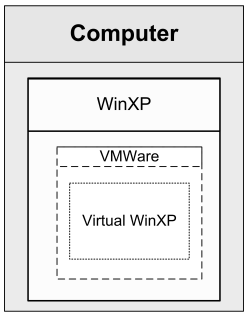

Figure 2.4: Illustration of a virtual machine

Another positive aspect with how VMware operates, is that it is easy to install monitoring software on the host OS outside of the VMs. This makes it very easy to see how the VM interacts with the external environment and take actions on unwanted communication.

2.6.2 Wireshark

According to *wireshark.org*, Wireshark is a popular network protocol analyser, previously known as Ethereal. A popular way of using Wireshark is to install it as a packet sniffer. This means that the computer administrator can get a complete picture of the traffic going from or to a computer.

Wireshark makes it possible to inspect traffic patterns, and it is a helpful tool when conducting the experiments described in Chapter 3. As will be shown, Wireshark makes it easy to distinguish between the traffic originating from the VM operating in normal mode and when the different bots are started.

2.6.3 Cerebus

According to *dragos.com/cerebus*, Cerebus is a "full screen, GUI and text-based unified IDS alert file browser and data correlator".

Cerebus makes it possible to read Snort alert files saved in the unified format. The unified format is written in binary and is amongst one of fastest (if not the fastest)[3] logging format available in Snort. Cerebus is a very useful tool when conducting the experiments throughout the report.

[3] According to [Pro06]; configuring Snort, output modules, unified.

2.7 UNINETT

"The UNINETT group supplies network and network services for universities, university colleges and research institutions and handles other national ICT[4] tasks. The Group is owned by the Norwegian Ministry of Education and Research and consists of a parent company and four subsidiaries" [5].

Uninett's network structure is very comprehensive and the network bandwidth is continuously increasing, as the network equipment is replaced with high-speed lines and new hardware. The fastest links are currently operating at a speed as high as 10 Gbps, with the majority of the network operating on speeds around 150-1000 Mbps and some on 2.5 Gbit.

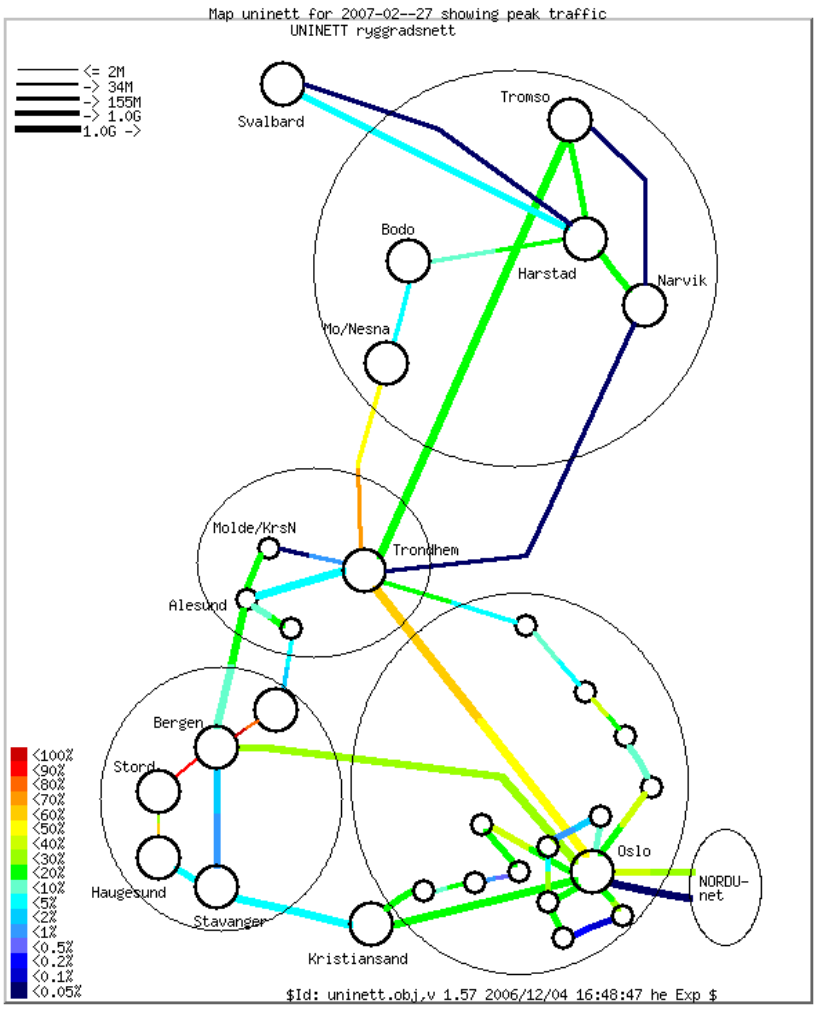

Figure 2.5: Overview over UNINETT's backbone[6]

The GigaCampus programme[7] is a four year initiative put forward by UNINETT. The over-

[4] ICT - Information and Communication Technology
[5] Quoted from http://www.uninett.no/om.en.html
[7] http://www.gigacampus.no

all objective is to address key networking challenges on the campus networks of Norwegian universities and university colleges towards 2009. One goal in this initiative is to increase the overall speed on the network. This means that gigabit networks should be a reality for most of the connected users, and all the big universities should be connected with 10 Gbps links.

2.7.1 Passive sensor

As mentioned above, UNINETT has an extensive network which requires monitoring on different levels. UNINETT has placed passive sensors throughout their network. These are fast computers with one or two DAG-card(s) installed. Information about DAG cards and how they operate are further elaborated in section 2.8. The experiments in this report will be carried out on a sensor connected to one of the most busy lines on the UNINETT backbone.

As a reference; the average traffic load[8] on the sensor on a typical day day, shows that approximately 160.000 packets is sent trough each seconds. The size of these packets sums up to 125MB each second - giving a total of 10.8 TB data per day.

[8] http://drift.uninett.no/

2.8 DAG CARDS

DAG cards are basically advanced network monitoring cards with support for bandwith up to 10 Gbps. They are, however, quite more sophisticated than a normal network interface card, in the sense that it is possible to program these cards to do packet processing directly on-board.

As mentioned in Section 2.7.1, UNINETT has placed passive sensors with DAG cards different places in their network. The DAG model 4.3S, as shown in Figure 2.6, was used in the experiments described in this thesis. This card supports 2.5 Gbps speeds on a Packet Over Sonet (Pos) link.

Figure 2.6: DAG 4.3s Networking monitoring card
Copyright Endace [Ltd05] 2005

DAG 4.3S can be extended with an extra co-processor, which allows for even more packet processing directly on the card. The main advantage of moving the processing to the card, is that hardware processing can be much faster than software processing. Another advantage is that it is possible to use the card as a filter, where packets which are not of interest are dropped instead of being forwarded to the software.

2.8.1 Filters

The DAG 4.3S card is, as mentioned, capable of storing filters on the card. When the card operates in default mode with one filter set, the card can store up to 16.384 filters. The filters are basically simple, one-line specifications used to describe characteristics of packets considered to be a match. In addition there is an action specified for each filter. The two possible actions are defined as:

- **Accept**
  Accepted packets are passed on to the host computer.
- **Reject**
  Rejected packets are dropped and not delivered to the host.

DAG 4.3S supports filtering on the different categories listed below. It should be underlined that if the categories are not present in the filter, they are just ignored when the filters are checked in the packet processing.

- **Ingress interface**
  This filtering option is not relevant for research presented in this report, because the tested DAG 4.3S card only has one interface.

- **Protocol**
  It is possible to filter on the different protocols ICMP, IGRP, TCP/RawIP or UDP.

- **Source and destination IP addresses**
  Filtering can be done according to which source and destination IP address the packet is going to and from. In addition, filtering can be done on whole or parts of a subnet.

- **TCP and UDP source and destination port numbers**
  It is possible to filter the packets on one or many given port numbers, either on the source or destination address - or both.

- **TCP flags**
  The TCP flags are typically SYN, ACK, FIN, PSH, URG or RST and can be used in the filtering.

2.8.2 Making filters

The filters on the DAG card can be written in different "languages". As can be seen in Figure 2.7 below, there are basically three different ways to make a filter.

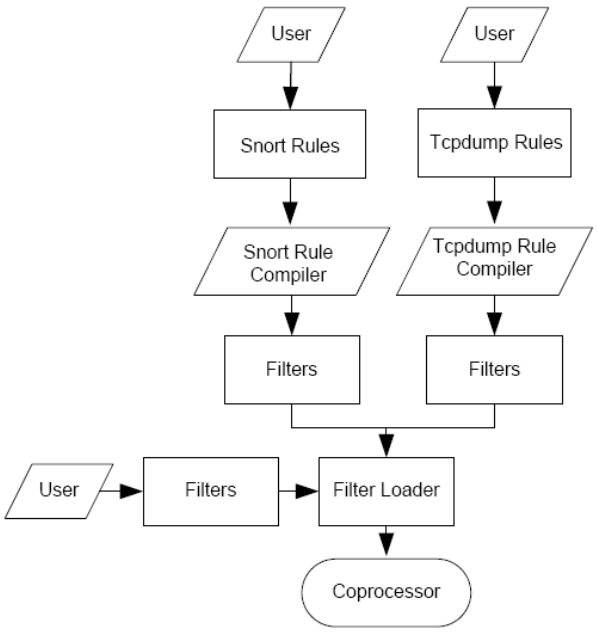

Figure 2.7: Options for making filters.
Copyright Endace [Ltd05] 2005

First, there are the written rules on the native DAG card format. As seen on the bottom left

side in Figure 2.7, these rules are basically filters which can be loaded directly into the co-processor by the filter loader.

Second, there are the Snort rules. These rules are written in the Snort format and can be copied directly from the rules used by Snort. Before it is possible to use the rules as filters on the DAG card, they have to be translated into DAG format. This is done using the Snort rule compiler, a program shipped with the DAG software. After compilation, the transformed Snort rules can be read by the filter loader and copied into the co-processor.

The third and last option, is to use rules written for the Tcpdump program. Just like the Snort rules, these rules need to be compiled before they can be used. This is done with the Tcpdump rule compiler, which generates DAG filters that can be read into the co-processor by the filter loader.

2.8.2.1 Filter syntax

Figure 2.8 shows the filter syntax for the filters used in the co-processor.

DAG Filter syntax

```
<number> <action> <protocol> <src-ip> <src-port> <dst-ip> <dst-port> <tcp-flags>
```

Figure 2.8: DAG Filter syntax

Some syntax fields are already mentioned in Section 2.8.1, but are somewhat further elaborated in the list below.

- **<number>** is the filter number which is an integer between 0 and 16384. The highest numbers are processed first. Filter number 0, is often a "reject all" filter. This means that the packets that reach this filter, are basically packets that did not match any of the other filters and can therefore be discarded.
- **<action>** is either reject or accept the packet if the filter is matched.
- **<protocol>** is the protocol which is to be filtered on, and can be either ALL or one of ICMP, IGRP, TCP/RawIIP and UDP.
- **<src-ip>** is a source IP address or subnet. The field is a 32 bit binary string, with the most significant bit starting at position 0. The following symbols can be used: *0*, *1* or -, where *0* and *1* represents binary bits while - represents a "do not care bit". This means that a complete 32 bit string represented with *1*s and *0*s is a complete IPv4 address. If the last eight bits are "don't care" bits, the filtering will match all IP addresses in a C-class subnet instead of one single IP address.
- **<src-port>** is the source port(s). The field is a 16 bit binary string, with the most significant bit starting at position 0. The following symbols can be used: *0*, *1* or -, where *0* and *1* represents binary bits while - represents a "do not care bit". This means that a complete 16 bit string with *1*s and *0*s refers to one specific port. 0000000000010101 equals port 21, while 00000000010— means that the filter will match port numbers between 16 and 23.

- **<dst-ip>** is a destination IP address or subnet. See **<src-ip>** for more details.
- **<dst-port>** is the destination port(s). See **<src-port>** for more details.
- **<tcp-flags>** is the TCP flag that is set. This is an eight bit field which, e.g., represents the values SYN, ACK, FIN, PSH, URG or RST.

Figure 2.9 below shows an example of a filter. Note that there are two entries, namely line 1 and 0.[9]

DAG Filter syntax example

```
#1
1 accept  tcp
src-ip {--------------------------------} src-port {----------------}
dst-ip {1100000010101000----------------} dst-port {0000000000010101}
tcp-flags {------1-}

#0
0 reject all
src-ip {--------------------------------} src-port {----------------}
dst-ip {--------------------------------} dst-port {----------------}
tcp-flags {--------}
```

Figure 2.9: DAG Filter syntax example

This first entry matches packets where the traffic is TCP, the destination subnet is 192.168.0.0/16 and destination port is 21. In addition, the TCP-flag SYN has to be set in the packet header. The second entry, line 0, basically rejects all packets which are not matched by line 1, regardless of each source or destination IP, traffic type or port numbers.

[9]Because of available page width in the report, the filters are split over four lines, while they originally are written on one line each.

2.9 BOTNET

A botnet is a network of bots, i.e., a collection of computers which are infected with some kind of malicious distributed software. This means that a bot is actually a program running on the infected host. As previously mentioned in Section 1.3.1, botnets are considered a significant threat on the Internet. The following sections elaborates the concept of botnets and the consequences they have on the end-users and the network.

2.9.1 Botnet Protocols

There are several ways to implement a botnet, and several communication protocols which may be used between the bots and the C&C servers. The following points list some various protocols used, as mentioned in [SBH+07].

- **Internet Relay Chat**
  According to [RZMT06], [GH07] and [KRH07], to name a few, most botnets are based on the IRC protocol. According to [GSN+07], using IRC makes it attractive to the botmaster due to its redundancy, scalability and versatility. Beacause of its importance, the IRC protocol is further explained in Section 2.10.

- **Web-based**
  According to [SBH+07], web-based C&Cs are the second most common type. [SBH+07] continues by mentioning two types of web-based botnets. The first one, *echo-based*, means that the bot simply announces its existence to the C&C server. This may for example be achieved by sending the information as an URL to the web server.

  The second type, *command-based*, uses GUI web interfaces to issue instructions to the bots.

- **Peer-to-peer**
  As mentioned in [GSN+07], a peer-to-peer network is a network in which any node in the network can act as both client and server. Botnets based on peer-to-peer architecture are more resilient, as there is no centralized C&C server.

- **Instant messaging**
  Computers infected with a bot would in this case communicate using known instant messaging protocols, such as AIM, Yahoo!, ICQ and MSN.

- **FTP-based**
  According to [SBH+07], there are cases in which the FTP protocol is used. The bot may collect information from the host, and then upload the captured information to an FTP server.

[SBH+07] also claims that IRC is going to stay the leading protocol and application for C&C servers for a long time to come, even though uses of other protocols are increasing. In the following, all the use of the term botnet is restricted to refer to an IRC based botnet.

2.9.2 The Botnet Lifecycle

Any vulnerable computer could become a part of a botnet, and an example on how the computer is compromised and turned into a bot is given below. This example is an excerpt from [RZMT06] and is illustrated in Figure 2.10 below.

1. As shown in the figure, the first step is to exploit a vulnerability in the software running on the victim's computer. This can be achieved by using self-replicating worms, e-mail viruses, etc.
2. After the attack, the exploited host will execute a script which will download a bot binary. The bot binary will install itself on the computer. Once this is installed, the bot software will start every time the computer is rebooted.
3. Most commonly, the bot running on the host will connect to a remote server. The IRC protocol is, in this example, used for communicating and controlling the bots. At this point, a communication channel between the infected host and the IRC server is set up.
4. The vulnerable computer is now part of the botnet and is under control of the botmaster via the IRC server. The botmaster has full control over the IRC server and thus all bots in her network. The botmaster may now send commands to her bots and prepare and execute a full blown attack. In addition, it is possible to modify the bots if necessary.

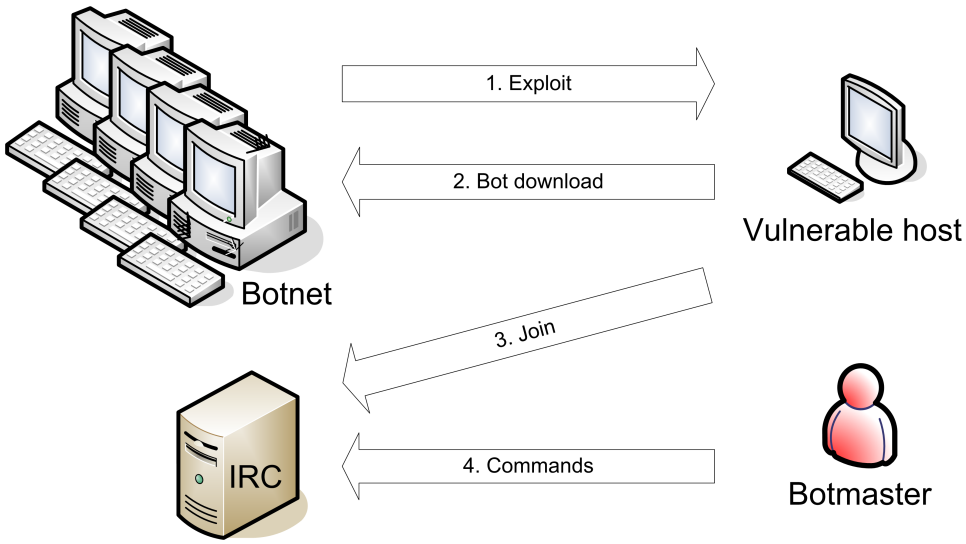

Figure 2.10: Stepwise procedure on how host become part of a botnet.

It is worth mentioning that this is only one example of how a host is turned into a bot. As mentioned, several other methods and communication protocols may be used, but this will not be elaborated any further in this report.

2.9.3 Consequences

As mentioned in [KRH07], a botnet may be used to launch distributed DoS attacks, send spam, trojan, phishing e-mails, and so on. All of which may have devastating effect on the networks.

As mentioned in [IH05], botnets may also be used to perform click frauds and stealing personal information. Software programmed to capture sensitive information such as bank account numbers, PIN codes, passwords, etc., may have financial implications to the end-users. In addition to loosing private and personal data, the users may be employees of companies whose business strategies and other sensitive information may be compromised.

2.9.4 Botnet Families

Today there exists several botnet families. [BY07] describes four of the biggest ones; Agobot, SDBot, SpyBot, and GT Bot. It is, however, difficult to get hold of the source codes to the different bots as a researcher with non-criminal intentions. Some are, however, available and will be used in the research presented in this report. The following is a list of these botnets, based on IRC, with a short explanation:

- **SDBot**
  This bot appeared in October 2002. According to [BY07], SDBot is a fairly simple bot, consisting of about 2.000 lines of C code. It is, however, claimed to be easily extendable.

  [SBH+07] states that SDBot's key to success is due to poor security on the compromised systems, and also the fact that the author released it as an open source. SDBot spreads itself by exploiting software using blank or common passwords on the host. The bot also includes a backdoor that allows an attacker to gain access to the infected host.

- **RBot**
  As mentioned in [SBH+07], the RBot family is one of the most pervasive and complex of the ones existing today. Filenames and techniques used are different from one variant to another. This bot was, according to [SBH+07], the first one to use compression and encryption.

  This bot includes functionalities that make it possible for the botmaster to download and execute files, creating a SOCKS proxy, participating in DDoS attack, logging keystrokes on the hosts, to name a few. In addition to using weak passwords as SDBot does, RBot also exploits vulnerabilities in the Windows operating system and common software applications. [SBH+07] also claims that RBot can termintate many antivirus and security products to ensure that it can continue to run undetected.

- **DBot**
  Unfortunately, there is not much documentation available about this bot, which makes it all the more interesting. This bot also provides a wide set of functionality. Some of the features are multicommand topic and chat parsing, IRC connection timeout and the ability to use unlimited number of irc servers, to name a few. Maybe the most threatening feature is that this bot also includes a bypass to Windows XP Service Pack 2's firewall. At least this is claimed in *features.txt*, which is appended to the source codes.

2.10 THE IRC PROTOCOL

As the focus of this thesis is on IRC-based botnets, this section provides some background knowledge of the IRC protocol.

According to [OR93], "IRC itself is a teleconferencing system, which (through the use of the client-server model) is well-suited to running on many machines in a distributed fashion." The backbone of IRC is the connection of the IRC servers. Clients connect to these servers and may then chat with other clients connected to the same network. An example of a small IRC network is given in Figure 2.11 below.

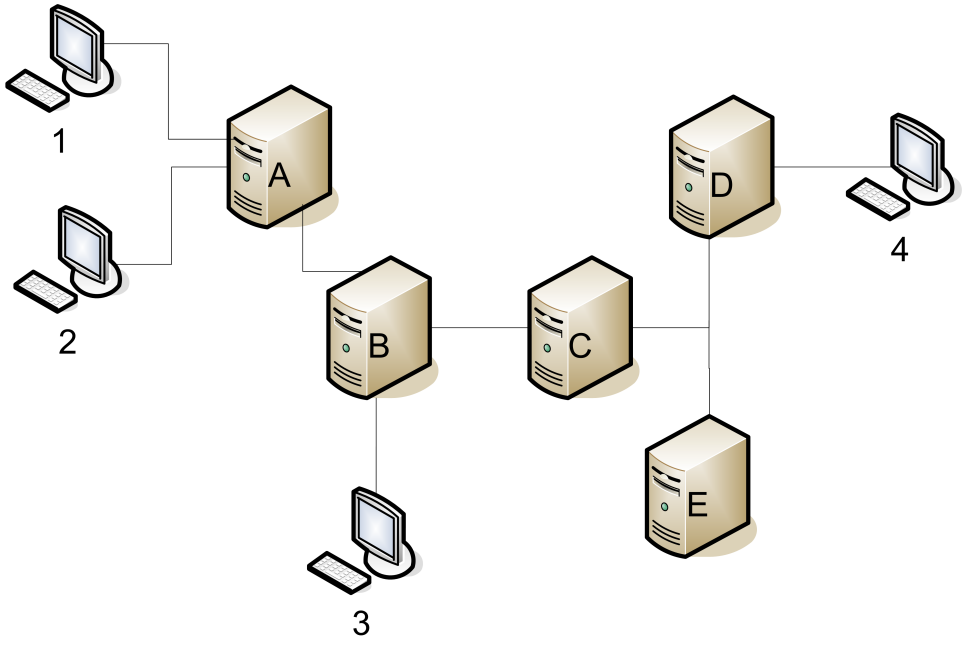

Figure 2.11: Example of a small IRC network, as presented in [OR93]. A, B, C, D and E are all servers, while 1, 2, 3 and 4 represent clients connected to the servers.

When clients are communicating with each other, the messages traverse the relevant servers. If for example clients 1 and 3 are communicating, the messages are sent through both server A and server B.

The clients have chosen unique nicknames as identifications, and may join or form their own channel. When connected to a channel, clients may send messages to each other, either publicly to all clients in the channel, or to specific clients privately. If, for example, clients 1, 2 and 3 are in the same channel, all messages to that channel are sent to all clients and the relevant servers.

In connection with botnets, the bots will sign on to one or more specific IRC channels which are maintained by the botmaster. The botmaster can now control the bots by sending commands through the channel.

See [OR93] for further elaboration of the IRC protocol.

CHAPTER 3
APPROACH

This chapter describes how the different implementations of the IDSs on the high-speed network is carried out. First, an introduction is given, followed by the experimental approach. The different implementations are described in Section 3.4 and Section 3.5. The results from these implementations are given in Chapter 4, and these results are again analysed and discussed in chapter 5.

3.1 INTRODUCTION

As mentioned in Chapter 2, installing an IDS on a high-speed network is not trivial. The main issue is the high data rate, which makes it hard to compare the packets and rules in real time. This probably sets a limit to the number of rules that simultaneously can be checked.

In addition, there are some challenges when it comes to choosing which rules to use. Because of the comparison problem, it is preferred to use as few rules as possible to find the desired traffic. That means that the rules should not only be able to identify the traffic, but also identify the traffic without generating too many false positives[1].

The following list is a point-by-point overview of the experimental approach described further in Section 3.3.

1. Set up a bot, an IRC server and botmaster on three different locations.
2. Set up an IDS; either as a MAPI implementation or by using Snort.
3. Run a mIRC script on the botmaster computer. The script generates malicious traffic between the botmaster and IRC server.
4. Detect the malicious traffic using the IDS.
5. Analyse and compare the detected traffic.

3.2 ARCHITECTURE

Figure 3.1 presents an overview of the experimental setup. The botmaster, bot, and IRC server are installed on computers located on three different places; the botmaster and bot will be located on one side of the IDS while the IRC server is located on the other side. This is done in

[1] According to [SBH+07], a false positive is an event that the system reported that appears bad and in point of fact is benign. [SBH+07] mentions, however, that a false negative is when the system reports that something is okay (or does not report anything) and in point of fact the event is bad

such a way that commands sent from the botmaster to the IRC server also will be detected by the IDS, in addition to the traffic between the IRC server and bot.

The communication between the different computers will mainly take place on the Uninett network, as previously mentioned in Section 2.7. Even though two of the computers are located on other networks, the traffic is routed through the sensor where the IDS is installed.

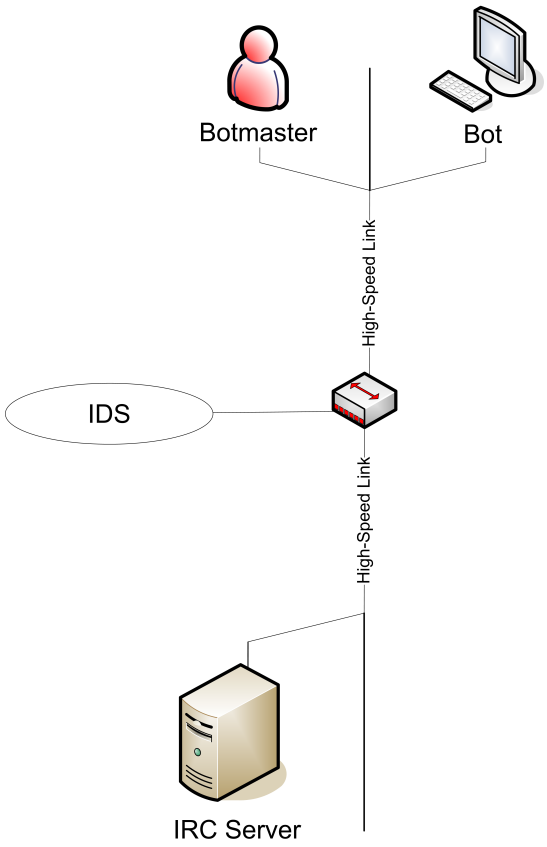

Figure 3.1: Overview of the experimental architecture.

3.2.1 Security Aspects

An important point when it comes to the architecture is to make sure that the security is attended to. As can be seen from Figure 3.1, there are basically two components which are especially vulnerable; the bot and the IRC server. The computer running the botmaster only utilizes well known programs, and is not more exposed than any other computer connected to the Internet. The IDS is only available through the SSH protocol and therefore not considered to be any threat to the experiments.

The computer running the bot is probably the most vulnerable part in the architecture. Bots can be harmful and if precautions are not made, one could end up giving away control over the computer to another botmaster. To prevent this from happening, the bot executables used

in the experiments were all compiled by the authors. This is further elaborated in Section 3.3.1. Because of this, it was guaranteed that the bots at least used a configuration file configured by the authors.

To deal with any unwanted behaviour, the bot programs were all run in a test environment where the traffic generated by the bots was monitored. First after the traffic pattern and behaviour was thoroughly checked, the bot was allowed to run on the Internet as described in Figure 3.1. In addition, only safe commands were tested. These commands are basically only commands which make the bots reply with their, e.g., version and uptime information. However, some scanner and vulnerabilities commands are included in the detection rules, which are further explained in 3.3.2.

The IRC server is the other vulnerable component besides the bot. The main reason for this, is that the bot is connected to the IRC server and a channel on the server. If someone were to compromise the computer and take control over the IRC server, they would in theory control the connected bot as well.

To prevent this from happening, some measures were taken. First of all, the IRC server was only running when the experiments were conducted, and not left on after the experiments were conducted. In addition, the server was configured to only allow traffic from the IP addresses belonging to the botmaster- and the bot computer. Last, the IRC server was protected with a server and a channel password.

More on the security for the different components, is to some extent described in the subsections below.

3.2.2 Botmaster

The botmaster is a standard Windows XP computer with an IRC client program installed. As mentioned, the botmaster is located at the opposite side of the IRC server according to the IDS.

The botmaster is only responsible for sending commands to the IRC server. This is done from mIRC, an IRC client program. As explained on section 3.3.3, these commands are sent from a mIRC script.

Since the tasks performed on the botmaster are so limited, there are minimal requirements to hardware setup.

3.2.3 Bot

The computer where the bots are executed, is a standard Windows XP installation. As with the botmaster, the bot computer is located at the opposite side of the IRC server according to the IDS in figure 3.1.

The bot executables require very little when it comes to system resources, so there are minimum requirements when it comes to the hardware setup. However, the test bots used in this article were designed for the Windows platform and therefore a Windows installation is re-

quired when choosing an operating system.

As mentioned in Section 2.9, bots can be very harmful and extra caution is required when working on them. In that connection, VMware was used to control the environment when running the different bots. This means that the bots were installed and run inside of a VM. More details on VMware are found in Section 2.6.1.

To get an overview over the traffic originating from the bot, Wireshark was used. As explained in Section 2.6.3, this program makes it possible to see every packet going to and from a computer. By installing Wireshark on the host OS outside the VM, a complete picture of the traffic pattern generated from the bot was acquired.

In addition to the traffic monitoring outside the VM, Process Exlorer from Sysinternals[2] was used to keep an eye on the behaviour of the bot executables inside the virtual machine. This made it easy to control the environment and to check if other processes were spawned from the bot executables.

The precautions and traffic sniffing, mentioned above, was mainly conducted to check the behaviour of the bot. After the initial testing was finished and no malicious traffic detected, a connection to the real IRC server (in Figure 3.1) was allowed and established.

3.2.4 IRC Server / C&C Server

It is important for the reader to understand that, in this connection, the IRC server and the Command & Control server are two names for the same thing. As previously mentioned in Section 2.9, the IRC Server is the place the bots connect to. Since the botmaster can control and command the bots from this server, the name C&C server is commonly used.

The computer hosting the IRC server is, similar to the botmaster- and the bot computer, also a plain Windows XP. Running an IRC server, that at least in the magnitude needed for the experiments in this report, requires no significant hardware worth mentioning.

In contrast to the botmaster and bot, the IRC server is located on a subnet on the opposite side of the IDS. This is due to fact that the IDS needs to be able to capture both traffic going to and from the server.

The IRC server software used throughout the experiments is a program called UnrealIRCd[3]. The configuration of this software is pretty much a default installation and are therefore not elaborated any further.

There are, however, a couple of modifications which have been made. Even though the server was completely controlled by the authors throughout the experiments, the security still had to be attended to. As mentioned in Section 3.2.1, only the IP addresses of the botmaster- and the bot computer were allowed to connect to the server. In addition, the UnrealIRCd was closed down between the experiments.

[2]Available at http://www.microsoft.com/technet/sysinternals/utilities/ProcessExplorer.mspx
[3]Available at http://www.unrealircd.com/

3.2.5 IDS

The IDS is probably the most important part of the architecture seen in Figure 3.1. Because of the high speed of the link where the IDS is installed, this computer requires fast hardware. See also Section 2.7.

The computer is a Linux Debian/sarge, running on a Pentium Xeon 3.2 GHz CPU with 3 GB memory and 1 GB swap. The PCI bus is a 64bit PCI-X bus. This component is responsible for delivering the packets from the DAG cards, see below, to the software where the processing is done.

As discussed in Section 2.8, there are two DAG cards installed on the IDS; one for each traffic direction on the link. This means that *dag0* is responsible for the traffic from A to B, while *dag1* is responsible for the traffic from B to A.

Figure 3.2 below, is a sketch of the computer running the IDS and illustrates how the packets arriving via the fiber tab is processed in the computer via the DAG card.

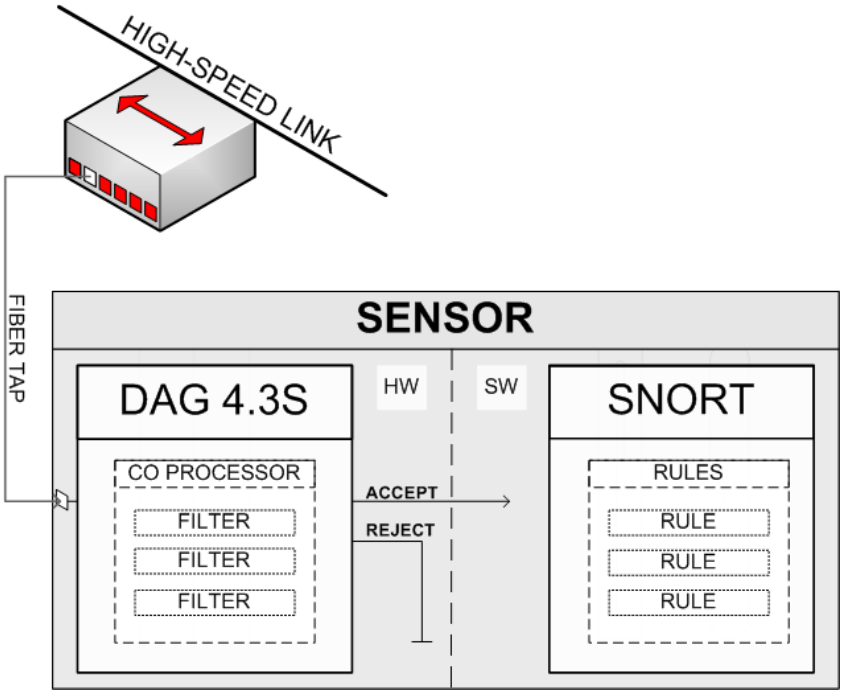

Figure 3.2: Uninett sensor

Both Snort and Mapi need to be able to pick up the packets from the bus for processing. Snort uses a library called libpcap[4] modified for the DAG card, while Mapi talks directly to the DAG card via the DAG API.

As described in Section 2.8.1, it is possible to load filters on the DAG card and co-processor. The DAG card processes the packets according to the filters and either rejects or accepts them, as figure 3.2 illustrates. Rejected packets are discarded and accepted packets are delivered to

[4]Available from http://sourceforge.net/projects/libpcap/

the software, in this case Snort, for further processing.

Hardware processing in the DAG card is much faster and less CPU intensive than processing done in software and Snort. However, the DAG filters are not as advanced as the Snort rules, making the filters less suited for, e.g., filtering on the packet payload.

3.2.6 Pros and Cons

One of the advantages with this architecture is that the experiments are conducted in a genuine environment. This means that the IDS is tested in a network with real traffic and real external users connected to it. The authors of this report have no controll over the traffic generated by these external users - making the experiments very realistic.

Another advantage, is that the security is attended to with this architecture. The securtiy aspects mentioned in Section 3.2.1 make the experiments as safe as possible - also when working on potentially "dangerous" bots.

By using only one bot and one C&C server at the time, the experiments are easily controlled. However, this also implies that the IDS will not be tested on a full-grown bot network.

Another disadvantage is that there are potentially many sources of error which may affect the experiments. These sources are further discussed in Section 5.3.

3.3 Implementation Preparations

Before the experiments can begin, some preparations need to be done. These preparations are discussed in the following sections.

3.3.1 Compiling the Bots

Since the experiments in this report required the use of bots, these had to be downloaded. Precompiled bots can be programs designed to do a lot of damage, and they are not in any way safe to install. By compiling the bots from source, however, it is much easier to control their behaviour. In addition, all events may, in principle, be verified by static analysis of the source code. To ensure the safety in the experiments throughout this report, only self-compiled bots were used.

The source code to the tested bots were downloaded from two internet sites, namely *darksun.ws* and *ryan1918.org*. These sites contained source code to a lot of different bots and bot versions. In addition to the bots mentioned in Section 2.9.4, several other bots were compiled; *Phatbot-st0ney-Fixed*, *spybot1.4*, *agobot3-0.2.1-pre4-priv* and *pBot_v2* amongst others. However, because of different runtime- and connection problems these were not used in the final experiments.

Most of the downloaded source code was made with Visual Studio 6. However, the bots used in this report were all compiled on Vistual Studio .NET 2003 without any problems worth mentioning.

Figure 3.3 on the following page, shows the RBot configuration file in the *rx-asn-2-re-worked v3* version. The file displayed is pretty much the same as the default configuration shipped with the source code, with some modifications. The values one line 26 to 45 are more or less changed to reflect on the set up used in the experiments, e.g., the IP address to the IRC server has been changed. Also, all the password fields have been set for maximum safety.

In addition to the bot compiled from *rx-asn-2-re-worked v3*, bots from source code *Dbot.v3.1* and *sdbot05b[skbot]_mods_by_sketch* were compiled and tested. The original source codes, in addition to the modified source code used for building the bots, are included as a digital attachment to this report.

RBot configuration file

```
// bot configuration (generic) - doesn't need to be encrypted
int port = 6667;                                    // server port
int port2 = 6667;                                   // backup server port
int socks4port = 2005;                              // Port # for sock4 daemon to run on  - CHANGE THIS!!!
int tftpport = 69;                                  // Port # for tftp daemon to run on
int httpport = 5678;                                // Port # for http daemon to run on
int rloginport = 514;                               // Port # for rlogin daemon to run on
BOOL topiccmd = TRUE;                               // set to TRUE to enable topic commands
BOOL rndfilename = FALSE;                           // use random file name
BOOL AutoStart = FALSE;                             // enable autostart registry keys
char prefix = '.';                                  // command prefix (one character max.)
int maxrand = 2;                                    // how many random numbers in the nick
int nicktype = CONSTNICK;                           // nick type (see rndnick.h)
BOOL nickprefix = FALSE;                            // nick uptime & mirc prefix


#ifdef DEBUG_LOGGING
char logfile[]="%temp%\\bot.log";
#endif

#ifndef NO_CRYPT // Only use encrypted strings or your binary will not be secure!!

#else  // Recommended to use this only for Crypt() setup, this is unsecure.

char botid[] = "rx-asn-2-re-worked by IDS ";        // bot id
char version[] = "Version 3 Mod by IDS";            // Bots !version reply
char password[] = "IDSbotS";                        // bot password
char server[] = "IRC-SERVER";                       // server
char serverpass[] = "IDSbotS";                      // server password
char channel[] = "#IDSbot";                         // channel that the bot should join
char chanpass[] = "IDSbotS";                        // channel password
char server2[] = "";                                // backup server (optional)
char channel2[] = "";                               // backup channel (optional)
char chanpass2[] = "";                              // backup channel password (optional)
char filename[] = "IDSbot.exe";                     // destination file name
char keylogfile[] = "keylog";                       // keylog filename
char valuename[] = "IDSbot";                        // value name for autostart
char nickconst[] = "IDSbot";                        // first part to the bot's nick
char szLocalPayloadFile[]="IDSbot.exe";             // Payload filename
char modeonconn[] = "-xi+B";                        // Can be more than one mode and contain both + and -
char exploitchan[] = "#IDSbot";                     // Channel where exploit messages get redirected
char keylogchan[] = "#IDSbot";                      // Channel where keylog messages get redirected
char psniffchan[] = "#IDSbot";                      // Channel where psniff messages get redirected

char *authost[] = {
    "*@*.*"
};

char *versionlist[] = {
    "mIRC v6.12 Khaled Mardam-Bey",
};

char regkey1[]="Software\\Microsoft\\Windows\\CurrentVersion\\Run";
char regkey2[]="Software\\Microsoft\\Windows\\CurrentVersion\\RunServices";
char regkey3[]="Software\\Microsoft\\OLE";
char regkey4[]="SYSTEM\\CurrentControlSet\\Control\\Lsa";

#endif

#ifdef PLAIN_CRYPT
char key[16] = "9jah3msnso23kam2";                   // CHANGE THIS!!! hmmm..Do I even need this now?
#endif
```

Figure 3.3: Rbot configuration file

3.3.2 IDS Rules

The IDS rules are very important when it comes to detecting unwanted traffic. Since this thesis centers around botnets, the test rules used throughout the experiments are to some degree bot focused. However, a part of the testing is done with respect to measuring the performance capabilities of Snort and MAPI and therefore requires different rules.

Control commands sent from the botmaster to the bots are sent in clear text and thus easily readable. This is however not the case if the traffic is encrypted. Encrypted traffic is not relevant in this report, and the experiments further on will only involve unencrypted packets.

There are many ways to make rules. One could focus on everything from the IP address itself to, e.g., the source or destination port number, IP address/subnet, content matching or TCP flags.

The rules used throughout the experiments, are all content matching rules. This means that the content of the IP packet payload is matched with the given content specified in each rule. The content can be binary code as well as normal ASCII strings. In this report, only rules with bot commands as ASCII strings are used.

Figure 3.4 shows an example of a Snort rule used for testing. Note that a properly formatted rule should be written on one line.

An example of a content matching Snort rule

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any
      (msg:"Bot Test"; content:".getcdkey"; classtype: bot-test; sid:9000001;)
```

Figure 3.4: An example of a content matching Snort rule

In addition to the content matching rule in figure 3.4, it is important to explain the first part of the rule, namely; alert tcp $HOME_NET any -> $EXTERNAL_NET any. *Alert* means that the rule will cause an ALERT action in Snort. *Tcp* means that Snort should only continue to process the rule if it infact is a tcp packet.

The variables *$HOME_NET* and *$EXTERNAL_NET* are set to value *any* in the Snort configuration, meaning that the rule is processed as *alert tcp any any -> any any*. This instructs Snort to match the rule with **all** tcp packets, regardless of what their source or destination port or address might be.

Using *any any* instead of specific ports or addresses is a deliberate choice by the authors. One of the main points with this report is to discover botnet activity, regardless of which protocol that is utilized for the transfer of bot commands. Even though a big percentage of the commands are sent via the IRC protocol, botmasters often tend to use IRC servers communicating on other ports than the standard 6667 IRC server port. Using *any any* makes sure that these packets being sent over non-standard ports are analysed as well.

As is mentioned above, a part of the experiments are conducted to measure the performance in Snort and MAPI. Since it was preferred to keep the false positive alerts to a minimum, rules with random content were generated. The point was therefore not to get many alerts, but to test the software with many rules.

See Section 2.5.1 for how these rules can be implemented in MAPI.

3.3.3 Malicious Traffic Script

To test the rules mentioned in Section 3.3.2, it was necessary to send a number of bot commands from the botmaster. The IRC client program, mIRC, was used in this connection. By using mIRC's built in timer and script support, it was possible to make a simple script which could send commands repeatedly.

Figure 3.5 shows how the timer command was run from mIRC, and what the script sending the bot command looked like.

mIRC bot command script

```
# Starting the timer from the mIRC command line
/timer1 60 1 /.bot

# Script sending a private message to the channel #IDSBot with a bot command
/bot {
  /privmsg #IDSBot .getcdkey
}
```

Figure 3.5: mIRC bot command script

In the figure above, mIRC repeats the bot-command every second, 60 times. The timer also supports milliseconds by setting a `-m` switch, which is a nice feature when many packets are to be tested against the IDS. As can be seen in the figure, mIRC sends a private message to the channel *IDSBot*. This is the channel that the bots are connect to, as mentioned in Section 3.3.1. The IRC server receives the private message and then forwards it to the bots connected.

3.4 IMPLEMENTING SNORT

This section explains how Snort was configured, implemented and tested in the high-speed backbone network. In addition, the procedure for how the different experiments were conducted is elaborated here. Notice that the configuration files used throughout the experiments, are included as digital attachments to this report.

A Snort installation is controlled by the configuration file, snort.conf. This report started out with a default configuration file, namely snort.conf shipped with snort-2.6.1.5.tar.gz[5].

As mentioned, there are a lot of options which need to be set. After some reading in [Pro06], some modifications to the original snort.conf was made. The most important modifications

[5] Available from snort.org

are listed in Figure 3.6, while the complete, modified Snort configuration file, snort.noflow.conf, is available in the appendix.

Snort.noflow.conf

```
1
2  # Where to store the unifed log files,
3  config logdir : /home/ids/snort/logs
4
5  # Alerts to file snort.alert. max 128Mb in size
6  output alert_unified: filename snort.alert, limit 128
7
8  # Logs to file snort.log. max 128Mb in size
9  output log_unified: filename snort.log, limit 128
10
11
12 # Which interface to listen on
13 config interface: dag0
14
15
16 # Flow module - not used in the experiments
17 # preprocessor flow: stats_interval 0 hash 2
18
19
20 # Different alert which had to be disabled
21 config disable_decode_alerts
22 config disable_tcpopt_experimental_alerts
23 config disable_tcpopt_obsolete_alerts
24 config disable_tcpopt_ttcp_alerts
25 config disable_ttcp_alerts
26 config disable_tcpopt_alerts
27 config disable_ipopt_alerts
28
29
30 # IP defragmentation module. See snort.noflow.conf in appendix for full documentation
31 preprocessor frag3_global: max_frags 65536
32 preprocessor frag3_engine: policy linux
33
34
35 # IP packet reassembly module. See snort.noflow.conf in appendix for full documentation
36 preprocessor stream4: disable_evasion_alerts, max_sessions 50000
37 preprocessor stream4_reassemble: both, ports all, noalerts
38
39
40 # Where the rules are located
41 var RULE_PATH /home/ids/snort/rules
42
43 #include $RULE_PATH/noflow-5.rules
44 #include $RULE_PATH/noflow-30.rules
45 #include $RULE_PATH/noflow-50.rules
46 #include $RULE_PATH/noflow-100.rules
47 include $RULE_PATH/noflow-200.rules
48 #include $RULE_PATH/noflow-1000.rules
```

Figure 3.6: Snort.noflow.conf

First of all the configuration file was changed to make Snort disregard whether a flow was established, hence the name snort.noflow.conf. When running Snort with flows enabled, the authors discovered that packets in an already established flow were not checked. Since this was not the wanted functionality (the IDS is, as mentioned, supposed to inspect all packet) the flow module was commented out in line 17-18.

Second, it was desirable to reassemble packets on, again, all ports and to/from all addresses. Thus, line 37: *both* and *ports all*, where *both* represents traffic to all addresses and *ports all* means to check all ports.

Third, some strange false alerts had to be eliminated. This was done by disabeling the various alerts on line 21-27. In addition, *stream4* was configured to not alert on different events such

as TCP overlap (*disable_evasion_alerts*, line 36) and insertion or evasion attacks (*noalerts*, line 37).

The following sections present the experiements which were conducted in this report. The results from these experiments are given in Chapter 4, and the results are further analysed in Chapter 5.

Note that one planned experiment had to be omitted, and is thus removed from the report. This was experiment 8: *Testing DAG hardware filtering*. It turned out that that the DAG firmware supporting filtering, did not work as expected and therefore had to be replaced with another firmware version. Unfortunately, this latter firmware did not support filtering.

3.4.1 Experiment 1 - Testing The Number of Rules

The purpose of this experiment is to test Snort's detection ability when different number of rules are used. To check whether Snort alerts on these rules, malicious packets generated by the script in Section 3.3.3 are sent. 50, 100, 250, 500, 750 and 1000 packets will be sent during the tests. Statistics, including the number of detections, and the number of processed and dropped packets in Snort will be presented.

The number of rules to be tested are 5, 30, 50, 100, 200 and 1000. The rules are all content based, as previously mentioned in Section 3.3.2, using a string search algorithm. To avoid false positives during testing, the rules have different content and there is only one rule matching the malicious packet being sent. However, all alerts are always checked for false positives or other correct detections.

In experiment 1, Snort is run with the modified configuration file mentioned in section 3.3, namely snort.noflow.conf. The results of this experiment are given in tables in Section 4.2.1. To make the experiments as accurate as possible, three trials are conducted for each number of malicious packets being sent. The results given in section 4.2.1 are calculated averages of these three trials.

3.4.2 Experiment 2 - Optimising Snort Configuration

The tests conducted in experiment 1, gave a good impression of Snort's performance when different number of rules were used. Due to the results, listed in section 4.2.1, it was however likely that improvements could be made to the configuration which again would boost the general performance. After further study of [Pro06], a few options in the module *stream4* stood out.

Figure 3.7 lists the most important part of the file snort.tweak.conf. Snort.tweak.conf is used in this experiment, and is basically a modified version of snort.noflow.conf.

The three most important options from the figure are *timeout*, *server_inspect_limit* and *noinspect*. In short, these three options control, respectively; how long time inactive streams should be kept in the state table, how many bytes of each packet Snort should inspect, and whether Snort should perform stateful inspection. The impact these modifications had on the performance

Snort.tweak.conf

```
1 preprocessor stream4: disable_evasion_alerts, max_sessions 10000, noinspect,
2                       timeout 1, server_inspect_limit 1540, memcap 1048576
```

Figure 3.7: Snort.tweak.conf

is further elaborated in Section 5.2.2.

Experiment 2 is conducted in the same manner as experiment 1, namely by sending malicious commands generated by the botmaster's mIRC script a given number of times. However, the rules tested in this experiment are limited to 200 and 1000. The reason to why only these rulesets were chosen, is explained in Section 5.2.2.

3.4.3 Experiment 3 - Testing Impact of Common Strings

The purpose of this experiments is so check Snort's sensitivity to excessive detections. To achieve this, rules for detecting the strings *GET*, *POST* and *password* were created, in addition to the *.getcdkey* rule used in experiment 1 and 2. The total number of rules used in this test is 200.

Even though the unified log setting specified in the configuration file is among the fastest offered in Snort, logging is still a performance issue. In order to rule out logging as a possible source of error to the experiment, the log setting was disabled. The results will be compared to another 200-ruleset, also with logging disabled. This ruleset does however not include the common words.

3.4.4 Experiment 4 - Testing With Minimum External Traffic

This small experiment is simply to test Snort's detection ratio in a controlled environment, with a minimum amount of external traffic.

In experiment 4, Snort is installed on the same machine as the bot computer. Therefore only a minimum of packets traversing the local area network will be processed.

This experiment will use the 5-ruleset and the modified configuration file from experiment 1. 100 bot commands will be sent via the mIRC script.

3.4.5 Experiment 5 - Testing Packet Length Input

As Snort reads all incoming packet headers and payloads, it should be possible to limit how much of the packets Snort will process. In theory, this should increase the processing and possibly make Snort faster.

Slen is a modifiable parameter on the DAG card, which is set via one of the DAG utilities. This parameter basically controls how many bytes of a packet which is transmitted to the

software. DAG's default slen value is 1540.

This experiment will test slen values from 1540 and down to 200. In addition, the packet drop ratio for each slen value will be recorded. The test will be conducted in two rounds, using both the modified configuration file and the optimised one.

Notice that the IDS looses some of its impact when reducing the slen value, as only a part of the packet is processed by Snort. However, it is reasonable to assume that bot commands will not fill an entire packet payload of 1540 bytes.

3.4.6 Experiment 6 - Testing Impact of the TCP Protocol

This experiment is more an observation than an experiment. The idea here is to get a better understanding of how Snort works and reacts in a high-speed environment. As most packets are TCP packets, this experiment analyses the effect these packets has on Snort. In this connection, especially the ratio of packets dropped by Snort will be studied.

To check whether this observation actually is correct, Snort will be run in different time intervals at random times. No bot commands will be sent in this experiment, but the 200-ruleset is still included for performance monitoring. The ratio of TCP packets recorded will then be compared to the ratio of packets dropped by Snort.

3.4.7 Experiment 7 - IDS Stand-Alone Test

This final experiment is conducted in order to observe in which degree Snort handles a high-speed environment over a given period of time. Therefore Snort will run with a small set of botnet rules, which identify actual RBot commands.

These commands are:

.capture screen
.capture frame
.capture video
.getcdkeys
.getclip
.keylog
.secure
.ddos.stop
.ddos.syn
.ddos.ack
.ddos.random
.icmpflood
.pingflood
.pingstop
.synflood
.synstop
.tcpflood

.udpflood
.udpstop

The measurement data in this experiment is the number of dropped and processed packets, the time spent and the number of detections made. The detections, should there be any, will be checked with Uninett's list of known botnets.

Another aspect of this experiment is to see whether these rules will provide false positives.

3.5 IMPLEMENTING MAPI

Unfortunately, as will be explained in the last part of this section, the testing of MAPI had to be omitted. However; the configuration, implementation and methods for testing will still be examined because of its relevance for this report. Notice that a MAPI test program, is included as a digital attachment to this report.

MAPI is quite different than Snort when it comes to how the program is run. While Snort is controlled via a configuration file where every option can be tweaked, MAPI is basically stand-alone programs. These programs communicate with a MAPI daemon running in the background, as elaborated in section 2.5. Even though this daemon has a small configuration file, the MAPI programs do not and are individually customized to solve the tasks they are made for.

Another big difference is how the rules works. While Snort has rule files with one rule per line, MAPI programs have to make flows and apply a string search function on the flow, as mentioned in Section 2.5. As of today, this string search function does only support AND, and not OR. This implicates that if multiple string search functions are added to one flow, they will behave as *str_search1 AND str_search2 AND ...* To get OR functionality, one has to make as many flows as there are rules and apply one string search function to each rule.

A five rule MAPI program is included in Appendix B. As can be seen from the code, five different flows are connected to the network interface dag0. In contrast to the code in Figure 2.3, line 2, the five rule program does not add BPF-filters to the flow. It is purely a content matching program, meaning that it only adds string search functions on the five flows.

The MAPI programs work on a per-packet basis in default mode. This means that it will only look at one packet at the time. This is often not the wanted behaviour on an IDS; if a packet is split up into several small pieces, the IDS will fail to alert on the activity because the commands or malicious code are split over several packets.

MAPI provides a function called *cooking* which offers both packet defragmentation and stream reassembly. Figure 3.8 shows how this functionality can be applied to a flow.

Mapi cooking function

```
1 mapi_apply_function(fd, "COOKING", -1, -1, 1, BOTH_SIDE)
```

Figure 3.8: Mapi cooking function

In short terms, Figure 3.8 adds the *cooking* function to the flow *fd*. The parameters given with the function are defaults, except for *BOTH_SIDES* which "cooks" both client and server side data. More on *cooking* is found in [MAP06] appendix C.

As mentioned in the start of this section, the testing of MAPI had to be omitted. The main reason for this was unstabilities in the MAPI implementation. The first test failed on the packet counter function and to some extend the string search function. To get around this problem, a new version of MAPI was installed. The new version worked great until another memory related bug terminated the test programs.

The mentioned bugs could unfortunately not be corrected in a reasonable time frame, meaning that the testing was cancelled. However, MAPI is still discussed in Section 6 and also suggested as further work in chapter 7.

CHAPTER 4
RESULTS

This chapter presents the results, given in raw data, from the experiments conducted in Section 3.4. The results are further calculated and analysed in Chapter 5.

4.1 INTRODUCTION

The results throughout the Chapter are presented in tables for easy reading. The measurement data used in these tables are:

- **# sent packets**
  the number of malicious packets (i.e., botnet commands) that was sent.
- **# detected packets**
  the number of packets that was detected by Snort.
- **# processed packets**
  the total number of packets that was analysed by Snort.
- **# dropped packets**
  the number of packets that was dropped by Snort.
- **# Time (s)**
  the time spent, given in seconds.
- **Slen**
  the value of *slen*, used in experiment 5.
- **Drop ratio**
  the ratio of dropped packets in experiment 5.
- **TCP ratio in %**
  the ratio of TCP packets processed by Snort, used in experiment 6.
- **Packets dropped in %**
  The packets dropped by Snort in experiment 6, given in percent.

Note that not all of these items will be included in every table due to the different types of experiments.

4.2 RESULTS FROM THE EXPERIMENTS

The results are printed in their respective sections, as they were presented in sections 3.4.1 - 3.4.7.

4.2.1 Experiment 1 - Testing Number of Rules

Table 4.1 presents the data gathered when using 5 rules.

# sent packets	# detected packets	# processed packets	# dropped packets	Time (s)
50	27,3	3945998	1071600	55,7
100	53,3	7609318	2337236	103,7
250	128,3	18540047	6084694	254,7
500	213,7	10185766	3139288	146,3
750	338,3	12684324	3792193	192,3
1000	458,0	14573446	4752877	141,0

Table 4.1: Results when using 5 rules

Table 4.2 presents the data gathered when using 30 rules.

# sent packets	# detected packets	# processed packets	# dropped packets	Time (s)
50	30,0	8558052	2255167	118,3
100	57,7	3927710	965597	38,0
250	135,0	10296377	3005477	149,0
500	281,3	10291044	3014016	143,3
750	448,3	14633768	4462452	198,7
1000	544,7	16227945	4712382	225,0

Table 4.2: Results when using 30 rules

Table 4.3 presents the data gathered when using 50 rules.

# sent packets	# detected packets	# processed packets	# dropped packets	Time (s)
50	35,3	4722290	1289989	69,3
100	70,3	3537676	982207	54,0
250	152,7	8227898	2849350	131,7
500	313,0	15820640	5362863	256,3
750	483,3	15121603	5082254	230,7
1000	563,0	13524141	4694245	211,3

Table 4.3: Results when using 50 rules

Table 4.4 presents the data gathered when using 100 rules.

# sent packets	# detected packets	# processed packets	# dropped packets	Time (s)
50	28,7	3757943	1079614	58,0
100	58,7	7589512	2435632	110,3
250	139,0	9612739	3330650	139,3
500	300,3	9357781	3174200	134,0
750	461,0	14412979	4516658	201,7
1000	577,0	15604236	4876274	215,7

Table 4.4: Results when using 100 rules

Table 4.5 presents the data gathered when using 200 rules.

# sent packets	# detected packets	# processed packets	# dropped packets	Time (s)
50	30,7	4344739	1214148	59,7
100	60,7	8277882	2149331	116,0
250	149,0	9827296	2636264	136,7
500	294,0	9074404	3097138	129,7
750	459,7	15206487	4732788	206,3
1000	572,7	16493553	5666268	220,0

Table 4.5: Results when using 200 rules

Table 4.6 presents the data gathered when using 1000 rules.

# sent packets	# detected packets	# processed packets	# dropped packets	Time (s)
50	15,7	3193019	1463615	60,0
100	38,7	7149154	3223847	117,7
250	104,0	9422250	4390845	144,3
500	206,3	8584910	4063170	132,7
750	293,7	12876115	6007375	198,0
1000	358,7	15680484	7490446	355,7

Table 4.6: Results when using 1000 rules

4.2.2 Experiment 2 - Optimising Snort Configuration

This section presents the results from running the experiment with the optimised version of the Snort configuration file. The test is performed using the same 200 and 1000 rules as in experiment 1.

Table 4.7 presents the data gathered when using 200 rules.

# sent packets	# detected packets	# processed packets	# dropped packets	Time (s)
50	41,0	9487701	1947048	75,3
100	79,3	6789107	1558987	55,0
250	198,0	8725457	2045654	70,3
500	399,0	9074404	4076847	135,0
750	598,7	23508501	5832509	193,0
1000	794,3	26710527	6510127	219,3

Table 4.7: Results when using 200 rules with optimised configuration file

Table 4.8 presents the data gathered when using 1000 rules.

# sent packets	# detected packets	# processed packets	# dropped packets	Time (s)
50	35,3	5912337	2253011	57,7
100	69,0	11245336	4033848	108,0
250	176,7	14964240	5245812	146,7
500	339,0	14317250	6273053	140,7
750	520,0	21815652	9169182	213,0
1000	682,3	21909508	9317177	215,0

Table 4.8: Results when using 1000 rules with optimised configuration file

4.2.3 Experiment 3 - Testing Impact of Common Strings

This section presents the results from experiment 3. As can be seen from Table 4.9, there was a great number of detections of the common words that were included in the rules.

Table 4.9 presents the data gathered when using in the ruleset.

# detected packets	# processed packets	# dropped packets	Time (s)
18302,3	3612454	1168767	34,0
45839,0	8320440	2127811	48,7
64251,7	12583371	4103625	112,7
94895,7	18954585	6662638	170,3
135563,3	28100965	9280670	249,0

Table 4.9: Results when common words are inlcuded in the ruleset

4.2.4 Experiment 4 - Testing With Minimum External Traffic

This section presents the results from experiment 4. As seen in table 4.10, Snort can easily handle detections when it is not running in a high-speed environment.

Table 4.10 presents the data gathered when using 5 rules and a minimum of external traffic.

# sent packets	# detected packets	# processed packets	# dropped packets	Time (s)
100	100,0	4046	0	101,0

Table 4.10: Results when using 5 rules and a minimum of external traffic

4.2.5 Experiment 5 - Testing Packet Length Input

This section presents the results from experiment 5. As can be seen from both tables, the packet drop ratio decreases drastically when slen is close to 1500. Testing was conducted on slen values down to 200. These are however excluded as the values starting with 1492 and lower all had a drop ratio of 0.0.

Table 4.11 presents the data gathered when using different *slen* values with the modified configuartion file.

Slen	Drop ratio
1536	0,254115
1524	0,282133
1508	0,415595
1504	0,365214
1500	0,136623
1496	0,057712
1492	0
1488	0
1480	0
1472	0

Table 4.11: The effect of the slen value when using the modified configuartion file

Table 4.12 presents the data gathered when using different *slen* values with the optimised configuartion file.

Slen	Drop ratio
1536	0,221902
1524	0,235542
1508	0,237085
1504	0,159672
1500	0,000396
1496	0
1492	0
1488	0
1480	0
1472	0

Table 4.12: The effect of the slen value when using the optimised configuartion file

4.2.6 Experiment 6 - Testing Impact of the TCP Protocol

This section presents the results from experiment 6. The results are presented in the order they were gathered.

Table 4.13 presents the data gathered when testing the impact of the TCP protocol on Snort.

TCP ratio in %	Packets dropped in %
66,908	20,644
68,857	34,163
69,833	31,195
69,304	32,461
67,852	29,151
68,764	40,117
69,019	35,883
69,656	33,041
69,122	35,899
70,057	30,518
61,396	0,000
60,973	0,000
60,824	0,000
61,902	0,000
61,784	0,000
62,904	0,000
62,688	0,000
62,515	0,000
62,246	0,000
61,705	0,000
63,447	0,390
62,84	0,005
64,487	0,000
64,673	0,000
64,65	0,000
63,006	0,144
60,272	0,000
60,825	0,000
60,859	0,000
62,574	0,676
62,465	2,004
65,542	15,960
65,973	20,484

Table 4.13: The impact the TCP protocol has on the ratio of dropped packets

4.2.7 Experiment 7 - IDS Stand-Alone Test

This section presents the results from experiment 7.

Table 4.14 resents the data gathered when running the IDS at night, without injecting any malicious packets.

# detected packets	# processed packets	# dropped packets	Time
1	3,95E+09	0	10h 32m 45s
2	3,89E+09	1041118	10h 41m 14s
1	2,54E+09	21	12h 35m 1s
8	3,91E+09	24420344	11h 4m 46s

Table 4.14: The IDS stand-alone test

CHAPTER 5
ANALYSIS

This chapter presents an analysis of the results found in chapter 4. Section 5.3 discusses some possible sources of error before the conclusion in Section 5.4 sums up the experiments.

5.1 INTRODUCTION

The following sections analyses the results in the same order as they were presented in sections 4.2.1 - 4.2.7.

5.2 ANALYSING THE EXPERIMENTS

All figures in the following sections are based on calculations of the data presented in the tables given in the different sections in Chapter 4.

5.2.1 Experiment 1 - Testing Number of Rules

Figure 5.1 shows the detection ratio for each ruleset test conducted in experiment 1. As can be seen in the figure, most of the tests have a detection ratio close to 0.60. However, when using 5 and 1000 rules, the detection ratio is somewhat low.

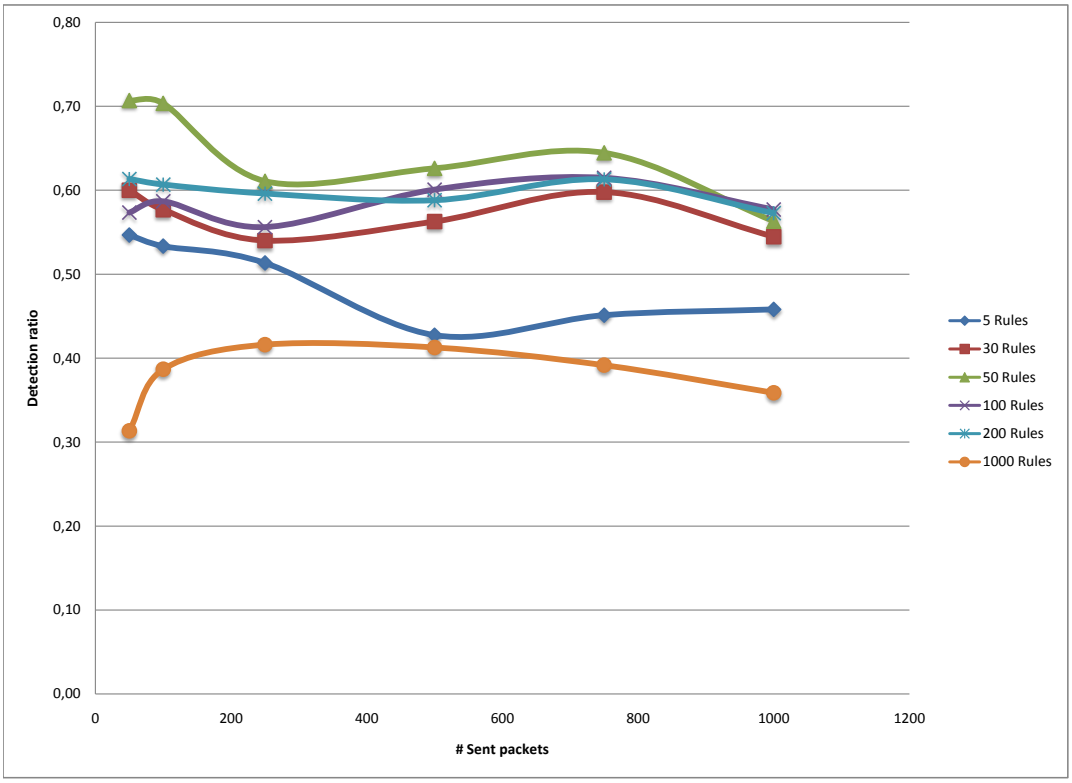

Figure 5.1: The detection ratio for each of the ruleset tests in experiment 1.

The low detection ratio when using 5 rules is probably due to statistical deviations as Snort does not drop particular more packets than the other rule-tests, as can be seen in Figure 5.2. From Figure 5.2, increasing the number of rules from 5 to 200 does not seem to have a negative effect on dropped packet ratio. However figure 5.2 shows that if 1000 rules are used, the dropped packet ratio increases dramatically.

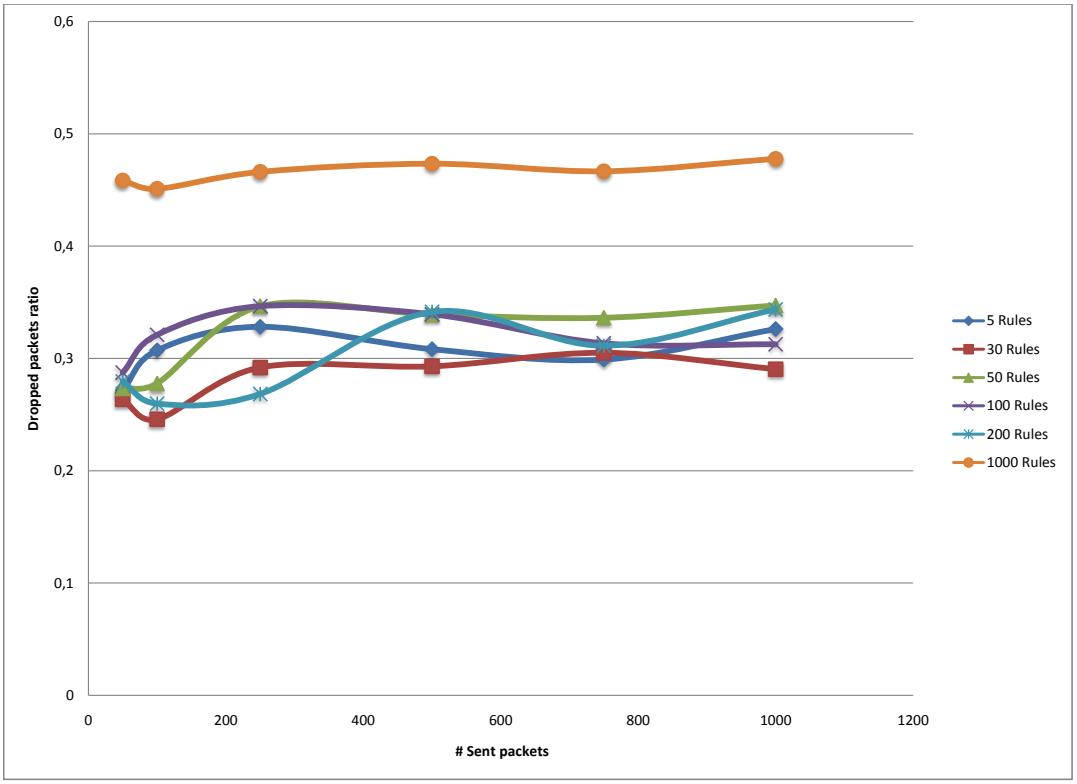

Figure 5.2: The ratio of packets dropped by Snort for each of the ruleset tests.

Figure 5.3 shows the number of processed packets per second during the experiment. As the figure shows, the number is mainly between 60.000 and 75.000 packets per second. This figure is simply included to show that it is not variations in the traffic load that causes the results given here. In fact, the deviations in the 5- and 1000-ruleset test in Figure 5.3 does not have noticeable effect when compared to the detection ratio in Figure 5.1.

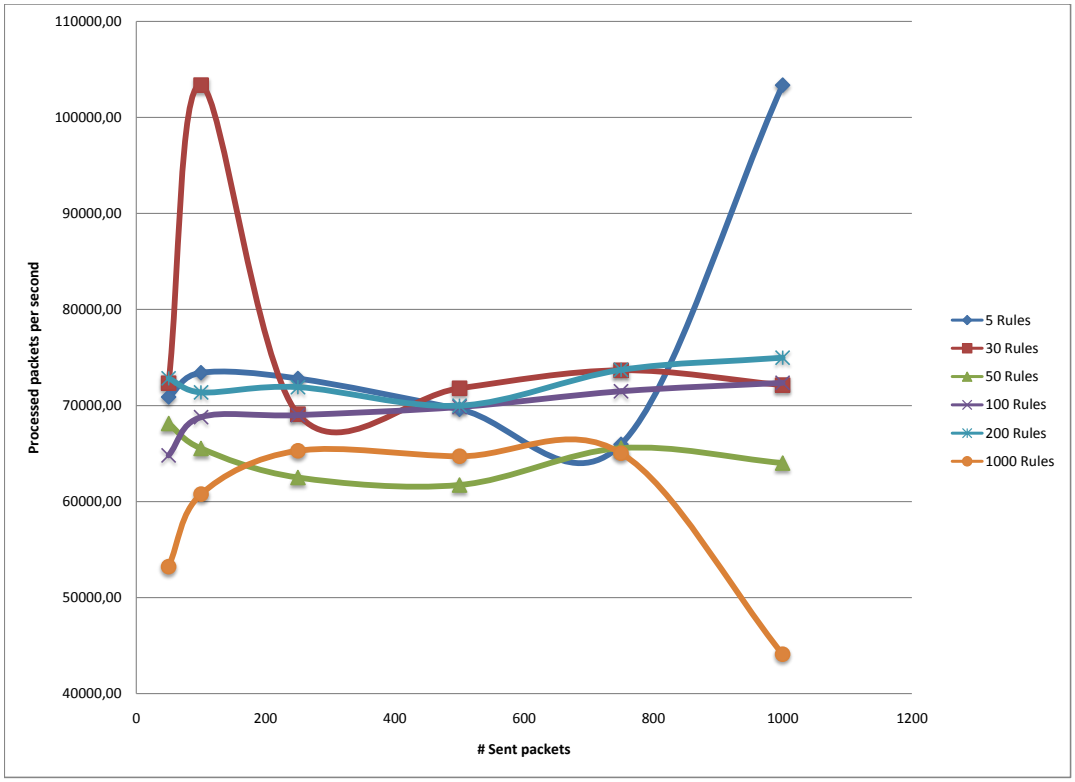

Figure 5.3: Processed packets per second divided on each of the ruleset tests.

From Figure 5.1 and 5.2, it is shown that Snort is still far from working optimally in a high-speed network. The detection ratio is low, and the drop ratio is high.

This experiment reveals that the number of rules has an impact on the processing done by Snort. The reason for this is probably that more rules increases the total number of comparisons needed per incoming packet. A valid question is then how many rules an IDS needs for satisfying botnet detection. Whatever the answer is, experiment 2 shows that it is possible to have 1000 rules, and even get a higher detection rate that with the other rulesets in this experiment.

5.2.2 Experiment 2 - Optimising Snort Configuration

Figure 5.4 shows the packet detection ratio of using the optimised Snort configuration file. As shown, it has a clear positive effect when compared to the respective ruleset-tests in experiment 1 which used the modified configuration file.

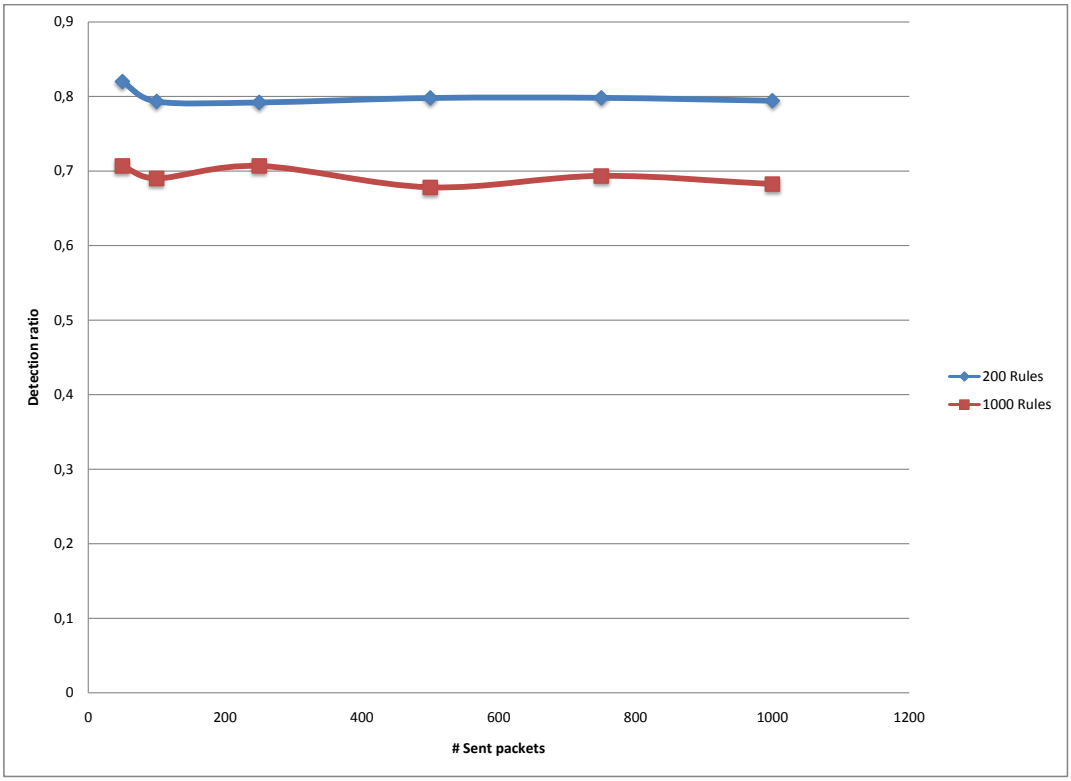

Figure 5.4: The detection ratio for using 200 and 1000 rules using the optimised Snort configuration file.

Because there was not much difference between using 5 and 200 rules in experiment 1, this experiment only used the 200 and 1000 ruleset. As can be seen from Figure 5.4, the detection ratio of the test using 200 rules has increased from about 0.60 in experiment 1 to about approximately 0.80 in experiment 2. The positive effect is even greater with the 1000-ruleset test. As can be seen, the detection ratio has increased from approximately 0.40 in experiment 1 to 0.70 in experiment 2. Notice that the 1000-ruleset now actually has a higher detection ratio than the highest ratio presented in experiment 1.

The effect is also shown in Figure 5.5, where Snort now has a lower packet drop ratio compared to the ratio in Figure 5.2 from experiment 1. However, the drop ratio is still relatively high.

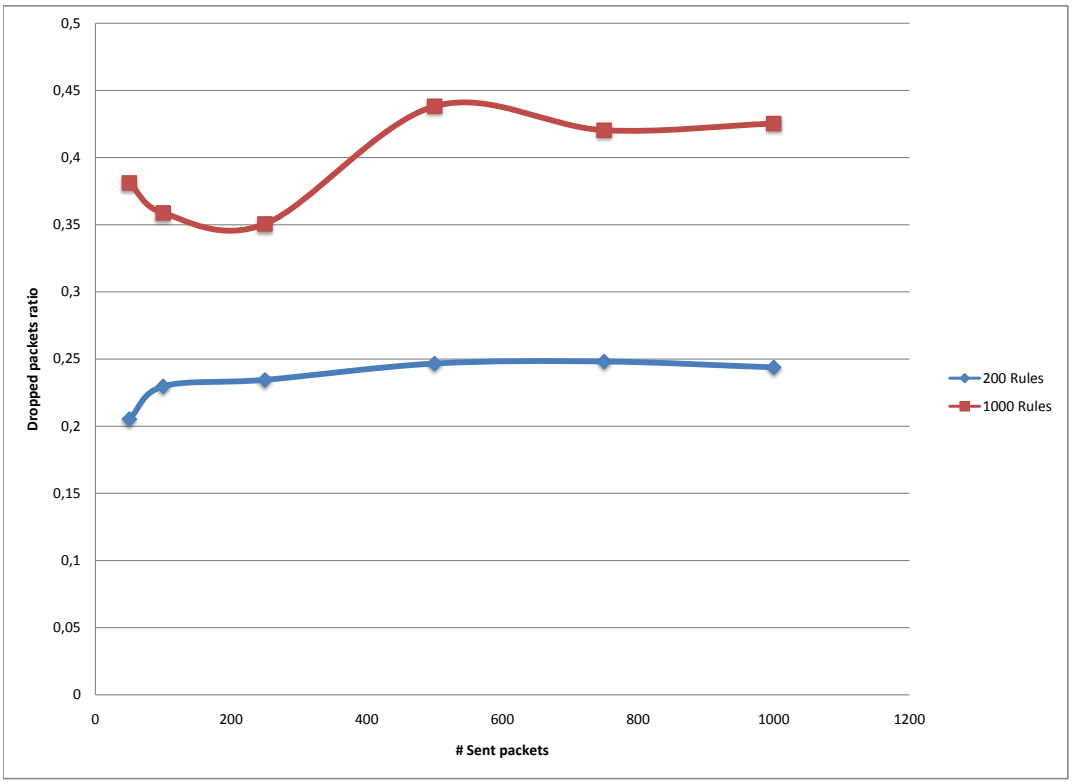

Figure 5.5: The ratio of packets dropped by Snort when using the optimised version of the Snort confiuration file on the 200- and 1000-rule-test.

This experiment implies that the Snort configuration has an effect on the packet processing, and thus the detection ratio. Some of the options mentioned in Figure 3.7 in Section 3.4.2, especially *max_session*, *disable_evasion_alerts* and *memcap*, had very little or no effect on the performance and are not further discussed. *Timeout*, *server_inspect_limit* and *noinspect*, on the other hand, showed to have a very positive effect.

Timeout basically means that Snort keeps an inactive stream in a state table for a given number of seconds, before the stream is deleted. This value was changed from the the default 30 seconds to 1 second in the experiment.

Server_inspect_limit is probably the most important modification. This instructs Snort to only inspect the first given number of bytes of the packet. The smaller *server_inspect_limit*, the faster Snort performs. However, there is a downside, namely that Snort only inspects the first n-bytes. This makes it possible to hide commands further back in the packet payload.

Stateful inspection is built upon keeping a record of the different IP sessions. Since Snort utilizes stateful packet inspection by default, and Snort's state-table only handles up top 100.000 simultaneous sessions (according to [Pro06]), the performance was far from optimal. *Noin-*

spect disabled this stateful inspection, meaning that Snort did not have to keep a record of streams in its memory, and thus work faster.

Even though the *server_inspect_limit* is one the most powerful modifications in connection with enhancing Snort's performance, this was deliberately not used. This is due to the fact that the authors focused on making Snort as fast as possible without limiting the packet payload size in the software.

5.2.3 Experiment 3 - Testing Impact of Common Strings

Figure 5.6 shows the packet drop ratio in experiment 3, which included a few common rules in the ruleset. The included words were *GET*, *POST*, *password* and *.getcdkey*.

One of the purposes with this test was to see the importance of using well written rules. With poorly written rules, processing may increase resulting in packet loss.

As Figure 5.6 shows, it may seem that more packets are dropped when alerts are frequent.

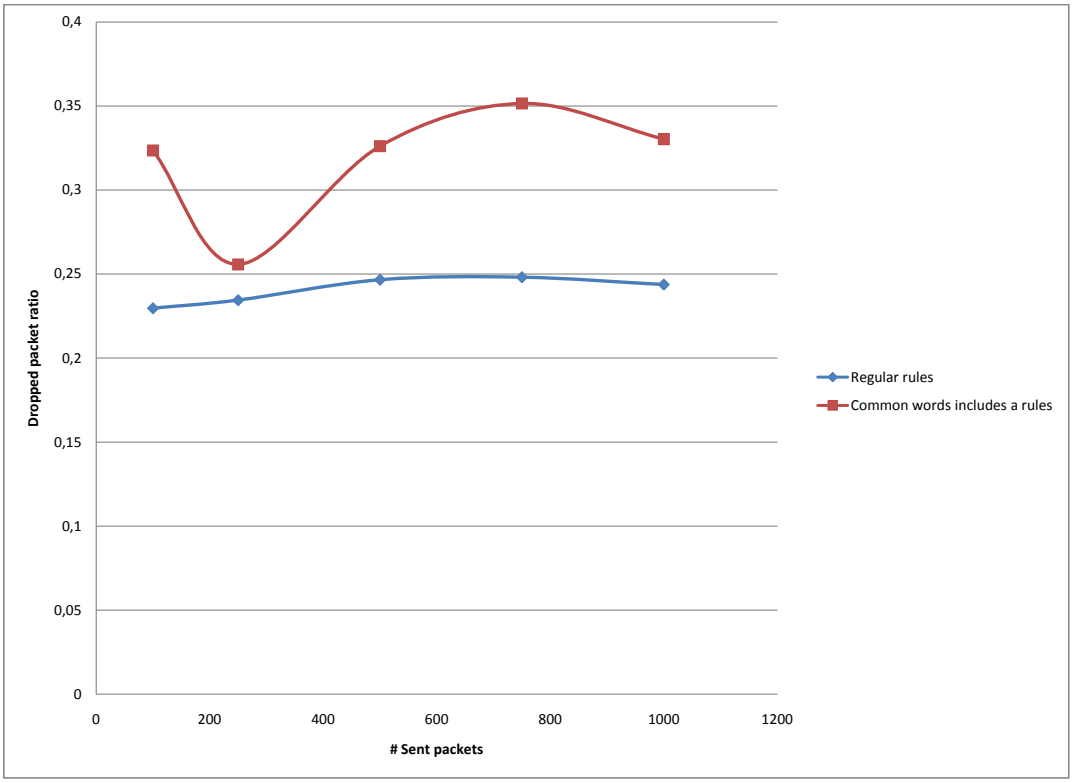

Figure 5.6: Packet drop ratio when the common words GET, POST, password and .getcdkey are included in the rulseset.

This experiment was conducted several times. In some of the trials there was no difference in the packet drop ratio when using the common words rules. This may indicate that there is no significant variations in the processing time. Because of multiple possible sources of errors, it is therefore hard to draw a conclusion based on this experiment.

5.2.4 Experiment 4 - Testing With Minimum External Traffic

This small experiment was conducted simply to test if Snort actually performs well in a more normal environment, as opposed to the high-speed environment.

As shown in Table 4.10 in Section 4.2.4, it is shown that Snort can easily detect intrusions when it is not implemented on a high-speed network. Not surprisingly the experiment gave a 100% detection ratio, and no packets were dropped by Snort.

The reason to why Snort processed more packets than the 100 bot commands that were sent is simply that network messages, retransmissions, ping messages, etc., was captured and analysed as well.

5.2.5 Experiment 5 - Testing Packet Length Input

The result from this experiment is probably the most important ones analysed. It turned out that by varying the *slen* value, it was possible for Snort to analyze and process up to and beyond 150.000 packets per second.

As can easily be seen from Figure 5.7, the packet drop ratio escalates when the *slen* value exceeds 1500. It seems that as long as the value is kept below 1500, the drop ratio decreases to 0.0, Snort is thus able to process all incoming packets.

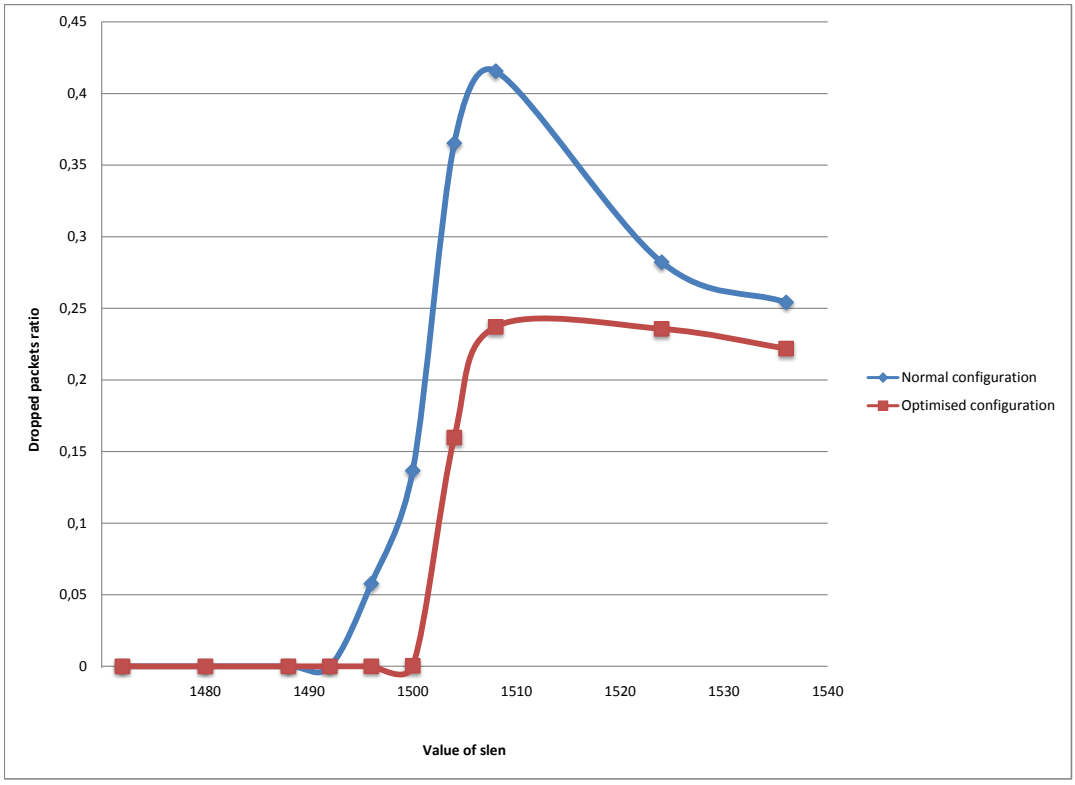

Figure 5.7: The effect of the slen value.

Notice that the packet drop ratio is noticeably lower when using an optimised Snort configuration file, as was shown in experiment 2.

The reason for the result in this experiment is a bit unclear. Most likely is it due to hardware limitations on the IDS computer, more specificly the 64-bits PCI-X bus (as mentioned in Section 3.2.5). Since a DAG card header is appended to the packet, an extra transfer on the bus is necessary should the packet exceed 1500 bytes.

5.2.6 Experiment 6 - Testing Impact of the TCP Protocol

Figure 5.8 presents the result, given in Table 4.13. The respective tests are sorted in ascending order.

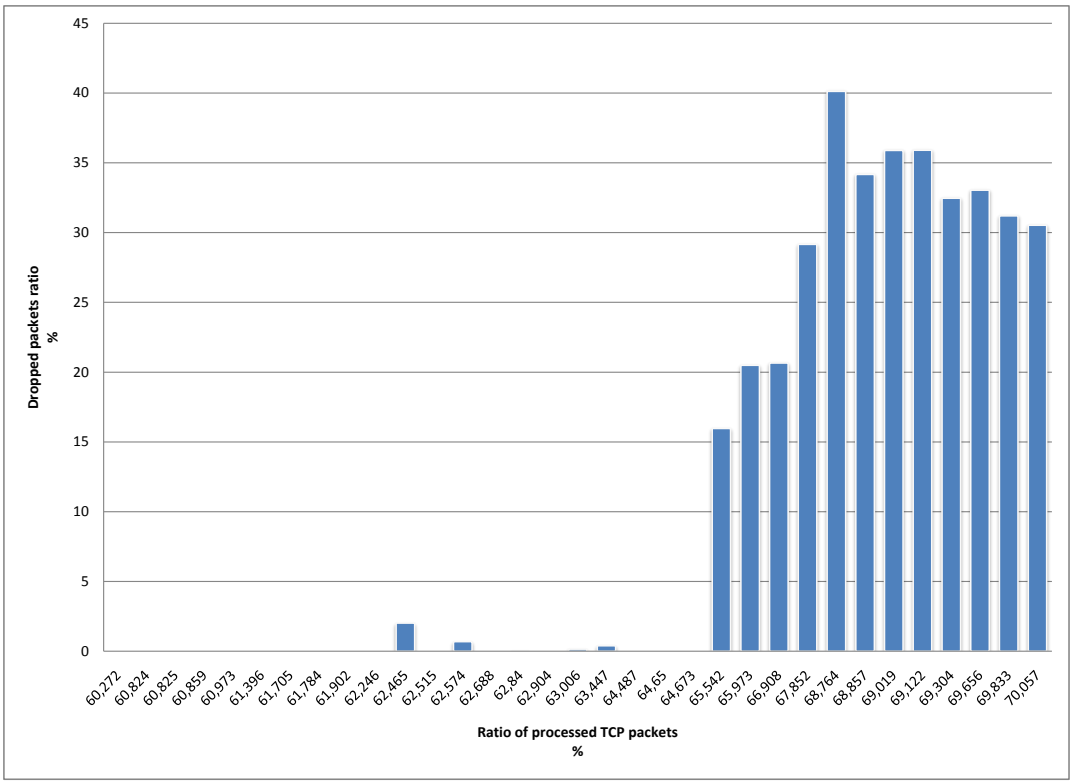

Figure 5.8: The impact the TCP protocol has on the ratio of dropped packets

It seems like the ratio of TCP packets has a clear impact of the ratio of dropped packets when studying the figure. When the ratio of TCP packets exceeds approximately 65%, Snort has to drop packets in order to keep up with the traffic load.

This result is not very surprising as TCP packets have bigger payloads, and thus need more processing time than for example a small UDP packet. However, it is important to mention that the rules used in the testing are limited to checking TCP packets.

5.2.7 Experiment 7 - IDS Stand-Alone Test

The purpose of this experiment was to test if the IDS can handle the traffic load without generating too many false positives.

Table 4.14 in Section 4.2.7 presents four runs conducted at night time. As can be seen from the table, a very large number of packets have been processed, with only a few packets dropped (all have a drop ratio below 1%).

However, all of the four runs had false positives. The log file contained alerts which were results of website URLs containing different words in the ruleset. Even though some false positives were documented, the IDS could handle the traffic load. The false postives ratio was however quite manageable when compared to the total number of processed packets.

Notice that these tests were conducted at night, which means that the traffic pattern differs from the daytime traffic.

Worth mentioning is that the plan was to check detections with Uninett's list of known C&C servers as a measure for false positives. However, the number of detections were so low that they could easily be identified as false positives when reviewing the log file.

Unfortunately, due to workload limitations, experiment 7 had to be conducted over a few nights. This is not really a satisfying scenario, as the result does not provide statistical significant values. Although, it gives a small picture of Snort's capabilities.

5.3 SOURCES OF ERROR

As presented in [AAM04] and [AAMP04], there are some problems regarding efficient and accurate methods for evaluating IDSs.

Firstly, as mentioned in [AAMP04], an IDS's performance varies significantly with different distributions of packets. This has, however, been taken into consideration by continuously monitoring both packet count and packet loss on the backbone network. Since these data were of no significant value when evaluating Snort, it has been excluded from this report.

Secondly, as shown in [AAPM02], IDSs performance is sensitive to packet and ruleset content. The packet contents analysed in the experiments originates from the end users and are thus uncontrollable. However, as seen in experiment 3, multiple detections may cause higher drop ratio.

Thirdly, [AAPM02] mentions that IDS performance varies with the choice of processor architecture. This report is based on using the currently available software and hardware used by Uninett, so processor architecture is not taken in consideration in the experiments conducted. It is however discussed in sections 6.3.

When it comes to measuring false positives, the problem of efficiency arises. Even though an IDS is tested in both a controlled and a real, uncontrollable environment, it is not possible

to test how the IDS will respond in every possible case.

The IDS rules are also subject to errors. In cases where string searching algorithms are used, there is always a possibility for false positives. Consider for instance if there is a rule for detecting DBot's command "!r". This would probably trigger an excessive amount of false positives. Also worth mentioning in the case of string searches, is that false positives will occur if someone is accessing a website accidentally describing bot commands or similar. This for example the case when conducting experiment 7.

Another important source of error to take into consideration is the fact that the experiments were conducted over several days and in different time periods of the day. As the traffic patters varies over the day, this will again cause the testing to vary.

Lastly, it is important to mention that only a limited number of tests were conducted. This is due to the limited workload in writing this report. Ideally, in order to rule out other sources of errors, and to get statistically significant results, the experiments should have been repeatedly conducted over a longer period of time.

5.4 CONCLUSION

The following is a summary from the findings in the experiments conducted:

1. The number of rules included in the IDS matter, but not significantly.
2. Optimising the Snort configuration file has had a positive effect. It is for example possible to include more rules using the optimised version. Worth mentioning, is that the optimised configuration file used in this report is not claimed to be *fully* optimised. It may be possible to tweak the configuration and enhance the performance even more.
3. It is difficult to say to which degree frequent detections causes Snort to drop packets, although it is most likely as frequent detections leads to more processing. Writing good rules should be regarded as an important factor in order to reduce the risk for false positives.
4. By limiting the *slen* value in the DAG card, it is possible for Snort to process all incoming packets. This effect may also be possible using other network interface cards or the server inspect limit in Snort.
5. The ratio of TCP packets matter. A higher ratio of TCP packets causes the packet drop ratio to increase.

Based on these findings, it is plausible to use Snort as botnet detection on a high-speed link. By using an optimised configuration file and writing good relevant rules, and especially setting the *slen* value to 1500 or less, it is possible to at least use Snort up to 150.000 packets per second.

Unfortunately, the maximum traffic load that Snort could handle with the given settings, was not possible to find. This was due to the decreasing load[1] on the backbone when the exper-

[1]During the period of conducting the experiments, the average traffic load was about 120,000 packets per second. One month earlier the traffic load had an average of about 160,000 packets per second, according to drift.uninett.no.

iments were conducted; the number of required packets for the testing could simply not be fulfilled.

The experiments described in this report where conducted on one of the most loaded links in the country. This obviously implies that Snort can run on links that have less traffic than in the experiments conducted in this report - at least based on the experience from this report, and the hardware available.

Chapter 6 discusses some more elements surrounding this specific field of research.

CHAPTER 6
DISCUSSION

This chapter discusses some of the elements in connection with the preceding experiments.

6.1 INTRODUCTION

The following sections presents some arguments, i.e., the authors's opinions, based on the results and analysis presented in Chapter 4 and chapter 5, respectively.

6.2 SNORT VS. MAPI

This report has evaluated both Snort and MAPI used as an IDS in a high-speed environment. Snort has been tested quite thoroughly, while MAPI unfortunately could not be tested due to the bugs mentioned in Section 3.5. The following sections lists some important aspects with an IDS, and discusses these from Snort's and MAPI's point of view.

6.2.1 The Rules

As mentioned in Section 3.3.2, 3.4, and 3.5, rules are handled differently; Snort has its own rule files while MAPI connects flows and then add string search functions on these.

Snort is per today very easy to handle when it comes to managing the rules. If a new rule is needed, it is simply added to a rule file, and after a restart, Snort has included the new rule in the ruleset.

Rules in MAPI, on the other hand, are not so easy to handle. The string search function can only be applied to one flow at the time, as discussed in section 3.5. Adding a new rule therefore implies that a new flow has to be written into the program.

If the string search function is rewritten to handle *OR*, this would be a major improvement because one would only need to apply one flow. String search functions may then be added to this flow.

6.2.2 Performance

When Snort operates with default settings, Snort performs poorly, and is not suited for running in high-speed environments. Even with the modifications done in *snort.noflow.conf*, Snort still gave an unsatisfactory high drop ratio. However, as is shown in the final experiment,

Snort demonstrated that it can be used in high-speed environments, as it had a drop ratio less than 1%.

MAPI could not, as mentioned, be tested to the same extend as Snort. Therefore no results documenting its performance is presented in this report. However, UNINETT claims to previously have run MAPI with over 40 flows connected simultaneously. This information, combined with the observations from the previously MAPI experiments, makes MAPI a promising candidate at least.

6.2.3 Distributed approach

As of today, Snort has no built-in support for a distributed architecture, i.e, one central application controlling the monitoring process on the other computers included in the network. There are however some alternatives, like SURF IDS[1].

SURF IDS has different sensors scattered around in the network, all their running own instances of the Snort application. The difference with this setup, compared to a normal setup, is that the logging is sent to a central logging server which handles the processing. Even though this provides a good picture of the whole network, the downside is that if, e.g., rules are to be changed or the configuration updated, Snort has to be updated and restarted on all of the involved computers.

As will be further elaborated in Section 6.8.1, MAPI offers an extension called DiMAPI. DiMAPI makes it possible to write one MAPI program which connects to a given number of network interfaces on a given number of computers. This means that it is possible to control everything from one central server, making it very easy to, e.g, update the rules for all of the involved computers.

6.3 DAG CARDS AND OTHER HARDWARE

There are basically two components which are especially important regarding an IDS; namely the IDS software and the hardware. The hardware in this connection is the network interface card. As mentioned in Section 2.8, the NIC used in this report is the Endance DAG 4.3S network card. The details can be found in Section 2.8, and is not further elaborated here.

The DAG card played a major role in achieving the results found in the experiments. This especially relates to Snort's performance. It is unlikely that another NIC could cope with the high data rate and deliver the same performance as the DAG card did.

The sensor where the experiments were conducted is connected to a 2.5 Gbps link. When the GigaCampus programme, mentioned in Section 2.7, is completed, this link will most likely have been upgraded to a 10 Gbps link. If Snort is going to have any chance of processing this traffic, it is important that the hardware is able to handle up to 10 Gbps.

As mentioned in Section 2.8, hardware filters can be loaded into the DAG card. These filters were supposed to be tested as a separate experiment in chapter 3. Unfortunately, as

[1]SURF IDS homepage at http://ids.surfnet.nl

mentioned in section 3.4, this experiment was omitted due to firmware problems with the DAG card and co-processor. It is nevertheless the authors' opinion that this, if deployed, would boost the performance of the IDS. By simply filtering out protocols which constitute a significantly amount of the network traffic, a lot of processing is eliminated in the software. The downside is that botnet traffic could be sent over the filtered protocols, and thus go by undetected.

6.4 CIRCUMVENTING THE IDS

IDSs that use string search functions may easily be circumvented by using special crafted and untypical commands. For example by renaming RBot's ".getcdkeys"-command to, e.g., ".qwert", the IDS will not detect this as a malicious command as it is not part of the ruleset. [GH07] however describes an IRC nickname evaluation method based on the untypical commands and nicknames often used in IRC based botnets, as mentioned in Section 2.3.2.

The string search will also be circumvented if the botnet is encrypting its communication. Encrypted botnets are a great challenge for IDS developers, since they also can circumvent anomaly based IDSs.

The IDS tested in this report is based on botnets using the IRC protocol. Botnets using other communication protocols will therefore not be detected, unless they by coincidence uses the same commands as the IRC based botnets.

If the input of packet payload is limited to a fixed value by, e.g., setting the *slen* or *server_inspection_limit* value, the IDS may easily be circumvented by placing the malicious payload at the end of the packet. In order for this to work, the botmaster must know how much of the packet payload that is analysed by the IDS.

6.5 BOTNET SIMILARITIES

There are, as mentioned in Section 2.9.4, several kinds of botnets and botnet families.

A botnet detection system should aspire to having rules that includes signatures for as many bots as possible. By choosing commands which are similar between the different families, the IDS can detect more botnets using fewer rules. Due to the fact that almost every bot family and bot distribution have different commands, this can, however, be very difficult.

The IDS should also include the individual commands as well. However, commands such as ".version" should not be included as this rule would generate a lot of false positives. The experiments conducted in this report indicates that it is possible to run Snort with 1000 rules in a high-speed environment. With such a large number of rules, it should be possible to include most of the malicious botnet commands from the most popular botnet families.

6.6 ALTERNATIVE DETECTION ALGORITHMS

Some of the LOBSTER partners presents in [AAMP03] an alternative string matching algorithm implemented in Snort. It is claimed in this article that the algorithm improves the performance by 10% - 36%. There are however other ways to detect malicious traffic than using

string search.

The article [BS06] presents an anomaly based IRC botnet detection algorithm. This algorithm combines an IRC mesh detection component with a TCP scan detection heuristic they have called the *TCP work weight*. This algorithm is, at the time of writing, not tested in a high-speed environment.

6.7 Handling the Detection Data

In a high-speed environment the logging of detected data can in some cases get quite excessive. It is therefore important to have good routines to handle the detected data.

In this connection it is important that responsible personnel are notified as fast as possible. Quick alerting is necessary in order to make countermeasures before the indications evolve into a full scale attack.

For administrators it is also important that the detected data is handled in such a matter so that it is possible to perform statistical analysis of this data. This makes it possible to discover and handle vulnerabilities in the network. Captured data may also be used in network research or to get a general view of the threat level.

As mentioned in [SBH+07]; if the data is to be used in relation with digital forensics, the data needs to be handled with care. It is important not to jeopardise the integrity of the collected evidence. One possible way is to use hash function on the captured data. These functions makes a unique hash value of the given data. It is then not possible to tamper with the data without changing the hash value.

6.8 Future Prospects

The research presented in this report was the first step in implementing an IDS in Uninett's backbone network. There is however still some research that needs be conducted and this is suggested in Chapter 7. The following Section discussed some future prospects in this connection.

6.8.1 A distributed Approach

As mentioned in Section 2.5, MAPI offers a distributed extension called DiMAPI. DiMAPI enables flow creation and manipulation of local and remote monitoring sensors, as mentioned in [TPP+06]. The article also claims that the response latency is very close to the actual round trip time between the monitoring application and sensors - making it usable in a distributed IDS implementation.

A distributed IDS is very useful for detection large scale attacks. There are however some challenges in this connection. As more IDSs are interconnected, more data is processed and needs to be handled and correlated.

An distributed botnet dection system combining all the sensors in the LOBSTER project,

would for example lead to huge tasks when it comes to process the information gathered. Detected botnet traffic from multiple sensors would need to be compared and correlated, in order to check whether the alert found in location A is the same as the alert in location B.

Another distributed IDS is presented in [SYL03]. This article suggests a peer-to-peer IDS architecture which can be used in a Gigabit network. It is claimed in this article that the IDS has a packet loss ratio less than 10%.

6.8.2 IPv4 vs. IPv6

Even though IPv4 is most commonly used today, it is obvious that the use of IPv6 will increase in near future.

As of today, MAPI only supports IPv4, while Snort is already IPv6 compatible. MAPI needs to be adapted to support IPv6, if it is to be used in the future. The DAG card is already IPv6 compatible.

The main challenge with IPv6 is that the size of the average packets increases. The IPv6 packet header itself, is twice the size as the packet headers in IPv4. In addition, the payload size will increase, making it tougher for both hardware and software to keep up with the processing.

CHAPTER 7
FURTHER WORK

This Chapter presents some further work in connection with the research presented throughout this report.

7.1 INTRODUCTION

Due to the limited time span and workload, some limitations of the research had to be made. As a consequence this Chapter presents suggested research, which could not be conducted by the authors.

7.2 IMPLEMENTING MAPI AS AN IDS

The research presented in this report was planned to include a MAPI implementation. Because of bugs in the software, this had to be omitted. If a future release of MAPI fixes the mentioned bugs and is more stable, MAPI should definitely be retested.

A retest should include the same parameters as Snort did in the experiments, so that a comparison of the two IDS implementations is possible.

7.3 TESTING MAXIMUM TRAFFIC LOAD

The experiments mentioned in this report was conducted on a busy backbone link. At the time of conducting the experiments, the traffic unfortunately started to decrease due to summer vacation. This basically meant that it was not possible to test Snort's maximum capacity. The number of packets per second, was simply too low to challenge Snort.

If a maximum limit should be found, one could expand the experiments by varying the *slen* or *server_inspect_limit* value on respectively the DAG card and in the Snort configuration even further.

Further work should include testing of Snort's maximal capacity, by using sensors located at busier links.

7.4 BOTNETS BASED ON OTHER PROTOCOLS

As the research presented in this report was only focused on IRC based botnet, further work should include botnets based on other protocols. E.g., the protocols mentioned in Section 2.9.1. This is also due to the fact that many botnets tend to utilze other protocols than IRC.

7.5 Increased experiment length and frequency

One of the drawbacks with this report, is that the number of experiments was limited due to the available time, as mentioned in Section 5.3. To get results of significant statistical values, these experiments should be repeated, i.e., the experiments should be conducted both multiple times and over longer time periods.

These experiments should take variations in the traffic pattern into consideration.

7.6 Dynamic Rules

It is possible to use dynamic rules in Snort. A dynamic rule is a rule which is loaded, but never used before it is activated by some sort of a trigger. This trigger may be a special event or another rule.

An idea could be to make dynamic rules which changes the configuration if a special condition is met. Even though this is not tested, it may be possible to write a script that decreases the *server_inspect_limit* value should the TCP ratio exceed a certain point, or for example if the dropped packet ratio exceeds an undesirable value.

7.7 Notification Upon Detection

The IDS implementations presented in this report should include functionality for notifying responsible personnel. This functionality should also include appropriate handling of the detection data, as discussed in Section 6.7.

7.8 Detection of Other Types of Intrusion

The main purpose of this research was to investigate high-speed IDS implementations, but was limited to detecting botnets due to scope limitation. However, real high-speed IDSs should handle several other threats in addition to the botnets. It is the authors's opinion that a *complete* high-speed IDS should be able to detect all known threats.

7.9 Sampling

A possible research suggestion to look into, is sampling. This basically means that instead of analysing each packet, every given number of packet is inspected. It is also possible to use other parameters, e.g., inspecting at random times or random flows.

By using sampling the processing performance is less critical, since the amount of packets are significantly lowered.

REFERENCES

[AAM04] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating Relistic Workloads for Network Intrusion Detection Systems. *Proceedings of Fourth International Workshop on Software and Performance (WOSP)*, 2004.

[AAMP03] K. G. Anagnostakis, S. Antonatos, E. P. Markatos, and M. Polychronakis. E2XB: A Domain-specific String Matching Algorithm for Intrusion Detection. *Proceedings of the 18th IFIP International Information Security Conference*, 2003.

[AAMP04] Spyros Antonatos, Kostas G. Anagnostakis, Evangelos P. Markatos, and Michalis Polychronakis. Performance Analysis of Content Matching Intrusion Detection Systems. *Proceedings of the IEEE/IPSJ Symposium on Applications and the Internet (SAINT)*, 2004.

[AAPM02] Spyros Antonatos, Kostas G. Anagnostakis, Michalis Polychronakis, and Evangelos P. Markatos. Benchmarking and design of string matching intrusion detection systems. *Technical Report 315 ICS-FORTH*, 2002.

[BS06] James R. Binkley and Suresh Singh. An Algorithm for Anomaly-based Botnet Detection. *FLOCON CERT/SEI, Vancouver WA*, 2006.

[BY07] Paul Barford and Vinod Yegneswaran. An Inside Look at Botnets. *Advances in Information Security , Vol. 27*, 2007.

[CAM04] Ioannis Charitakis, Kostas Anagnostakis, and Evangelos P. Markatos. A Network-Processor-Based Traffic Splitter for Intrusion Detection. *ICS-FORTH Technical Report 342*, 2004.

[Cor06] Symantec Corporation. Symantec Internet Security Threat Report. *Volume X*, 2006.

[Cor07] Symantec Corporation. Symantec Internet Security Threat Report. *Volume XI*, 2007.

[GH07] Jan Goebel and Thorsten Holz. Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation. *HotBots 07*, 2007.

[GSN+07] Julian B. Grizzard, Vikram Sharma, Chris Nunnery, Brent ByungHoon Kang, and David Dagon. Peer-to-Peer Botnets: Overview and Case Study. *HotBots 07*, 2007.

[IH05] Nicholas Ianelli and Aaron Hackwort. Botnets as a Vehicle for Online Crime. *CERT Coordination Center*, 2005.

[KRH07] Anestis Karasaridis, Brian Rexroad, and David Hoeflin. Wide-scale Botnet Detection and Characterization. *HotBots 07*, 2007.

[LA00] Brian Laing and Jimmy Alderson. How To Guide - Implementing a Network Based Intrusion Detection System. *Internet Security Systems*, 2000.

[Ltd05] Endance Measurements System Ltd. Co-processor IP Filter Software User Manual. *http://www.endance.com*, 2005.

[MAP06] MAPI. A Tutorial Introduction to MAPI. *http://mapi.uninett.no*, 2006.

[OR93] J. Oikarinen and D. Reed. Internet Relay Chat Protocol. *Request for Comments: 1459, http://www.ietf.org/rfc/rfc1459.txt*, 05 1993.

[Pro06] The Snort Project. Snort User Manual. *http://www.snort.org*, 2006.

[RZMT06] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. *Internet Measurement Conference 2006*, 2006.

[SBH+07] Craig A. Schiller, Jim Binkley, David Harley, Gadi Evron, Tony Bradley, Carsten Willems, and Michael Cross. Botnets: The Killer Web App. *Syngress Publishing, Inc.*,, 2007.

[SM07] Karen Scarfone and Peter Mell. Guide to Intrusion Detection and Prevention Systems (IDPS). *National Institute of Standards and Technology*, 2007.

[SYL03] Bo Song, Ming Ye, and Jie Li. Intrusion Detection Technology Research based High-Speed Network. *IEEE*, 2003.

[TPP+06] Panos Trimintzios, Michalis Polychronakis, Antonis Padadogiannakis, Michalis Foukarakis, Evangelos P. Markatos, and Arne Øslebø. DiMAPI: An Application Programming Interface for Distributed Network Monitoring. *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2006.

[XCA+06] Konstantinos Xinidis, Ioannis Charitakis, Spiros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. An Active Splitter Architecture for Intrusion Detection and Prevention. *IEEE Transactions on Dependable Secure Computing*, 2006.

APPENDIX A
MODIFIED SNORT CONFIGURATION FILE

This appendix lists the modified Snort configuration file.

A.1 SNORT.NOFLOW.CONF

Figures A.1, A.2, A.3, A.4 and A.5 present the code for `snort.noflow.conf`.

snort.noflow.conf (continues on page 72)

```
1  #--------------------------------------------------
2  #   http://www.snort.org      Snort 2.6.1.5 Ruleset
3  #     Contact: snort-sigs@lists.sourceforge.net
4  #--------------------------------------------------
5  # $Id$
6
7  var HOME_NET any
8  var EXTERNAL_NET any
9
10 config logdir : /home/ids/snort/logs
11 config interface: dag0
12
13 config disable_decode_alerts
14 config disable_tcpopt_experimental_alerts
15 config disable_tcpopt_obsolete_alerts
16 config disable_tcpopt_ttcp_alerts
17 config disable_ttcp_alerts
18 config disable_tcpopt_alerts
19 config disable_ipopt_alerts
20
21 var RULE_PATH /home/ids/snort/rules
22
23 # Flow module
24 # preprocessor flow: stats_interval 0 hash 2
25
26 # frag3: Target-based IP defragmentation
27 # ---------------------------------------
28 #
29 # Frag3 is a brand new IP defragmentation preprocessor that is capable of
30 # performing "target-based" processing of IP fragments.  Check out the
31 # README.frag3 file in the doc directory for more background and configuration
32 # information.
33 #
34 # Frag3 configuration is a two step process, a global initialization phase
35 # followed by the definition of a set of defragmentation engines.
36 #
37 # Global configuration defines the number of fragmented packets that Snort can
38 # track at the same time and gives you options regarding the memory cap for the
39 # subsystem or, optionally, allows you to preallocate all the memory for the
40 # entire frag3 system.
41 #
42 # frag3_global options:
43 #   max_frags: Maximum number of frag trackers that may be active at once.
44 #              Default value is 8192.
45 #   memcap: Maximum amount of memory that frag3 may access at any given time.
```

Figure A.1: snort.noflow.conf part 1

snort.noflow.conf (continued from page 71, continues on page 73)

```
46 #           Default value is 4MB.
47 #    prealloc_frags: Maximum number of individual fragments that may be processed
48 #                    at once.  This is instead of the memcap system, uses static
49 #                    allocation to increase performance.  No default value.  Each
50 #                    preallocated fragment eats ~1550 bytes.
51 #
52 # Target-based behavior is attached to an engine as a "policy" for handling
53 # overlaps and retransmissions as enumerated in the Paxson paper.  There are
54 # currently five policy types available: "BSD", "BSD-right", "First", "Linux"
55 # and "Last".  Engines can be bound to standard Snort CIDR blocks or
56 # IP lists.
57 #
58 # frag3_engine options:
59 #    timeout: Amount of time a fragmented packet may be active before expiring.
60 #             Default value is 60 seconds.
61 #    ttl_limit: Limit of delta allowable for TTLs of packets in the fragments.
62 #               Based on the initial received fragment TTL.
63 #    min_ttl: Minimum acceptable TTL for a fragment, frags with TTLs below this
64 #             value will be discarded.  Default value is 0.
65 #    detect_anomalies: Activates frag3's anomaly detection mechanisms.
66 #    policy: Target-based policy to assign to this engine.  Default is BSD.
67 #    bind_to: IP address set to bind this engine to.  Default is all hosts.
68 #
69 # Frag3 configuration example:
70 #preprocessor frag3_global: max_frags 65536 prealloc_frags 262144
71 #preprocessor frag3_engine: policy linux \
72 #                           bind_to [10.1.1.12/32,10.1.1.13/32] \
73 #                           detect_anomalies
74 #preprocessor frag3_engine: policy first \
75 #                           bind_to 10.2.1.0/24 \
76 #                           detect_anomalies
77 #preprocessor frag3_engine: policy last \
78 #                           bind_to 10.3.1.0/24
79 #preprocessor frag3_engine: policy bsd
80
81 preprocessor frag3_global: max_frags 65536
82 preprocessor frag3_engine: policy linux
83
84
85
86 # stream4: stateful inspection/stream reassembly for Snort
87 #-------------------------------------------------------------------
88 # Use in concert with the -z [all|est] command line switch to defeat stick/snot
89 # against TCP rules.  Also performs full TCP stream reassembly, stateful
90 # inspection of TCP streams, etc.  Can statefully detect various portscan
91 # types, fingerprinting, ECN, etc.
92
93 # stateful inspection directive
94 # no arguments loads the defaults (timeout 30, memcap 8388608)
95 # options (options are comma delimited):
96 #    detect_scans - stream4 will detect stealth portscans and generate alerts
97 #                   when it sees them when this option is set
98 #    detect_state_problems - detect TCP state problems, this tends to be very
99 #                            noisy because there are a lot of crappy ip stack
100 #                           implementations out there
101 #
102 #    disable_evasion_alerts - turn off the possibly noisy mitigation of
103 #                             overlapping sequences.
104 #
105 #    ttl_limit [number]     - differential of the initial ttl on a session versus
106 #                             the normal that someone may be playing games.
107 #                             Routing flap may cause lots of false positives.
108 #
109 #    keepstats [machine|binary] - keep session statistics, add "machine" to
110 #                          get them in a flat format for machine reading, add
111 #                          "binary" to get them in a unified binary output
112 #                          format
113 #    noinspect - turn off stateful inspection only
114 #    timeout [number] - set the session timeout counter to [number] seconds,
115 #                       default is 30 seconds
116 #    max_sessions [number] - limit the number of sessions stream4 keeps
117 #                          track of
118 #    memcap [number] - limit stream4 memory usage to [number] bytes (does
119 #                      not include session tracking, which is set by the
120 #                      max_sessions option)
121 #    log_flushed_streams - if an event is detected on a stream this option will
122 #                          cause all packets that are stored in the stream4
123 #                          packet buffers to be flushed to disk.  This only
124 #                          works when logging in pcap mode!
125 #    server_inspect_limit [bytes] - Byte limit on server side inspection.
126 #    enable_udp_sessions - turn on tracking of "sessions" over UDP.  Requires
127 #                          configure --enable-stream4udp.  UDP sessions are
128 #                          only created when there is a rule for the sender or
129 #                          responder that has a flow or flowbits keyword.
130 #    max_udp_sessions [number] - limit the number of simultaneous UDP sessions
```

Figure A.2: snort.noflow.conf part 2

snort.noflow.conf (continued from page 72, continues on page 74)

```
131 #                               to track
132 #   udp_ignore_any - Do not inspect UDP packets unless there is a port specific
133 #                    rule for a given port.  This is a performance improvement
134 #                    and turns off inspection for udp xxx any -> xxx any rules
135 #   cache_clean_sessions [number] - Cleanup the session cache by number sessions
136 #                                   at a time.  The larger the value, the
137 #                                   more sessions are purged from the cache when
138 #                                   the session limit or memcap is reached.
139 #                                   Defaults to 5.
140 #
141 #
142 #
143
144 preprocessor stream4: disable_evasion_alerts, max_sessions 50000
145
146 # tcp stream reassembly directive
147 # no arguments loads the default configuration
148 #   Only reassemble the client,
149 #   Only reassemble the default list of ports (See below),
150 #   Give alerts for "bad" streams
151 #
152 # Available options (comma delimited):
153 #   clientonly - reassemble traffic for the client side of a connection only
154 #   serveronly - reassemble traffic for the server side of a connection only
155 #   both - reassemble both sides of a session
156 #   noalerts - turn off alerts from the stream reassembly stage of stream4
157 #   ports [list] - use the space separated list of ports in [list], "all"
158 #                  will turn on reassembly for all ports, "default" will turn
159 #                  on reassembly for ports 21, 23, 25, 42, 53, 80, 110,
160 #                  111, 135, 136, 137, 139, 143, 445, 513, 1433, 1521,
161 #                  and 3306
162 #   favor_old - favor an old segment (based on sequence number) over a new one.
163 #               This is the default.
164 #   favor_new - favor an new segment (based on sequence number) over an old one.
165 #   overlap_limit [number] - limit on overlaping segments for a session.
166 #   flush_on_alert - flushes stream when an alert is generated for a session.
167 #   flush_behavior [mode] -
168 #           default      - use old static flushpoints (default)
169 #           large_window - use new larger static flushpoints
170 #           random       - use random flushpoints defined by flush_base,
171 #                          flush_seed and flush_range
172 #   flush_base [number] - lowest allowed random flushpoint (512 by default)
173 #   flush_range [number] - number is the space within which random flushpoints
174 #                          are generated (default 1213)
175 #   flush_seed [number] - seed for the random number generator, defaults to
176 #                         Snort PID + time
177 #
178 # Using the default random flushpoints, the smallest flushpoint is 512,
179 # and the largest is 1725 bytes.
180 preprocessor stream4_reassemble: both, ports all, noalerts
181
182
183
184 ##################################################################
185 # Step #4: Configure output plugins
186 #
187 # Uncomment and configure the output plugins you decide to use.  General
188 # configuration for output plugins is of the form:
189 #
190 # output <name_of_plugin>: <configuration_options>
191 #
192 # log_tcpdump: log packets in binary tcpdump format
193 # ----------------------------------------------------
194 # The only argument is the output file name.
195 #
196 # output log_tcpdump: tcpdump.log
197
198 # database: log to a variety of databases
199 # -----------------------------------------
200 # See the README.database file for more information about configuring
201 # and using this plugin.
202 #
203 # output database: log, mysql, user=root password=test dbname=db host=localhost
204 # output database: alert, postgresql, user=snort dbname=snort
205 # output database: log, odbc, user=snort dbname=snort
206 # output database: log, mssql, dbname=snort user=snort password=test
207 # output database: log, oracle, dbname=snort user=snort password=test
208
209 # unified: Snort unified binary format alerting and logging
210 # -------------------------------------------------------
211 # The unified output plugin provides two new formats for logging and generating
212 # alerts from Snort, the "unified" format.  The unified format is a straight
213 # binary format for logging data out of Snort that is designed to be fast and
214 # efficient.  Used with barnyard (the new alert/log processor), most of the
215 # overhead for logging and alerting to various slow storage mechanisms such as
```

Figure A.3: snort.noflow.con part 3

snort.noflow.conf (continued from page 73, continues on page 75)

```
216 # databases or the network can now be avoided.
217 #
218 # Check out the spo_unified.h file for the data formats.
219 #
220 # Two arguments are supported.
221 #    filename - base filename to write to (current time_t is appended)
222 #    limit    - maximum size of spool file in MB (default: 128)
223 #
224  output alert_unified: filename snort.alert, limit 128
225  output log_unified: filename snort.log, limit 128
226
227
228 # prelude: log to the Prelude Hybrid IDS system
229 # ----------------------------------------------
230 #
231 # profile = Name of the Prelude profile to use (default is snort).
232 #
233 # Snort priority to IDMEF severity mappings:
234 # high < medium < low < info
235 #
236 # These are the default mapped from classification.config:
237 # info   = 4
238 # low    = 3
239 # medium = 2
240 # high   = anything below medium
241 #
242 # output alert_prelude
243 # output alert_prelude: profile=snort-profile-name
244
245
246 # You can optionally define new rule types and associate one or more output
247 # plugins specifically to that type.
248 #
249 # This example will create a type that will log to just tcpdump.
250 # ruletype suspicious
251 # {
252 #   type log
253 #   output log_tcpdump: suspicious.log
254 # }
255 #
256 # EXAMPLE RULE FOR SUSPICIOUS RULETYPE:
257 # suspicious tcp $HOME_NET any -> $HOME_NET 6667 (msg:"Internal IRC Server";)
258 #
259 # This example will create a rule type that will log to syslog and a mysql
260 # database:
261 # ruletype redalert
262 # {
263 #   type alert
264 #   output alert_syslog: LOG_AUTH LOG_ALERT
265 #   output database: log, mysql, user=snort dbname=snort host=localhost
266 # }
267 #
268 # EXAMPLE RULE FOR REDALERT RULETYPE:
269 # redalert tcp $HOME_NET any -> $EXTERNAL_NET 31337 \
270 #   (msg:"Someone is being LEET"; flags:A+;)
271
272 #
273 # Include classification & priority settings
274 # Note for Windows users:  You are advised to make this an absolute path,
275 # such as:  c:\snort\etc\classification.config
276 #
277
278 include classification.config
279
280 #
281 # Include reference systems
282 # Note for Windows users:  You are advised to make this an absolute path,
283 # such as:  c:\snort\etc\reference.config
284 #
285
286 include reference.config
287
288 ######################################################################
289 # Step #6: Customize your rule set
290 #
291 # Up to date snort rules are available at http://www.snort.org
292 #
293 # The snort web site has documentation about how to write your own custom snort
294 # rules.
295
296 #include $RULE_PATH/local.rules
297 #include $RULE_PATH/bad-traffic.rules
298 #include $RULE_PATH/exploit.rules
299 #include $RULE_PATH/scan.rules
300 #include $RULE_PATH/finger.rules
```

Figure A.4: snort.noflow.con part 4

snort.noflow.conf (continued from page 74)

```
300 #include $RULE_PATH/finger.rules
301 #include $RULE_PATH/ftp.rules
302 #include $RULE_PATH/telnet.rules
303 #include $RULE_PATH/rpc.rules
304 #include $RULE_PATH/rservices.rules
305 #include $RULE_PATH/dos.rules
306 #include $RULE_PATH/ddos.rules
307 #include $RULE_PATH/dns.rules
308 #include $RULE_PATH/tftp.rules
309
310 #include $RULE_PATH/web-cgi.rules
311 #include $RULE_PATH/web-coldfusion.rules
312 #include $RULE_PATH/web-iis.rules
313 #include $RULE_PATH/web-frontpage.rules
314 #include $RULE_PATH/web-misc.rules
315 #include $RULE_PATH/web-client.rules
316 #include $RULE_PATH/web-php.rules
317
318 #include $RULE_PATH/sql.rules
319 #include $RULE_PATH/x11.rules
320 #include $RULE_PATH/icmp.rules
321 #include $RULE_PATH/netbios.rules
322 #include $RULE_PATH/misc.rules
323 #include $RULE_PATH/attack-responses.rules
324 #include $RULE_PATH/oracle.rules
325 #include $RULE_PATH/mysql.rules
326 #include $RULE_PATH/snmp.rules
327
328 #include $RULE_PATH/smtp.rules
329 #include $RULE_PATH/imap.rules
330 #include $RULE_PATH/pop2.rules
331 #include $RULE_PATH/pop3.rules
332
333 #include $RULE_PATH/nntp.rules
334 #include $RULE_PATH/other-ids.rules
335 # include $RULE_PATH/web-attacks.rules
336 # include $RULE_PATH/backdoor.rules
337 # include $RULE_PATH/shellcode.rules
338 # include $RULE_PATH/policy.rules
339 # include $RULE_PATH/porn.rules
340 # include $RULE_PATH/info.rules
341 # include $RULE_PATH/icmp-info.rules
342 # include $RULE_PATH/virus.rules
343 # include $RULE_PATH/chat.rules
344 # include $RULE_PATH/multimedia.rules
345 # include $RULE_PATH/p2p.rules
346 # include $RULE_PATH/spyware-put.rules
347 #include $RULE_PATH/experimental.rules
348
349
350 #include $RULE_PATH/community-bot.rules
351 #include $RULE_PATH/community-web-client.rules
352 #include $RULE_PATH/bot-our-noflow.rules
353
354 #include $RULE_PATH/noflow-5.rules
355 #include $RULE_PATH/noflow-30.rules
356 #include $RULE_PATH/noflow-50.rules
357 #include $RULE_PATH/noflow-100.rules
358 include $RULE_PATH/noflow-200.rules
359 #include $RULE_PATH/noflow-1000.rules
```

Figure A.5: snort.noflow.con part 5

APPENDIX B
MAPI IDS PROGRAM

This appendix lists a MAPI IDS program with five rules.

B.1 MAPI IDS PROGRAM

Figures B.1 and B.2 present the code for the `MAPI IDS program`.

MAPI IDS program (continues on page 78)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <signal.h>
5  #include <net/ethernet.h>
6  #include <sys/time.h>
7  #include <netinet/ip.h>
8  #include <sys/socket.h>
9  #include <netinet/in.h>
10 #include <arpa/inet.h>
11 #include <mapi.h>
12 #include <string.h>
13
14 // declare some functions
15 static void terminate();
16
17 // globally initialize the mapi-flows
18 int aa, ab, ac, ad, fd;
19
20 // global packet counters, fid
21 int fid;
22
23 // global packet counter
24 mapi_results_t *dres;
25
26 // time variables
27 time_t start, end;
28
29 // program starts here
30 int main(int argc, char *argv[]) {
31
32     // define signals which calls terminate() and aborts the program; e.g, ctrl-c
33     signal(SIGINT, terminate);
34     signal(SIGQUIT, terminate);
35     signal(SIGTERM, terminate);
36
37     // create a flow using the dag0 interface
38     fd = mapi_create_flow ("/dev/dag0");
39     aa = mapi_create_flow ("/dev/dag0");
40     ab = mapi_create_flow ("/dev/dag0");
41     ac = mapi_create_flow ("/dev/dag0");
42     ad = mapi_create_flow ("/dev/dag0");
43
44     // check whether we could create the flows
45     if( aa < 0 && ab < 0 && ac < 0 && ad < 0 && fd < 0) {
```

Figure B.1: MAPI IDS program part 1

Mapi 5-rule program (continued from page 77)

```
46          printf("Could not create flow %d\n", aa);
47          exit(EXIT_FAILURE);
48      }
49
50      // Apply packet counting on the fd flow
51      fid = mapi_apply_function(fd,"PKT_COUNTER");
52
53      // Apply the string search function (content rule matching) to the flows
54      mapi_apply_function (aa, "STR_SEARCH", ".getcdkey",          0, 1500);
55      mapi_apply_function (ab, "STR_SEARCH", ".synflood.start",    0, 1500);
56      mapi_apply_function (ac, "STR_SEARCH", ".synflood.stop",     0, 1500);
57      mapi_apply_function (ad, "STR_SEARCH", ".ddos.ack",          0, 1500);
58
59      // Apply the file writer function to the differentflows
60      mapi_apply_function (aa, "TO_FILE", MFF_PCAP, "aa.pcap", 0);
61      mapi_apply_function (ab, "TO_FILE", MFF_PCAP, "ab.pcap", 0);
62      mapi_apply_function (ac, "TO_FILE", MFF_PCAP, "ac.pcap", 0);
63      mapi_apply_function (ad, "TO_FILE", MFF_PCAP, "ad.pcap", 0);
64
65      // start the aa flow
66      if(mapi_connect(aa) < 0) {
67         printf("Could not connect to flow %d\n", aa);
68         exit(EXIT_FAILURE);
69      }
70
71      // start the ab flow
72      if(mapi_connect(ab) < 0) {
73           printf("Could not connect to flow %d\n", ab);
74           exit(EXIT_FAILURE);
75      }
76
77      // start the ac flow
78      if(mapi_connect(ac) < 0) {
79         printf("Could not connect to flow %d\n", ac);
80         exit(EXIT_FAILURE);
81      }
82
83      // start the ad flow
84      if(mapi_connect(ad) < 0) {
85         printf("Could not connect to flow %d\n", ad);
86         exit(EXIT_FAILURE);
87      }
88
89      // start the fd flow
90      if(mapi_connect(fd) < 0) {
91         printf("Could not connect to flow %d\n", fd);
92         exit(EXIT_FAILURE);
93      }
94
95      // start script
96      printf ("\n Working...\n");
97
98      // start the timer
99      time(&start);
100
101     // make the script run forever until we terminate
102     while(1){
103         sleep(1);
104     }
105
106  return 0;
107 }
108
109
110 void terminate() {
111
112     printf ("\n Finished..\n");
113
114     time(&end);
115     double dif = difftime (end,start);
116     dres = mapi_read_results(fd, fid);
117
118     printf ("\n=== Statistics ===\n");
119     printf ("Run-time: %.2lf seconds.\n\n", dif );
120
121     printf("Total packets : %llu\n", *((unsigned long long*)dres->res));
122
123     mapi_close_flow(aa);
124     mapi_close_flow(ab);
125     mapi_close_flow(ac);
126     mapi_close_flow(ad);
127     mapi_close_flow(fd);
128
129     exit(EXIT_SUCCESS);
130 }
```

Figure B.2: MAPI IDS program part 2