TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science
Chair of Network Software

A COMPARATIVE ANALYSIS OF OPEN-SOURCE INTRUSION DETECTION SYSTEMS

Master's Thesis

ITI70LT

Student: Mauno Pihelgas
Student code: 106497IVCMM
Advisor: Risto Vaarandi, Ph.D

Tallinn
2012

Declaration

I hereby declare that I am the sole author of this thesis. The work is original and has not been submitted for any degree or diploma at any other University. I further declare that the material obtained from other sources has been duly acknowledged in the thesis.

……………………………………. ………………………………
(date) (signature)

MAGISTRITÖÖ ÜLESANNE

Üliõpilane: **MAUNO PIHELGAS** Matrikkel: **106497**

Lõputöö teema eesti keeles:
Võrdlusanalüüs vabatarkvaralistest ründetuvastussüsteemidest

Lõputöö teema inglise keeles:
A Comparative Analysis Of Open-Source Intrusion Detection Systems

Juhendaja (nimi, töökoht, teaduslik kraad, allkiri):
Risto Vaarandi, Arvutiteaduse Instituut, PhD

Konsultandid:

Lahendatavad küsimused ning lähtetingimused:
Anda ülevaade populaarsetest vabatarkvaralistest ründetuvastussüsteemidest.
Testida ning võrrelda ründetuvastussüsteemide jõudlusomadusi.

Eritingimused:

Nõuded vormistamisele: Vastavalt Arvutitehnika instituudis kehtivatele nõuetele

Lõputöö estamise tähtaeg: 07.12.2012

Ülesande vastu võtnud: ______________________________kuupäev: 29.08.2012
(lõpetaja allkiri)

List of Acronyms and Abbreviations

BSD license	A class of extremely simple and very liberal licenses for computer software. Acronym BSD is short for Berkley Source Distribution. [1]
CERT	Computer Emergency Response Team.
CPU	Central Processing Unit.
DoS	Denial of Service.
GNU	A recursive acronym for *GNU's Not Unix.* [2]
GNU GPL	GNU General Public License. The most widely used license for free software. [2]
GNU GRUB	A multi-boot boot loader responsible for loading and transferring control to the operating system kernel.
GPU	Graphics processing unit. A processing unit on a graphics cards.
HDD	Hard Disk Drive.
ICMP	Internet Control Message Protocol. One of the core protocols of the Internet Protocol suite.
IDS	Intrusion Detection System.
IPS	Intrusion Protection System.
md5sum	A program used to calculate and verify 128-bit MD5 hashes.
NIC	Network Interface Card.
NIDS	Network-based Intrusion Detection System.
Nmap	Nmap ("Network Mapper") is a free and open-source utility for network discovery and security auditing. [3]
PCAP	A libpcap library file format that is the primary capture format for many networking tools. [4]
RAM	Random Access Memory.
RPM	A Package management system for many Linux distributions. The name also refers to software packaged to files in the .rpm format. It is a recursive acronym for "RPM Package Manager".
SSH	Secure Shell. A network protocol used primarily for remote access.

TCP	Transmission Control Protocol. One of the core protocols of the Internet Protocol suite.
UDP	User Datagram Protocol. One of the core protocols of the Internet Protocol suite.
yum	Open-source automatic updater and package management utility for RPM systems.

Abstract

This thesis focuses on comparing three popular open-source network intrusion detection systems (NIDS) called Snort, Suricata and Bro. The aim of this thesis is to find out the advantages and disadvantages of each system. Performance evaluation was performed on a 1Gbit/s network with several experiments.

Snort has become the industry standard open-source intrusion detection technology over the last decade and is the most widely deployed IDS worldwide.

Suricata is a newer intrusion detection engine that is intended to bring new ideas and technologies like multi-threading to the field of IDS's. It uses roughly the same set of rules as Snort.

Bro is slightly alternative compared to Snort and Suricata. While focusing on network security monitoring, it also provides a comprehensive platform for more general network traffic analysis.

Experiments demonstrated that all three systems with their default configuration were only able to handle about 100Mbit/s network traffic. After numerous optimizations and with the use of PF_RING network socket, performance increased at least ten-fold. Transmission speeds of 1,000Mbit/s were handled without any dropped packets.

Table of Contents

1 Introduction ... 11
1.1 Problem Statement ... 11
1.2 Objective of the Thesis ... 12
1.3 Related Work ... 13
1.4 Outline of the Thesis ... 15
1.5 Acknowledgements ... 15
2 Introduction to Intrusion Detection Systems ... 16
2.1 Choosing Between an IDS or IPS Solution ... 16
2.2 Selection of IDS Sensors for Testing ... 20
2.2.1 Snort ... 21
2.2.2 Suricata ... 23
2.2.3 Bro ... 24
2.3 Comparison of Features Side-by-side ... 26
3 Testing Implementation ... 27
3.1 Environment ... 27
3.2 Factors Affecting IDS Performance ... 28
3.3 Input Data ... 28
3.4 Rule Set ... 30
3.5 Experiment Setup ... 31
4 Results ... 34
4.1 Experiment 1 - Default OS & IDS Configuration ... 34
4.2 Experiment 2 - Optimize IDS Configuration ... 36
4.2.1 Snort ... 37
4.2.2 Suricata ... 39
4.2.3 Bro ... 42
4.2.4 Optimization Results ... 42
4.3 Experiment 3 – Modify or Replace Network Packet Capture Module ... 44
4.3.1 Increase libpcap Buffer Size ... 44
4.3.2 Use AF_PACKET Network Socket ... 46
4.3.3 Use PF_RING Network Socket ... 48
5 Discussion of Results ... 52
5.1 Dropped Packets ... 52
5.2 CPU Usage ... 52
5.3 Memory Usage ... 53
6 Future Research ... 54
6.1 Snort (DAQ) with PF_RING ... 54

6.2 Use Suricata with Kernel Versions Above 3.1 ... 54
6.3 Experiment with Speeds Up To 10Gbit/s ... 54
6.4 GPU Processing ... 55
6.5 Compare Different Rule Sets ... 55
7 Conclusion ... 56
Résumé (in Estonian) ... 58
List of References ... 60
Appendices ... 63
Annex 1 – Hardware Specification ... 63
Annex 2 – Software Versions ... 65
Annex 3 – Suricata CPU Affinity Configuration ... 67

List of Figures

Figure 1: Possible placements of the IDS/IPS node on the network 16
Figure 2: Typical IDS deployment 18
Figure 3: IDS setup with the ability of active response 19
Figure 4: Typical IPS setup 20
Figure 5: Snort structure and operation [13] 22
Figure 6: PCAP file protocol statistics 29
Figure 7: PCAP file frame lengths in bytes 30
Figure 8: Percentage of dropped packets with default IDS configuration 34
Figure 9: Percentage of CPU utilized by each IDS in default configuration 36
Figure 10: Percentage of dropped packets after optimizations 43
Figure 11: Percentage of CPU utilized by Snort and Suricata after optimizations 43
Figure 12: Percentage of dropped packets using previous optimizations and a 2GiB libpcap buffer size 46
Figure 13: Percentage of dropped packets using previous optimizations and a 2GiB AF_PACKET buffer size 47
Figure 14: Percentage of CPU utilized by Suricata and Bro using PF_RING 51
Figure 15: IDS process memory usage in experiments 53

List of Tables

Table 1: Feature comparison of Snort, Suricata and Bro 26
Table 2: Percentage of dropped packets with default settings 35
Table 3: Percentage of dropped packets after optimizations 43
Table 4: Percentage of dropped packets using previous optimizations and a 2GiB libpcap buffer size 45
Table 5: Percentage of dropped packets using previous optimizations and a 2GiB AF_PACKET buffer size 47
Table 6: Percentage of dropped packets using previous optimizations and PF_RING .. 50

1 Introduction

The topic of this thesis is "A Comparative Analysis of Open-Source Intrusion Detection Systems". It will give a comprehensive comparison of three popular open-source intrusion detection systems and describe their ability to detect malicious activity.

1.1 Problem Statement

The Internet is a hostile environment for networked computers. What is more, computer network security has been an afterthought to combat all the exploits that have been discovered in the last decades. In the early days of the Internet the security relied on knowing and trusting the other person and their computers. However, when the Internet became available to the masses this was no longer possible; people could not always identify the other person or establish trust with them. Malicious users have taken advantage of this to achieve financial gain or accomplish some corporate or personal agenda. Curiosity and amusement are also possible reasons for malicious activity.

In spite of the many developments in IT security over the past years, cybercriminals have got bolder in their attacks and there has been a significant growth in the volume of malware and infections. Gerhard Eschelbeck (CTO at Sophos) has said "/…/ For 2012, I anticipate growing sophistication in web-borne attacks, even broader use of mobile and smart devices, and rapid adoption of cloud computing bringing new security challenges. The web will undoubtedly continue to be the most prominent vector of attack. Cybercriminals tend to focus where the weak spots are and use a technique until it becomes far less effective." [5]

Computer networks around the world are constantly being probed and attacked in an attempt to penetrate the security defenses and gain access to information on the network. Institutions maintaining these networks have to continuously monitor and adapt to the threats as they change to protect their users, information and other valuable assets from these attacks. However, it is increasingly difficult to keep up with the rapidly growing volume of network traffic and the number of attacks.

Firewalls are efficient for addressing a wide range of network filtering problems. However, firewalls make filtering decisions only based on network packet header data – packet content data are not inspected. Analyzing packet payload is often essential for detecting packets with malicious content.

This is where intrusion detection systems (IDS) can be helpful. An IDS monitors and logs the network traffic for signs of malicious activity and generates an alert upon discovery of a suspicious event.

There are many different intrusion detection systems available. One has to analyze the requirements and make a decision about which system fulfills their requirements the best.

The problem is that often there just is not enough information available to make a decision about which software to choose. Moreover, in recent years some new systems have entered the competition, so there is little information on how they compare to the older, more mature systems. Furthermore, these comparisons often come from people involved with some IDS community, so they often seem biased. To offer a solution, this thesis will focus on comparing some of the more popular open-source IDS solutions available at the moment.

1.2 Objective of the Thesis

This thesis is primarily focusing on open-source network-based intrusion detection systems, because maintaining host-based systems does not scale well in large networks. The aim is not to name the best open-source IDS available, but rather come to an unbiased conclusion about the advantages and disadvantages of different systems.

The main objective of this thesis is to present an overview of popular open-source IDS solutions and carry out their comparative evaluation with a number of tests. The tests were designed to satisfy the following conditions:

- Reliability – Reliable test results;
- Repeatability – Tests can be run again when needed;
- Reproducibility – Provide configuration instructions.

There are a number of different variables that affect the performance and reliability of an IDS setup. Some more clearly distinguished ones are the following:

- IDS engine effectiveness;
- IDS configuration;
- Quality and amount of the detection rules;
- Amount of network traffic;
- Host system performance.

The practical part of this thesis will focus on comparing how the performance and reliability of the IDS changes when IDS configuration and the amount network traffic is changed. This will be measured in the number of packets dropped. More detailed explanations are given further in the thesis.

Less will be focused on the accuracy (e.g. false positives/negatives) of the IDS solutions, because for definitive results this would require a separate analysis of the quality of the detection rules, which is not considered part of this thesis.

1.3 Related Work

The general idea of testing intrusion detection systems is not new or unique. However, intrusion detection is difficult to accomplish perfectly and that makes the IDS testing procedure an interesting research topic now and again.

Intrusion detection systems are different when it comes to software design (e.g. single- or multi-threaded). Additionally, each IDS offers many configuration and optimization possibilities. Characteristics of the analyzed network traffic and the underlying hardware performance have a clear impact on the overall IDS performance. Also, the amount and the quality of the detection rules are of key importance. Last but not least, newer versions of the IDS software and supporting packages can make a notable difference in test results.

There are several related research papers and thesis that will be analyzed and compared in this thesis.

There was an interesting master's thesis from Alar Kvell titled "A high-performance network intrusion detection solution for S4A software". The study focused on testing Suricata as a possible replacement for Snort in the S4A (Snort for All) software. S4A is an open-source network analyzer and intrusion detection system, which is used and maintained by CERT Estonia to detect security incidents in the networks of institutions like the government, local municipalities, police etc. The S4A hardware platform used in the testing environment was rather conservative by recent standards. It had a dual-core Intel processor and only 4GiB of memory. This certainly sets some limits for the IDS solutions. Furthermore, the testing rule set contained 5373 detection rules. The thesis concluded that Suricata slightly outperformed Snort in some configurations, while making better use of resources available on the system. [6]

A thesis by Eugene Albin compared Snort and Suricata in a variety of tests for performance, resource consumption and accuracy. The testing was performed on two separate systems. One a virtual machine running on a quad-core server with 96GB memory, the other a node on the Hamming supercomputer with 48 AMD 12-core CPUs with 125GiB of memory available. Results regarding performance concluded that Suricata has a high processing overhead compared to Snort, due to Suricata's multi-threaded design. However, on systems with plenty of processing resources, Suricata is able to analyze network traffic at a much higher rate. In terms of detection engine accuracy, the author E. Albin noted that a definitive answer was not reached, because the accuracy and the effectiveness of the rules were not verified as part of the thesis. [7]

Additionally, several smaller articles were examined while working on this thesis. These were mostly in the form of a blog post and focused on testing and improving only one IDS solution at a time. While not giving any comparative results between solutions, they still provided useful tips and knowledge about these systems. Some of the more important articles are mentioned in the following paragraph.

A white paper by Steven Sturges on the topic of tuning Snort configuration to improve performance shared important pointers that the user manual did not cover as thoroughly. [8] A post from Éric Leblond, a member of the OISF team provided important configuration tips for running Suricata on networks as fast as 10Gbit/s. [9] Leblond also determined that hyper-threading technology on Intel CPUs might cause variations in results and therefore using fixed CPU affinity might result in better and more stable

performance. [10] Martin Holste published an article on running the single-threaded Bro IDS with multiple worker processes to cope with higher bandwidth than a single process could handle. [11]

Finally, there is also a Linux distribution called Security Onion, which is dedicated to intrusion detection and network security monitoring. It is based on Ubuntu and contains Snort, Suricata and Bro IDS engines along with many other security analysis tools. It is a good example on how to configure IDS solutions and an easy-to-use quick start solution for someone, who does not want to delve into compiling and installing dependencies for the IDS solution to work. [12]

1.4 Outline of the Thesis

This thesis is organized as follows. Chapter 1 gives an introduction to the thesis and states the problem that the thesis handles. Thesis objectives and previous work on related topics are also discussed. In chapter 2 we describe intrusion detection systems in general. A selection of popular open-source IDS solutions is made and each solution is analyzed. Next, chapter 3 gives an overview of the testing environment and describes the testing procedure. Input data and rule set used during the testing is analyzed. The fourth chapter is devoted to test results and describes any configuration changes that were made to improve IDS performance. Results are usually given as graphs or tables and are briefly interpreted. Chapter 5 briefly outlines the test results again to give a better overview of the improvements. In chapter 6 we offer suggestions for future research topics that were not handled in this thesis. And finally, chapter 7 gives a summary and concludes the thesis.

1.5 Acknowledgements

We would like to thank Elion Ettevõtted Aktsiaselts for providing the testing hardware and environment.

2 Introduction to Intrusion Detection Systems

This chapter will give an overview of different open-source intrusion detection systems. For better understanding of different topics discussed later in this thesis, this section will also compare basic differences between intrusion detection and intrusion prevention systems.

2.1 Choosing Between an IDS or IPS Solution

IDS/IPS deployment typically consists of one or more sensors placed strategically on the network (see Figure 1). Additionally, the solution may contain an optional central console for easier management of all sensor nodes. The sensor placement on the network can of course differ, but in a situation where the objective is to protect internal network from external threats, these would be the optimal choices for the IDS and IPS nodes. It is sensible to place IDS/IPS sensor after the firewall for incoming traffic, because it is not necessary to analyze and trigger alarms for traffic that the firewall would block anyway.

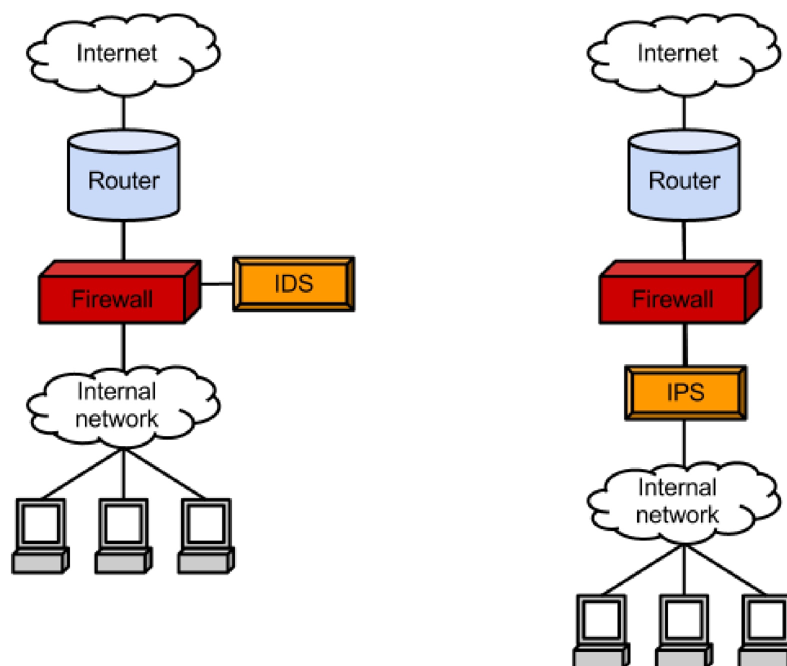

Figure 1: Possible placements of the IDS/IPS node on the network

There are several different methods of physically connecting an IDS sensor to the monitored network [13]:

- **Monitoring interface** – Usually a configurable interface on a network device (switch, router, firewall, etc.), which copies all packets passing the device to that interface;
- **Network tap** – A dedicated device on a network link that transparently mirrors all packets on the link to the IDS sensor;
- **Ethernet hub** – A simple device which rebroadcasts all incoming traffic to other ports on the device. It is important to note that this is not a good solution if all network traffic passes through the device. However, it can be useful for replicating a single monitoring interface in a fairly low-bandwidth network.

Typically setting up an IDS sensor involves connecting the node's one network interface to the monitored network segment and the other interface to the network where the central management console can be accessed (e.g. the internal network segment). The interface connected to the monitored network does not need an IP address to function, but has to be set into promiscuous mode to listen for all packets transmitted on the network link.

The IDS engine analyzes packets collected from the interface in promiscuous mode. The criteria which packets should trigger an alert are usually specified as rules. Alerts are logged and sent to the central console or directly to the security analyst responsible for the system. It is important to note that IDS systems have no effect if triggered alarms are not monitored by someone on a daily basis.

See Figure 2 for a descriptive schematic on a typical IDS sensor deployment and operation.

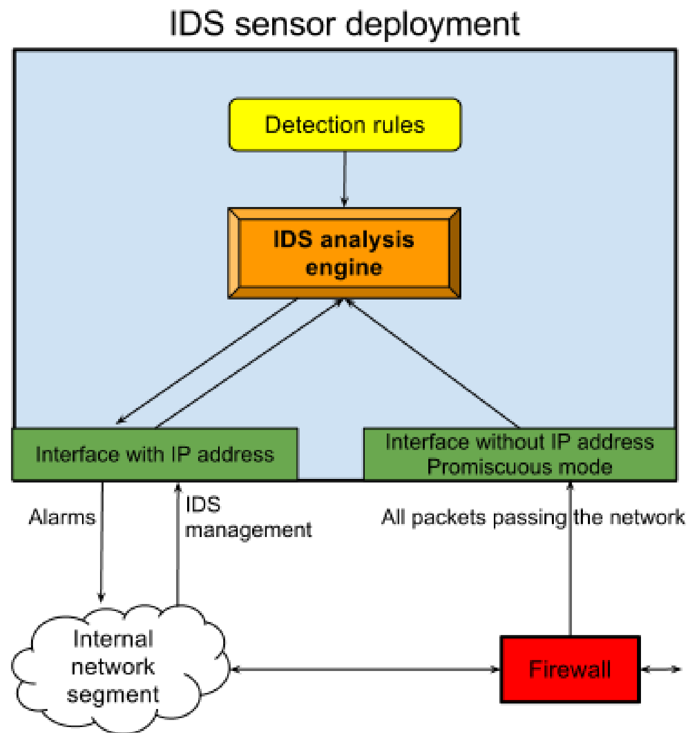

Figure 2: Typical IDS deployment

Bear in mind that pure IDS deployments cannot protect networks on their own. They can only alert the security analyst that a malicious activity took place at a certain time. Therefore IDS sensors are sometimes augmented with capabilities for firewall interaction. For example, block the source IP address of a DoS attack. However, this is a post-factum measure that cannot stop the malicious packets that triggered the creation of the firewall rule.

Additionally, it is important to remember that these blocking rules have to be tested very thoroughly in order to avoid false negatives, which can result in being falsely blocked out of the network.

See Figure 3 for a descriptive schematic on an IDS sensor with an active firewall response.

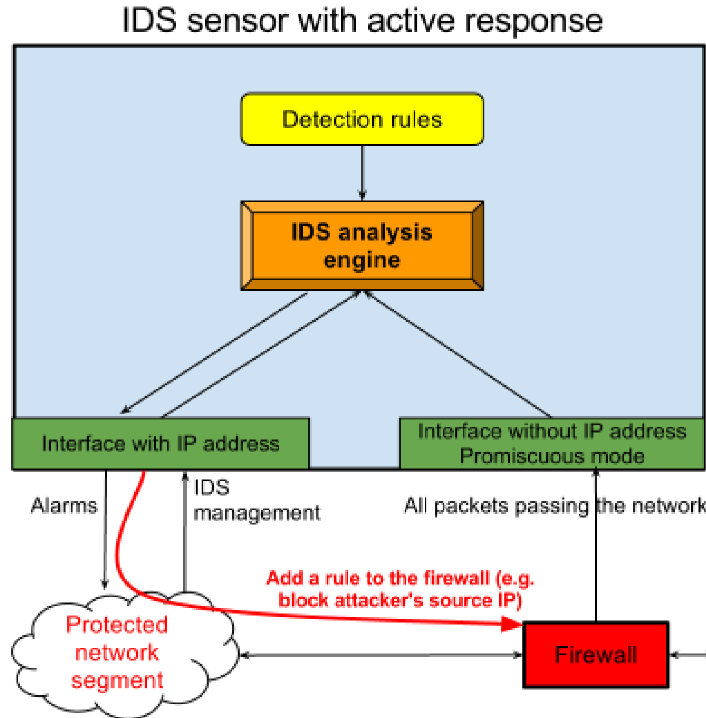

Figure 3: IDS setup with the ability of active response

In order to improve the level of protection even more, many IDS solutions can also be configured to run in intrusion protection system (IPS) mode. IPS node is connected inline to the network segment. IPS sensor acts as an OSI layer 2 device. All traffic on the network goes through the analysis engine, which decides if the packet is forwarded or not. Dropped packets are usually logged. IDS functionality remains in a way, that some rules might not drop packets but only produce alerts.

See Figure 4 for a descriptive schematic on a typical IPS sensor deployment and operation.

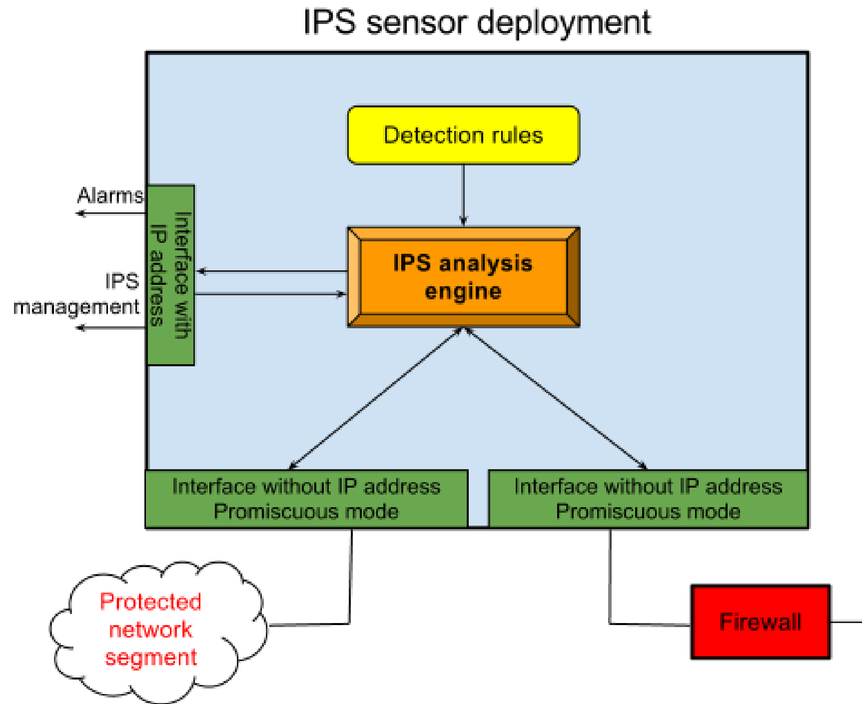

Figure 4: Typical IPS setup

In short, IDS is a network security monitoring solution that does not interfere with network traffic, but reports incidents to central console. These incidents have to be investigated for possible consequences by the security analyst.

Many IDS solutions can also be configured to function as intrusion protection systems. IPS sensor can protect the network against threats as they are detected. Although, IPS solution can reduce personnel workload, it requires well-tested and very precise rules to avoid any false negatives.

Furthermore, it is important to choose a solution, whose vendor puts in the effort to develop the product, while regularly updating and testing the rules.

2.2 Selection of IDS Sensors for Testing

There are many open-source and proprietary intrusion detection systems available. However, it could prove difficult to configure, manipulate and measure the effectiveness of a proprietary product. By any means, this is not to say that any proprietary solution does not offer this functionality, but for sake of equal comparison, this thesis will only be focusing on open-source IDS solutions.

To limit the scope and not lose focus, this thesis will only concentrate on the first described solution, which is the typical IDS sensor deployment (see Figure 2 in the previous chapter). This setup will be thoroughly tested on a selection of open-source IDS sensors. The testing procedure and the results will be discussed in the following chapters.

Research into which open-source solutions have been used around the world and which of them are still actively developed resulted in three IDS engines named Snort, Suricata and Bro. Related works on similar topic mentioned above have also tested these IDS engines.

The search revealed that Snort is a popular IDS that has been around for more than a decade. It is probably the most well-known open-source IDS software available today. [14]

In recent years Suricata has often been compared with Snort. The two systems share the same rule syntax, but Suricata seems to be favored by the multi-threaded design compared to Snort’s single-threaded analysis engine. It would be interesting to see how they perform against each other. [15]

Finally, there is also an IDS sensor and network analyzer called Bro. It is definitely not a new software, its research and development dates back more than 15 years. [16]

Next we will describe and analyze each of the three solutions named above in greater detail.

2.2.1 Snort

Snort is an open-source intrusion detection system that is developed by Sourcefire. Snort was created in 1998 by Martin Roesch. It is capable of performing real-time traffic analysis and packet logging on IP networks. Snort is compatible with most operating systems (e.g. Linux, Mac OS X, FreeBSD, OpenBSD, UNIX and Windows). [17]

The Snort detection engine and the Community Snort Rules are GNU GPL v.2 licensed. Sourcefire also offers proprietary Snort Rules which are licensed by Non-Commercial Use License. [17]

The two major components of Snort are the following: [17]

1. Detection engine that utilizes modular plug-in architecture;
2. Flexible rule language to describe traffic to be collected.

Snort structure is illustrated on Figure 5. The preprocessor, the detection rules, and the alert output components of Snort are all plug-ins, which can be individually configured and turned on or off.

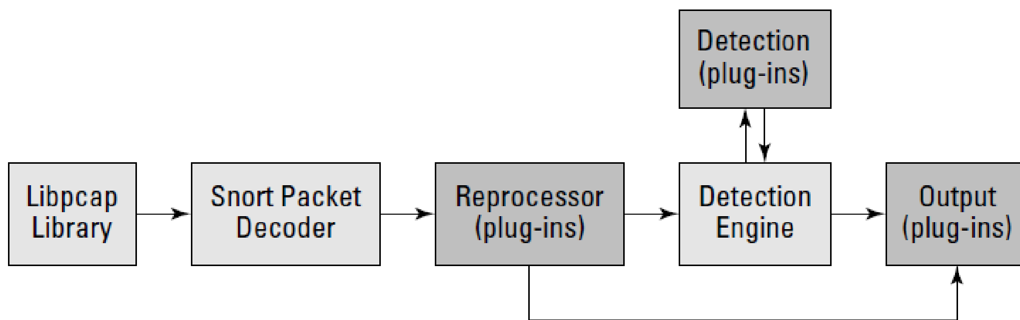

Figure 5: Snort structure and operation [13]

Figure 5 also shows how a network packet is handled if it is received by the network interface on which Snort is listening. The handling process is similar for all three chosen IDS solutions, but will be described here using Snort as an example.

1. **Packet capture library** is a software module that gathers packets from the network adapter. On UNIX and Linux systems Snort uses libpcap library. On Windows systems WinPcap is used.
2. **Packet decoder** receives the OSI layer 2 frame, analyzes packet headers and looks for any anomalies. Packet data is then decoded and prepared for further processing.
3. **Preprocessors** are plug-ins that operate on the decoded data. Preprocessors can alert on, classify, or drop a packet before sending it to the more CPU-intensive detection engine. By default, Snort comes with a variety of preprocessors, some of which are the following.
    a. Frag3 preprocessor addresses problem of overlapping fragmented IP packets that could be used to avoid IDS/IPS detection.

   b. Stream5 preprocessor makes Snort state- and session aware. For instance, it can detect out-of-state packets created by Nmap tool.
   c. HttpInspect preprocessor handles HTTP traffic. It extracts compressed data and decodes any hexadecimal or other expressions in the Universal Resource Identifier (URI).
4. **Detection engine** is the most important part of Snort. It operates on the OSI transport and application layers, analyzing packet contents based on the detection rules. The rules contain signatures for attacks.
5. **Output plug-ins** support a variety of alert and logging methods. When a preprocessor or rule is triggered, an alert is logged in Snort's own text or binary file logging formats, database or syslog.

Snort uses a single-threaded engine, which seems outdated, considering that nowadays multi-CPU and multi-core hardware is commonplace. As a result, by default Snort can only fully utilize one processor core. Snort developers are working on multi-threaded solution, however stable version has not yet been released. To alleviate this problem Snort can be run as multiple processes; each process utilizing a different processor core. This, however, increases the level of complexity, because the default network socket packet capture library needs to be replaced. [18]

2.2.2 Suricata

The Suricata Engine is a fairly new open-source intrusion detection and prevention engine. The initial beta release was made available for download on January 1, 2010. It is developed by Open Information Security Foundation (OISF), which is a non-profit foundation supported by the US Department of Homeland Security (DHS) and a number of private companies. [15] [19]

Suricata is compatible with most operating systems (e.g. Linux, Mac, FreeBSD, UNIX and Windows). The Suricata Engine is available to use under the GPL v.2 license. [15]

OISF claims that "The Suricata Engine is not intended to just replace or emulate the existing tools in the industry, but will bring new ideas and technologies to the field". However, the industry considers Suricata a strong competitor to Snort and thus they are often compared with each other. Both systems seem to have their advantages and strong community support. [19]

The operation modes of Suricata are the same as Snort's. It can be used either as an IDS or IPS system. There are no differences when connecting Suricata to the network. Suricata even has basically the same rule syntax as Snort (although not 100%), which means that both systems can use more or less the same rules.

The general data flow through Suricata is similar to Snort. Packets are captured, decoded, processed and analyzed. However, when it comes to the internals of the Suricata Engine, differences become apparent.

Suricata also features the HTP Library that is a HTTP normalizer and parser written by Ivan Ristic for the OISF. This integrates and provides advanced processing of HTTP streams for Suricata. The HTP library is required by the engine, but can also be used as an independent tool. [19]

Suricata uses a multi-threaded approach opposed to the Snort's single threaded engine. Threads use one or more Thread Modules for this. Threads have an input queue handler and an output queue handler. These are used to get packets from other threads, or from the global packet pool. [19]

Taking these few, but significant differences into account, it is probable that Snort and Suricata perform differently when it comes to the speed and efficiency of network traffic analysis. This will be tested later in the practical phase of this thesis.

2.2.3 Bro

Bro intrusion detection system is focusing on network security, but also provides a comprehensive platform for more general network traffic analysis. Bro has been developed over 15 years. Bro was created by Vern Paxson, who is still leading the project jointly with a team of researchers and developers at the International Computer Science Institute (ICSI) in Berkeley and the National Center for Supercomputing Applications in Urbana-Champaign. [16] [20]

Bro and its pre-written policy scripts ("rules") come with a BSD license, allowing free use with even less restrictions than the GPL v.2 license of Snort and Suricata. [16]

Moreover, it is important to note that Bro policy scripts ("rules") are written in its own Bro scripting language that does not rely on traditional signature detection. It analyzes

network while trying to detect anomalies, e.g. attacker installing hacked SSH daemon. It is said that: “Bro language takes some time and effort to learn, but once mastered, the Bro user can write or modify Bro policies to detect and alert on virtually any type of network activity.” [21]

Bro is not a full-blown IPS, but can function as an IDS with active response. Its policy scripts have the functionality to execute programs, which can, in turn, perform a variety of tasks (e.g. send e-mail or SMS, insert new rules to the firewall). [21]

Furthermore, Bro comes with a useful tool called BroControl which enables the administrator to manage multiple Bro nodes at once. In addition to being able to controlling the Bro instances, it could even execute shell commands on all nodes.

Similar to Snort, Bro is also single-threaded. Although, the developers of Bro have implemented a proof-of-concept multi-threaded version of Bro, it is not yet ready for release. Therefore, once the limitations of a single processor core are reached, the only option is to spread the workload across many cores or even many physical nodes. The accompanying BroControl tool provides the means to easily manage many Bro processes. However, similar to Snort, this method significantly increases the level of system complexity. [22]

Interestingly, Bro does not seem to have gained the popularity of Snort or Suricata. Maybe this is due to the fact that the developers of Bro have stayed clear of the debates that people behind Snort and Suricata have engaged in. [21]

2.3 Comparison of Features Side-by-side

Now that each of the intrusion detection systems has been described independently, a descriptive table (see Table 1) that would give an overview of the different parameters can be assembled.

Table 1: Feature comparison of Snort, Suricata and Bro

Parameter	**Snort**	**Suricata**	**Bro**
IPS feature	Snort_inline or snort used with -Q option	optional while compiling (--enable-nfqueue)	No
Rules	VRT::Snort rules SO rules EmergingThreats rules	VRT::Snort rules EmergingThreats rules	Pre-packaged scripts
Threads	Single-thread	Multi-thread	Single-thread
Installation complexity	Relatively straightforward. Installation also available from packages.	Relatively straightforward. Not available from packages (except Ubuntu).	Relatively straightforward. Installation also available from packages.
Documentation	Well documented on the official website and many resources on the Internet.	Well documented on the official website. Some resources on the Internet.	Satisfactory documentation on the official website. Very few resources on the Internet
Event logging	Flat file, database, unified2 logs		Flat file, database, barnyard2 integration
IPv6 support	Yes, when compiled with --enable-ipv6 option.	Yes	Yes
Capture accelerators	Yes (e.g. PF_RING)		
Offline analysis (pcap file)	Yes, multiple files consecutively	Yes, only single file	Yes, only single file
Frontends	Sguil, Aanval, BASE, FPCGUI (Full Packet Capture GUI), Snortsnarf		Brownian
License	GNU GPL v.2		BSD

Table 1 shows that all three intrusion detection systems have their merits and there is no system with a clear advantage over the others.

3 Testing Implementation

In this chapter the testing environment and procedure is described. Test results will be given in chapter 4.

3.1 Environment

The testing infrastructure consisted of three rack-mounted servers and one manageable switch. The hardware specification of the server that was running the IDS software is of key importance and will be listed here. More detailed information about the testing environment can be found in the Annex 1 – Hardware Specification.

The server running the IDS software was a Hewlett-Packard ProLiant DL320 Generation6 server in the following hardware configuration.

- Single CPU - Intel® Xeon® Processor E5630 (12M Cache, 2.53 GHz);
  - 4 cores;
  - 8 threads (Hyper-Threading enabled).
- 72GB PC3-10600 (DDR3-1333) Registered Memory;
- NIC: Two embedded NC373i Multifunction Gigabit Network Adapters;
- RAID1 – 2x Seagate Barracuda 750GB, 3.5” LFF, 7200RPM, 16MB cache, SATA 3.0Gb/s.

Each IDS was set up on an individual CentOS 6.3 (64-bit) installation to avoid any anomalies or uncertainty from installing multiple intrusion detection systems side-by-side. GNU GRUB boot loader was used to load the different operating systems installations.

Note that some of the testing and tweaking was performed on Oracle VM VirtualBox virtual machines to make use of the ability to create snapshots and easily roll back changes. This was mostly to verify how any configuration changes affect the whole system, before issuing them in the actual physical testing environment.

3.2 Factors Affecting IDS Performance

Judging by the info gathered from the previous chapters, conclusion can be made that the performance of intrusion detection systems depends mostly of the following factors.

1. Software architecture and implementation;
    - 1.1. Detection algorithm optimization;
    - 1.2. Software configuration options;
    - 1.3. Amount of modules/preprocessors loaded;
        - 1.3.1. Preprocessor configuration.
    - 1.4. Operating system's network socket packet capture module efficiency;
        - 1.4.1. PCAP
        - 1.4.2. AF_PACKET
        - 1.4.3. PF_RING
2. Hardware performance;
    - 2.1. CPU;
    - 2.2. RAM;
    - 2.3. NIC speed;
    - 2.4. HDD speed;
    - 2.5. Bandwidth of buses connecting the previous elements.
3. Detection rules;
    - 3.1. Amount of rules loaded;
    - 3.2. Quality (efficiency) of loaded rules.

3.3 Input Data

A way to test how all of these factors come together is by testing different IDS setups at different transmission speeds with identical pre-captured network traffic in PCAP format. To accomplish this, we will be using software named tcpreplay to replay the PCAP file to the network for the IDS to analyze. Transmission speeds will also be regulated by tcpreplay software.

The PCAP file ictf2010pcap.tar.gz used in this thesis was acquired from the International Capture The Flag (iCTF) 2010 security exercise [23].

```
md5sum:
b4ecab2caf05420a8ab372b1afaf3e73  ictf2010pcap.tar.gz
```

The file was an archive of network traffic capture session that was split into 79 PCAP files. When extracted, the files were 37GiB in size. However this amount of data would have taken too long for the slower tests. For this reason only 30 first PCAP files were selected and merged for the tests forming a 14.36GiB file. The packets in the PCAP files were originally in RAW format and were modified to Ethernet packet by tcprewrite software to be compatible with our thesis testing environment.

The PCAP file used in each test has the following characteristics:

- Packets (total): 26,316,950;
- Protocol statistics (see Figure 6):
  - TCP: 93,67%;
  - UDP: 4,62%;
  - ICMP: 1,71%;
- Frame length (see Figure 7 on the next page):
  - 40-79 bytes: 10,759,421 packets;
  - 80-159 bytes: 2,042,414 packets;
  - 160-319 bytes: 3,113,723 packets;
  - 320-639 bytes: 1,508,691 packets;
  - 640-1279 bytes: 183,920 packets;
  - 1280-1500 bytes: 8,708,781 packets.

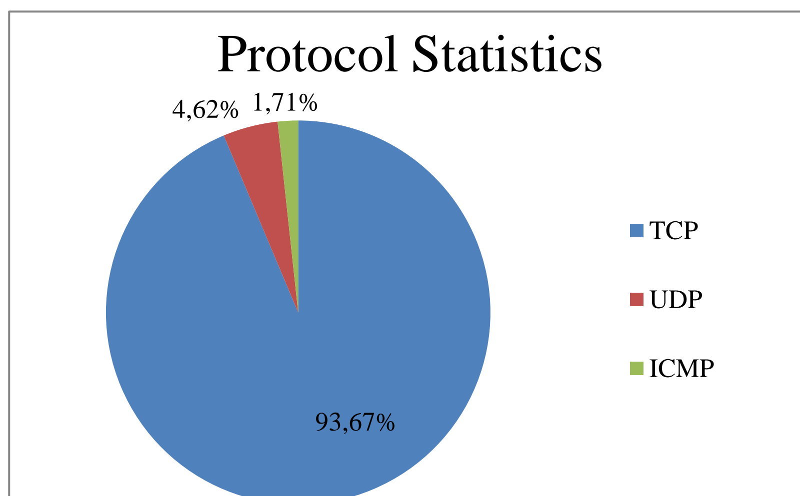

Figure 6: PCAP file protocol statistics

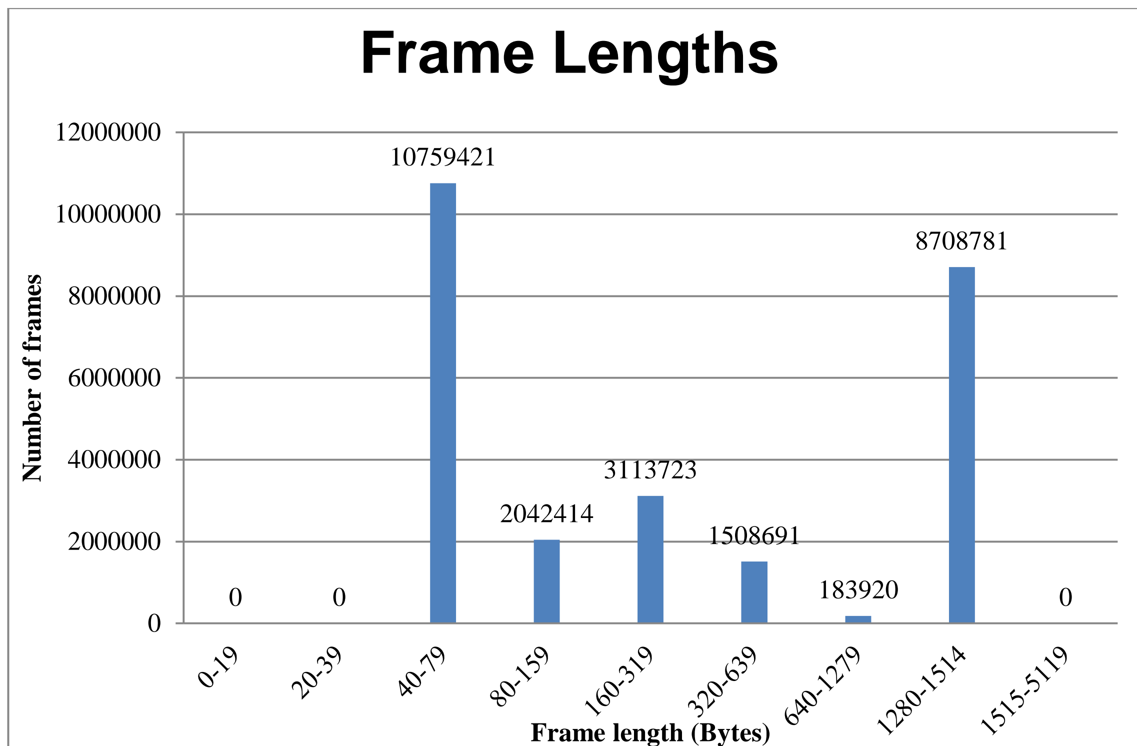

Figure 7: PCAP file frame lengths in bytes

The PCAP file is befitting for a company or data center network, where most of the traffic uses TCP protocol. Packet lengths and other statistics also seem reasonable, thus this will be the file that will be used for testing all the IDS engines.

3.4 Rule Set

In order to achieve comparable performance results, Snort and Suricata were both using Emerging Threats detection rules dating from November 15, 2012 17:53. The amount of rules is roughly the same for both files – 11,719 for Snort and 11,703 for Suricata.

Additionally for Snort, Preprocessor and Shared Object rules dating from October 30, 2012 were also downloaded packaged as snortrules-snapshot-2931.tar.gz. Detection rules from this package were not included into Snort configuration.

Snort:
http://rules.emergingthreats.net/open/snort-2.9.0/emerging-all.rules

http://www.snort.org/downloads/2063

```
md5sum:
92d2f0ae03b98d6a014dc378b81aaf00  emerging-all.rules
```

```
ae8def31513acb12dc21d72c7e88a10a  snortrules-snapshot-
2931.tar.gz
```

Snort output regarding loaded rules:

```
11719 Snort rules read
    11719 detection rules
    0 decoder rules
    0 preprocessor rules
11719 Option Chains linked into 351 Chain Headers
0 Dynamic rules
```

Suricata:
http://rules.emergingthreats.net/open/suricata-1.3/emerging-all.rules

```
md5sum:
4e72c97bce5569d18884341549314797  emerging-all.rules
```

Suricata output regarding loaded rules:

```
1 rule files processed. 11703 rules succesfully loaded, 0
rules failed
11711 signatures processed. 4 are IP-only rules, 3809 are
inspecting packet payload, 9484 inspect application layer,
0 are decoder event only.
```

Unfortunately, Bro uses different rule (policy) syntax and there are no Emerging Threats rules available for Bro. Therefore, we are unable to test Bro with similar rule set. However, Bro ships with many pre-written policy scripts that are suitable for most analysis needs. Scripts are customizable to support traffic analysis for specific requirements. [24]

3.5 Experiment Setup

In the practical phase of this thesis all three IDS solutions were tested in the following experiments.

1. Default OS & IDS configuration (only necessary changes – e.g. select correct network interface);
2. Optimize each IDS configuration by consulting manuals and online discussions;

3. Modify or replace network socket packet capture module to improve capturing performance;
    - Increase libpcap buffer size;
    - Use AF_PACKET network socket;
    - Use PF_RING network socket.

The experiments were carried out with the following transmit speeds set by the tcpreplay software All tests were performed at least three times to detect and avoid any anomalies in the results.

It is important to note that tcpreplay tool allows specifying the maximum transmission speed (using `--mbps`), which it will try to match, but will not exceed. However, testing revealed that depending on the network traffic characteristics inside the PCAP file, the actual speeds are somewhat slower. Testing also revealed that for each defined maximum speed, the average for each test cycle is always roughly the same (differences under ± 5Mbit/s). This actual tested average transmission speed for each replay cycle is indicated in the parenthesis.

- 50Mbit/s (50Mbit/s);
- 100Mbit/s (99Mbit/s);
- 150Mbit/s (147Mbit/s);
- 200Mbit/s (193Mbit/s);
- 300Mbit/s (277Mbit/s);
- 400Mbit/s (358Mbit/s);
- 500Mbit/s (433Mbit/s);
- 600Mbit/s (493Mbit/s);
- 700Mbit/s (562Mbit/s);
- 800Mbit/s (618Mbit/s);
- 900Mbit/s (655Mbit/s);
- *As fast as possible – 1,000Mbit/s* (895Mbit/s)

Interestingly, when setting the maximum transmission speed to 1,000Mbit/s or even more, tcpreplay was not actually able to reach the speeds achieved in the “as fast as possible” (`--topspeed`) mode. Nevertheless, this mode will be denoted as 1,000Mbit/s further on in the thesis.

After each test the following resulting information was checked and noted.

- Actual average transmit speed;
- Number of packets received;
- Number of packets dropped;
    - By the operating system kernel;
    - By the IDS.
- Average CPU usage;
- Memory usage.

We also monitored hard disk read and write operations, but since we were only writing generated log information to the hard disk and not the whole captured network traffic, we verified that our HDD configuration was fast enough to handle these operations. Therefore, further analysis for HDD usage seemed unnecessary.

4 Results

This chapter describes the test results. For each test cycle many performance indicators (e.g. CPU usage, memory usage, analyzed/dropped packages) were gathered, but in order to be concise and focused, the level of detail presented for each experiment had to be adjusted accordingly.

A performance baseline will be established in experiment 1. For the following experiments, changes made to the systems will be described and the results will be compared to the previous measurements.

4.1 Experiment 1 - Default OS & IDS Configuration

In first part of testing, all three intrusion detection systems were set up following required steps in corresponding installation manuals for Snort [25], Suricata [26] and Bro [24].

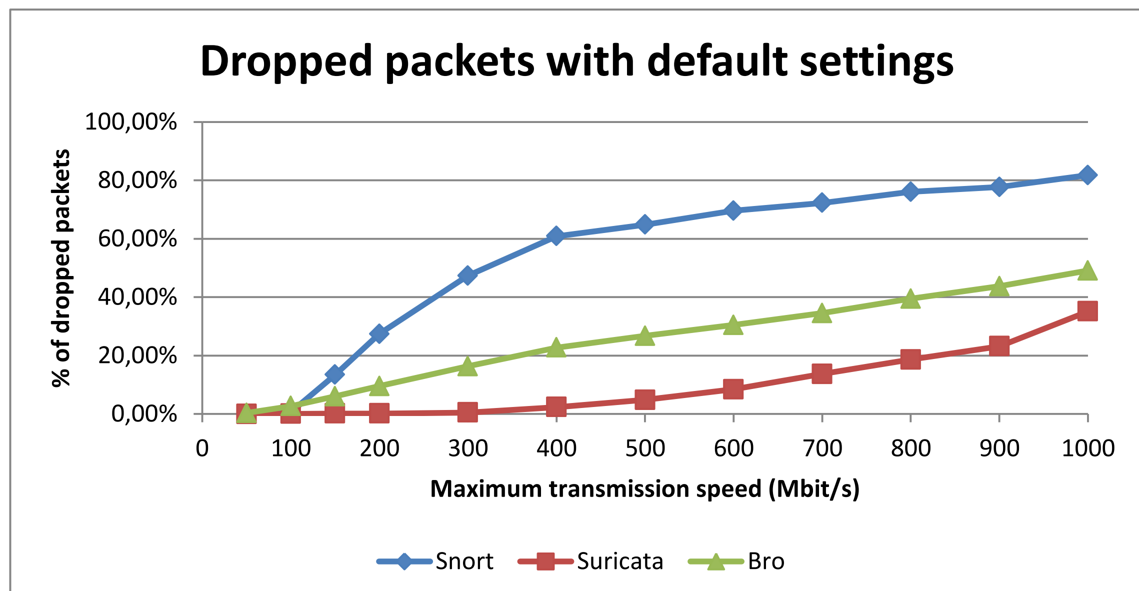

Figure 8: Percentage of dropped packets with default IDS configuration

Figure 8 shows that all IDS solutions in their default configuration can handle bandwidth up to 100Mbit/s with little to no dropped packets. At higher speeds results start to change drastically. Snort drops packets at the highest rate of the three systems. Bro comes in second and Suricata achieved the best results. More detailed results can be seen in Table 2.

Table 2: Percentage of dropped packets with default settings

	50M	100M	150M	200M	300M	400M	500M	600M	700M	800M	900M	1000M
Snort	0.0%	0.6%	13.4%	27.4%	47.3%	60.9%	64.8%	69.6%	72.3%	76.0%	77.7%	(81.7%)
Suricata	0.0%	0.0%	0.1%	0.2%	0.5%	2.3%	4.8%	8.4%	13.7%	18.6%	23.2%	(35.1%)
Bro	0.3%	2.6%	6.0%	9.5%	16.3%	22.7%	26.7%	30.5%	34.5%	39.4%	43.7%	(49.1%)

It is important to note that at the fastest transmission speed (1,000Mbit/s) about 2,000 packets were dropped on the kernel level before reaching the Suricata software. For Snort this was around 90,000 packets. However, this is only a fraction (0.0077% and 0.35% correspondingly) of all packets and could thus be considered of little magnitude in the overall statistics. Nevertheless, these values have been included in the statistics above, hence the value in parenthesis. Reason for this could be because of buffers filling up faster than software could read from them. At other transmission speeds, this problem did not occur.

Reason, why Snort and Bro started dropping packets much faster, is because they are single-threaded and their single instances were overwhelmed by the traffic. This is distinctly illustrated on Figure 9.

To explain the CPU utilization graph on Figure 9, it is important to bear in mind that this host has a quad-core CPU with Hyper-Threading enabled, which means that 8 logical processors are available to the operating system. When one of the eight logical processors is fully utilized, the overall CPU utilization is 12.50% (i.e. 1/8).

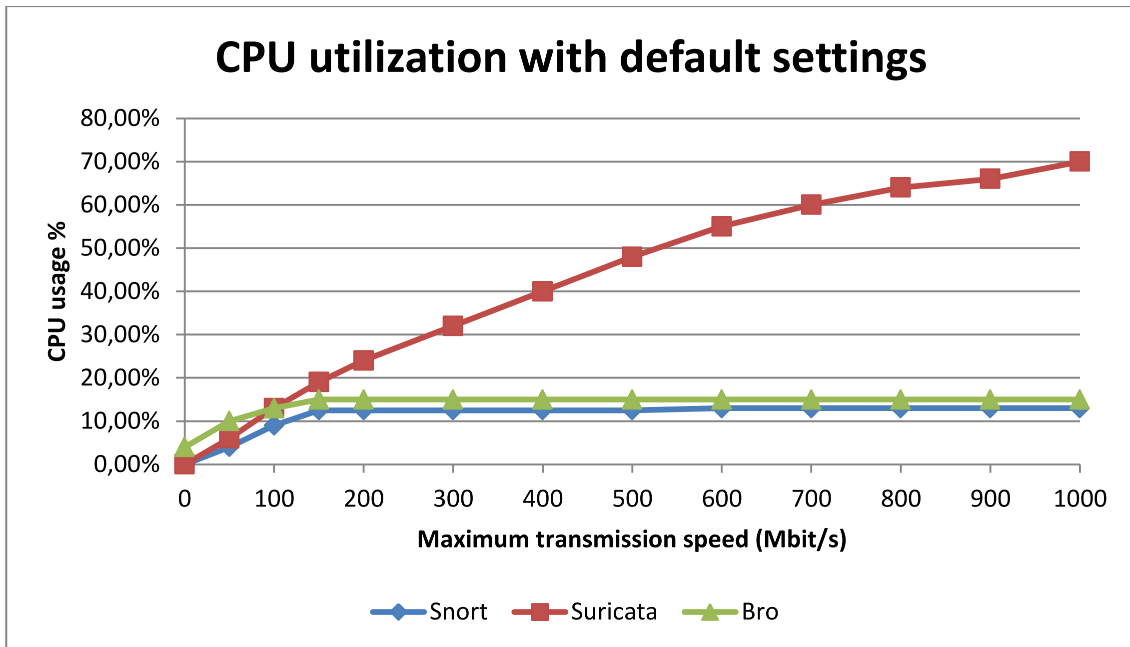

Figure 9: Percentage of CPU utilized by each IDS in default configuration

The graph clearly depicts that single-threaded Snort is only able to make use of one processor. Even as the transmission speed increases, the CPU load that Snort generates does not exceed 12.50%.

Bro's detection engine is single-threaded as well, but its manager and communications protocol, which are running as a separate process, use some additional resources. It is important to note that this communication protocol loop creates some load (about 4% CPU utilization) on the system even when no traffic is being analyzed.

Suricata demonstrates that it is able to make use of more processing power than the other two systems. The CPU usage increases linearly. However, it seems that there is still room for improvement, since it is dropping some of the packets and not using the CPU at full capacity.

4.2 Experiment 2 - Optimize IDS Configuration

The first experiment revealed that the default configuration for IDS solutions is not optimal for analyzing high-speed network traffic exceeding 100Mbit/s. Therefore, this chapter focuses on simply changing the IDS configuration to achieve better results.

Changes made to the configuration in order to improve performance are derived from consulting best-practice guides, user manuals and online support forums. Any specific changes will be noted in their corresponding chapters. Results will be outlined in chapter 4.2.4 Optimization Results.

It is important to note that intrusion detection system optimization largely depends on the network traffic characteristics (packet size, session length, protocols, etc.) and what are the systems the IDS is supposed to be protecting (web server, mail server, etc.). In this thesis we tried to be applicable for most solutions, therefore we did not deliberately discard any traffic, but rather tried to analyze as much as possible while remaining reasonable.

4.2.1 Snort

This chapter focuses on changes made Snort configuration and how it affected its performance.

Interestingly, many performance improvement suggestions for Snort are actually about excluding specific traffic from being analyzed by the CPU-intensive detection engine. This can be done by using special preprocessors or just excluding certain ports. It can be a good way to optimize, however one has to be careful, because this can potentially lead to ignoring malicious packets. This tradeoff must be considered by every IDS administrator. There are also some suggestions for improving the performance by increasing memory limitations in the Snort configuration. These suggestions are discussed and tested in this chapter. [8] [27]

Warning messages regarding exceeded memory cap

During the first experiment Snort (Stream5 preprocessor) produced warning messages that a preprocessor memory cap was reached and therefore some of the sessions were pruned (not fully analyzed) to free memory for new ones.

Following warning messages were reported by Snort:

```
S5: Pruned X sessions from cache for memcap.
S5: Session exceeded configured max bytes to queue X using
10yyyyy bytes (client queue).
S5: Pruned session from cache that was using 10yyyyy bytes
(stale/timeout).
```

To overcome this problem, the Stream5 preprocessor global memory limit was raised to 512MiB (default is 8MiB). The number of simultaneous sessions to track was set to 1,048,576 (maximum supported by Snort) for TCP and to 524,288 for UDP.

```
preprocessor stream5_global: memcap 536870912, \
   max_tcp 1048576, \
   max_udp 524288, \
```

Additionally, Stream5 TCP module was configured to allow up to 32MiB or 1,048,576 segments to be queued for reassembly of any TCP session. However, in terms of performance it might not be a good idea to increase the reassembly limit too high, because most detection signatures only focus on the beginning of a stream. For instance, in most cases there is no point to reassemble the whole packet stream of a 4,7GiB DVD image download.

```
preprocessor stream5_tcp: max_queued_bytes 33554432,
max_queued_segs 1048576, ...
```

Further increasing the memory cap and limits for Stream5 preprocessor did not produce verifiable improvement in results. However, as mentioned above, this configuration largely depends on the characteristics of the traffic.

Detection engine pattern matcher algorithm

By default Snort detection engine uses ac-split (Aho-Corasick Full with separate ANY-ANY port group) fast pattern matcher algorithm search method. According to Snort user manual, this method offers high performance with relatively low memory consumption. It is a good compromise between memory consumption and performance. [28]

The detection engine was tested with all the search methods available. As also pointed out by Snort user manual, the ac (Aho-Corasick Full) queued search method showed the best results in performance (about 15% less dropped packets). However, it also requires the highest amount of memory of all the search methods. Compared to ac-split method, additional 1.4GiB was utilized with the given set of aforementioned rules. Since our test host had 72GiB of memory, this was not a problem.

```
config detection: search-method ac search-optimize max-
pattern-len 20
```

HTTP preprocessor http_inspect optimization

In Snort configuration that came with the default installation there was no limit set to how much Snort would analyze HTTP traffic. This turned out to load the CPU heavily for large HTTP transfers.

```
server_flow_depth 0 \
client_flow_depth 0 \
```

Snort user manual revealed that Snort's initial values for these two are 300 bytes. This means that Snort will only inspect the first 300 bytes of the client request or server response packet. We did not want to risk missing any potential attacks, so we increased the client flow depth to 1,460 bytes, which is the maximum that could be specified for this flow. Snort rules usually only analyze HTTP packet headers, so this limit would be reasonable in most cases.

```
server_flow_depth 300 \
client_flow_depth 1460 \
```

Results in our tests were impressive. Overall there was about 30% less dropped packets. This can of course differ when analyzing traffic that contains different proportion of HTTP traffic.

HTTP preprocessor can be optimized even further. For example, by disabling unlimited payload decompression and limiting it to the Snort default values of 1,460 and 2,920 bytes, 4-5% less packets are dropped.

```
compress_depth 1460 \
decompress_depth 2920 \
```

4.2.2 Suricata

This chapter focuses on changes made Suricata configuration and how it affected its performance.

Warning messages regarding exceeded memory cap

Similar to Snort in the first experiment, Suricata (Flow engine) produced warning messages that a memory cap was reached and therefore some of the sessions were pruned (not fully analyzed) to free memory for new ones.

Following warning messages were reported by Suricata:

```
flow.emerg_mode_entered
Flow emergency mode over, back to normal... unsetting
FLOW_EMERGENCY bit
```

To overcome this problem, the Flow engine memory cap was set from the default of 32MiB to 512MiB, which is similar to what was specified for Snort. As a result no more sessions were pruned during the test cycles.

```
flow:
  memcap: 512mb
```

Further increasing the memory cap improved the results minimally, therefore there was no point in raising it higher than 512MiB for our tests. However, this largely depends on the characteristics of the network traffic being currently analyzed.

Simultaneous packet processing

Default number of packets allowed to be processed simultaneously by Suricata is 1,024, which according to configuration comments is rather conservative. Increasing this limit to 4,096 showed a slight improvement in performance.

```
max-pending-packets: 4096
```

Increasing this limit negatively impacts caching, so there is a fine line between improving and degrading performance with this parameter.

Detection engine configuration

By default Suricata uses the ac (Aho-Corasick) multi pattern matcher (mpm) algorithm with "single" distribution context that offers good performance with low memory consumption. By setting the distribution context to "full", the number of dropped packets reduced twofold. However, for the ac algorithm this required a large amount of memory (over 30GiB) and several minutes to load our set of rules. Considering that other detection algorithms offered nearly similar performance with much lower memory consumption (3-7GiB), it was reasonable to continue with some of the alternatives. Testing showed that the b2gc algorithm produced the best results in our environment.

```
mpm-algo: b2gc
```

```
detect-engine:
  - profile: high
  - sgh-mpm-context: full
```

As can be seen above, the detection engine profile was set to high from the default medium setting. According to the configuration comments, this should efficiently manage system memory use to ensure good performance. This setting directly affected the memory allocated during traffic analysis. Higher memory use resulted in better performance.

Multi-threading and CPU affinity

Suricata is multi-threaded. By default configuration, the ratio of detect threads created for each processor available was 1.5. On our system this meant that 12 detect threads were created. First, we discovered that increasing the number of detection threads surprisingly resulted in more dropped packages. This is probably due to high overhead of managing multiple threads.

```
detect-thread-ratio: 1.5
```

Furthermore, as already mentioned in the paragraph 1.3 Related Work, a study by Éric Leblond suggested that CPUs with Hyper-threading can cause variations in Suricata's performance. By default Suricata tries to balance load on each available processor equally. This means that Hyper-threading is used even if the physicals cores are not fully utilized. It was recommended to ignore the detect-thread-ratio and use fixed CPU affinity to limit the amount of processors utilized to the number of physical cores on the CPU. [10]

This indeed improved performance. When dividing Suricata threads between the 4 cores on the system, we aimed that each core would be equally loaded. We finally configured Suricata to use only three detection threads. Management, receive, decode and other threads were configured to use the fourth core. Our exact CPU affinity configuration can be found in Annex 3 – Suricata CPU Affinity Configuration.

4.2.3 Bro

It is important to bear in mind that unlike Snort and Suricata, Bro does not have a main configuration file, where most of the settings can be altered. Instead, Bro has several files for different purposes. Some of the more important ones are the following.

- broctl.cfg – BroControl management configuration file (e.g. log directory);
- node.cfg – Configuration for standalone or clustered Bro nodes;
- local.bro – Local site policy that states which detection policies/scripts are loaded during Bro startup.

Interestingly, there are very few optimization guides available for Bro. Even the official documentation states that a single Bro instance can handle approximately 80Mbit/s of traffic. Note that this verifies the results we acquired for Bro in experiment 1. In order to efficiently analyze traffic faster, using some of Bro Cluster solutions is recommended. These solutions will be tested in experiment 3 (chapter 4.3.3). [22]

In terms of optimizing, there were some suggestions how to improve Bro's performance. Unfortunately, similar to Snort, most of them involved turning off detection policies, so that some of the traffic would not be analyzed. For instance, disabling the CPU-intensive HTTP data processing would probably result in many missed attacks. Optimizing the policies (rules) did not fall into the scope of this thesis.

Therefore, we were unable to perform any optimizations that were reasonable in this context and thus Bro will not be included in the results in the following chapter.

4.2.4 Optimization Results

This chapter outlines the results of the optimizations described in this experiment. Note that Bro is excluded because we were unable to optimize its configuration (see chapter 4.2.3 for explanation).

Table 3 comprises of the percentages of dropped packets after optimizations had been applied to the systems. Overall, both systems showed significant improvements. Suricata nearly achieved the perfect result of no dropped packets, however Snort still had much room for improvement. See Figure 10 for a more graphical view of the results.

Note that at the fastest transmission setting (1,000Mbit/s), about 2,000 and 90,000 packets were dropped on the kernel level before reaching the Suricata and Snort correspondingly. These values have been included in the following statistics.

Table 3: Percentage of dropped packets after optimizations

	50M	100M	150M	200M	300M	400M	500M	600M	700M	800M	900M	1000M
Snort	0.00%	0.02%	0.65%	1.97%	5.49%	11.31%	18.59%	22.31%	29.56%	36.02%	42.27%	(58.36%)
Suricata	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.01%	0.01%	0.02%	0.02%	(1.84%)

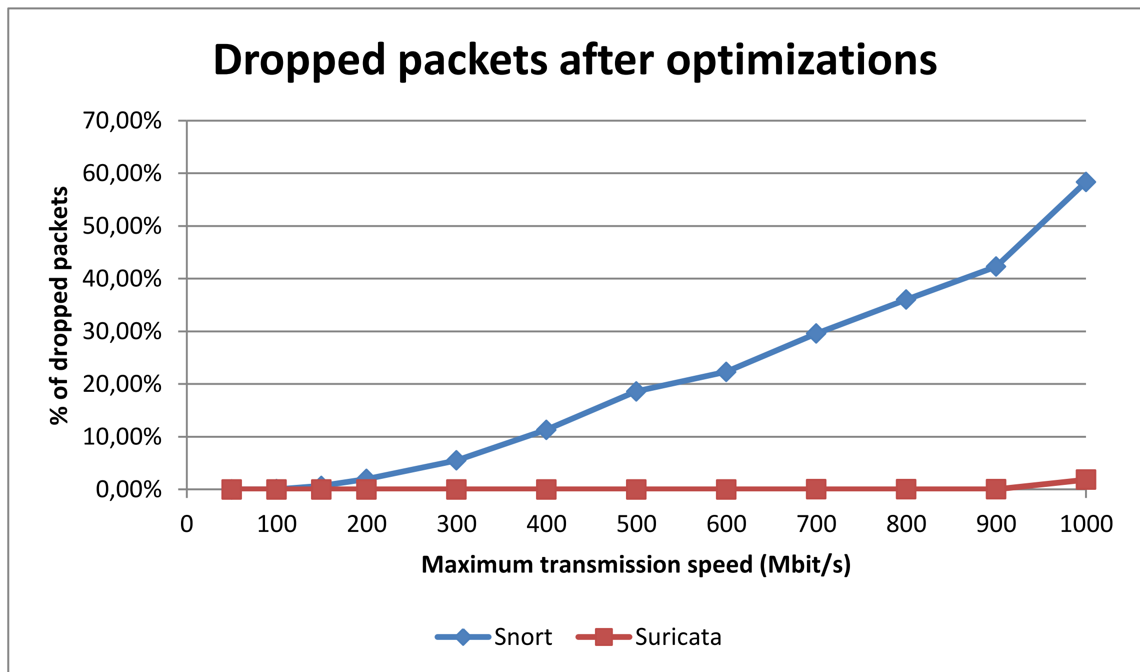

Figure 10: Percentage of dropped packets after optimizations

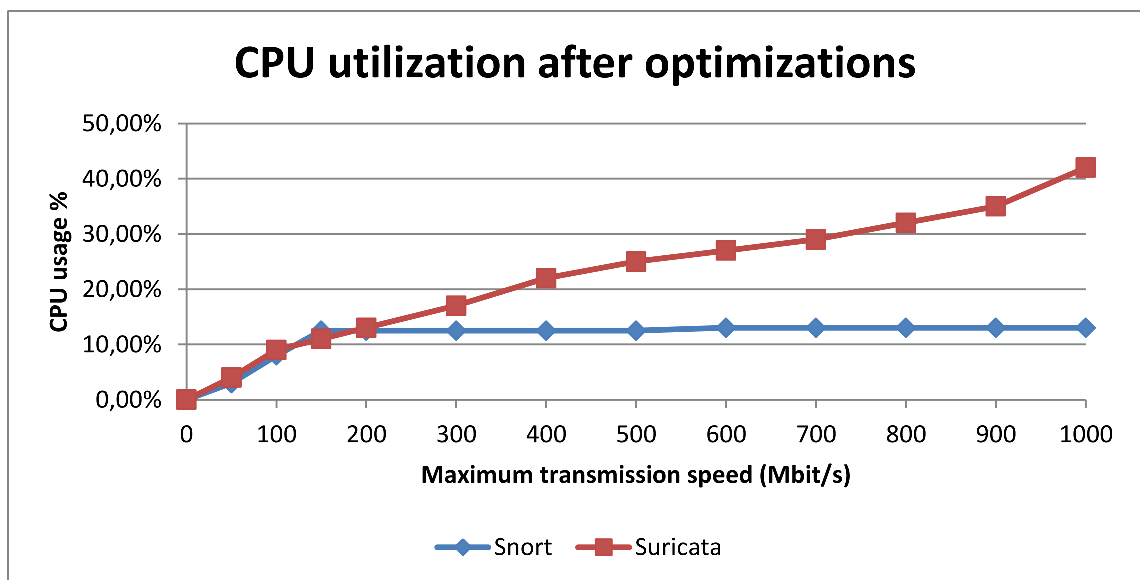

Figure 11: Percentage of CPU utilized by Snort and Suricata after optimizations

Figure 11 shows that Snort CPU usage remained the same, since it was still using only one processor. After optimizations Suricata demonstrated about 20-30% lower CPU utilization. This was mostly due to the CPU affinity settings we applied to limit the Suricata to using only 4 processors and avoid Hyper-Threading.

4.3 Experiment 3 – Modify or Replace Network Packet Capture Module

Third and final experiment describes some of the possibilities for improving IDS performance by the means of modifying or replacing the network packet capture software.

4.3.1 Increase libpcap Buffer Size

Another possibility to reduce the amount of dropped packets was to increase the capture buffer size of libpcap. By default, libpcap has a buffer size of only 32KiB, which is good for a variety of portable solutions, however for our tests this limit could be significantly higher.

For Linux, the maximum buffer size is 2GiB. It is important to note that this buffer will always be allocated from system memory, even when not fully in use. However, on our test system there was plenty of free memory, so this was not a problem.

Moreover, it is important to bear in mind that this is only a buffer to handle spikes in the network traffic. If the transmission rates are constantly faster than what the IDS can handle, then this buffer will eventually fill up and the packets will be dropped anyway.

For Snort (DAQ) and Suricata this limit can be easily set in their configuration files or when starting the IDS process with the following command line arguments.

Snort configuration:

```
config daq_var: buffer_size=2147483647
```

or from command line:

```
--daq pcap --daq-var buffer_size=2147483647
```

Suricata configuration:

```
pcap:
  - interface: eth1
    buffer-size: 2147483647
```

or from command line:

```
--pcap-buffer-size 2147483647
```

In our tests this configuration showed about 5-15% less dropped packets for Snort compared to the previous results. However, this improvement did not prove to be very stable. Results varied significantly, therefore many tests had to be performed to get a solid average result. Nevertheless, with this addition, Snort was able to analyze 400Mbit/s traffic without any drops – a remarkable improvement.

For Suricata, the configuration that we optimized in the previous chapter and this additional buffer was enough to achieve zero dropped packets on all test scores. Note that at the highest transmission speed around 2,000 and 90,000 packets were dropped on the kernel level for Suricata and Snort correspondingly. However, as mentioned before, this is marginal compared to total number of packets. These values have been included in the following statistics.

Unfortunately, Bro does not yet have the functionality to set libpcap memory buffer size from the configuration or command line when starting the process. Feature request has been submitted to Bro Trac and the milestone has currently been set to Bro version 2.2. [29]

Since we were considering some alternative possibilities to improve Bro's performance, we were not going to modify the source code for Bro and libpcap to statically set a larger buffer size. Therefore Bro will again be excluded from the results presented here.

Exact numbers for Snort and Suricata are shown in Table 4 and graphed on Figure 12.

Table 4: Percentage of dropped packets using previous optimizations and a 2GiB libpcap buffer size

	50M	100M	150M	200M	300M	400M	500M	600M	700M	800M	900M	1000M
Snort	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	5.45%	16.15%	27.22%	37.89%	43.72%	(49.86%)
Suricata	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	(0.01%)

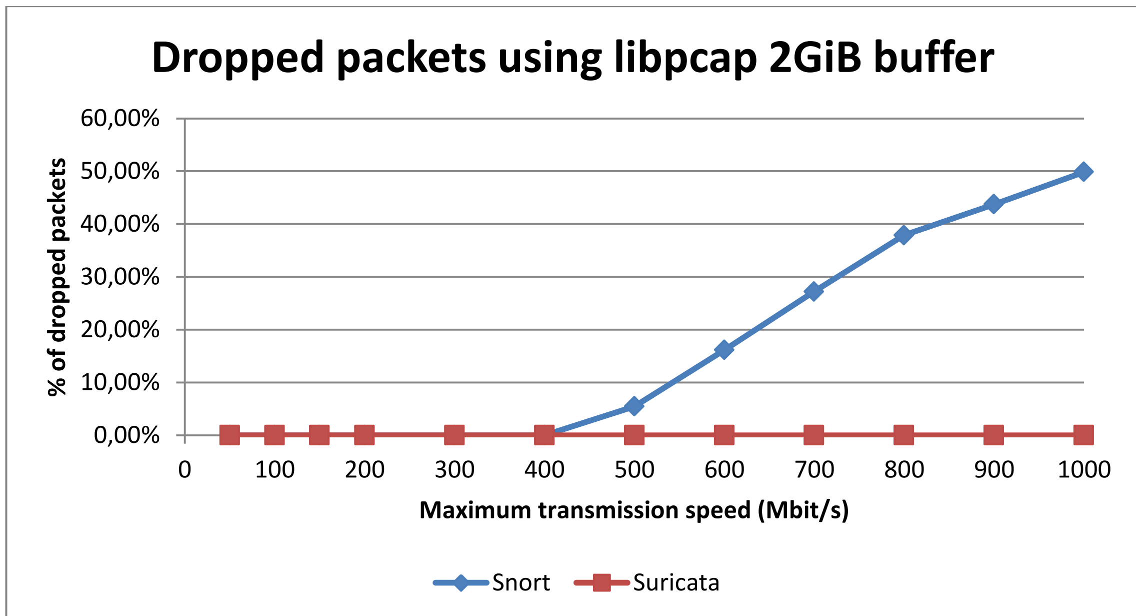

Figure 12: Percentage of dropped packets using previous optimizations and a 2GiB libpcap buffer size

Since we did not change much with the IDS process itself, the CPU load remained similar to previous experiments for both systems.

4.3.2 Use AF_PACKET Network Socket

AF_PACKET is the Linux native network socket. It functions similar to the memory mapped PCAP, but no external libraries are required.

Similar to libpcap, AF_PACKET enables the user to configure a memory buffer for captured packets. By default, the Snort with DAQ allocates 128MiB for packet memory, which is significantly higher than PCAP default of 32KiB. In our tests, we set the memory buffer to 2GiB in order to compare the results with libpcap. The buffer size can be modified as follows.

Snort command line arguments:

```
--daq afpacket --daq-var buffer_size_mb=2048
```

Suricata configuration file:

```
af-packet:
  - interface: eth1
    buffer-size: 2147483647
```

Furthermore, recent versions of Suricata include a ring buffer feature for AF_PACKET capture. This is a memory mapped buffer similar to PF_RING that will be discussed in

the next chapter. Additionally, a mode named Zero Copy is also provided with this buffer. This means that the memory allocated for the buffer is shared with the capture process, so instead of kernel sending packets to the capture process, the process can just read the packets from their original memory address. This method saves time and is less consuming in terms of CPU resources. These features are only supported on kernel versions above 3.1, but unfortunately we are using kernel version 2.6.32. [30]

Bro did not yet seem to support the AF_PACKET capturing mode. However, we were still able to use the AF_PACKET for Snort and Suricata and see how it compared to the previous methods. See Table 5 and Figure 13 for the test results.

Table 5: Percentage of dropped packets using previous optimizations and a 2GiB AF_PACKET buffer size

	50M	100M	150M	200M	300M	400M	500M	600M	700M	800M	900M	1000M
Snort	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	2.50%	12.84%	16.97%	26.59%	36.46%	(41.00%)
Suricata	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	(0.44%)

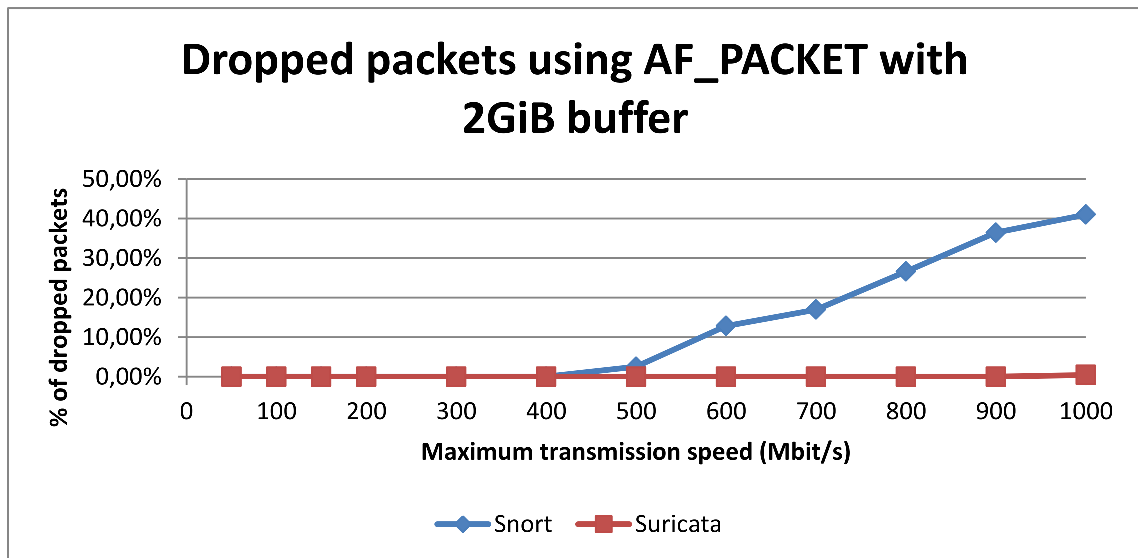

Figure 13: Percentage of dropped packets using previous optimizations and a 2GiB AF_PACKET buffer size

Snort improved the results compared to libpcap capture socket. With AF_PACKET it was now dropping about 10% less packets overall. Suricata did not manage to achieve a perfect result using AF_PACKET. It was again dropping around 110,000 packets (about 0,4%) at the fastest transmission speed. This is probably due to our older kernel version and Suricata not being able to make use of newer advanced features. Similarly to

libpcap, once the 2GiB buffer filled up AF_PACKET produced quite unstable results and tests had to be performed many times in order to get a decent average.

CPU usage remained the similar to what it was after optimizations. Thus, there is no need to repeat the results here.

4.3.3 Use PF_RING Network Socket

Final solution that we considered in this thesis was using the PF_RING network socket. PF_RING is a new type of socket from a research company called "ntop". One of its main traits is that it should significantly improve packet capture speed. [31]

PF_RING is a complex software, therefore we will only be able to cover some of the main aspects. As already mentioned in the previous chapter, it features a circular (ring) buffer and the applications read packets from this buffer. Purpose of this is that PF_RING can distribute packets to multiple application processes simultaneously. More detailed explanations can be found from source [31], the PF_RING project homepage.

PF_RING package includes source code for many PF_RING-aware NIC drivers, modified versions of some necessary software modules (e.g. libpcap, tcpdump, pfring-daq-module) and even tools for testing and debugging.

Operating modes

There are three different operating modes for PF_RING, which can be chosen when loading the PF_RING kernel module with `insmod pf_ring.ko` command. These modes are as follows. [32]

- `transparent_mode=0` – Default mode, which means that packets are sent to PF_RING via the standard kernel mechanisms. Packet capture is not accelerated, but PF_RING features can be used. All NIC drivers support this mode.
- `transparent_mode=1` – In this mode NIC driver sends packets directly to PF_RING, however packets are still propagated to other kernel components. Packet capture is accelerated because packets are copied by the NIC driver itself without passing through the usual kernel path. In order to use this mode, a PF_RING-aware NIC driver has to be used.
- `transparent_mode=2` - Packets are sent directly by the NIC driver to PF_RING and are not propagated to other kernel components. This mode is the

fastest, because packets are copied only to PF_RING and discarded after processing. Again, the NIC driver has to support PF_RING to enable this mode.

Installation
Getting PF_RING to function properly proved to be a quite complicated procedure. There are some incomplete and outdated guides available on the Internet, which can cause problems. To assist in this matter, we will point out some important steps. We followed the guide on the official ntop site [33], however this does not explain how to do any of the configuration on the IDS solution.

After compiling and installing all necessary packages, PF_RING module should be loaded to the kernel. There are some important arguments that could be passed to the module. The transparent mode has to be selected. To cope with peaks in the network traffic, a buffer size can be specified for PF_RING as well.

Load PF_RING kernel module:
```
insmod /lib/modules/2.6.32-
279.11.1.el6.x86_64/kernel/net/pf_ring/pf_ring.ko
transparent_mode=2 min_num_slots=16384
```

Note that, this guide instructs the user to unload the old network driver. Bear in mind that after issuing the command the host will lose network connectivity. This might be avoided when the newly compiled driver is ready to be loaded. To avoid the risk of being cut off from the server, access to the host console should be available before attempting this.

Unload the old driver and load the newly compiled driver with a single invocation:
```
rmmod bnx2; insmod /lib/modules/2.6.32-
279.11.1.el6.x86_64/updates/bnx2.ko
```

IDS configuration
Suricata and Bro were both able to make use of the improved performance. Unfortunately, we were not able to get PF_RING working with DAQ for Snort. For some reason DAQ would not load the PF_RING capture library.

For Suricata no extra configuration was necessary and PF_RING run-mode could be selected from the command line.

Suricata command line arguments:

```
--pf-ring=eth1
```

Bro required some extra configuration. Bro had to be run in Cluster mode, in order to enable PF_RING. There is already an example configuration in place which can be modified. The last two rows indicate a load balancing method and the number of Bro worker processes to create for load balancing.

Bro node.cfg configuration:

```
[manager]
type=manager
host=172.16.16.1

[proxy-0]
type=proxy
host=172.16.16.1

[worker-0]
type=worker
host=172.16.16.1
interface=eth1
lb_method=pf_ring
lb_procs=8
```

Results using PF_RING

Using PF_RING resulted in excellent performance from both Suricata and Bro. Neither dropped any packets, but at the fastest transmission speed around 15,000 packets were dropped on the kernel level. This is quite much considering the overall good performance. Table 6 shows the exact numbers, although with such good results there is not much to look at.

Table 6: Percentage of dropped packets using previous optimizations and PF_RING

	50M	100M	150M	200M	300M	400M	500M	600M	700M	800M	900M	1000M
Suricata	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	(0.06%)
Bro	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	(0.05%)

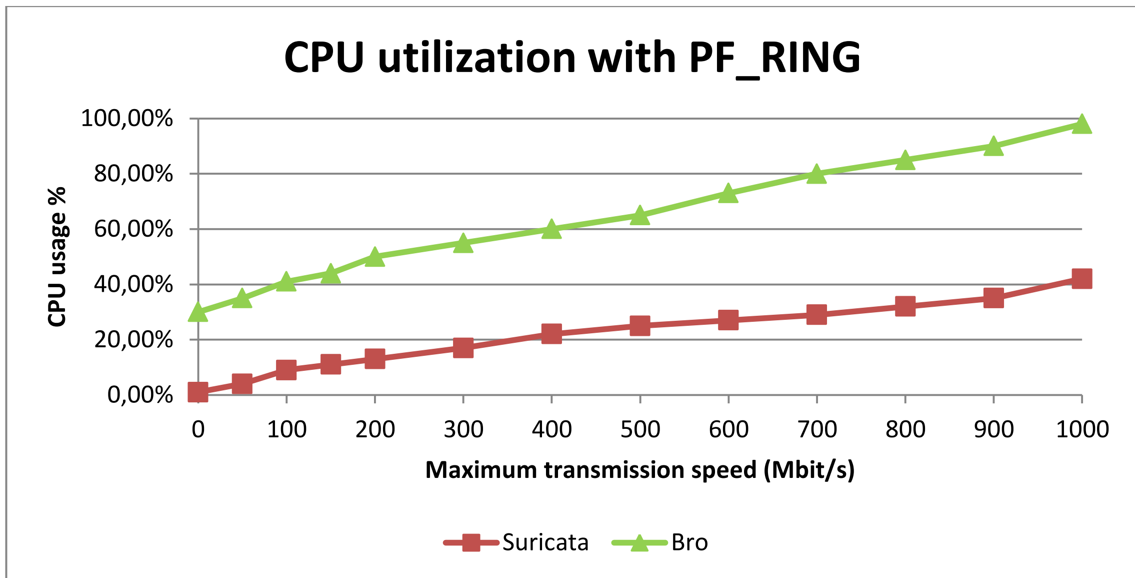

Figure 14: Percentage of CPU utilized by Suricata and Bro using PF_RING

Figure 14 shows that Bro and its eight worker processes are now using a lot more resources. Linearity of the CPU usage graph hints that 1,000Mbit/s is probably the limit that Bro can handle with this configuration on this hardware. Note that Bro was always using about 30% of CPU resources, even when no packets were being processed. This is said to be due to communication overhead between the multiple processes.

When it comes to Suricata, we were using the same optimized configuration from experiment 2, so there was not much change in the results.

5 Discussion of Results

This chapter will summarize and interpret the results gathered during all experiments.

5.1 Dropped Packets

In terms of dropped packets, Suricata and Bro achieved the desired result – no dropped packets at the fastest transmission speed on a 1,000Mbit/s network. The actual average speed was rated at 895Mbit/s. This should be more than adequate to perform intrusion detection on a mid-sized company network link.

Unfortunately, we were unable to test PF_RING with Snort (DAQ) due to unknown errors. DAQ would not load the PF_RING capture module. We believe that when using PF_RING, Snort would have also achieved the result of no dropped packets.

Next we will compare the results of the highest transmission speed for each IDS and name the experiment where each IDS achieved its best results. This could be considered as a top result for each IDS.

- Snort – Experiment 3 – AF_PACKET with 2GiB buffer – 41% dropped packets;
- Suricata:
  - Experiment 3 – libpcap with 2GiB buffer – 0% dropped packets;
  - Experiment 3 – PF_RING – 0% dropped packets.
- Bro – Experiment 3 – PF_RING – 0% dropped packets.

5.2 CPU Usage

When it comes to CPU usage, lower results are better. Of course, this is only true as long as no packets are dropped. It seems that multi-threaded Suricata was able to achieve the best results in terms of dropped packets while using the least amount of CPU resources.

Snort and Bro are single-threaded applications, so by default, they were utilizing only one of eight logical processors (i.e. around 12.5%). When using PF_RING socket, we were able to create multiple worker processes for Bro. This allowed Bro to use all available CPU resources. We tried the same approach for Snort, but were unsuccessful. Therefore, we cannot be definitive about the results.

5.3 Memory Usage

While Suricata seemed to be using the least amount of CPU resources, it was just the opposite in terms of memory usage. Figure 15 depicts the average memory consumption of IDS process(es) after each experiment. Note that different transmission speeds did not have any significant effect on memory consumption.

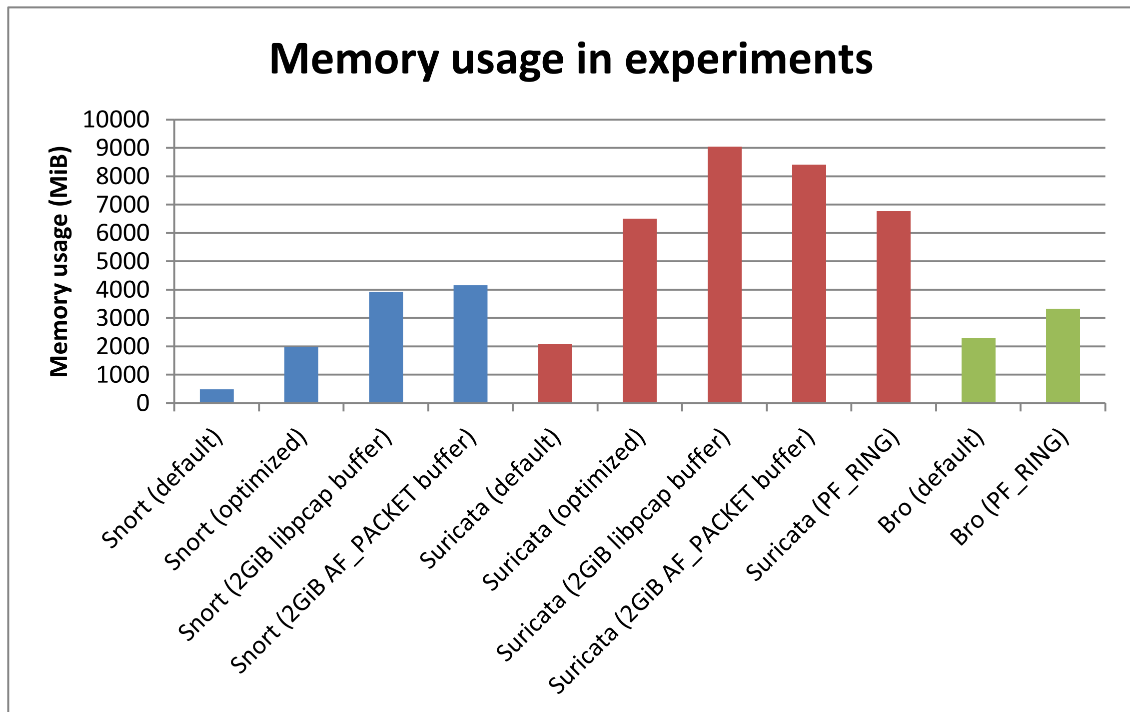

Figure 15: IDS process memory usage in experiments

Higher memory consumption is not necessarily a negative aspect as long as there is memory to spare. Memory can be used to enable more efficient detection algorithms, increase buffers and improve caching.

6 Future Research

This chapter offers some ideas on future research topics that were not covered in this thesis.

6.1 Snort (DAQ) with PF_RING

In this thesis we were unable to get PF_RING network socket to function with DAQ. The PF_RING modules compiled and installed successfully, however DAQ would not load the PF_RING module.

It would be interesting to see, whether or not Snort can achieve the same result as Suricata and Bro – no dropped packets at the highest transmission speed on a 1,000Mbit/s network. The same or similar hardware would have to be used for the results to be comparable.

6.2 Use Suricata with Kernel Versions Above 3.1

Suricata had many features that were only functional on kernel versions above 3.1. We were not able to test them in this thesis. Enabling those newer features would probably increase performance even more.

6.3 Experiment with Speeds Up To 10Gbit/s

Nowadays many network backbones already operate at 10Gbit/s bandwidth. It would be interesting to experiment with the same IDS engines on a 10Gbit/s network. The testing environment would require to be replaced with more powerful hardware and special 10Gbit/s network adapters.

6.4 GPU Processing

It is known that Graphics Processing Units (GPUs) are often used in computational tasks that can be parallelized (parallel computing). In situations like these GPUs are usually many times faster than CPUs. However, using GPU instead of CPU complicates the configuration and can be difficult to manage.

Suricata already has support for NVIDIA CUDA (Compute Unified Device Architecture) parallel computing platform. Other intrusion detection systems have experimental releases that are not yet stable. Experiments in this field could help contribute to this research. One would require testing hardware with a compatible graphics card.

6.5 Compare Different Rule Sets

In this thesis we did not concentrate on the accuracy of the rules. This would have widened the scope and increased the risk of losing focus.

There are a few major contributors to rules for Snort and Suricata. Comparing systems with similar and/or different rule sets could reveal which engines and rule sets are more effective in detecting intrusions.

Testing could be performed with software called Pytbull, which is a python based flexible IDS/IPS testing framework. It comes packaged with more than 300 tests. Additional tests can be added if necessary. [34]

7 Conclusion

As a result of this thesis, an overview of three popular open-source intrusion detection systems (IDS) was provided along with their comparative performance benchmarks. The thesis addressed a problem that there practically were no recent and unbiased comparisons available for intrusion detection systems.

On the one hand, the problem arises from a fact that most articles and comparisons are often written by people involved with some IDS community. On the other hand, intrusion detection is difficult to accomplish perfectly and that makes the IDS testing procedure an interesting research topic now and again.

The thesis was set to achieve the following objectives:

- Present an overview of popular open-source IDS solutions;
- Carry out their comparative evaluation that satisfies the following conditions:
    - Reliability – Reliable test results;
    - Repeatability – Tests can be run again when needed;
    - Reproducibility – Provide configuration instructions.

The aim of the thesis was not to determine the best open-source IDS, but rather bring out the advantages and disadvantages of each system.

We analyzed three network-based intrusion detection systems and gave a brief description of each system. Snort is probably the most widely deployed IDS worldwide. Suricata is a younger competitor of Snort that offers many improvements. Bro is an alternative to Snort and Suricata that also provides a comprehensive platform for more general network traffic analysis.

We performed the comparative evaluation on a 1Gbit/s network with a number of experiments. Note that testing the accuracy of the detection rules was not included in the scope of this thesis. We used the percentage of dropped packets as the primary metric for measuring the IDS performance.

With the first experiment, a performance baseline was established for all three systems in their default configuration. As the second experiment, optimizations were applied to the configuration. Finally, three different network packet capture modules (PCAP, AF_PACKET, PF_RING) were tested as the third experiment.

Experiments demonstrated that systems in their default configuration were only able to handle about 100Mbit/s network traffic. Number of dropped packets increased significantly beyond this limit. At this point we began applying the optimizations and testing different network sockets. All modifications were documented in the thesis.

As a result, Snort paired with AF_PACKET network socket was able to handle about 450Mbit/s of traffic. Unfortunately, we were unsuccessful in running Snort with PF_RING socket due to unresolved errors. We are planning to solve this problem as part of our future research.

Suricata with its multi-threaded architecture achieved good results in all experiments. Only exception being that it was consuming nearly double the amount of memory compared to Snort or Bro. Suricata did not drop any packets at 1,000Mbit/s when using the improved libpcap module or the PF_RING network socket.

Bro was not able to take part in many of the experiments due to compatibility issues. However, similarly to Suricata, it was able to achieve the perfect result of no dropped packets at 1,000Mbit/s using PF_RING socket.

In short, we saw a four- to ten-fold increase in performance. Transmission speeds up to 1,000Mbit/s were handled without any dropped packets.

It can be concluded that all the objectives of the thesis were achieved as described.

Võrdlusanalüüs vabatarkvaralistest ründetuvastus-süsteemidest

Magistritöö (30 EAP)

Mauno Pihelgas

Kokkuvõte

Käesoleva magistritöö tulemusena anti ülevaade kolmest populaarsest vabatarkvaralisest ründetuvastussüsteemist ning teostati nende jõudlusmõõtmised. Lõputöö lahendas probleemi, et erinevatest ründetuvastussüsteemidest ei leidunud erapooletuid ja ajakohaseid võrdlusi.

Ühest küljest tuleneb probleem sellest, et enamik artiklite ja võrdluste autorid on ise seotud mõne ründetuvastussüsteemi kogukonnaga ning nende arvamus ei pruugi alati olla objektiivne. Teisalt tuleneb probleem asjaolust, et ründetuvastust on tänases kiiresti muutuvas keskkonnas keerukas teostada. Just seetõttu on teema üha uuesti aktuaalne.

Lõputöö ülesande püstitusel määratleti kaks põhieesmärki:

- Anda ülevaade populaarsetest vabatarkvaralistest ründetuvastussüsteemidest;
- Testida ning võrrelda nende ründetuvastussüsteemide jõudlusomadusi, mis rahuldaksid järgnevaid alamtingimusi:
    - Usaldusväärsus – Usaldusväärsed testi tulemused;
    - Korratavus – Samu teste saab vajadusel korduvalt käivitada;
    - Taastoodetavus – Paigaldusjuhiste olemasolu.

Magistritöö eesmärk ei olnud parima ründetuvastussüsteemi leidmine, vaid pigem iga süsteemi tugevamate ja nõrgemate omaduste väljaselgitamine.

Töö käigus analüüsiti ning kirjeldati kolme võrgupõhist ründetuvastussüsteemi. Snort on üks suurima kasutajaskonnaga ründetuvastussüsteem kogu maailmas. Suricata on

Snort-i suurim konkurent, omades mõningaid eeliseid Snort-i ees. Bro on alternatiivne lahendus Snort-i ja Suricata asemel, pakkudes ühtlasi laialdasema kasutusalaga võrguanalüüsi platvormi.

Töö praktilises osas teostati 1Gbit/s ribalaiusega võrguühendusel süsteemide jõudlusmõõtmisi. Eksperimentide käigus ei mõõdetud mitte tuvastusreeglite täpsust, vaid hoopis analüüsimata jäänud võrgupakettide hulka.

Esimeses eksperimendis, tuvastati kõigi kolme süsteemi võimekus nende vaikeseadistuses. Teine eksperiment sisaldas endas süsteemide seadistuse optimeerimist. Viimaks testiti ründetuvastussüsteeme koostöös erinevate võrgusoklite (*ingl. k. network socket*) lahendustega (PCAP, AF_PACKET, PF_RING).

Tulemused näitasid, et vaikeseadistuses suudavad süsteemid edukalt töödelda vaid võrguliiklust kuni 100Mbit/s. Suurematel kiirustel kasvas analüüsimata jäänud võrgupakettide hulk märgatavalt. Tulemuste parandamiseks optimeeriti seadistust ning katsetati erinevaid võrgusoklite lahendusi. Kõik muudatused kirjeldati käesolevas töös.

Kasutades AF_PACKET soklit, suutis Snort analüüsida kuni 450Mbit/s võrguliiklust. Kahjuks ei õnnestunud tehniliste probleemide tõttu kasutada Snort-i koos PF_RING sokliga, mis tõenäoliselt oleks parandanud tulemust veelgi. See probleem on plaanis edasise uurimustöö käigus kõrvaldada.

Mitmelõimelise arhitektuuriga Suricata saavutas igas testis head tulemused. Erandiks oli vaid enam kui kahekordne mälukasutus võrreldes kahe teise süsteemiga. Kasutades täiustatud libpcap või PF_RING võrgusoklit, suutis Suricata analüüsida kiirusel 1000Mbit/s kõik talle saadetud võrgupaketid.

Ühilduvusprobleemide tõttu ei õnnestunud Bro testimine mõnes eksperimendis. Aga kasutades PF_RING võrgusoklit, suutis ka Bro kiirusel 1000Mbit/s analüüsida kõik võrgupaketid.

Lühidalt öeldes, töö käigus saavutati 4-10kordne kasv kõikide ründetuvastussüsteemide jõudluses. Eelnevast lähtudes võib väita, et kõik töös püstitatud eesmärgid on nõuetekohaselt täidetud.

List of References

1. **The Linux Information Project.** BSD License Definition. [Online] [Cited: November 29, 2012.] http://www.linfo.org/bsdlicense.html.

2. **Free Software Foundation, Inc.** The GNU Operating System. [Online] [Cited: November 29, 2012.] http://www.gnu.org/.

3. **Lyon, Gordon.** Nmap - Free Security Scanner For Network Exploration & Security Audits. [Online] [Cited: December 1, 2012.] http://nmap.org/.

4. **Wireshark Wiki.** Libpcap File Format. *Wireshark Wiki.* [Online] [Cited: December 1, 2012.] http://wiki.wireshark.org/Development/LibpcapFileFormat.

5. **Sophos.** Security Threat Report 2012. [Online] 2012. [Cited: August 27, 2012.] http://www.sophos.com/medialibrary/PDFs/other/SophosSecurityThreatReport2012.pdf.

6. **Kvell, Alar.** *A high-performance network intrusion detection solution for S4A software.* Tartu : Tartu Ülikool, 2012.

7. **Albin, Eugene A.** A Comparative Analysis of the Snort and Suricata Intrusion-Detection Systems. [Online] September 2011. [Cited: August 27, 2012.] http://faculty.nps.edu/ncrowe/oldstudents/ealbin_thesis_final.htm.

8. **Sourcefire.** Using Perfmon and Performance Profiling to Tune Snort Preprocessors and Rules. [Online] November 6, 2009. [Cited: November 19, 2012.] http://www.snort.org/assets/163/WhitePaper_Snort_PerformanceTuning_2009.pdf.

9. **Leblond, Éric.** Suricata, to 10Gbps and beyond. *To Linux and beyond!* [Online] July 30, 2012. [Cited: November 15, 2012.] https://home.regit.org/2012/07/suricata-to-10gbps-and-beyond/.

10. —. Optimizing Suricata on multicore CPUs. *To Linux and beyond!* [Online] January 26, 2011. [Cited: November 16, 2012.] http://home.regit.org/?p=438.

11. **Holste, Martin.** Bro Quickstart: Cluster Edition. *Open-Source Security Tools.* [Online] September 26, 2011. [Cited: November 16, 2012.] http://ossectools.blogspot.com/2011/09/bro-quickstart-cluster-edition.html.

12. **Burks, Doug.** Security Onion Homepage. [Online] [Cited: November 29, 2012.] http://securityonion.blogspot.com/.

13. **Καρυπιδης, Χαραλαμπος.** Snort, IDS, IPS, NSM, hacking and...beyond. *Harrykar's Techies Blog.* [Online] [Cited: October 2012, 25.] http://harrykar.blogspot.com/2009/05/snort-ids-ips-nsm-andbeyond.html.

14. **Sourcefire.** Snort :: Home Page. [Online] [Cited: September 17, 2012.] http://www.snort.org/.

15. **The Open Information Security Foundation.** Suricata Homepage. [Online] [Cited: September 18, 2012.] http://www.openinfosecfoundation.org/.

16. **Bro.** The Bro Network Security Monitor. [Online] [Cited: October 18, 2012.] http://bro-ids.org/.

17. **Sourcefire.** Snort :: License. [Online] [Cited: October 2012, 16.] http://www.snort.org/snort/license.

18. **Roesch, Martin.** Snort 3.0 Beta 3 Released. *Security Sauce.* [Online] April 2, 2009. [Cited: November 18, 2012.] http://securitysauce.blogspot.com/2009/04/snort-30-beta-3-released.html.

19. **The Open Information Security Foundation.** What is Suricata. *Suricata.* [Online] [Cited: October 2012, 22.] https://redmine.openinfosecfoundation.org/projects/suricata/wiki/What_is_Suricata.

20. **Paxson, Vern.** Bro: A System for Detecting Network Intruders in Real-Time. [Online] January 26, 1998. [Cited: November 1, 2012.] http://static.usenix.org/publications/library/proceedings/sec98/full_papers/paxson/paxson.pdf.

21. **Gerber, John.** Three Open Source IDS/IPS Engines: The Setup. *Security Advancements at the Monastery.* [Online] August 26, 2010. [Cited: September 15, 2012.] http://web.archive.org/web/20100831151023/http://blog.securitymonks.com/2010/08/26/three-little-idsips-engines-build-their-open-source-solutions/.

22. **Bro.** Bro Cluster. *Bro 2.1 documentation.* [Online] [Cited: November 2012, 18.] http://www.bro-ids.org/documentation/cluster.html.

23. **The UCSB International Capture The Flag.** The 2010 iCTF Data. *The UCSB iCTF.* [Online] [Cited: September 20, 2012.] http://ictf.cs.ucsb.edu/data/ictf2010/.

24. **Bro.** Quick Start Guide. *Bro 2.1 documentation.* [Online] [Cited: October 6, 2012.] http://www.bro-ids.org/documentation/quickstart.html.

25. **Parker, William.** Snort 2.9.3.1 on CentOS 6.3. [Online] [Cited: October 3, 2012.] http://www.snort.org/assets/202/snort2931_CentOS63.pdf.

26. **The Open Information Security Foundation.** Suricata - CentOS 5.6 Installation. [Online] [Cited: October 4, 2012.] https://redmine.openinfosecfoundation.org/projects/suricata/wiki/CentOS_56_Installation.

27. **Kemp, Juliet.** Use Profiling to Improve Snort Performance. *OpenLogic.* [Online] [Cited: November 20, 2012.] http://www.openlogic.com/wazi/bid/188092/Use-Profiling-to-Improve-Snort-Performance.

28. **Sourcefire.** SNORT Users Manual 2.9.3. *The Snort Project.* [Online] [Cited: November 20, 2012.] http://manual.snort.org/node16.html.

29. **Bro.** Script variable to set pcap's buffer size. *Trac Trickets.* [Online] [Cited: December 03, 2012.] http://tracker.bro-ids.org/bro/ticket/553.

30. **Leblond, Éric.** Using AF_PACKET zero copy mode in Suricata. *To Linux and beyond!* [Online] [Cited: December 3, 2012.] https://home.regit.org/2012/02/using-af_packet-zero-copy-mode-in-suricata/.

31. **ntop.** PF_RING Homepage. [Online] ntop. [Cited: December 3, 2012.] http://www.ntop.org/products/pf_ring/.

32. —. PF_RING and Transparent Mode. [Online] [Cited: December 3, 2012.] http://www.ntop.org/pf_ring/pf_ring-and-transparent-mode/.

33. —. Installation Guide For PF_RING. [Online] [Cited: December 3, 2012.] http://www.ntop.org/pf_ring/installation-guide-for-pf_ring/.

34. **Damaye, Sébastien.** Pytbull Homepage. [Online] [Cited: December 4, 2012.] http://pytbull.sourceforge.net/.

35. **Hewlett-Packard Development Company, L.P.** HP ProLiant DL320 Generation 6 (G6). [Online] [Cited: October 4, 2012.] http://h18004.www1.hp.com/products/quickspecs/13344_na/13344_na.html.

36. —. HP ProLiant DL360 Generation 5 (G5). [Online] [Cited: October 4, 2012.] http://h18004.www1.hp.com/products/quickspecs/12476_div/12476_div.HTML.

37. **Linksys.** Product Data. [Online] [Cited: October 4, 2012.] http://www.andovercg.com/datasheets/linksys-srw2016-switch.pdf.

Appendices

Annex 1 – Hardware Specification

Role: IDS
Hewlett-Packard (HP) ProLiant DL320 Generation6 server [35]

- CPU: Intel® Xeon® Processor E5630 (12M Cache, 2.53 GHz);
  - 1 Physical CPU;
  - 4 cores;
  - 8 threads (Hyper-Threading enabled).
- RAM: 72GB PC3-10600 (DDR3-1333) Registered CAS-9 Memory;
- NIC: Embedded NC326i Dual Port Gigabit Server Adapter;
- HDD Controller: HP Smart Array P410/512 BBWC Controller;
- HDD: RAID1 – 2x Seagate Barracuda 750GB, 3,5" LFF, 7200RPM, 16MB cache, SATA 3.0Gb/s;
- GPU: Integrated ATI ES1000, 64 MB;
- Management: HP Integrated Lights Out 2 (iLO2).

Role: Supporting Host1 & Supporting Host2
HP ProLiant DL360 Generation5 server [36]

- CPU: Intel® Xeon® Processor E5450 (12M Cache, 3.00 GHz);
  - 2 Physical CPUs;
  - 4 cores per CPU;
  - 8 threads.
- RAM: 32GB PC2-5300 (DDR2-667) Fully Buffered Memory;
- NIC: Two embedded NC373i Multifunction Gigabit Network Adapters;
- HDD Controller: HP Smart Array P400i/256MB BBWC Controller;
- HDD:
  - RAID1 – 2x HP 72GB, 2,5" SFF, 15K RPM, SAS;
  - RAID5 – 3x HP 146GB, 2,5" SFF, 15K RPM, SAS;

- GPU: Integrated ATI ES1000, 32MB;
- Management: HP Integrated Lights Out 2 (iLO2).

Looking at the CPU specifications of the servers, it might seem that two physical E5450 CPUs could perform better than one E5630. Some tests were run to figure out, which machine performs faster.

In synthetic single-threaded benchmarks the 3GHz E5450 processors performed better than the 2,53GHz E5630, however when running any of the intrusion detection systems, the newer E5630 had better results (dropping less packets).

This is probably because E5450 is about three years older than E5630. Moreover, the DL320 G6 server has newer HDD controller and DDR3 memory instead of DDR2. The difference in amount of memory did not have much effect, because most of it was not utilized.

Network switch

Linksys 16-Port 10/100/1000 + 2-Port MiniGBIC Gigabit Switch with WebView [37]

- Ports:
    - 16x 10/100/1000 RJ-45 ethernet ports;
    - 2 shared MiniGBIC slots for optical interfaces;
- Switching Capacity: 32 Gbps, non-blocking
- MAC table size: 8KiB

Network cable type used in the testing environment was straight-through CAT5E Ethernet cable.

Annex 2 – Software Versions

This chapter lists the software names and versions used during the testing phase of this thesis.

Operating System

- CentOS 6.3 64-bit
    - Basic server installation
- Linux version 2.6.32-279.11.1.el6.x86_64 (mockbuild@c6b9.bsys.dev.centos.org) (gcc version 4.4.6 20120305 (Red Hat 4.4.6-4) (GCC)) #1 SMP Tue Oct 16 15:57:10 UTC 2012

To install prerequisite packages required by the IDS solutions, Extra Packages for Enterprise Linux (EPEL) repository was added to the yum package manager configuration.

http://mirror.switch.ch/ftp/mirror/epel/6/x86_64/epel-release-6-7.noarch.rpm

Measurement utilities

```
dstat-0.7.0-1
htop-1.0.1-2
```

Snort

`snort-2.9.3.1-1` (released August 6, 2012)
`daq-1.1.1-1`

Prerequisites

```
flex-2.5.35-8.el6.x86_64
bison-2.4.1-5.el6.x86_64
zlib-1.2.3-27.el6.x86_64
zlib-devel-1.2.3-27.el6.x86_64
libpcap-1.0.0-6.20091201git117cb5.el6.x86_64
libpcap-devel-1.0.0-6.20091201git117cb5.el6.x86_64
tcpdump-4.0.0-3.20090921gitdf3cb4.2.el6.x86_64
pcre-7.8-4.el6.x86_64
pcre-devel-7.8-4.el6.x86_64
```

Suricata

`suricata-1.3.2` (released October 03, 2012)

Prerequisites
libpcap-1.0.0-6.20091201git117cb5.el6.x86_64
libpcap-devel-1.0.0-6.20091201git117cb5.el6.x86_64
libnet-1.1.5-1.el6.x86_64
libnet-devel-1.1.5-1.el6.x86_64
pcre-7.8-4.el6.x86_64
pcre-devel-7.8-4.el6.x86_64
gcc-4.4.6-4.el6.x86_64
gcc-c++-4.4.6-4.el6.x86_64
automake-1.11.1-1.2.el6.noarch
autoconf-2.63-5.1.el6.noarch
libtool-2.2.6-15.5.el6.x86_64
make-3.81-20.el6.x86_64
libyaml-0.1.3-1.el6.x86_64
libyaml-devel-0.1.3-1.el6.x86_64
zlib-1.2.3-27.el6.x86_64
zlib-devel-1.2.3-27.el6.x86_64
libcap-ng-0.6.4

Bro
bro-2.1 (released August 29, 2012)

Prerequisites
cmake-2.6.4-5.el6.x86_64
make-3.81-20.el6.x86_64
gcc-4.4.6-4.el6.x86_64
gcc-c++-4.4.6-4.el6.x86_64
flex-2.5.35-8.el6.x86_64
bison-2.4.1-5.el6.x86_64
libpcap-devel-1.0.0-6.20091201git117cb5.el6.x86_64
openssl-devel.x86_64 0:1.0.0-25.el6_3.1
python-devel.x86_64 0:2.6.6-29.el6_3.3
swig.x86_64 0:1.3.40-6.el6
zlib-devel-1.2.3-27.el6.x86_64
file-devel-5.04-13.el6.x86_64
gperftools-libs-2.0-3.el6.2.x86_64
gperftools-devel-2.0-3.el6.2.x86_64
ipsumdump-1.82

PF_RING
PF_RING 5.5.1 (released November 24, 2012)
libpcap-1.1.1-ring
tcpdump.4.1.1

Annex 3 – Suricata CPU Affinity Configuration

Here are listed our modified CPU affinity settings from the suricata.yaml file.

```
  set-cpu-affinity: yes
  cpu-affinity:
    - management-cpu-set:
        cpu: [ 0 ]  # include only these cpus in affinity
settings
    - receive-cpu-set:
        cpu: [ 0 ]  # include only these cpus in affinity
settings
    - decode-cpu-set:
        cpu: [ 0 ]
        mode: "balanced"
    - stream-cpu-set:
        cpu: [ 0 ]
    - detect-cpu-set:
        cpu: [ 1, 2, 3 ]
        mode: "exclusive" #run detect threads in these cpus
        #Use explicitely 3 threads and don't compute number
by using detect-thread-ratio variable
        threads: 3
        prio:
          #low: [ 0 ]
          medium: [ "1-2" ]
          high: [ 3 ]
          default: "medium"
    - verdict-cpu-set:
        cpu: [ 3 ]
        prio:
          default: "high"
    - reject-cpu-set:
        cpu: [ 3 ]
        prio:
          default: "low"
    - output-cpu-set:
        cpu: [ 3 ]
        prio:
           default: "medium"
```