TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
IT College

Balint Adam 201839IVSB

Performance and Applicability Analysis of Open-source Intrusion Detection Systems in Special-purpose Networks

Bachelor Thesis

Supervisor
Gabor Visky
MSc
Co-supervisor
Risto Vaarandi
PhD

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
IT Kolledž

Balint Adam 201839IVSB

Vabatarkvaraliste ründetuvastussüsteemide jõudlus- ja sobivusanalüüs eriotstarbelistes võrkudes

Bakalaureusetöö

Juhendaja
Gabor Visky
MSc
Kaasjuhendaja
Risto Vaarandi
PhD

Tallinn 2022

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author:	Balint Adam	.... (signature)
Date:	04.25.2022	

Annotatsioon

Soodne pind küberrünneteks merenduse valdkonnas on kiirelt kasvamas. Navigeerimise parendamiseks ning käitluskulude vähendamiseks kasutatakse nii reisi- kui ka kaubalaevadel üha enam infotehnoloogilisi abisüsteeme. Käesoleva lõputöö eesmärk on tuvastada sobiv monitooringusüsteem merenduses kasutatavate infosüsteemide turvalisuse tagamiseks.

Abstract

The cyberattack surface in a maritime environment is constantly growing. More modern information and computer technologies are being used on cargo and passenger ships to save on operational costs and increase navigational safety. The goal of this thesis is to find an optimal security monitoring software that could be deployed in such an environment.

List of abbreviations and terms

IDS	Intrusion Detection System
IPS	Intrusion Prevention System
IOC	Indicators of Compromise
NGIPS	Next Generation Intrusion Prevention System
HIDS	Host-Based Intrusion Detection System
NIDS	Network-Based Intrusion Detection System
AIDS	Application-Based Intrusion Detection System
DoS	Denial of Service
URL	Uniform Resource Locator
IP	Internet Protocol
NNIDS	Network Node Intrusion Detection System
HTTP	Hypertext Transfer Protocol
OS	Operating System
SEM	SolarWinds' Security Event Manager
SRS	Snort Rule Subscription
OISF	Open Information Security Foundation
IOM	International Maritime Organisation
NATO CCD COE	NATO Cooperative Cyber Defence Centre of Excellence
RPI	Raspberry Pi
PCAP	Packet Capture
SWaT	Secure Water Treatment
CPU	Central Processing Unit
VM	Virtual Machine
AMD	Advanced Micro Devices
ARM	Advanced RISC Machines
RISC	Reduced Instruction Set Computer
RAM	Random Access Memory
PC	Personal Computer
ANS1	Abstract Syntax One
TCP	Transmission Control Protocol
BPF	Berkeley Packet Filter

TLS	Transport Layer Security
SNMP	Simple Network Management Protocol

Table of Contents

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 What is an IDS? 1

1.2 The background and purpose of IDSs 1

1.3 Main working principles 2

1.4 History of IDSs 2

2 Various IDS types 4

2.1 Place of deployment 4

2.1.1 Application-based IDSs 5

2.1.2 Network-based and Network Node IDSs 6

2.1.3 Host-based IDSs 8

2.2 Intrusion detection methods 8

2.2.1 Signature-based IDSs 9

2.2.2 Anomaly-based IDSs 9

2.3 Marketing model 10

2.3.1 Commercial and open-source IDSs 10

3 Latest and Top IDS Software 12

3.1 Common commercial IDSs 12

3.1.1 Security Event Manager 12

3.1.2 Cisco Firepower NGIPS 13

3.1.3 Snort rule subscription 13

3.2 Open-source IDSs 14

3.2.1 Snort 14

3.2.2 Suricata 14

3.2.3 Zeek 15

4 Limitations 16

4.1 Environment of deployment 16

4.2 Intended task 16

5 Method 18

5.1 Process of experiments 18
5.2 Monitored resources 19

6 Research Environment **20**
6.1 Virtual environment 20
6.2 Physical environment 20

7 Results **23**
7.1 Results in the virtual environment 23
7.2 Results in the physical environment 28
7.2.1 Without attacks 28
7.2.2 Results with attacks 31
7.3 Result comparison between environments 33

8 Applicability **38**

9 Discussion **40**

10 Further works **43**

11 Conclusion **44**

Acknowledgment **46**

Bibliography **47**

Appendices **50**

Appendix 1 - Non-Exclusive License **50**

List of Figures

1 Placement of NIDS and NNIDS 6
2 Placement of NIDS on a network 7

3 Virtual testing environment. 21
4 The physical testing environment in practice. 21
5 Physical testing environment. 22

6 Total time elapsed 24
7 CPU usage 24
8 Time the CPU spent in kernel mode 26
9 Time the CPU spent in user mode 26
10 Memory used 27
11 Number of dropped packets per run with PCAP 1 30
12 Number of dropped packets per run with PCAP2 31
13 Number of dropped packets per run with PCAP 1 with attacks 33
14 Number of dropped packets per run with PCAP2 with attacks 34

List of Tables

1 Summary of the results 37

1. Introduction

As technology evolves, so do the threats related to them. For every new solution, there is going to be a new problem to be solved. The paper aims to lean in on the mostly untouched world of the maritime cyber environment and find answers to some of the obstacles it presents. The safe movement of ships greatly depends on their navigational systems and the integrity of that data, to ensure that, the introduction of intrusion detection systems is necessary.

1.1 What is an IDS?

An intrusion detection system (IDS) detects and logs predefined activities on a network. It monitors system and user actions and recognises patterns typical of attacks[1]. If needed and programmed to, it will notify necessary staff members to check on certain alerts and take further actions. It is important to mention intrusion prevention systems (IPS). The goal of IPSs is similar to what IDSs do however, it is also permitted to take such predefined actions as blocking or altering the given traffic[2]. From here forward, the term IDS will also include functionalities of IPS without explicitly mentioning it.

1.2 The background and purpose of IDSs

The background will be introduced by the following scenario, a ship is cruising, it is the middle of the night, no land in sight, no light sources or anything that would help determine the position and all of a sudden every navigational device goes dark. This is a dangerous situation for multiple reasons: 1) The ship has no way of determining directions so it does not know where to go. 2) It has no way of telling if where it is heading is safe or not. 3) The crew may not be able to call for help. What could have been done to avoid this situation assuming we are dealing with a cyberattack? This is where the use of an IDS and its purpose comes into play.

An IDS monitors and logs the network traffic for signs of malicious activity and generates an alert upon discovery of a suspicious event[3]. Although it means we are not preventing but catching the act, this could help to avoid such disasters from happening. If appropriate personnel is notified in time, the mischievous action could potentially be stopped and

identified before major damage is done to the ship or anyone aboard. So it could be stated that the purpose of the IDSs is to record specified traffic on a given network and, when needed, alert professionals to overview it.

1.3 Main working principles

The IDSs can be differentiated based on their main working principles.

The network-based IDS analyse and detects malicious activity on the network, while the host-based solutions analyse the different resources of the host computer (memory content and usage, CPU load, network traffic, processes, etc.) to identify hostile activity. To achieve this goal the anomaly-based systems detect the deviation of the aforementioned parameters to the state that is considered normal. The signature-based solutions look for signatures [1] in the network traffic or in different data that can be the sign of the presence of malicious software. The application-based IDSs analyse the application-related network data and find different types of protocol abuses.

1.4 History of IDSs

The goal of intrusion detection is to detect unauthorised activities on a given host or network. The concept of IDS was introduced in the 1980s[4]. The task seems simple but in reality, it is much more complicated than it first meets the eye. One could argue that true intrusion detection does not exist since it only identifies evidence of intrusion when ongoing or after[5]. As Jeffrey R. Yost said in an article published about the early intrusion detection systems *"James P. Anderson alone recognised and pioneered the articulation of an early vision for intrusion-detection systems"*, meaning this area of computer science was neglected for a long time[6]. This comes as no surprise since previously computer- and computer network security was based on locked doors and guards. In 1983 SRI International and Dorothy Denning started working on a project about intrusion detection systems[4]. She was one of the first computer scientists to focus on computer security[6]. The very first version of the intrusion detection system that could evaluate audit data in real-time was introduced in the early 1990s[5]. This shows us that the earliest IDSs were host-based (HIDS) but more on that later.

As time passed, more and more features were introduced to the world of IDSs, and their versions and capabilities started to vary. As of today, there are a relatively high number of them available and also a huge amount of studies, papers, reviews etc. The manner of these

[1] Pre-programmed list of known threats and their indicators of compromise (IOCs).

studies is very specific like this paper from Garand Vira Yudha and Rini Wisnu Wardhani *Design of a Snort-based IDS on the Raspberry Pi 3 Model B+ Applying TaZmen Sniffer Protocol and Log Alert Integrity Assurance with SHA-3*[7]. This research focuses on a subset of IDSs and their deployments. Papers like these help a great deal in determining what had and had not been covered in this field. Every researcher can use relevant parts and avoid doing work that has already been done. There are publications that focus on optimisation such as *An IDS Rule Redundancy Verification*[8] by Piyawat Noiprasong and Assadarat Khurat. As more and more IDSs were released there was a need for rule conversion. A fairly up to date paper that analyses the before mentioned topic was released in 2020 [9]. The list could go on about the articles these days.

2. Various IDS types

We can sort IDSs into multiple categories, based on different characteristics. First and foremost, we should take a look at where the IDS is deployed. According to the place of deployment, we can divide the types of software into three main groups:

- Application-based IDSs;
- Network-based IDSs;
- Host-based IDSs.

These classes cover every place an IDS can be deployed. The other major category is how is it deployed or the main working principle of an IDS. We can divide them into two subcategories:

- Signature-based;
- Anomaly-based.

Last, but not least the before mentioned IDSs could be sorted into two main groups:

- Open-source;
- Commercial.

In this section, the meaning of these properties going to be explained along with their introduction, main features, how they work, what are their strengths and what are their disadvantages.

2.1 Place of deployment

From the perspective of where an IDS is used three groups were determined: host-based intrusion detection system (HIDS), network-based intrusion detection system (NIDS) and application-based intrusion detection system (AIDS). Although they are all IDSs their working principles are different.

2.1.1 Application-based IDSs

An AIDS is deployed on layer 7 of the OSI model, the application layer[10]. The OSI model defines the application layer as the interface responsible for communicating with host-based and user-facing applications. Attacks against OSI layer 7 are including finding weaknesses on a web server, not sanitising user input etc[10]. One could argue that this should be considered a NIDS but there are some key differences in their main features.

As for most of the IDSs, the main features are data collection, feature selection, analysis and action[4]. These four main characteristics are going to be referred to as core features and will not be defined later in the document. The core features of an IDS:

Data collection: the IDS captures data that is analysed. Feature selection: basically rule or rule set selection, deciding how to deal with the data. Analysis: data is compared to a predefined signature or pattern. Action: based on the result of the analysis, the IDS decides on appropriate actions, that do not include countermeasures and altering the web traffic.[4]

The unique key feature of an AIDS is that, although it is deployed on a network, it does not actually analyse network traffic but instead the misuse of protocol and service ports[10].

The main working principles of an AIDS are including, but are not limited to:

- Misuse of protocol and service ports: an attacker could modify these to tunnel through firewalls and to make network traffic appear as normal traffic.[10]
- Denial of service (DoS) based on crafted payload: when an intruder creates an attack with crafted internet protocol (IP) packets, the DoS can appear as overuse of computational resources.[10]
- DoS based on volume: in this case, the anomaly is detected because the network overflowed with a large volume of traffic. It is relevant because based on packet content the traffic might not look malicious but the volume is.[10]
- Buffer overflow: this is one is a commonly exploited vulnerability. In certain cases, it can result in the execution of malicious software on the victim system.[10]

By detecting these types of network application-layer attacks the AIDS is able to prevent a great variety of malicious activity either signature or anomaly-based. To stop certain types of attacks a specific version of IDS must be selected. In larger organisations ADISs are usually deployed as an application-layer proxy, that can analyse all incoming and outgoing web requests. One company that provides such appliances is Blue Coat Systems, a company focused on cyber security and network management software and hardware.

Some advantages include but are not limited to detecting attacks that would potentially slip through NIDSs or would not be analysed by HIDS. When placed correctly it can analyse a great amount of network traffic. Can be signature or anomaly-based and is able to detect a significant amount of attacks (Misuse of protocol, DoS etc.). Like every software, AIDS also have drawbacks that include among others that malicious traffic might slip through if the network traffic is encrypted. It may not get all network traffic to analyse when the volume of traffic is overwhelming. To detect application-based attacks it needs a huge number of rules or training sets and that can lead to a large number of false positives before rules and training sets are refined.[10],[4]

2.1.2 Network-based and Network Node IDSs

As for our second category based on deployment, NIDSs are fairly similar to AIDSs however, there are major differences, but more on the working principles later. Another variation of NIDS is the network node intrusion detection system (NNIDS). The difference between a NIDS and a NNIDS is instead of being placed anywhere on the network they are placed on the hosts of the protected network, or in other words, they are placed on the nodes (vertices)[4] as shown in Figure1. Their purpose is to monitor the incoming and outgoing network traffic of the given node.

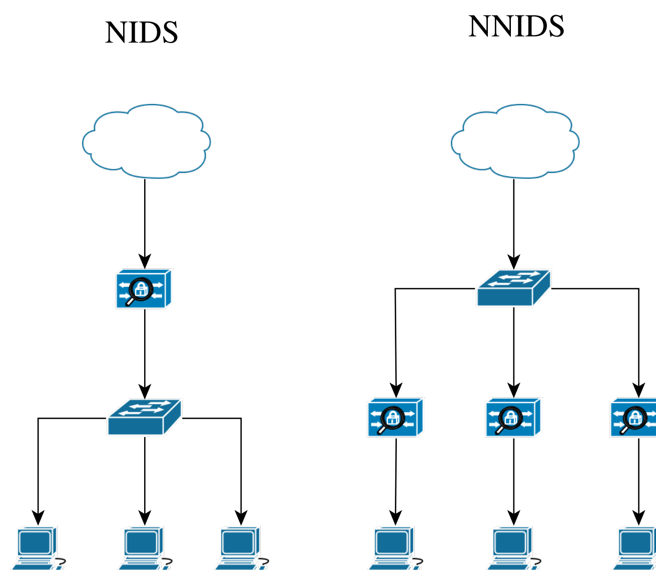


Figure 1. Placement of NIDS and NNIDS

A NIDS could be placed anywhere on the network, it all depends on what we want to monitor and how we want to do it. It can be next to the firewall however that is not an optimal solution since it would be monitoring packets the firewall might block anyway[3]. Another way of deployment is after the firewall. If it is there, the NIDS monitors all

network traffic passed the firewall[11]. A common practice is to have a NIDS in front of the firewall, to gain an understanding of all the incoming attacks against the organisation, and one behind the firewall to gain knowledge of attacks that the firewall was unable to filter. One other common way of placement is on a specific segment of the network, yet not in a way that all network communication flows through it, but in a way that the network traffic is mirrored and forwarded to the NIDS[3]. Figure 2 is an example of the before mentioned positioning of a NIDS. NIDSs are deployed on the network with an aim to protect all devices and monitor traffic before that could reach the hosts[11]. NIDSs audit network traffic as it flows on the network and evaluates the content based on predefined rules[4]. These predefined rules or rule sets can be modified by users if they wish to take care of problems the NIDS was not programmed to deal with. It can detect an ongoing or failed attack against the network as well as log them and alert appropriate users to deal with the problem[4].

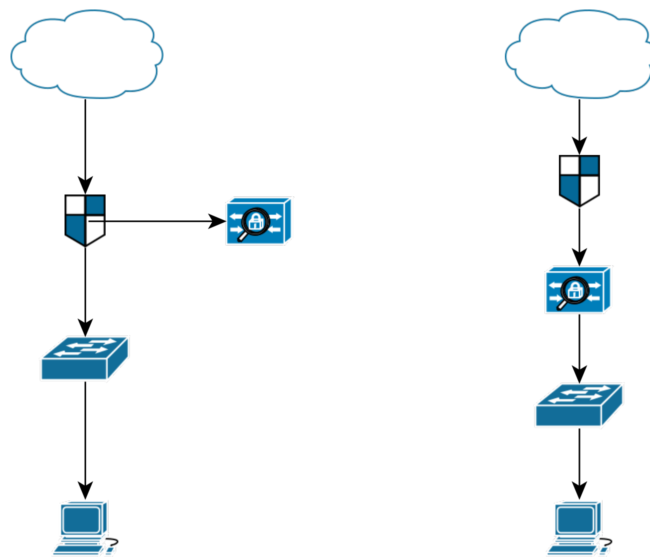


Figure 2. Placement of NIDS on a network

Some advantages of a NIDS include that a single node may cover a whole network or a network segment. Cheaper to deploy and provides real-time monitoring. Fast response, ongoing network operations will not be disrupted when installing nodes. It is possible that attackers cannot find it on the network. If there is a device on the network that requires extra network traffic monitored, NNIDS could be deployed. On the contrary, with huge network traffic, nodes might start dropping packages. Typically cannot analyse the content of encrypted packages. Attacks including legitimately looking traffic may not be detected. Since some switches do not support port monitoring some networks cannot provide all the required data for NIDSs to analyse. Cannot react to malicious uniform resource locator (URL) requests.[10],[4]

2.1.3 Host-based IDSs

The last category based on where an IDS is deployed is HIDSs. Unlike NIDSs, NNIDSs and AIDSs, HIDSs are placed on a single host to monitor. Why they are necessary is, if the monitoring of a whole network or a network segment is not possible, or there is no need to monitor a network, just some hosts on it, HIDS are the best solution?

One could ask the question, why is there a need for a HIDS when there are anti-virus systems? The answer for that is an anti-virus monitors all activities inside a system but that it is not sufficient enough to analyse memory attacks, function calls inside the system, system-specific files, problems with operating system (OS) processes and many more[12]. This is where HIDSs come into play.

HIDS are monitoring the local file system for signs of intrusions or irregularities. They take great advantage of host-side logging such as system logs and other predefined recording mechanisms on the host's side. HIDS can very precisely report what the security breach was, even the details of what the intruder did and how it was done. They are also very efficient in analysing network attacks in the same way, the only drawback is that the attack must reach the host where the HIDS is located. HIDS can also detect improper use of computational resources, this is especially useful for companies to monitor violations of user privileges.

HIDS can also do a detailed analysis of an attack, precisely pinpoint what and how an attack was done, monitor system activities, report attacks against the network originating from the host machine or read data that was encrypted since the host decrypts it. On the other hand, a HIDS require an agent on every monitored host which leads to being more expensive on a relatively large amount of hosts. Host audit logs may occupy a large amount of disk space and it takes more effort to maintain multiple HIDS.[4],[10],[12]

2.2 Intrusion detection methods

As stated earlier, IDSs could be placed into groups based on how they detect an intrusion. It includes two main groups: signature-based and anomaly-based IDSs. They are fundamentally different and have little to nothing in common. One of them is analysing attacks based on predefined rules and rule sets, the other one using "default behaviour" samples to determine if there is something unauthorised going on[13]. Both of them can be any IDSs from the previously mentioned categories.

2.2.1 Signature-based IDSs

One of the two major categories based on how an IDS detects intrusion is signature-based or misuse detection. Most of the attacks have unique signatures based on which they can be identified and this type of IDS uses this to its advantage. They are easy to understand and use since the only thing needed is to learn how to specify a pattern and to decide what types of patterns we want the IDS to trigger.

Signature-based IDSs are using rule sets, which are files or a file, containing a defined traffic pattern that will trigger a reaction that is specified. These rules can contain previous attack patterns, known malicious intrusion sequences, system vulnerabilities etc.

Each intrusion triggers unique reactions like denied access to a file or directory, failed login attempt, failed attempt to run an application etc. These unique patterns are then used to detect and alert when a similar attack is happening in the future. Data is collected from different places such as network traffic, particular resource usage, number of requests from the same IP, etc. After this data is collected and analysed, it could be added to the knowledge base for future uses.

Some advantages of such an IDS include easier detection of attacks if the signatures are well known. In modern systems, pattern matching is optimised thus making it effective and quick. Such systems are versatile and rules might be easier to understand compared to anomaly baselines. On the other hand, the collection of signatures must be constantly updated. It is possible that a signature-based IDS fails to identify unique attacks. Rules may be redundant and use up computational resources to calculate an already evaluated result in a different way.[12],[13]

2.2.2 Anomaly-based IDSs

The other section of IDSs based on how they detect intrusions are the anomaly-based IDSs. Their techniques are fundamentally different from signature-based IDSs so their usage differs greatly. Their perspective is opposite to the signature-based variants, they do not monitor the individual packets one by one, but instead, they are comparing their behaviour to a standard pattern, and if that differs it gets classified as an anomaly.

They cannot rely on rule sets like signature-based versions. They must use a behaviour profile of the system, that defines normal activities and anything that greatly varies from this pattern triggers the countermeasures.

The IDS refers to a behaviour profile that defines the normal behaviour of the system. Any variation from the normal will trigger alarms. This version can detect zero-day exploits. Anomaly detection can be done during run time or later down the line. Just like AIDSs, anomaly-based IDSs are great against protocol or port missuses, detecting DoS attacks with crafted IP packets and other network or resource failures[10]. Attacks that do not have known signatures are also detectable, like new worms or viruses.

Advantages of anomaly-based IDSs include detecting new versions of attacks with no known fingerprints. Behaviour profiles can get more advanced as time passes on. Customised profiles for different networks and applications can be created to detect attacks that are unique to a system. However, it might be hard to understand and make a profile. It is challenging to find the boundary between normal and abnormal behaviour. All protocols analysed must be well defined and tested for accuracy, otherwise, it might associate malicious behaviour as normal.[10],[12],[13]

2.3 Marketing model

Some of the important properties of software were discussed but a key property is still unspoken, the marketing model. As in every field, there are commercial and open-source versions of IDSs. Which one is better greatly depends on factors like, what is it going to be used for? Or who is going to use it?

2.3.1 Commercial and open-source IDSs

The main difference between these two types is that most of the time the source code of a commercial IDS is not bought with the product and an open-source IDS is free. There are of course exceptions. An open-source model usually means that the source code is open to the public, so anyone can download and modify it to their liking. It is also commonly true that the development of this software is community-driven. Some well known open-source licenses are GNU general public license and Apache License. More about these licenses can be found on their official website at [14] and at [15].

Most of the time commercial software is developed by a given company, they do not sell the source code, they sell the software. As mentioned previously, there are exceptions, there is open-source software that is monetised and there is commercial software that is highly modifiable and arrives with source code. Open-source software is usually highly customisable, free and has good community support. However, problems might not have solutions documented. Finding the correct version for an individual task might be

challenging. Might require a lot of extra research to make such software work properly.

Commercial license software usually has strong support from the entity that developed it. They are mostly well documented and contain quality guarantee, on the other hand, commercially licensed software might be expensive. It might not contain options to customise the product.

A common practice is to use an open-source IDS with commercial signature packs. Signature packs are significantly cheaper than purchasing a commercial IDS so this combination would result in an optimal cost-benefit ratio.

3. Latest and Top IDS Software

The paper covered various types of IDSs based on deployment, working principle and marketing model. In the next section, the focus is shifted toward specific IDSs. Some commercial and open-source IDSs are going to be reviewed focusing on their main features, strengths, disadvantages and where they belong on the spectrum discussed so far.

3.1 Common commercial IDSs

Most of the commercial IDSs are not transparent, meaning there is not much information about what principles they use and there are not many studies on them since they are expensive. They are hard to categorise because they are made for multiple purposes and IDS functionalities are just one part of them. Most of the information about them is only on their websites.

3.1.1 Security Event Manager

SolarWinds' security event manager (SEM) is a commercial IDS. Because of how commercial products work, it is very hard to find research related to them but fortunately, the company's official documentation is available to the public. SolarWinds is a software company that mainly develops system management, monitoring and other technical tools. Their product, the SEM, is a versatile application for a lot more than just being an IDS, but it is part of its main features.

It can collect logs from the system to monitor and analyse them as well as application or security logs. It can also log network devices like firewalls, routers, switches etc. SEM can monitor events in real-time and also safely transfer data. It is not stated specifically what type of IDS is the SEM, but according to the documentation, it seems close to a signature-based NIDS or NNIDS.

The document mentions the biggest advantages of the software include capturing events in real-time, buffering the events locally if network connectivity is lost to SEM and encrypting and compressing the data for efficient and secure transmission to SEM. However, these might come at a cost, for example, encryption and compression take time and computational

resources. The installation documentation is vague, and the SEM reports application is Windows only. [16]

3.1.2 Cisco Firepower NGIPS

Cisco Firepower NGIPS is also a very versatile software. The name NGIPS stands for new generation intrusion prevention system. An IPS differ from an IDS but to no great extent, the main difference is that an IPS does an active countermeasure, beside making a log or sending an alert, it may drop a package or do something else that is specified as action.

Cisco Firepower NGIPS can be bought as an individual software or with hardware as well. It monitors the target in real-time and automates security. Cisco Firepower NGIPS could be deployed as HIDS or as a NIDS as well. The software's base is Snort, one of the most popular IDS. Cisco Firepower NGIPS can be configured as signature or anomaly-based making it a very flexible application. It automatically upgrades the rule sets and the behavioural profile, by constantly monitoring the traffic, discovering information about the network, the OS, devices etc.

Cisco Firepower NGIPS has constant upgrades, protects against advanced threats, is versatile and could be deployed almost anywhere in any way according to the documentation. However it has little to no documentation on operation, it is heavy on the budget, and creating accurate behavioural profiles for a new network is time-consuming. [17]

3.1.3 Snort rule subscription

Snort rule subscription (SRS) is the commercial version of snort. Since in the next chapter one of the IDSs in focus going to be snort, this section will not contain specific information about it, just the things included in the subscription. SRS provides the user with three subscription options: personal, business, and integrator.

A personal subscription provides the user with a time advantage because new rules will be available immediately, not thirty days after release. The users can submit if a rule provides false positives or negatives and help improve the rule sets.

The business version offers instant rule set usage from a higher-level group. They also receive priority customer support status when reporting false positives or negatives.

Integrator's subscription is for those who would like to integrate snort into their application

or server and release it for commercial use. It enables them to integrate official rule sets and distribute them. [18]

3.2 Open-source IDSs

They are more transparent, a lot of communities use them, and there are a greater number of studies conducted using these. They are mostly free and highly customisable. Use these to solve problems that commercial versions are not fit for, because the source code is modifiable to a great extent. Exact solutions to problems might not be available, yet due to the high community support solving those should be easier.

3.2.1 Snort

Snort is one of the most popular open-source IDSs available. It was originally created by a company named Sourcefire that now belongs to Cisco. As it is stated in the official Snort documentation, the software is a NIDS that uses signature-based detection to alert in case of malicious activities. Snort has versions 2 and 3 with the latter being the more advanced, including features like multithreading[19].

It has five main components: packet capture, packet decoder, preprocessor, detection engine, and output. First Snort will collect all network traffic, analyse the type of packets then forward it to the processor. The processor will reassemble the packets and if any of them triggered a rule, appropriate actions will be taken[19]. Snort can also run on Linux, BSD, macOS X and Windows[20].

Snort has good documentation online that is easily accessible and describes most features in detail. On the other hand, created rules might be redundant and contain no HIDS and anomaly-based capabilities.

3.2.2 Suricata

In Suricata's official documentation, it is stated that it also is a signature-based NIDS. It is being developed by the Open Information Security Foundation (OISF). Actually, it is very similar to Snort in a lot of senses for example it is also licensed under version 2 of the GNU general public licence. Suricata runs on Linux, BSD, macOS X and Windows platforms. Suricata uses a packet decoder and detection engine[19].

Since Suricata and Snort are similar there is a study on how to convert rules between the

two IDSs called *"Rule conversion mechanism between NIDPS engines"*[9]. IDS Suricata uses af_packets to improve performance[19]. In Suricata's packet decoder, every packet is converted to a data structure supported by Suricata. Rules used by the IDS support layer 3, layer 4 and layer 7 of the OSI model[19]. Suricata can keep up with dense network traffic. It has a multi-threading feature and good documentation with video guides by the OISF. Something that might be inconvenient is that it generates big log files by default but it has good documentation that describes configurations thoroughly.

3.2.3 Zeek

Zeek has a very long history as an IDS. The very first version dates back to the 1990s. It was designed and developed by Vern Paxson. The original name of the software was called Bro but its name was later changed to Zeek. The success and popularity of Zeek reside in top tear academic studies as well as in the fact that they try to connect academic knowledge with everyday problems and solutions.

Zeek is a NIDS that uses anomaly-based detection and can run on UNIX based OSs[21]. Zeek is designed to analyse live network traffic although it can process captured network traffic. Zeek is a great tool to use in an industrial environment since it can be deployed using the Modbus protocol[22]. Zeek uses its own scripting language.

Monitoring is highly customisable and it is possible to detect attacks without known signatures. Zeek has good official documentation. However, it needs a lot of storage space and takes more effort to install. It can only run on UNIX based OSs.

As we can see every one of these studies are focusing on one very specific argument and that is why this paper is created, to summarise, cover the research gaps and update the information regarding its topic.

4. Limitations

In this research, three NIDSs will be analysed: Snort version 3, Zeek and Suricata. The reason why this software was chosen is that all of them are open-source and highly customisable. There are a lot of aspects that must be considered when executing a project and among those are the limitations. There are additional reasons why only NIDSs were selected for testing, like the environment of deployment and the main characteristics of the system, in which the IDS will be deployed.

4.1 Environment of deployment

On larger networks, for example on an autonomous vehicle, there are a lot of nodes, so inspecting every single one of them requires a large number of HIDSs or NNIDSs. This would require a lot of time to set up and maintain. Besides time, it would increase the cost to monitor such a network since an IDS should be deployed on or in front of every node.

There are other circumstances that must be taken into consideration. In safety-critical industries, head organisations may require special specifications to be met. For example in off-shore transportation International Maritime Organisation (IMO) require ship producers to install certified components on the ships, meaning that for every node where a HIDS would be installed, negotiations should be done with the manufacturers so their product is certified with the HIDS installed. This process would be too complicated to carry out. An AIDS would be impractical because the content of the network traffic must be analysed. AIDS solutions exist for common networking protocols, such as hypertext transfer protocol (HTTP), while the internal network of a ship uses less common, specialized solutions to which, there is no AIDS solution. This leaves the NIDSs as the only viable option.

4.2 Intended task

Current research will focus only on the speed of the IDSs, along with their resource usage and not their performance from the perspective of intrusion detection, meaning that the detection and false-positive rates are not tested. This is due to the fact that the final product will use a custom function to evaluate the data passing through it, so the effectiveness of the base IDS is not in the first place to be analysed. However, if an IDS cannot keep up

with the traffic due to speed limitations, some important data may not be evaluated and attacks could slip through.

5. Method

The method was designed to align with other research conducted at the NATO Cooperative Cyber Defence Centre of Excellence (NATO CCD COE). The research environment was realised as a virtual environment first, to allow the method testing, while the components of the physical environment were procured.

The virtual environment is based on VMware player and the installed OS is an Ubuntu server. The physical environment contains Raspberry Pis (RPI) and the installed OS is Raspbian Lite (Legacy) which is a Debian version 10 (buster) based OS. The equipment to create both the virtual and the physical environment was provided by NATO CCD COE as well as the source traffic used to simulate a live network in the form of packet capture (PCAP) files. These PCAPs were recorded in a live network of Crossed Swords Cyber Exercise. To test the behaviour of the system with malicious packets Secure Water Treatment (SWaT) dataset was used[23].

5.1 Process of experiments

According to [24], traditionally networks have been analysed in two ways. One of them is using software tools, running on off-the-shelf workstations and network cards. This approach is easy on the budget but usually not considered a good approach on high-speed networks due to hardware limitations. The other is using custom hardware which is suitable to use on high-speed networks but considered expensive. To measure the performance of such tools on a network, a utility must be selected, that is able to perform calculations necessary for the experiment. An example of such a tool is PFCount[1].[25].

To measure speed and hardware usage, the GNU time tool was selected. Since every NIDS is on a different VM, and later on a different physical computer, the measurements can be conducted parallel because it does not affect the data. For every different setup, an average is calculated at the end of their testing phases.

The GNU time tool runs another program and then displays information about the resources used by that command, and the execution time[26].

[1]PFCount is a command used to return the number of unique entries from a HyperLogLog data structure.

A detailed description of how the measurements were conducted, including script templates and the log files, can be found in the public GitHub repository of the project at [27].

5.2 Monitored resources

In the case of this research, the measurements contain the following values: total time elapsed (real-time), the total number of central processing unit (CPU) seconds that the process spent in user mode, the total number of CPU seconds that the process spent in kernel mode, average CPU used in percentage, maximum resident set the size of the process during its lifetime (maximum memory used). These monitored resources cover the critical parts which could be bottlenecks of the hardware in live deployment.

To gain accurate results, the tests are automated. They are started with a cron job[2] and stopped automatically when the given script finished execution. The GNU time utility has a built-in option to save measurements to a file and to append further data without overwriting the previous ones. These logs contain all the data collected during the testing phase.

In every software, there are going to be bottlenecks. Identifying them is crucial if speed and performance should be increased. One way of doing that is altering the source code and finding optimal algorithms instead of those in use[28]. Another way is omitting unnecessary actions during execution. In the case of this research, these could be unnecessary rules which are included as a basic but not useful in the case of a special-purpose network.

[2] A job scheduler on Unix-like operating systems

6. Research Environment

In the early parts of testing, a virtual environment is used. All of the NIDSs are installed on a different virtual machine (VM) and they are placed on the same network. All VMs have the same specifications, 4GB random access memory (RAM), CPU with 4 cores and 30GB storage space.

After finishing the tests in the virtual environment the experiment will be transported to a physical environment. This physical environment contains three RPI4s and a desktop personal computer (PC). The network traffic is simulated with the PC and the IDSs are installed on the RPI4s which are placed on the same network, connected with a switch.

The results of these test runs are recorded, evaluated and used to select the final software that will be used to develop the final product as part of another research.

6.1 Virtual environment

The virtual environment, as can be seen in Figure 3, consists of three guest OSs. All of them have an IDS installed along with the traffic generator. All three of these VMs are placed on the same network. The OS of the VMs is a Debian based Linux distribution. It was chosen this way because in the physical testing environment the IDSs are installed on RPIs and their OS the Raspbian is also Debian based. In the virtual environment, there is no need for any extra equipment, tests could be conducted once everything is set up.

6.2 Physical environment

The physical testing environment is more complex. There is no virtual network that connects every host so the IDSs must be connected using a switch, as shown in Figure 4 and as it can be seen in Figure 5.Figure 4 shows the implementation of the testing environment.

After the RPIs are ready to be deployed with all the necessary software and IDSs installed, they are plugged into three different ports of the used switch. All those ports are getting the same traffic from the traffic generator, which is connected to another port of the device.

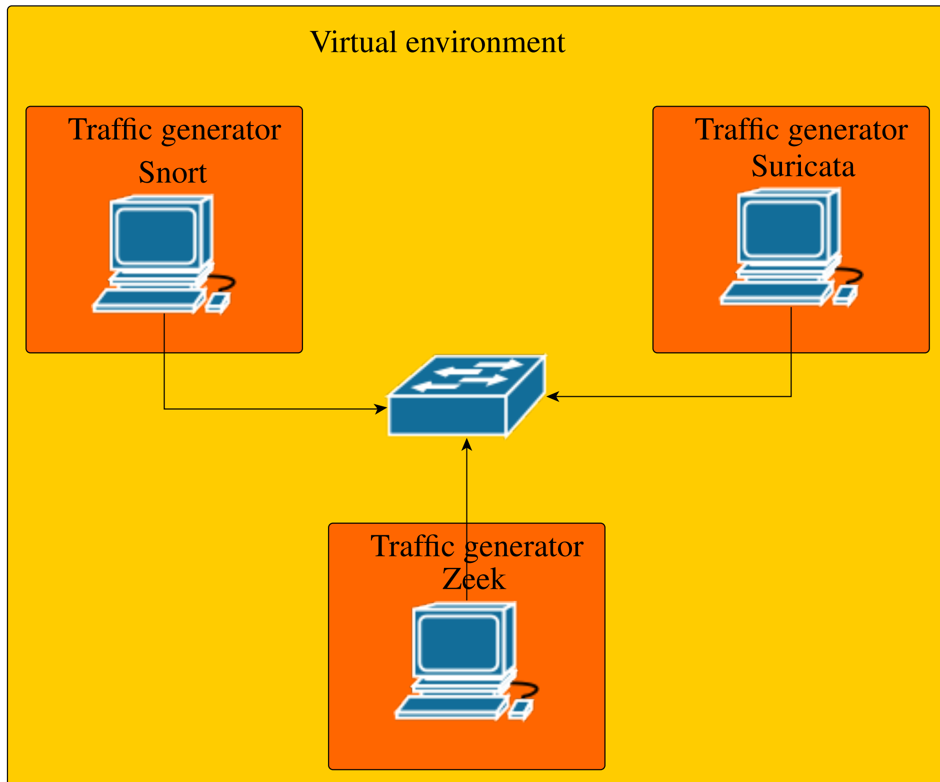


Figure 3. Virtual testing environment.

That one port to which the traffic generator is connected is then mirrored to all three ports used by the RPIs.

Figure 4. The physical testing environment in practice.

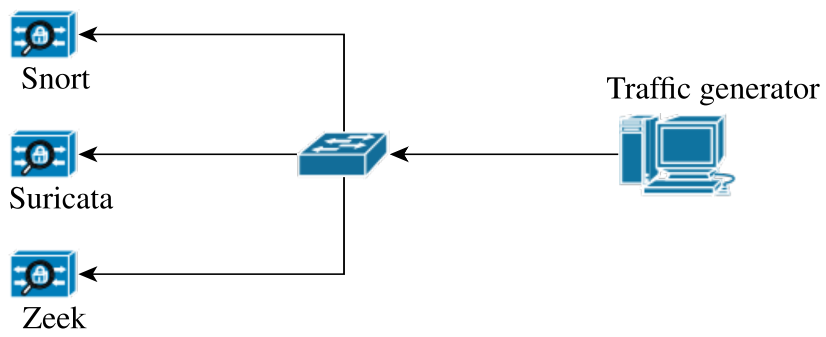


Figure 5. Physical testing environment.

7. Results

All the results are calculated based on the custom logs that were created during the measurements in the physical and virtual environments. These logs were separated into multiple files using unique python scripts that were created to sort the values of individual resources to separate files and calculate the mean value of the given resource. This method allows to minimise the possibility of human error during calculations and also speeds up the process to analyse logs.

The IDSs were deployed with their default configurations along with some minor changes to what logs they generate and changes so they could use all of their included default signatures. In the case of Snort version 3, there were no default signatures included, so the snort3-community-rules.tar.gz packet was used[29].

7.1 Results in the virtual environment

The first result from the virtual testing environment is the total time elapsed. These numbers are the real-time needed for the IDSs to finish their runs. Since the terminating processes were controlled by the traffic generators, a longer run time means heavier system resource usage by the IDS. In the log files these numbers are represented in humanly readable format *hh:mm:ss* and also as seconds to two decimal places. On the graphs, to represent the most accurate results, the time is represented as the later. According to the maximum values, Snort version 3 had the longest run time of all the IDSs with ($t_{max_{Snort}} = 12,197.93s$) with Suricata was the second ($t_{max_{Suricata}} = 10,561.78s$) and Zeek ($t_{max_{Zeek}} = 9,431.78s$) in the third place with the shortest runtime as shown in Figure 6 (a).

However, calculating the average total time elapsed from all the measured data reveals, that Snort is not much slower than the other IDSs. If we take a look at Figure 6 (b) it clearly shows that the difference between Snort ($\overline{t_{Snort} = 9,941.17s}, \sigma = 345.49$)[1] and Suricata ($\overline{t_{Suricata}} = 9,937.27s, \sigma = 171.82$) is just $\overline{t_{diff}} = 3.9s$, while Zeek ($\overline{t_{Zeek}} = 9,002.84s, \sigma = 150.47$) has a shorter runtime $\Delta t_{Snort,Zeek} = 938,33s$ on the same PCAP file.

[1] σ (sigma), standard deviation

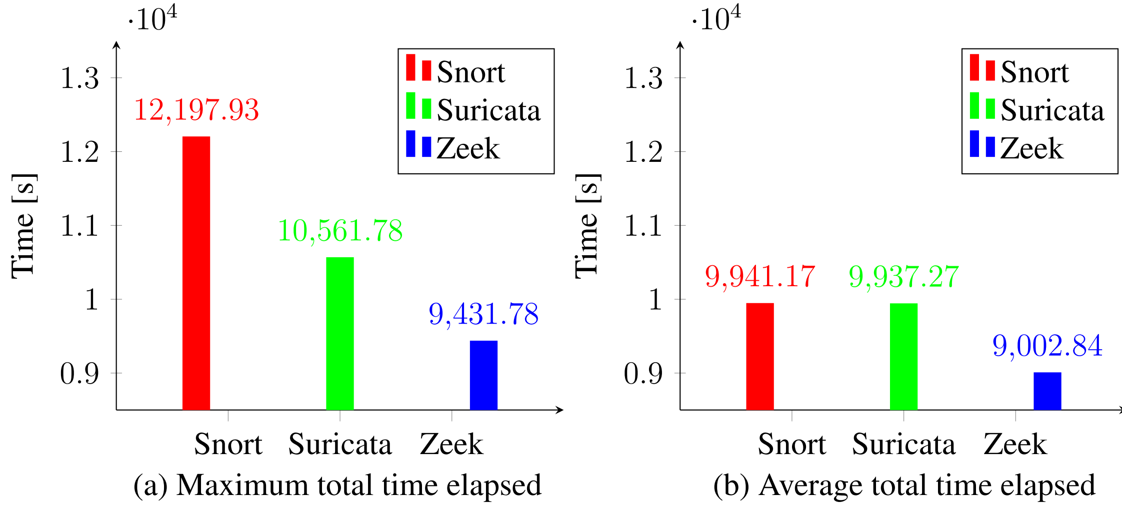


Figure 6. Total time elapsed

With the help of GNU time, overall CPU usage was also measured. This data is an average on its own, the time utility records what percentage of the CPU is in use during the runtime of the command and at the end calculates the average. Out of these measurements, the maximum values are also represented, for all the IDSs. As it can be seen in Figure 7 (a), the maximum CPU usage of the different software as follows: Snort $CPU_{max_{Snort}} = 21.12\%$, Suricata $CPU_{max_{Suricata}} = 23.23\%$, Zeek $CPU_{max_{Zeek}} = 8.08\%$. The comparison of the values shows the considerable differences for Zeek, but Snort and Suricata brought similar results.

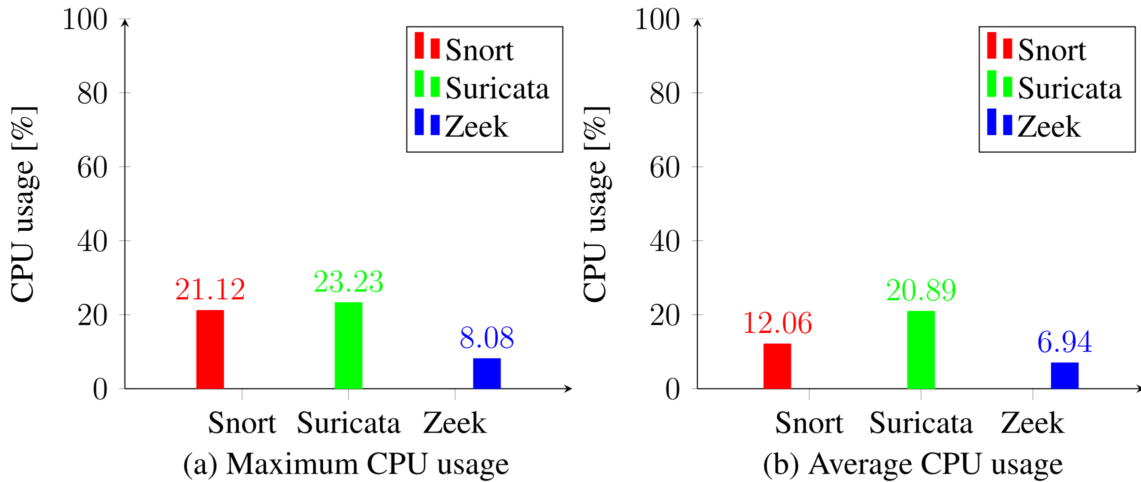


Figure 7. CPU usage

Just as in every other case, from all the logged data, an average was calculated for every IDS in use. With the average calculated it can be observed that Suricata ($\overline{CPU_{Suricata}} = 20.89\%, \sigma = 0.79$) has fallen behind Snort ($\overline{CPU_{Snort}} = 12.06\%, \sigma = 1.58$) compared to the maximum values however, Zeek ($\overline{CPU_{Zeek}} = 6.94\%, \sigma = 0.5$) still had the best numbers out of the three. These values are represented too in Figure 7 (b). In some cases,

it would be possible to have CPU usage above 100%, for example, if the monitored process is multithreaded, but this phenomenon did not appear during the virtual testing phase.

It is possible to dissect the overall CPU usage into smaller categories. A process can run in two ways on a system and these are user mode and kernel mode. With their names respectively a user process runs in user mode and a kernel process runs in kernel mode. The fundamental difference between the two is that a process running in kernel mode has unrestricted access to the processor and to the main memory. This is a very useful privilege because a process running in kernel mode can make system calls which are necessary to access system resources, for example read from and write to files. It could also crash the system if not configured correctly, but such unstable kernels are not used in production under normal circumstances. The user mode on the other hand has limited access to system resources that results in a safer execution since if a user process crashes the damage done can be cleaned up by the kernel.[30]

As mentioned above time spent in kernel and user mode was also measured during the testing phases. Figure 8 (a) shows the maximum time spent in kernel mode in seconds. As we can see the kernel-mode graph is proportional to the overall CPU usage. Snort ($t_{max_{kernel_{Snort}}} = 1,284.92s$) and Suricata ($t_{max_{kernel_{Suricata}}} = 1,299.96s$) once again are very close to one another with maximum values and Zeek ($t_{max_{kernel_{Zeek}}} = 251.09s$) spends the least amount of time in the kernel space.

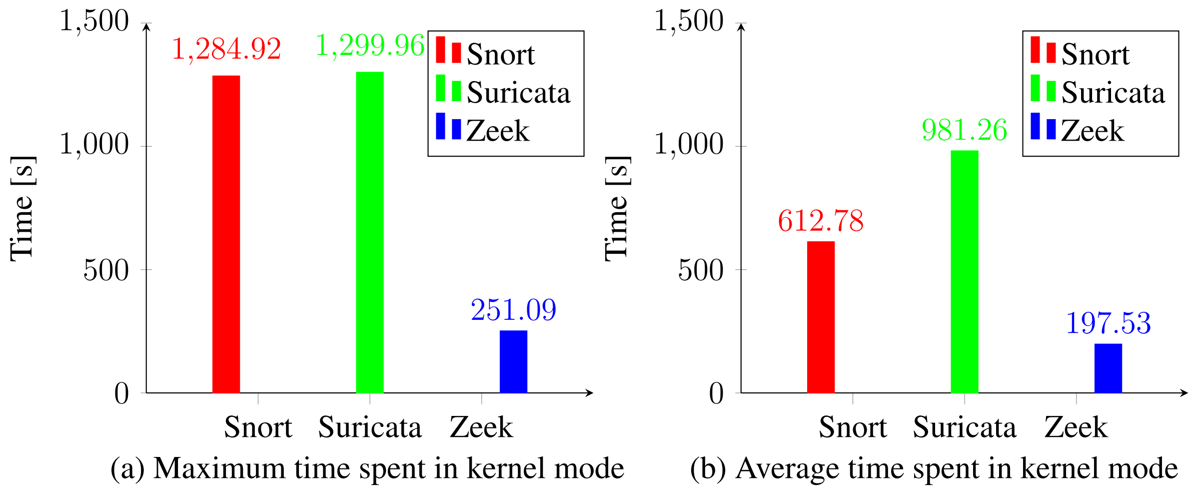
The graph representing the average values also fits the pattern mentioned in7 (b). Snort on average spends less time in kernel mode than Suricata with Zeek spending less amount of seconds there.

($\overline{t_{kernel_{Snort}}} = 612.78, \sigma = 102.47$) with Suricata ($\overline{t_{kernel_{Suricata}}} = 981.26, \sigma = 67.64$) being a few seconds behind while Zeek ($\overline{t_{kernel_{Zeek}}} = 197.53, \sigma = 40.20$).

In case of user mode, it can be observed in Figure 9 (a) that Snort spent the longest time in the user space ($t_{max_{user_{Snort}}} = 1,290.49$), Suricata was the next ($t_{max_{user_{Suricata}}} = 1,253.45$), and Zeek ($t_{max_{user_{Zeek}}} = 578.62$) spent the least time there.

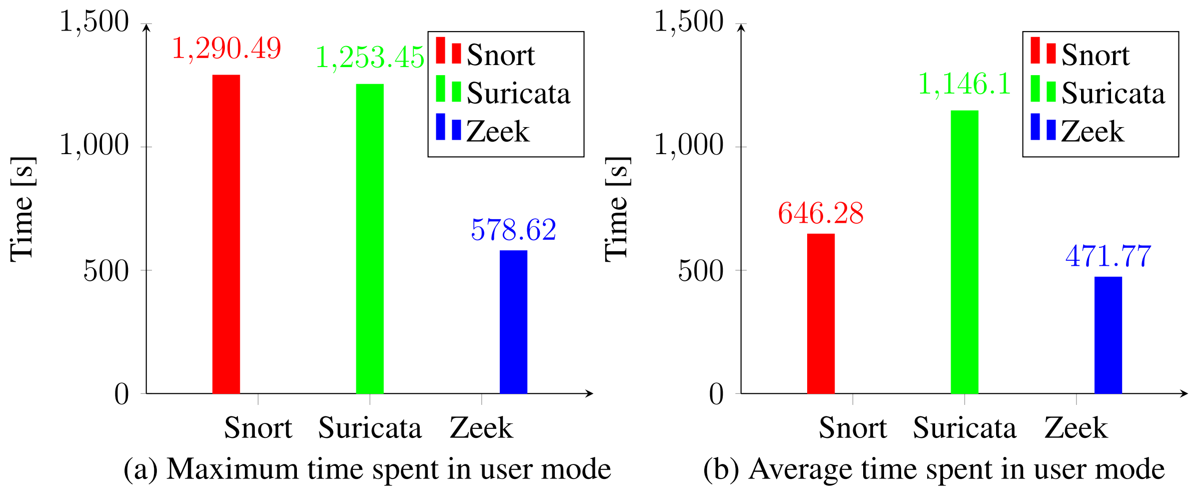
However, if we compare the graphs of time spent in kernel mode and time spent in user mode, the results show that, while Snort and Suricata spent nearly identical amounts of time in both spaces Zeek spent significantly more in user mode than in kernel mode.

The average graph in Figure 9 (b) follows the pattern of the average graph in kernel mode with Suricata spending the most time there ($\overline{t_{user_{Suricata}}} = 1,146.1, \sigma = 67.64$) ,



(a) Maximum time spent in kernel mode (b) Average time spent in kernel mode

Figure 8. Time the CPU spent in kernel mode



(a) Maximum time spent in user mode (b) Average time spent in user mode

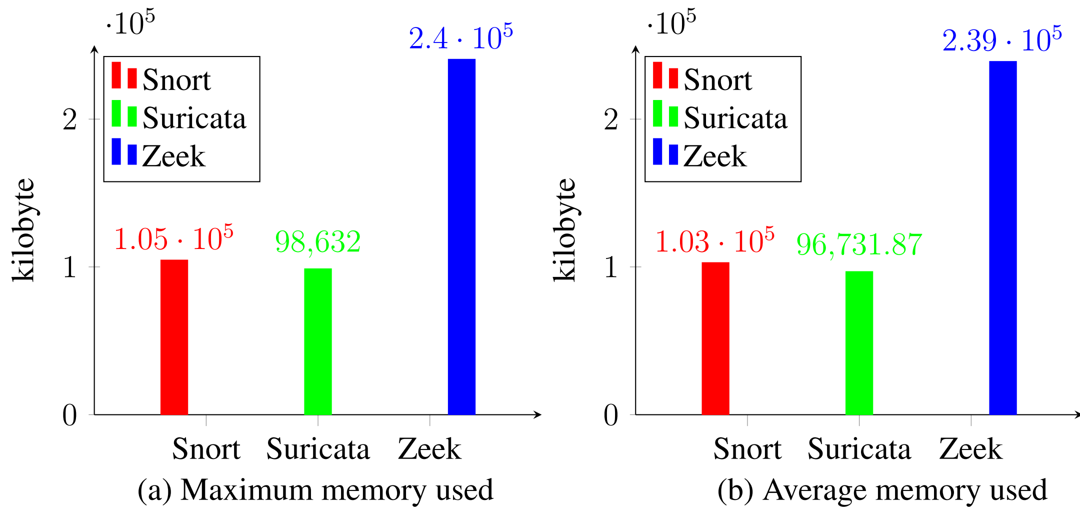
Figure 9. Time the CPU spent in user mode

Snort having a much better result ($\overline{t_{user_{Snort}}} = 646.28, \sigma = 102.47$) then Zeek comes ($\overline{t_{user_{Zeek}}} = 471.77, \sigma = 40.20$).

By monitoring these resources we can get a deeper understanding of how the process operates in the kernel and what risks they carry. If a process spends relatively a lot of time in kernel mode that could mean that there is a bigger chance of some operations failing and crashing the system. However, if the opposite is true then the process might not get access to resources that would be needed for optimal operation.

Only one monitored resource is left in the virtual environment and that is the memory consumption of the IDSs. The memory could also be a bottleneck when doing a huge number of calculations therefore it was also added to the list of things to be logged during the research. If the given IDS consumes a large amount of memory it slows down the system it is running on. This could potentially make the system vulnerable since if the

system cannot operate properly the IDS's efficiency will also decrease. This could lead to undetected activities on the network that could come with critical consequences.



(a) Maximum memory used

(b) Average memory used

Figure 10. Memory used

In the Figure 10 the recorded data shows that not relying on the processor that heavily did have some drawbacks. As is shown in the graph of the maximum memory consumption (a), Snort ($Memory_{max_{Snort}} = 104,524kB$) and Suricata ($Memory_{max_{Suricata}} = 98,6321kB$) are again produced very similar values, Suricata is a bit behind Snort. The same graph also shows that Zeek ($Memory_{max_{Zeek}} = 240,376kB$) used, by far, the most amount of memory during the run. The calculated average introduced in the graph (b) shows similar results. The values of Snort ($\overline{Memory_{Snort}} = 102,801.73kB, \sigma = 426.44$) and Suricata ($\overline{Memory_{Suricata}} = 96,731.87kB, \sigma = 359.42$) got closer to each other with the memory consumed by Snort and Suricata lowering to some extent. The average of Zeek ($\overline{Memory_{Zeek}} = 238,902.75kB, \sigma = 561.17$) changed also just a minor amount thus keeping the proportions as they were in the maximal values graph with Suricata needing the less amount of memory, Snort needing slightly more and a huge difference with Zeek needs the most out of all the IDSs.

7.2 Results in the physical environment

Results in the physical testing environment can be divided into two subcategories:

- Without attacks
- With attacks

During the physical testing phase, to stress test the IDSs, some simulations were done using PCAP files that contained malicious activity. The main goal of using maliciously crafted packets was to record how the IDSs process the packets and if they start to drop some of them. The resource consumption is not important in this case, since the end result of the parent research is going to be an IDS with a custom function that will evaluate data. However, to demonstrate the performance in a real-world-like environment, the same system resources will be analysed with the addition of the number of dropped packets.

The reason why stress testing is only applied in the physical environment is that the goal of the research is to determine the performance of the IDSs in limited hardware environments. Tests conducted in the virtual environment give an opportunity to compare the performance of the IDSs in both environments however, stress tests are not necessary since, with the understanding of the base performance, conclusions can be drawn about how the attacks affect the given IDS.

The total time elapsed is omitted in this part of the research since now there was only one traffic generator and when it finished the simulation of a given PCAP it stopped all the IDSs at the same time.

7.2.1 Without attacks

The PCAPs' sizes used to simulate the network traffic without attacks were significantly bigger than the ones that contained malicious crafted packets. In these tests, the first prerecorded network traffic (PCAP 1) is the same file that was used in the virtual environment. This way it can give a comprehensive result of how a given IDS can perform with limited hardware resources.

The CPU percentage used by the IDSs is as follows, Suricata used a slightly greater portion of the CPU with Snort being second and Zeek using the least when the maximum values are taken into consideration. When it comes to averages Suricata ($\overline{CPU} = 32.5\%, \sigma = 17.83$) outperformed all the IDSs, even Zeek ($\overline{CPU} = 39.20\%, \sigma = 0.40$), while Snort

($\overline{CPU} = 43.01\%, \sigma = 0.09$) used the most CPU percentage. The simulation with PCAP 2 has a very similar result, with only some minor deviation from the previous ones in Suricata ($\overline{CPU} = 31.91\%, \sigma = 18.00$), Snort ($\overline{CPU} = 43.02\%, \sigma = 0.16$) and Zeek ($\overline{CPU} = 39.60\%, \sigma = 0.49$).

The proportions of the maximum and average time spent in the kernel space with PCAP 1 being played as traffic are Snort ($\bar{t} = 870.87s, \sigma = 5.47$) spending the most and Suricata ($\bar{t} = 496.02s, \sigma = 293.21$) the least amount of time there while Zeek ($\bar{t} = 760.64s, \sigma = 7.18$) being in between. Results of the time spent in kernel mode while PCAP 2 was played on the network are very similar, the difference is minimal ($\overline{t_{kernel_{Snort}}} = 874.37s, \sigma_{Snort} = 4.63, \overline{t_{kernel_{Suricata}}} = 489.50s, \sigma_{Suricata} = 295.34, \overline{t_{kernel_{Zeek}}} = 768.67s, \sigma_{Zeek} = 7.84$).

Suricata spent the most time in user space by far, if maximum values are analysed, with Snort being second and Zeek being third with a few hundreds of milliseconds behind. Upon inspecting the averages the results show that the three IDS spent almost the same amount of time in user space with a few seconds difference ($\overline{t_{user_{Snort}}} = 537.34s, \sigma_{Snort} = 5.08, \overline{t_{user_{Suricata}}} = 569.31s, \sigma = 276.06, \overline{t_{user_{Zeek}}} = 525.27s, \sigma = 5.45$). When PCAP 2 was used to simulate the network traffic the results were very similar with every value being slightly less than in the case of PCAP 1 ($\overline{t_{user_{Snort}}} = 536.68s, \sigma_{Snort} = 5.55, \overline{t_{user_{Suricata}}} = 557.31s, \sigma_{Suricata} = 278.51, \overline{t_{user_{Zeek}}} = 524.43s, \sigma_{Zeek} = 5.66$).

The last hardware resource that was analysed without attacks in the simulation is memory consumption. Zeek ($\overline{Memory} = 212,634.69kB, \sigma = 61.53$) uses the most memory both when checking absolute maximum values and when comparing averages. Suricata ($\overline{Memory} = 66,522.72kB, \sigma = 354.19$) consumed the least while Snort ($\overline{Memory} = 85,655.53kB, \sigma = 131.29$) used more than Suricata but significantly less than Zeek when deployed on the network traffic simulated by using PCAP 1. When PCAP 2 was used to simulate a live network the memory usage of Zeek ($\overline{Memory} = 215,095.07kB, \sigma = 659.28$) increased further along with Suricata ($\overline{Memory} = 67,444.83kB, \sigma = 387.67$) and Snort ($\overline{Memory} = 89,055.65kB, \sigma = 136.83$).

Observing the data recorded during the simulation of PCAP 1 in Figure 11 it is clear that the results are in fact very different to those containing malicious activities. Suricata still did not drop any packets contrary Snort dropped 22-214 packets. This amount can still be considered minor but it is more than we experienced under the pressure tests when all the packets were processed. Although the graph of Zeek looks similar to the one that represents the results of pressure tests, however, the amount decreased significantly to 1990-1995 packets. Results representing when PCAP 2 was played on the network could be found in Figure 12. Here Suricata also drops 0 packets while Snort 39-208 and Zeek

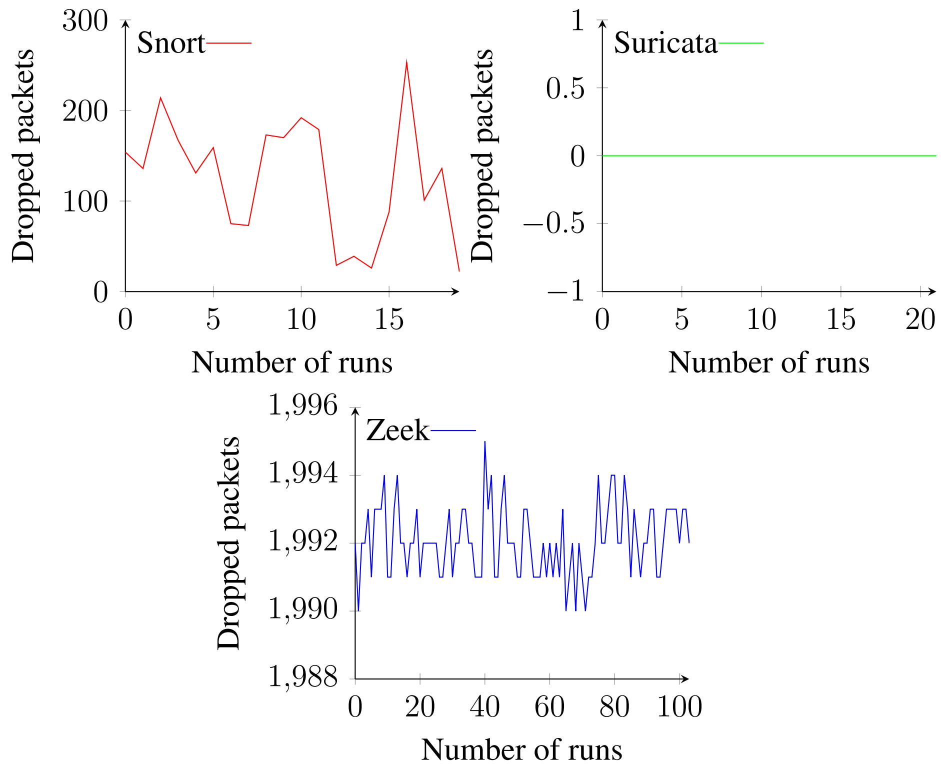


Figure 11. Number of dropped packets per run with PCAP 1

1991-1995 packets.

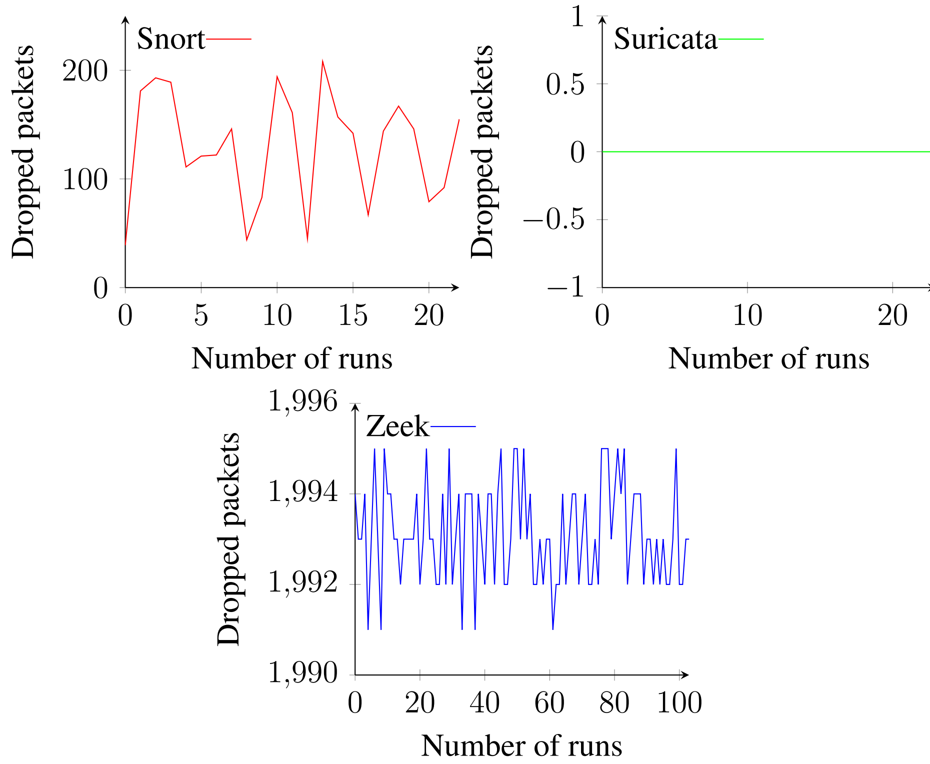


Figure 12. Number of dropped packets per run with PCAP2

7.2.2 Results with attacks

In this section, the two PCAP files that were used to simulate network traffic contained maliciously crafted packets. These PCAPs were used to test real industrial networks for vulnerabilities thus their content is fit to simulate authentic attacks. The length and size of these files are less than those that did not contain malicious traffic.

The results of the simulation with the first PCAP (PCAP 1) file, Snort ($\overline{CPU} = 29.0\%, \sigma = 0.0$) used the most amount of CPU percentage, Suricata ($\overline{CPU} = 15.0\%, \sigma = 0.0$) was the next in line and Zeek ($\overline{CPU} = 8.35\%, \sigma = 0.48$) used the least. The results of the second PCAP (PCAP 2) are nearly identical to the graph of PCAP 1. Suricata performed ($\overline{CPU} = 15.0\%, \sigma = 0.0$) the same but there were differences in the values of Snort ($\overline{CPU} = 28.0\%, \sigma = 0.0$) and Zeek ($\overline{CPU} = 8.14\%, \sigma = 0.47$), however, these are not significant, 1% or less ($\Delta_{Snort} = 1.0\%$, $\Delta_{Suricata} = 0.0\%$, $\Delta_{Zeek} = 0.21\%$).

Snort ($\overline{t_{kernel}} = 272.41s, \sigma = 1.37$) spent the most time in kernel mode while Suricata ($\overline{t_{kernel}} = 117.08s, \sigma = 1.66$) the second most and Zeek ($\overline{t_{kernel}} = 86.64s, \sigma = 1.4$) the least. With PCAP 2, the extracted data from the measurements are fairly similar to what is presented in the diagram with PCAP 1. The difference between the average of the tests of the two PCAPs are $\Delta_{Snort} = 11.6s$, $\Delta_{Suricata} = 4.07s$, $\Delta_{Zeek} = 0.11s$.

The time the CPU spent in user mode when the network traffic was simulated with PCAP 1 was Snort ($\overline{t_{user}} = 183.04s, \sigma = 1.54$) spending the most, Zeek ($\overline{t_{user}} = 52.58s, \sigma = 1.93$) spending the least time in user mode and Suricata ($\overline{t_{user}} = 127.49s, \sigma = 0.97$) being in between. The simulation done with PCAP 2 resulted in similar data ($\Delta_{Snort} = 6.81s$, $\Delta_{Suricata} = 4.24s$, $\Delta_{Zeek} = 4.45s$). With PCAP 2 Snort ($\overline{t_{user}} = 176.23s, \sigma = 2.18$) spent the most time in user mode by average followed by Suricata ($\overline{t_{user}} = 123.25s, \sigma = 1.10$) then Zeek ($\overline{t_{user}} = 48.13s, \sigma = 1.09$).

The maximum memory consumption with the network simulated using PCAP 1 clearly indicates that Zeek uses the most amount of memory of all the IDSs. Zeek used the least CPU and it used the most amount of memory ($\overline{Memory_{Zeek}} = 209,412.6kB, \sigma_{Zeek} = 87.41$) as well yet Suricata consumed less CPU overall than Snort and also used less memory ($\overline{Memory_{Suricata}} = 54,150.8kB, \sigma_{Suricata} = 215.80$, $\overline{Memory_{Snort}} = 83,729.8kB, \sigma_{Snort} = 1138.53$). The difference of this data with PCAP 2 is $\Delta_{Snort} = 7.279MB$, $\Delta_{Suricata} = 0.027MB$, $\Delta_{Zeek} = 3.185MB$.

As the Figure 13 clearly indicates Snort and Suricata performed outstandingly when the network simulation used PCAP 1, they did not drop a single packet. When it comes to Zeek the graph shows that it did drop packets during the runtime, and while more than eight thousand dropped, or not processed packets are greater than 0, the default logs that are printed to the terminal by Zeek, the software calculated that it is 0.07% of the total packets that the IDS dealt with. The simulation that used PCAP 2 provided results where Snort and Suricata dropped or not processed 0 packets and Zeek ($\Delta_{Zeek} = 216.69packets$) performed around the same level with just a few more processed packets than with the use of PCAP 1. These results are recorded in the Figure 14.

This concludes the individual analysis of the gathered data during the research. In the following sections, this data is going to be evaluated further, compared to each other where applicable and based on those results a final conclusion is going to be drawn.

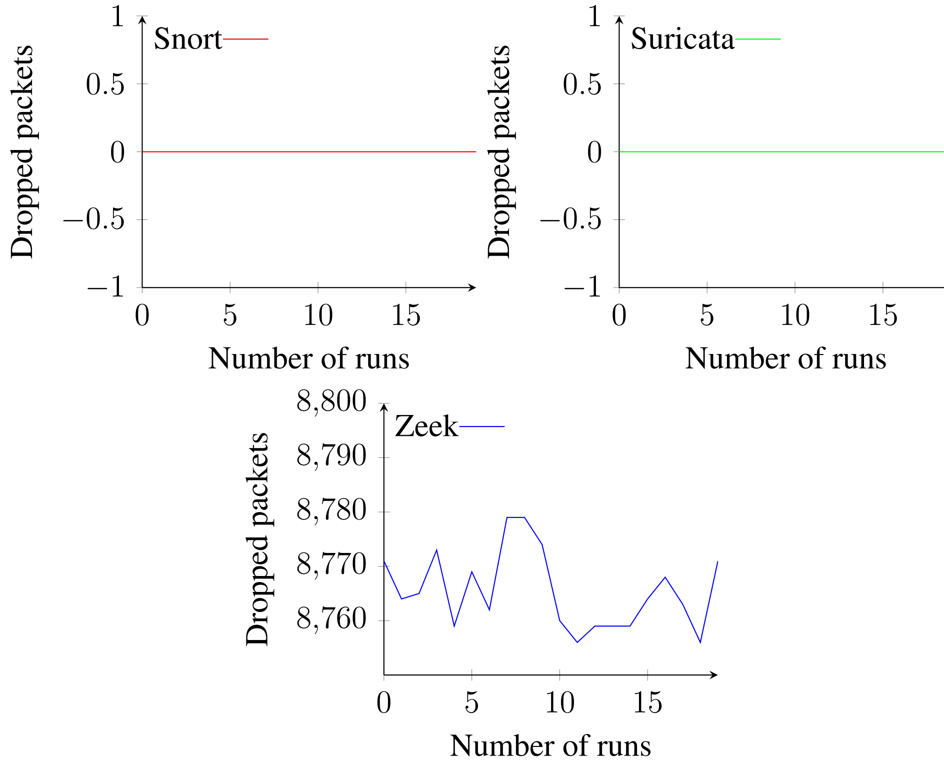


Figure 13. Number of dropped packets per run with PCAP 1 with attacks

7.3 Result comparison between environments

During the experiments, two prerecorded network traffic files were used. The PCAP 1, without additional malicious packets, was used both in the virtual and in the physical environment, providing a baseline for comparison of the different environments. In the physical environment, the test was conducted with two PCAPs. To gain inputs for comparative analysis the systems were tested with both PCAPs as well as with and without additional malicious content. Table 1 gives an overview of the results.

The difference between the physical and the virtual environment may be due to the difference in available hardware resources. The architecture of the CPUs also differs because the virtual environment is based on regular PCs with Advanced Micro Devices (AMD) architecture while the RPI4s have advanced RISC[2] machines (ARM) CPU architecture.

During the tests on the VMs Snort used 12.06% of the CPU, Suricata 20.89% and Zeek 6.94% on average. When compared to the physical environment without attacks the CPU consumption was 43.01% for Snort, 32.5% for Suricata and 39.2% for Zeek. These results have $\Delta_{Snort} = 30.95\%$, $\Delta_{Suricata} = 11.61\%$, $\Delta_{Zeek} = 32.26\%$ difference.

[2]Reduced Instruction Set Computer

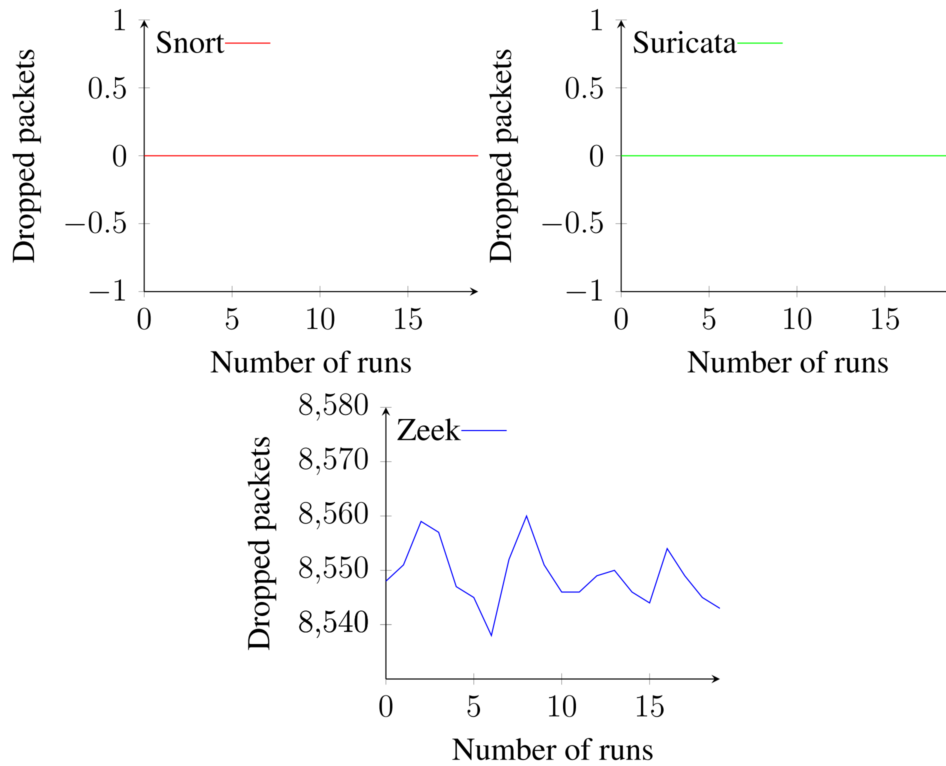


Figure 14. Number of dropped packets per run with PCAP2 with attacks

This CPU consumption can be broken down into two subsections, operating in kernel mode and in user mode. When the IDSs were tested in the virtual environment, Snort spent 48.67% of the time operating in kernel mode while Suricata spent 46.13% and Zeek 29.51%. In the physical environment, Snort spent 61.84% of the time working in the kernel space, Suricata 46.56% and Zeek 59.15%. This means that on average Snort spent $\Delta_{Snort} = 13.17\%$ more time in the kernel space in the physical environment, Suricata spent $\Delta_{Suricata} = 0.43\%$ less time in kernel mode in the real environment and Zeek spent $\Delta_{Zeek} = 29.64\%$ more time in the kernel space in the physical environment. When the IDSs were tested in the virtual environment Snort spent 51.33%, Suricata 53.87% and Zeek 70.49% of the time operating in user mode. In the physical testing, environment Snort spent 38.16%, Suricata 53.44% and Zeek 40.85% of the time working in the user space. So in total Snort spent $\Delta_{Snort} = 13.17\%$, Suricata $\Delta_{Suricata} = 0.43\%$ and Zeek $\Delta_{Zeek=29.64\%}$ less time in user mode in the physical environment.

The average memory used in the virtual environment by Snort was 1.03*10^6kB while Suricata used 0.9863*10^6kB and Zeek 2.39*10^6. In the physical environment, Snort used 0.657*10^6kB, Suricata consumed 0.666*10^6kB and Zeek occupied 2.13*10^6kB memory of the system. On average Snort needed $\Delta_{Snort} = 0.373$*10^6kB, Suricata $\Delta_{Suricata} = 0.32$*10^6kB and Zeek $\Delta_{Zeek} = 0.26$*10^6kB less memory to operate in the real environment.

In the physical environment, there were two kinds of tests, one with malicious packets and one without. In the tests where the traffic simulated did not contain malicious activity on the network, with PCAP 1 Snort used 43.01%, Suricata 32.5% and Zeek 39.2% of the CPU, while with PCAP 2 Snort consumed 43.02%, Suricata 31.91% and Zeek 39.6% on average. This gives a difference of $\Delta_{Snort} = 0.01\%, \Delta_{Suricata} = 0.59\%, \Delta_{Zeek} = 0.4\%$. In the physical environment with malicious packets with PCAP 1 Snort needed 29.00%, Suricata 15.00% and Zeek 8.35% of the CPU while with PCAP 2 Snort used 28.00%, Suricata 15.00% and Zeek 8.14% of the CPU. This gives a difference of $\Delta_{Snort} = 1.00\%, \Delta_{Suricata} = 0.00\%, \Delta_{Zeek} = 0.21\%$.

While the network had no malicious activity simulated, with the use of PCAP 1 Snort spent 61.84% of the time in kernel mode, while Suricata 46.56% and Zeek 59.15%. In the case of PCAP 2 Snort was operating in kernel mode 61.97% of the time, Suricata 46.76% and Zeek 59.44%. The differences between the two tests are $\Delta Snort = 0.13\%, \Delta Suricata = 0.2\%, \Delta Zeek = 0.29\%$. With the attacks, Snort spent 59.81%, Suricata 47.87% and Zeek 62.23% of the time operating in the kernel space when PCAP 1 was simulated. With PCAP 2 being used to simulate the network attacks Snort needed 59.68%, Suricata 47.83% and Zeek 64.26% of the time to work in kernel mode. The difference between the results of the two PCAPs containing malicious activity are $\Delta_{Snort} = 0.13\%, \Delta_{Suricata} = 0.04\%, \Delta_{Zeek} = 2.03\%$. Without attacks with PCAP 1 Snort spent 38.16% of the time, Suricata 53.44% and Zeek 40.85% in user mode, while with PCAP 2 Snort 38.03%, Suricata 53.24% and Zeek 40.56% of the total time. The difference of the two simulations are $\Delta_{Snort} = 0.13\%, \Delta_{Suricata} = 0.2\%, \Delta_{Zeek} = 0.29\%$. In the same resource category, when the network contained simulated attacks, with the use of PCAP 1, Snort occupied 40.19%, Suricata 52.13% and Zeek 33.77% of the time the user space. With the traffic of PCAP 2 Snort used 40.32%, Suricata 52.17% and Zeek 35.74% of their time operating in the CPU in user mode. The difference of the averages in the network under attack were $\Delta_{Snort} = 0.13\%, \Delta_{Suricata} = 0.04\%, \Delta_{Zeek} = 1.7\%$.

The last category of comparing the physical environment with attacks and without them is memory consumption. As long as the network did not contain attacks Snort needed 0.857*10^6kB, Suricata 0.666*10^6kB and Zeek 2.13*10^6kB of memory with PCAP 1 while with PCAP 2 Snort consumed 0.8906*10^6kB, Suricata 0.7644*10^6kB and Zeek 2.15*10^6kB of memory. This results in the following differences, $\Delta_{Snort} = 0.033kB*10^6$, $\Delta_{Suricata} = 0.098kB*10^6$, $\Delta_{Zeek} = 0.02kB*10^6$.

Unfortunately, the exact content of the PCAPs including attacks is unknown so the IDSs can only be compared based on their resource usage while they had to analyse packets with malicious content. With PCAP 1 Snort used 29.00% of the CPU while Suricata

used 15.00% and Zeek 8.35%. With PCAP 2 Snort needed 29.00%, Suricata 15.00% and Zeek 8.14% of the CPU. The difference between the two measurements are minimal ($\Delta_{Snort} = 1.00\%, \Delta_{Suricata} = 0.00\%, \Delta_{Zeek} = 0.21\%$).

The percentage of how much time a CPU spent in kernel mode are as follows, Snort 59.81%, Suricata 47.87% and Zeek 62.23% with PCAP 1, while with PCAP 2 Snort 59.68%, Suricata 47.83 and Zeek 64.26%. These results yield the following differences, $\Delta_{Snort} = 0.13\%, \Delta_{Suricata} = 0.04\%, \Delta_{Zeek} = 2.03\%$. In user mode with PCAP 1, Snort spends 40.19% of the time, Suricata 52.13% and Zeek 37.77% while with PCAP 2 Snort spends 40.32% of the total time, Suricata 52.17% and Zeek 35.74%. This results in the following differences, $\Delta_{Snort} = 0.13\%, \Delta_{Suricata} = 0.04\%, \Delta_{Zeek} = 1.7\%$.

The last part of the comparison is the memory consumption of the IDSs. When the simulations were done using PCAP 1 Snort needed 0.837*10^6kB of memory while Suricata 0.541*10^6kB and Zeek 2.09*10^6kB. The differences with PCAP 2 are $\Delta_{Snort} = 0.033kB$*10^6, $\Delta_{Suricata} = 0.098kB$*$10^6$, $\Delta_{Zeek} = 0.02kB$*10^6. The results of the data collection with PCAP 2 are Snort 0.756*10^6kB, Suricata 0.5412*10^6kB and Zeek 2.06*10^6kB.

	Virtual Environment			Physical environment											
				Without maliciopus packets						With malicous packets					
				PCAP1			PCAP2			PCAP1			PCAP2		
	Snort	Suricata	Zeek	Snort	Suricata	Zeek	Snort	Suricata	Zeek	Snort	Suricata	Zeek	Snort	Suricata	Zeek
Max CPU usage [%]	21	23	8	44	45	40	44	45	40	29	15	9	28	15	10
AVG CPU usage [%]	12.06	20.89	6.94	43.01	32.5	39.2	43.02	31.91	39.6	29.00	15.00	8.35	28.00	15.00	8.14
Max time in kernel mode [s]	1,284.92	1,299.96	251.09	895.97	716.99	778.91	888.98	718.96	787.26	275.73	119.35	89.37	263.15	116.94	103.85
AVG time in kernel mode [s]	612.78	981.26	197.53	870.87	496.02	760.64	874.37	489.5	768.67	272.41	117.08	86.64	260.81	113.01	86.53
AVG time in kernel mode [%], SD	48.67	46.13	29.51	61.84	46.56	59.15	61.97	46.76	59.44	59.81	47.87	62.23	59.68	47.83	64.26
Max time in user mode [s]	1,290.49	1,253.45	578.62	560.65	765.73	538.41	546.32	772.74	540.41	185.28	129.33	56.57	182.54	125.45	49.71
AVG time in user mode [s], SD	646.28	1,146.10	471.77	537.34	569.31	525.27	536.68	557.31	524.43	183.04	127.49	52.58	176.23	123.25	48.13
AVG time in user mode [%]	51.33	53.87	70.49	38.16	53.44	40.85	38.03	53.24	40.56	40.19	52.13	37.77	40.32	52.17	35.74
Max memory usage [kBx10^6]	1.05	0.9863	2.4	0.858	0.674	2.13	0.895	0.681	2.22	8.56	5.45	210.00	7.798	5.46	2.06
AVG memory usage [kBx10^6], SD	1.03	0.9673	2.39	0.857	0.666	2.13	0.8906	0.7644	2.15	0.837	0.541	2.09	0.765	0.5412	2.06

Table 1. Summary of the results

8. Applicability

In this section the applicability of the three observed IDSs going to be analysed. This analysis is based on the official documentation of Snort, Suricata and Zeek.

Zeek needs Spicy, a parser generator that makes it easy to create robust C++ parsers for network protocols, file formats, and more. Spicy parsers expose a C++ API that any application can leverage to send them data for processing. It has its own compiler that can compile and run protocol parser modules. These modules' complexity may vary from very simple to the extent of creating custom parser algorithms for existing protocols. It uses so-called unit hooks, conceptually, unit hooks are somewhat similar to methods, they have bodies that execute when triggered, and these bodies may receive a set of parameters as input. Units can provide metadata about their semantics through properties that both Spicy itself and host applications can access. It supports attaching filters to units that get to preprocess and transform a unit's input before its parser gets to see it. Spicy source code is structured around modules, which introduce namespaces around other elements defined inside. While Spicy itself remains application-independent, transparent integration into Zeek has been a primary goal for its development. In Zeek, the term "analyser" refers generally to a component that processes a particular protocol, file format, or low-level packet structure. The Spicy plugin makes it possible to provide the remaining pieces to Zeek, turning a Spicy parser into a full Zeek analyser.[31],[32]

Through Berkeley packet filters (BPF) it is possible to configure what traffic Suricata receives. The traffic not seen by Suricata is not inspected, logged or otherwise recorded. Suricata has its own solution to implement application-layer parsers and it makes it available to its users. Suricata is able to parse a good number of protocols such as HTTP, transport layer security (TLS), Modbus, simple network management protocol (SNMP) and many more[33]. LibHTP is one of the libraries being developed for Suricata, it is a security-aware parser for the HTTP protocol and the related bits and pieces. There are four basic principles LibHTP wants to achieve: it must be able to parse virtually all traffic that is found in practice, it must never fail to parse a stream that would be parsed by some other web server, it must be able to detect and effectively deal with various evasion techniques and the performance must be adequate for the desired tasks. It uses a standard notation structure named abstract syntax one (ANS1) to describe data. The parser uses

pattern-based protocol detection that is port independent. According to the official GitHub repository, LibHTP is not yet considered stable and should not be used in production. There is also little to no documentation, as the developers stated *"The best documentation at this time is the code itself..."*. [34],[35]

In the case of Snort, the official documentation provides a very detailed description of how to create preprocessors. They allow the functionality of Snort to be extended. The Session preprocessor is a global stream session management module. Since Session implements part of the functionality and API that was previously in Stream5 it cannot be used with Stream5 but must be used with the new Stream[1] preprocessor. It enables the creation of the session control block. Stream supports the modified Stream API that is focused on functions specific to reassembly operations. Stream TCP provides means on an IP address target to configure TCP policy. This can have multiple occurrences. One default policy must be specified. [36]

[1] A target-based transmission control protocol (TCP) reassembly module for Snort, it replaces the Stream5 preprocessor.

9. Discussion

In this section, a more in-depth analysis of the results is going to be conducted to point out the reasons and significance of the results.

The first experiment was conducted in the virtual environment, where the prerecorded network traffic captures (without additional malicious packets) were replayed for the different IDSs on similar virtual machines with identical resources.

The IDS's resource consumption limited the speed of the replaying, and consequently the execution itself as well. During the tests, the average CPU consumption of Zeek was 6.94%, while Snort used 12.06%, and Suricata 20.89% of the CPU. Snort and Suricata performed around the same execution time while Zeek delivered better results. The average execution times of Snort (9,941.17s) and Suricata (9,937.27s) are similar, while Zeek (9,002.84s) finished the task in a shorter time. Aligning with that, Suricata spent the most time in kernel space with 46.13% followed by Snort with 48.61% and Zeek with 29.51% on average. Compared to these results the time difference spent in kernel mode is significant, therefore there is a greater chance of a critical system failure when using Suricata.

As Zeek used the least amount of CPU, it makes sense that it spent the least time in the kernel space. The ratio of the times that the processes spent in user mode are similar to the kernel mode, Suricata takes the lead with 1146.1s, Snort is the second with 646.28s and Zeek with 471.77s. It could be clearly seen that although the execution times are almost identical Suricata spends a lot more time using the processor than Snort and Zeek.

When observing the maximum memory used by the IDSs it is obvious that Zeek does have some drawbacks. It is easier on the CPU but much heavier on the memory than Suricata and Snort. Suricata used on average 96,731.87kB (96.73MB) memory, while Snort used 102801.73kB (12.85MB) and Zeek used 238902.75kB (238.90MB).

The rest of the experiments were conducted in the physical environment, with two prerecorded network traffic files, that were replayed by an independent computer. This solution ensured the more precise IDS-related results since the resource consumption of the different IDSs neither influenced each other nor the traffic generation. Both three IDSs were

tested with two PCAPs, with and without malicious packets.

In the results of the physical environment when malicious packets were added to the PCAP files, in terms of average CPU consumption Zeek (8.25%) outperformed both Suricata (15%) and Snort (28.5%), while in the virtual environment, where the network traffic was greater none of the results went above 25%.

Suricata performed worse on a busier network than on one with malicious packets, however, Snort could handle more traffic with less CPU usage than it could deal with a packet containing attacks. The CPU consumption of Zeek increased slightly compared to a busier network. It is safe to assume that, in regards to CPU usage, Snort can deal with heavier normal traffic better than Suricata, however when it comes to analysing packets Suricata outperforms Snort while attacks also take a toll on Zeek but not to great extents.

The time spent in kernel mode corresponds to the CPU usage by the IDSs. Snort spent the most average time in Kernel mode (266.41s) while Suricata spent 115.05s and Zeek spent 88.08s in the kernel space. The difference between the CPU usage of Snort and Suricata is greater than it was in the virtual environment however the ratio between the time spent in kernel mode is not as great as it was in the virtual testing environment. It is safe to assume that, even though Snort uses more of the CPU it does not have to run as many kernel processes as Suricata. The graphs of average time spent in user mode look fairly similar to the kernel graphs of the kernel mode. Snort spent 183.04s and 176.23s on average in user mode, Suricata spent 127.49s and 123.25s while Zeek spent 52.58s and 48.13s in the user space. As was the case with kernel mode, the proportions between Snort and Suricata are much smaller than they were on a network with heavier traffic.

As it was the case in the virtual environment too, Zeek used the most amount of memory during the deployment with 209412.6kB (209.41MB) and 206226.66kB (206.22MB). Snort used 83729.8kB (83.72MB) and 76450.09kB (76.45MB) of memory while Suricata consumed 54150.8kB (54.15MB) and 54123.42kB (54.12MB). The overall amount of memory used by the IDSs decreased compared to the virtual environment. In proportion, Suricata decreased more compared to Snort. By this data, it is definable that heavier network traffic makes the IDSs consume more memory than the analysing packets do.

During the stress tests, Snort and Suricata did not drop any packets while Zeek did. As explained earlier the dropped packages are only around 0.07% of the whole traffic analysed, however, compared to the other IDSs, Zeek lost between 8760 and 8780 packets with PCAP 1 and between 8540 and 8560 with PCAP 2.

The results of the tests on the physical environment, where the PCAPs did not contain malicious packets shows, that on average, Snort used 43.01% of the CPU and Suricata used 32.5%. Not following the pattern of the previous tests, Zeek used 39.2% of the CPU, more than Suricata and not much less than snort. With the second PCAP file the results are similar, Snort utilised 43.02%, Zeek 39.6% and Suricata 31.91% of the CPU on average. The average time spent in kernel mode shows similarities to the percentage of the CPU and IDS used, Snort operated for 870.87s and 874.37s in kernel mode, and Zeek spent 760.64s and 768.67s in the kernel space while Suricata spent only 496.02s and 489.5s operating kernel processes. When it comes to user mode, Suricata took the lead with 569.31s and 557.31s. This value is very close to the time Snort and Zeek spent in the user space. Snort spent 537.34s and 536.68s running user processes while Zeek did it in 525.27s and 524.43s. The huge difference between the values disappeared compared to how much CPU an IDS used.

On average, the graph of the maximum memory that was used by a given IDS is very similar to the previous test runs. Zeek once again consumed the most memory with 212634.69kB (212.63MB) and 215095.07kB (215.09MB). Snort used 85655.53kB (85.65MB) and 89055.65kB (89.05MB) and Suricata, as before, needed the least amount of memory with 66522.72kB (66.52MB) and 67444.83kB (67.44MB). Considering how much more CPU Zeek utilised during these test runs the memory usage not only not decreased, it increased, compared to the runs where there were malicious packets on the network.

With heavier network traffic Suricata still managed to process all the packets passing through it. Snort started to drop some packets, between 20-200 were not processed according to the default data provided by the IDS, in later runs, this amount decreased to 0. Zeek did not process significantly fewer packets, around 1980-1996 packets. This number is still much greater than what other IDSs dropped or could not process.

10. Further works

The early stages of the research was focusing on the performance of the IDSs in virtual environments. This provided up-to-date data on how these systems perform in virtualised environments. This data gives ground for further research based on VMs or on regular hardware. Customisation has not been explored in virtual environments in this thesis, expanding on the idea of virtualisation may be one of the various topics that the gathered data could be used for.

The research contains data on the limited hardware performance of the IDSs. This performance is based on the basic settings of the software, meaning they are true if the IDSs are not modified. Further research that includes developing different parsers, preprocessors etc. could bring different results that could potentially change how an IDS works in limited hardware environments. As described earlier with certain IDSs it is possible to change the way a protocol is processed during deployment, this gives the opportunity to develop optimal ways of processing data in special-purpose environments such as industrial or maritime areas.

11. Conclusion

Before arriving at a conclusion, there were some phenomena, that are not related to the data collected but are worth mentioning. The installation of the IDSs are well documented on their official websites and could be followed to do so in a normal environment. When the IDSs needed to be installed on the RPI4s there were a couple of setbacks. The most up-to-date Raspbian OS could not be used, thus the legacy version was installed, because Snort could not be installed on a device running that OS. The problem was, that the *Atomic operations library* could not execute correctly on such a device and prevented the installation of Snort version 3 from the source code.

The Suricata IDS, by default, creates logs. One of the default logs is eve.json. This file contains detailed information about every packet the IDS analysed. The only problem is that under two weeks of operation this file reached the size of over 42GB. The recording of such logs can be turned on or off in the configuration file of Suricata.

Zeek, compared to the other IDSs, needs a lot of storage space. While testing virtual environments, Snort and Suricata could be installed on VMs that had 8GB of storage capacity while Zeek needs more than 10GB of free space for installation. The installation time of Zeek is also the longest, on the VMs it was about one hour while on the RPI4s it took more than two hours to install.

RPi4s also had some disadvantages. While it is easy to build, install and configure a device to work with the basic settings, microSD cards are unfit to be the storage for a device that completes read and write operations constantly for weeks. The devices had to be reinstalled because a huge number of files got corrupted and they could not execute basic functionalities anymore.

In a virtual environment, where the hardware is not limited as much as a RPi4 is, Suricata is the heaviest on the CPU while Zeek is more demanding with memory. Snort is an optimal solution as it uses significantly less CPU than Suricata and requires much less memory to operate than Zeek.

When it comes to the physical, very limited environment, Suricata outperformed both

Zeek and Snort when malicious packets were sent to the network. It used less CPU and less memory than snort and significantly less memory than Zeek. However, Zeek outperformed bot IDSs in terms of CPU usage. Despite the fact that Zeek used much fewer CPU resources, it did not perform that well because Snort and Suricata did not drop a single packet, while Zeek did. In conclusion, on a network where the traffic is not that heavy and is targeted by attacks, Suricata performed the best when installed on a RPI4.

In the other tests, where the network of the physical testing environment was not targeted by attacks but the network traffic was much heavier Zeek started to use significantly more of the CPU. While Snort still used more of the CPU than Suricata the gap was not as great as before. In the tests with attacks, the differences between the two IDSs CPU resource usage were 14% and 13%. In a busier network, this decreased to 11% and 12%, meaning that the heavier the network traffic, the more CPU resource Suricata will require. Zeek did not use less memory than in any of the previous tests, on the contrary, it used even more. With the CPU resource usage being also significantly high it puts it as the heaviest IDS in terms of hardware requirements. It also drops the most packets out of all the IDSs tested during the research. Suricata did outperform Snort version 3 in memory consumption too with about 22MB. Snort also started to drop packets in a few runs, when the network traffic increased.

Overall, Suricata performed better on the RPI4s than the other IDSs with the executed simulations, however, the documentation to develop new modules for the software is vague and, as stated earlier, should not be used in production in its current state. This leaves Snort and Zeek. Although Zeek performed worse, if the basic IDS functionalities are not needed and the goal is the development of a new method to process data, it could be considered since the documentation is very detailed. Snort performed better than Zeek and has great documentation on how to create certain types of preprocessors. Based on the data collected, Snort is the optimal solution if the IDS should be extended for deployment.

It can be concluded, that the thesis achieved its described objective.

Acknowledgment

I, the author, would like to express my deepest gratitude to my supervisors, Gabor Visky and Risto Vaarandi, their supervision over the thesis helped to improve the quality beyond what I could achieve on my own. I would also like to thank my unofficial supervisor, Mauno Pihelgas, his expertise, insights and guidance allowed me to improve both the thesis and myself during the research. He provided valuable, indispensable knowledge whenever it was needed, his help was crucial throughout the whole process. I also offer my gratitude to those friends and family members who were supporting me during this time.

Bibliography

[1] Asmaa Shaker Ashoor and Sharad Gore. "Importance of intrusion detection system (IDS)". In: *International Journal of Scientific and Engineering Research* 2.1 (2011), pp. 1–4.

[2] Indraneel Mukhopadhyay et al. "Heuristic Intrusion Detection and Prevention System". In: *2015 International Conference and Workshop on Computing and Communication (IEMCON)*. 2015, pp. 1–7. DOI: 10.1109/IEMCON.2015.7344479.

[3] Mauno Pihelgas. "A comparative analysis of open-source intrusion detection systems". In: *Tallinn: Tallinn University of Technology & University of Tartu* (2012).

[4] G Nanda Kishor Kumar and RK SHARMA. "AN OVERVIEW STUDY ON INTRUSION DETECTION SYSTEM (IDS)". In: *International Journal of Pure and Applied Mathematics* 118.24 (2018).

[5] Richard A Kemmerer and Giovanni Vigna. "Intrusion detection: a brief history and overview". In: *Computer* 35.4 (2002), supl27–supl30.

[6] Jeffrey R. Yost. "The March of IDES: Early History of Intrusion-Detection Expert Systems". In: *IEEE Annals of the History of Computing* 38.4 (2016), pp. 42–54. DOI: 10.1109/MAHC.2015.41.

[7] Garand Vira Yudha and Rini Wisnu Wardhani. "Design of a Snort-based IDS on the Raspberry Pi 3 Model B+ Applying TaZmen Sniffer Protocol and Log Alert Integrity Assurance with SHA-3". In: *2021 9th International Conference on Information and Communication Technology (ICoICT)*. 2021, pp. 556–561. DOI: 10.1109/ICoICT52021.2021.9527511.

[8] Piyawat Noiprasong and Assadarat Khurat. "An IDS Rule Redundancy Verification". In: *2020 17th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 2020, pp. 110–115. DOI: 10.1109/JCSSE49651.2020.9268269.

[9] Jang Hyeon Jeong et al. "Rule conversion mechanism between NIDPS engines". In: *2020 22nd International Conference on Advanced Communication Technology (ICACT)*. 2020, pp. 581–584. DOI: 10.23919/ICACT48636.2020.9061387.

[10] Karthikeyan KR and A Indra. "Intrusion Detection Tools and techniques–a Survey'". In: *International Journal of Computer Theory and Engineering* 2.6 (2010), p. 901.

[11] Zeeshan Ahmad et al. “Network intrusion detection system: A systematic study of machine learning and deep learning approaches”. In: *Transactions on Emerging Telecommunications Technologies* 32.1 (2021), e4150.

[12] Shijoe Jose et al. “A survey on anomaly based host intrusion detection system”. In: *Journal of Physics: Conference Series*. Vol. 1000. 1. IOP Publishing. 2018, p. 012049.

[13] Roshan Kumar and Deepak Sharma. “HyINT: Signature-Anomaly Intrusion Detection System”. In: *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2018, pp. 1–7. DOI: 10.1109/ICCCNT.2018.8494088.

[14] Inc. Free Software Foundation. *GNU Licenses*. Accessed: 2022-04-12. 2022. URL: https://www.gnu.org/licenses/.

[15] The Apache Software Foundation. *Apache Licenses*. Accessed: 2022-04-12. 2022. URL: https://www.apache.org/licenses/.

[16] SolarWinds Worldwide LLC. *GETTING STARTED GUIDE Security Event Manager Version 2021.2*. Accessed: 2021-11-03. 2021. URL: https://documentation.solarwinds.com/archive/pdf/sem/sem-getting-started-guide.pdf.

[17] Inc. Cisco Systems. *Cisco Firepower NGIPS data sheet*. Accessed: 2021-11-03. 2019. URL: https://www.cisco.com/c/en/us/products/collateral/security/ngips/datasheet-c78-742472.pdf.

[18] Cisco. *Rule Subscriptions*. Accessed: 2021-11-14. 2021. URL: https://www.snort.org/products%5C#rule%5C_subscriptions.

[19] Dede Fadhilah and Marza Ihsan Marzuki. “Performance Analysis of IDS Snort and IDS Suricata with Many-Core Processor in Virtual Machines Against Dos/DDoS Attacks”. In: *2020 2nd International Conference on Broadband Communications, Wireless Sensors and Powering (BCWSP)*. 2020, pp. 157–162. DOI: 10.1109/BCWSP50066.2020.9249449.

[20] Maham Qayyum, Wajeeha Hamid, and Munam Ali Shah. “Performance Analysis of Snort using Network Function Virtualization”. In: *2018 24th International Conference on Automation and Computing (ICAC)*. 2018, pp. 1–6. DOI: 10.23919/IConAC.2018.8749024.

[21] Hendrawan Hendrawan, Parman Sukarno, and Muhammad Arief Nugroho. “Quality of Service (QoS) Comparison Analysis of Snort IDS and Bro IDS Application in Software Define Network (SDN) Architecture”. In: *2019 7th International Conference on Information and Communication Technology (ICoICT)*. 2019, pp. 1–7. DOI: 10.1109/ICoICT.2019.8835211.

[22] Zachary Hill et al. "Using Bro with a Simulation Model to Detect Cyber-Physical Attacks in a Nuclear Reactor". In: *2019 2nd International Conference on Data Intelligence and Security (ICDIS)*. 2019, pp. 22–27. DOI: 10.1109/ICDIS.2019.00011.

[23] Singapore University of Technology and iTrust Design (SUTD). *Terms of Usage of Dataset*. Accessed: 2022-04-21. URL: https://itrust.sutd.edu.sg/itrust-labs_datasets/dataset_info/.

[24] Loris Degioanni and Gianluca Varenni. "Introducing scalability in network measurement: toward 10 Gbps with commodity hardware". In: *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*. 2004, pp. 233–238.

[25] Francesco Fusco and Luca Deri. "High speed network traffic analysis with commodity multi-core systems". In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 2010, pp. 218–224.

[26] Inc. Free Software Foundation. *GNU Time*. Accessed: 2021-12-08. 2018. URL: https://www.gnu.org/software/time/.

[27] Balint Adam. *Special-Purpose Network Intrusion Detection System*. 2022. URL: https://github.com/baadam3/SPNIDS.

[28] Neil Desai. "Increasing performance in high speed NIDS". In: *A look at Snort's Internals* (2002).

[29] cisco. *Snort rules*. Accessed: 2022-04-25. URL: https://www.snort.org/downloads/#rule-downloads.

[30] Brian Ward. *How Linux works : what every superuser should know*. San Francisco, CA: No Starch Press, 2015. ISBN: 978-1-59327-567-9.

[31] The Zeek Project. *Spicy documentation*. Accessed: 2022-04-21. URL: https://docs.zeek.org/projects/spicy/en/latest/index.html#.

[32] The Zeek Project. *Zeek documentation*. Accessed: 2022-04-21. URL: https://docs.zeek.org/en/master/.

[33] Open Information Security Foundation. *Complete list of Suricata Features*. Accessed: 2022-04-25. URL: https://suricata.io/features/all-features/.

[34] OISF. *Official LibHTP GitHub repository*. Accessed: 2022-04-21. URL: https://github.com/OISF/libhtp.

[35] OISF. *Suricata user guide*. Accessed: 2022-04-21. URL: https://suricata.readthedocs.io/en/suricata-6.0.0/index.html.

[36] Cisco. *Snort manual*. Accessed: 2022-04-21. URL: http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node1.html.

Appendices

Appendix 1 - Non-Exclusive License[1]

I Balint Adam

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Performance and Applicability Analysis of Open-Source Intrusion Detection Systems in Special-Purpose Networks" , supervised by Gabor Visky
   1.1 to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2 be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

Date: 04.22.2022

[1] A Non-Exclusive Licence grants to the licensee the right to use the intellectual property, but means that the licensor remains free to exploit the same intellectual property and to allow any number of other licensees to also exploit the same intellectual property.