Tuning Suricata Intrusion Detection System for High Performance on a Single Non-Uniform Memory Access Node

VEGARD HENRIKSEN & ROAR FØRDE

SUPERVISOR
Olaf Owe & Roger Skjetlein

University of Agder, 2021
Faculty of Engineering and Science
Department of Information and Communication Technology

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	**Vi erklærer videre at denne besvarelsen:** • Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • Ikke refererer til andres arbeid uten at det er oppgitt. • Ikke refererer til eget tidligere arbeid uten at det er oppgitt. • Har alle referansene oppgitt i litteraturlisten. • Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	Ja

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Preface

The thesis is a result of a research carried out at the department of Information and Communication Technology (ICT) at University of Agder (UiA) and Telenor Security Operations Center from January 2021 to June 2021. We want to thank Olaf Owe from UiA and Roger Skjetlein from Telenor for being our supervisors and assisting us throughout the project. We would also like to thank all of our co-workers at Telenor that have assisted us throughout the project.

Abstract

The rapid increase in network capacity poses a challenge in detecting cyber attacks. Suricata is a modern *intrusion detection system* (IDS) used to monitor network traffic to detect cyber attacks. Telenor is monitoring considerable amounts of network traffic with IDS-servers, commonly referred to as sensors. Occasionally, the traffic load reaches a point where the IDS drops some of the incoming packets, termed packet loss. This is a serious problem, as it can lead to undetected threats. With this in mind, Telenor wants to find out if a lossless detection can be achieved by tuning Suricata for high performance. This can be accomplished by pinning dedicated CPUs to Suricata and by optimizing the process from when the packets arrive at the sensor until they have been processed by Suricata. Additionally, a *non-uniform memory access* (NUMA) topology will be employed to reduce data latency. NUMA is a memory architecture where the processing units are divided into two NUMA nodes with their own local memory attached. By using the NUMA node where the packets arrive, Suricata should in theory operate more efficiently. The Suricata performance is affected by several factors, such as the hardware capability and capacity, BIOS settings, and the traffic type. The sensors also run other critical processes on the same NUMA node. An additional goal is therefore to limit the processing power reserved for Suricata.

Two Lenovo servers are used in this research. One of them will operate as the sensor, while the other will operate as a traffic generator. The traffic generation tool is called TRex, able to generate stateful, bidirectional traffic. Stateful means that TRex can dynamically alter the packet headers, such as the source and destination IP addresses. A traffic profile containing a combination of realistic network traffic is used for this purpose. TRex is also able to multiply the connections per second in order to generate a desirable amount of traffic. This is used to methodically stress test the sensor by gradually increasing the traffic to the limit where packet losses are observed. If no limit is met, the processing power will be gradually limited in the same manner. The automation tool Ansible is used to streamline the testing phase. It methodically configures the sensor, runs the test, and finally gathers the results.

A high performance Suricata configuration has been proposed, tested and evaluated. The solution is able to process 9 Gbps of data using only 10 CPUs on a single NUMA node, which allows the sensor to also run other critical processes on the node. Compared to the default configuration which uses all 64 CPUs, it is able to process 2 Gbps more data. This is a significant improvement, and Telenor should be able to mitigate the packet loss problem by applying this solution. However, optimizing the BIOS settings had no performance impact. This is unexpected, as previous research has proven considerable performance benefits by making the same changes. Additionally, the results reveal significantly higher last layer cache misses than expected, indicating that the packet pipeline does not work as intended. There is also a high number of TCP reassembly gaps reported in the results, which indicates a problem with the memory allocation. These problems should be investigated further.

Contents

List of Figures

List of Tables

Abbreviations

This list specifies the pages each abbreviation is mentioned.

CFS Completely Fair Scheduling Class. 8

DCA Direct Cache Access. 5, 20, 25

DMA Direct Memory Access. 5, 20

FIFO First-In-First-Out. 20

Gbps Gigabits per second. iii, 2, 13–15, 21, 25, 27, 29, 37, 38, 40

GPU Graphics Processing Unit. 15

IaC Infrastructure as Code. 11

IDS Intrusion Detection Engine. iii, 1, 3, 9, 13–16, 29, 38

IRQ Interrupt Request Line. 10, 11

LIFO Last-In-First-Out. 7

LLC Last Layer Cache. 4, 20, 25

LTS Long Time Support. 16, 17

LWP Light Weight Process. 7

NIC Network Interface Controller. 1, 2, 9–11, 13–16, 20–25, 27, 33, 40, 45, 46

NUMA Non-Uniform Memory Access. iii, 1, 2, 5, 6, 14, 18, 20, 25, 26, 33, 37, 39, 40

OISF Open Information Security Foundation. 9

OOTB Out-Of-The-Box. 1, 40

PCAP Packet Capture. 17, 38

RSS Receive Side Scaling. 9, 13–16, 20, 24, 40

RX Receive. 9, 36

SEPTun Suricata Extreme Performance Tuning. 14

SMP Symmetric Multiprocessing. 5

TX Transmit. 9

UMA Uniform Memory Access. 5, 6

XDP Express Data Path. 14, 41

Chapter 1

Introduction

1.1 Motivation

Telenor is monitoring considerable amounts of network traffic every day from their customers through network monitoring sensors deployed on their network. At this date, each sensor operates as a standard server with two CPU sockets. The sensors run an *intrusion detection engine* (IDS) called Suricata. When the load is so high that the sensor's bandwidth capacity is saturated, Telenor can see a high cost when it comes to context switching. This leads to loss of data which can result in an undetected threat. By optimizing the process from when the packets arrive at the sensor until they have been processed by Suricata, defined as the packet pipeline, Telenor should in theory be able to process more data.

Suricata throughput defines the amount of data that can be handled by the Suricata IDS engine without any packet drops. With this in mind, Telenor wants to verify whether it is possible to achieve a higher Suricata throughput while the sensor is still able to run other critical processes. An *out-of-the-box* (OOTB) Suricata configuration will be used to compare the performance difference, which is defined as the default configuration. The default configuration is similar to the configuration Telenor uses today. The processing speed may be affected by several factors such as the hardware capability and capacity, BIOS settings and the traffic type. This research is proposed by Telenor Security Operation Centre in Arendal.

1.2 Problem Statement and Research Goals

On a sensor, each CPU socket resides on a *non-uniform memory access* (NUMA) node with its own local resources. Communication between the nodes adds some latency. The first node has a local *network interface card* (NIC) that receives the packets to be processed by Suricata, hereby defined as the monitoring interface. Hence, Suricata should run on the same node. However, Telenor runs other applications on their sensors. Some of them need direct access to the same NIC, which means that they should run on the same node. Thus, Suricata should not occupy all of the processing power on the local node. From this background, the problem statement of this research is:

Is it possible to achieve a higher Suricata throughput than the default configuration while the sensor is still able to run other critical processes?

The problem statement can be divided into four main goals:

1. Set up a traffic generator on a separate system used to transmit bidirectional traffic to the sensor. This ensures that the sensor is not affected by the workload of generating traffic.
2. Tune Suricata for high performance on a single NUMA node. This includes modifications to Suricata and the NIC, as well as some adjustments in Linux and in the BIOS.
3. Compare the Suricata throughput with default and high performance configurations. Depending on how performant the high configuration is, processing power can be limited for Suricata. The remaining processing power is available for other critical processes.
4. Use the final configuration to develop a configuration management module used for automation. By using this module, the final configuration can be deployed on any sensor that satisfy the hardware requirements.

By tuning Suricata and optimizing the packet pipeline, it is expected to achieve a significantly higher Suricata throughput that also solves the packet loss problem for Telenor.

1.3 Limitations

The availability of equipment and hardware is limited to what Telenor has at hand. The required equipment consists of two servers, where one of them needs to satisfy a hardware requirement explained in section 4.1. Additionally, a switch that supports port mirroring is needed in order to route the bidirectional traffic between the servers. The available equipment has a common limitation of 10 Gbps bandwidth capacity.

1.4 Research Methodology

In order to start the research, it is necessary to select a research path. The main objective is to observe and study how Suricata behaves under extreme conditions with different configurations, and then compare the results. The motivation behind this objective is that the existing configuration on the network monitoring sensors results in loss of data. Because there is a preconceived problem that needs solving, *applied research* is the appropriate research type [1, p. 74].

The applied research type is usually associated with testing and evaluating an application of science to solve a real-world problem. There are two research methods that cover applied research: *applied experiments* and *applied observational studies*. Because it is desirable to understand how the solution performs in a simulated real-world scenario, an *applied observational study* is the appropriate research method [1, p. 89-90].

Going forward, there are two different categories of applied observational studies: *exploratory* and *descriptive*. Exploratory studies refers to how a system behaves. Descriptive studies describe how the results of foundational research is applied. Because this study introduces a specific change that is to be evaluated, an *applied exploratory study* is the most suitable. [1, p. 300-301]

Again, there are two types of applied exploratory studies: *operational bounds testing* and *sensitivity analysis*. Because this study will test the performance boundaries without considering the sensitivity of the data, *operational bounds testing* is the appropriate type. Furthermore, operational bounds testing is comprised of three different flavours of testing: *stress testing*, *performance testing* and *load testing*. Because the system is often saturated with data, it is natural to use *stress testing* in order to find the maximum traffic rate the IDS can process [1, p. 301-305].

The first part of the testing phase is to establish a baseline for comparison. A default Suricata configuration will be stress tested until a consistent throughput is found. The second part of the testing phase is to experiment with configurations until the most performant is found. Finally, the throughput of both configurations will be compared.

1.5 Thesis Outline

The thesis is outlined with the following chapters. Chapter 2 presents the related technologies used in this research. Next, chapter 3 introduces the primary research conducted in this field. Chapter 4 presents the system, the main behavior to study and the testing methodology. The results are presented in chapter 5. Then, in chapter 6, the test results and anomalies are discussed. Finally, chapter 7 concludes the findings in this thesis. The chapter winds up with a few considerations on future work.

Chapter 2

Background

The following chapter first gives an introduction to how memory caches and shared memory architectures work in computer systems. Then, the relevant parts of the Linux kernel are explained. Following, a fundamental network interface technology in this research is presented. Finally, a brief description of the applications Suricata, Ansible and TRex is given.

2.1 Memory Cache

Memory caches are reserved areas of memory located on the CPU, which is used to speed up processing. Caches have faster transmission speeds than the main memory that feeds them. Cache is a memory bank that bridges the processor to the main memory, and allows the CPU to execute instructions at a faster pace. If the next instruction is available in the cache, a *cache hit* occurs. CPUs usually contains two or three cache levels, where the levels closer to the CPU have faster transfer speeds at the cost of less memory. The higher levels supplies the lower levels with data. The three cache levels are called *L1*, *L2* and *L3*. The *L3 cache*, also called *last layer cache* (LLC), is slower than the lower levels. However, it has far more memory, meaning it can store more data. L3 is shared by all of the CPUs in contrary to the lower levels, which are pinned to a single CPU [2]. Figure 2.1 displays the cache topology.

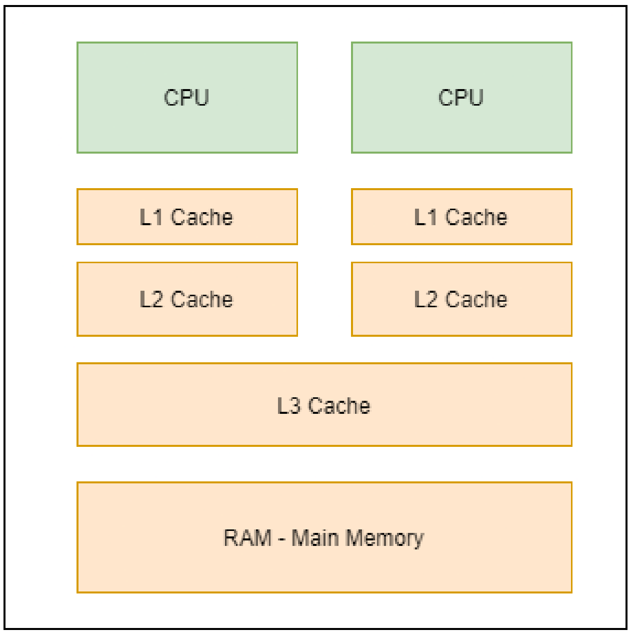

Figure 2.1: Cache topology. Cache levels closer to the CPU have faster transfer speeds at the cost of less memory.

Direct Memory Access & Direct Cache Access

Direct memory access (DMA) is the process where peripheral hardware, such as a NIC can read or write data directly to memory without involving the CPU. This prevents the processor to get interrupted each time an external device needs to transfer data. Hence, while the external device is copying data, the processor can still execute programs [3].

Direct cache access (DCA) is similar to DMA, but it allows peripheral hardware to transfer data directly to the CPU cache. Simply put, this is achieved with an accelerated copy by a dedicated DMA engine. This can improve response times as well as reducing CPU cache misses [4].

2.2 Shared Memory Architecture

The processors in a multiprocessing system need to access data on the system in one way or another. The most common architecture used to solve this problem is called the shared memory architecture. A system using this architecture consists of a set of independent processors that share a global memory. All communication between the processors are also done via the global memory [5, p. 77].

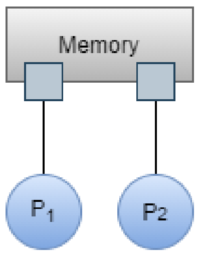

Figure 2.2: Shared memory via two ports. P = Processor.

A simple shared memory system consists of two processors connected to a memory module, as shown in figure 2.2. The memory controller can only handle one concurrent request from a processor at a time. When multiple processors are trying to access the shared memory at the same time, it can lead to performance degradation. The typical way to solve this is to implement caches. Each processor then has its own local cache that holds relevant copies of data. This solution can lead to another problem regarding coherence. When a processor writes to a value in the cache, the copy becomes inconsistent relative to the other copies on the surrounding CPUs [5, p. 78]. Caching snoop protocols were invented to solve this very problem. Snoop protocols is a set of protocols which ensures memory cache coherency in *symmetric multiprocessing* (SMP) systems. A widely used snoop protocol called *write invalidate* will erase all other copies of data before writing to the local cache [6].

There are several types of shared memory systems based on the interconnection network used. The two most common ones will be explained next, one of them being NUMA.

2.2.1 UMA

In a *uniform memory access* (UMA) system, the processors access the shared memory through an interconnection network, which is typically one or more buses. This is similar to the way a single processor accesses its memory. The memory access latency is balanced, hence the name, and this is why they are also called SMP systems. This memory organization is the most widespread among shared memory systems. An example model can be seen in figure 2.3 [5, p. 78, 79].

By adding more processors to a UMA system the bandwidth on the bus can be saturated. Implementing caching will minimize this risk. A high flow of data might also turn the bus into a bottleneck [6].

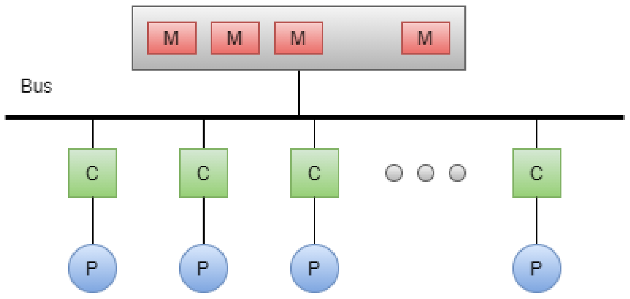

Figure 2.3: Bus-based UMA (SMP) shared memory system. P = Processor, C = Cache, and M = Memory.

2.2.2 NUMA

Non-uniform memory access (NUMA) is a shared memory system where the processing units are divided into NUMA nodes. Each node has its own part of the shared memory attached. This works by splitting the memory into different address spaces and distributing them to each node. However, nodes can still access other parts of the memory, but this will add some latency because the data has to traverse a proprietary interconnect link called NUMALINK [5, p. 79] [7, p. 1-2].

Figure 2.4 can be used as an example. This is a system with two processors that have four cores each. Each processor is a NUMA node with its own part of the shared memory. Node A can access its own local memory with very little latency. It can also access the memory in node B called foreign memory, but it has to traverse the interconnect which adds some latency [7, p. 2].

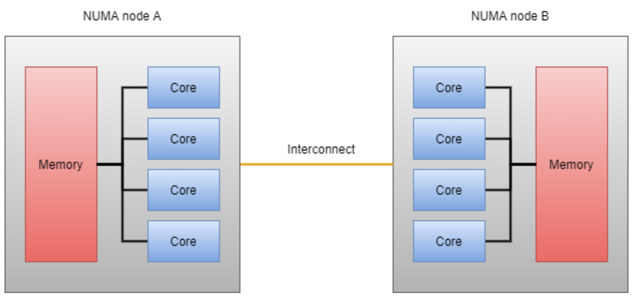

Figure 2.4: NUMA shared memory system.

The current processor speed is so fast that the signal path length from the processor to memory becomes the throughput bottleneck. NUMA effectively brings the memory closer to the processor cores which increases the overall bandwidth, as long as foreign memory access is minimal [7, p. 1].

The benefits from this memory organization depends heavily on the type of workload. It is more suitable for servers which typically manages specific tasks [8].

2.3 Linux Kernel

In order to understand how Suricata can be optimized in a Linux System, it is necessary to get a basic understanding of how the relevant parts of the Linux Kernel works.

2.3.1 Processes

When a computer program is executed, the program's code and data is fetched into memory regions and becomes a process. A process is a running instance of a program. The process is also assigned additional memory regions called stack and heap. The stack is a reserved, fixed region, and keeps track of the current state of the process by adding and removing data in a *last-in-first-out* (LIFO) manner. The heap is a non-fixed region visible to other processes, and is responsible for dynamic allocations during execution. Stack-memory allocation is faster than heap-memory allocation because of the less complicated architecture [9, p. 6-8].

2.3.2 Virtual Address Space

Virtual address space is an abstraction provided by the kernel between the process and the physical memory, which keeps concurrent processes from interfering with each other. The kernel allocates a virtual address space for every process on the system, and any attempt to access an address outside this space will trigger a hardware fault. For each process, an address is relative to it's own space and not the global space [9, p. 8-9].

2.3.3 Kernel and User Space

Besides protecting processes from one another, the kernel also prevents processes from manipulating kernel data and services by segmenting the memory into user and kernel space. Processes reside in user space and the CPU is operating in user mode. Processes can still interact with the kernel through system calls, a set of special CPU instructions that makes up the kernel interface. The CPU have to switch to a more privileged mode called kernel mode before requesting system calls, and then switch back to user mode. Fork is a system call used to create a new process called child from a running process called parent [9, p. 9-11].

2.3.4 Threads

A thread in Linux is the flow of execution in a process, and there has to be at least one thread for every process. All threads share their parent's virtual address space and system resources, but maintains it's own code, stack and thread local storage. However, threads are not suitable for concurrent computing, because they are locked to the same processor core as their parent. Therefore, Linux does not support threads directly. Instead, a special process called *light weight process* (LWP) is used to achieve the same goal. Briefly explained, they function similarly to a thread, but can be bound to any processing core. Since threads and LWP are similar, the terms are commonly interchangeable [9, p. 28].

2.3.5 Process Scheduler

Scheduler Design

Linux has evolved over the years, and has added support for several platforms with different scheduling requirements. The Linux process scheduler is responsible for allocating processing time among processes in an effective manner and consists of two layers. The generic scheduler is the first layer and functions as an entry point. The second layer is a set of scheduling classes which specializes in different kinds of processes, for example normal I/O bound and real-time [9, p. 44-45].

Runqueue

For every CPU core, there is a runqueue which contains all the concurrent processes. Because the scheduler classes have their own priorities and policies, it is not possible to have a common runqueue. The solution is to let every scheduler class have its own runqueue, while the general scheduler constructs the main runqueue by embedding all the scheduling class' runqueues [9, p. 48].

Process Priorities

In order for the scheduler to determine which process to run next, every process are given a priority. Normal processes are given dynamic priorities determined by the kernel, while real-time processes are given static priorities set by the user. Linux scales the priorities between 0 to 139 where a lower number indicates higher priority. 0 to 99 is assigned for real-time processes. 100 to 139 is for normal processes and are called the *nice* value range, because it implies how nice the process is to other processes. The *nice* value has an internal range from -20 to 19 in the global priority range from 100 to 139. A nice value of 0 means neutral priority, while a positive value means that the process is nicer to other processes [9, p. 52-53].

Completely Fair Scheduling Class

The *completely fair scheduling class* (CFS) is responsible for maintaining balance in allocating time to all processes with dynamic priorities. It is the most used class as most processes in Linux are non-realtime. The CFS uses a concept of virtual runtime for tracking the amount of CPU time each process has been given. A high virtual runtime means that the process has been given considerable CPU time. The scheduling class uses a combination of *virtual runtime* and the *nice* value to determine which process to run next and for how long. Another concept called sleeper fairness ensures that processes waiting for an I/O request gets extra processing time when they wake up. Group scheduling is a concept designed to prevent process starvation by making sure that every runnable process gets at least some CPU time under a defined time duration called the scheduling period. In addition, CPU time are distributed on group level rather than thread level to prevent allocation of unfair amounts of CPU time to processes with a large number of threads [9, p. 54-57].

Processor Affinity

By default, Linux will try to keep a process on the same CPU. In order to increase performance on multi-core systems, it is possible to determine a thread's processor affinity with a system call. This will keep the process on the same core, rather than on the same CPU [9, p. 64].

2.4 Network Interface Controller

The *network interface controller* (NIC) is a device that makes it possible for a computer to connect to the internet. Incoming and outgoing network packets are placed in *receive* (RX) and *transmit* (TX) queues.

2.4.1 Receive Side Scaling

Receive side scaling (RSS) distributes incoming traffic over several RX queues on the NIC. This allows multiple CPUs to process the inbound network traffic. In addition to reducing network latency, RSS can relieve overloaded CPUs caused by bottlenecks when handling interrupts [10].

RSS was originally designed for normal traffic, and not for IDS packet capturing. A hash algorithm is used to distribute the traffic over different queues by hashing the packet header. The RSS hash value is asymmetrical, which will cause packets from the same flow to be mapped to different queues depending on the direction. This will lead to an unpredictable order of packet processing, and Suricata will deem some of the traffic as invalid [11]. This problem can be solved by using a symmetric RSS hash value. The hash algorithm will result in the same hash for both sides of a flow, which causes the flows to end up in the same queue. In [12], Woo et al. found that the following hash key can be used by Suricata to achieve symmetrical RSS. The hash value is presented below:

```
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a
0x6d5a 0x6d5a 0x6d5a 0x6d5a
```

2.5 Suricata IDS

Suricata is an open source network threat detection engine that includes an *intrusion detection system* (IDS), and is developed by the *Open Information Security Foundation* (OISF). The Suricata IDS uses a signature language to match known threats and malicious behaviour on the network. The IDS can be configured to use a ruleset, which is a set of malware signatures [13][14].

2.5.1 Rules

Rules play an important part in Suricata's performance since it scans all the rules in order to perform inspections. A rule includes an action that determines what happens when a signature matches. It also includes a header defining the IP addresses, ports, direction and lastly the rule options defining the specifics of the rule. The command *suricata-update* will update the rules by scanning the defined sources. By default, only the source *Emerging Threats Open* is defined. This is an open source signature database featuring a large number of rules, which should satisfy most requirements [15].

2.5.2 Runmodes

Suricata is a multi-threaded process built up by threads, modules and queues. Each module is responsible for a specific functionality, like packet decoding and detection. Packets are processed by one thread at a time, and can be passed to a different thread through a queue. The arrangement of the threads, modules and queues is called the runmode.

There are three available runmodes: workers, autofp and single. Workers is the most CPU efficient mode, where each thread contains the packet pipeline. The NIC will ensure that the packets are balanced across each thread. Autofp uses one or more capture threads, which captures the packets and decodes it. Then, the packets are sent to the worker threads. The single runmode works like the worker runmode, but uses only one packet processing thread [16].

2.5.3 Performance Considerations

Aside from configuring Suricata itself, there are a few modifications that can be set in the Linux system worth considering.

Isolate CPUs

As mentioned in section 2.3.5, The Linux kernel scheduler is responsible for allocating processing time among processes in an effective manner. In many scenarios, the scheduler yields acceptable performance. However, when Suricata is affinitized to a CPU set, the scheduler can pin other processes to the Suricata worker threads. In order to achieve high performance, only Suricata should run on the threads. This problem can be solved by isolating the Suricata threads from the Linux kernel, which will prevent the scheduler to detect the CPUs.

Interrupt Handling

Aside from the scheduler, interrupt handling can greatly affect the system performance. An interrupt is sent to a processor whenever an event requires immediate attention [9, p. 257]. When an external device like a NIC issues an interrupt, it is sent to an output pin on a CPU called the *interrupt request line* (IRQ), which will signal an interrupt request [9, p. 260].

The Linux *irqbalance* service distributes IRQs across all CPUs on the system, attempting to balance the work load [17]. If there is a high frequency of IRQs, it can severely degrade the performance of cores under heavy load. In some situations it can therefore be necessary to stop the *irqbalance* service as well as setting the IRQ affinity in order to affinitize interrupts to cores [18]. For Intel NICs, this can be done using a script developed by Intel called *set_irq_affinity*, which is included with the Intel i40e driver source package [19].

2.6 Automation

In an IT system, repeatable tasks can be automated by using an automation engine. The software works within a set of instructions and executes them with little to no human interaction. This is key to IT optimization, as environments needs to be able to scale faster than ever [20].

2.6.1 Ansible

Ansible is created by RedHat and is an IT automation engine. It is designed based on the *infrastructure as code* (IaC) mindset. IaC is widely used within IT to deploy and orchestrate the infrastructure topology. It also ensures that the same environment is deployed every time[21]. Ansible uses files called playbooks to describe the automation job. Playbooks are written in an easy to read language called YAML. Ansible is agent-less, which means that no software needs to be installed on the target system. All communication uses SSH. [22][23].

Roles

Roles in Ansible let the user load related files, tasks, handlers and other artifacts based on a known file structure. Ansible have a defined directory structure shown below. *Common* and *webservers* are different roles with a different map structure based on the role.

```
roles/
common/
    tasks/
    handlers/
    library/
    files/
    templates/
    vars/
    defaults/
    meta/
webservers/
    tasks/
    defaults/
    meta/
```

Roles can be used in three ways. At play level and at task level with include_role or import_role [24].

2.7 TRex Traffic Generator

TRex is an open source traffic generator developed by Cisco [25]. It can generate and amplify bidirectional traffic up to 200 Gbps as long as the bandwidth capacity is sufficient. TRex uses traffic profiles to generate stateful traffic, dynamically changing the packet header parameters when replaying packets [26, Ch. 1.2]. Stateful traffic means it can generate client and server side of the traffic based on smart replays and real traffic templates. To make it possible for TRex to generate bidirectional traffic, it needs to be setup and configured for loopback [27, Slide 4]. This means that two ports on the system are physically linked by cable. A switch can be used to mirror this traffic and send it to a sensor, as suggested in [28, Slide 10-11].

Chapter 3

Related Work

3.1 Improvement of Hardware Firewall's Data Rates by Optimizing Suricata Performances

In [29], Jakimoski et al. studies the performance gain by optimizing Suricata. Their sources show that among the most popular IDSs, Suricata can process a higher rate of network traffic with a lower packet drop rate at the expense of a higher consumption of computational resources. It is stated that both the hardware and Suricata needs to be tuned in order to achieve a lossless detection when the network capacity increases to 40 Gbps and beyond. The test setup consists of a multi-core, high-end system suitable for telecommunications and large IT needs. Suricata's performance in this setup was dependent upon the below factors:

HW capability and capacity
In order for Suricata to make use of multiple cores, it is necessary to use a NIC that supports the RSS multi-queue feature to hash the incoming traffic to multiple receive queues.

Suricata version
By using the latest Suricata version, optimal performance is ensured as far as Suricata is concerned. The test setup used Suricata version 5.0.0 dev.

Rules
The number of rules affects the performance. The test setup used a ruleset of 14350 signatures.

BIOS settings
It was discovered that BIOS tuning was needed for speeds higher than 50 Gbps.

Traffic type
Higher throughput rates were observed with simple HTTP flows. Thus, the traffic profile will have an impact on the performance.

Most tuning efforts were done to ensure that the packets are read from the L3 cache in order to minimize the time required to fetch data. Some of the most important tuning considerations are:

- A single port of every NIC is used. The other ports are reserved for redundancy.

- A local NIC is used for each of the two NUMA nodes to reduce latency.
- All interface receive and transmit queues are pinned to cores local to the NUMA node for the interface. The traffic is evenly distributed using symmetrical RSS hashing.
- Suricata runs in worker mode where the worker threads are pinned to the cores local to the NUMA node for the interface.
- The *memcap* values in the Suricata configuration file are adjusted to handle high traffic load.
- Interface buffers and interrupts are adjusted in order to decrease packet drops and to endure higher performance.

By applying the tuning configurations specified above and using *Suricata extreme performance tuning* SEPTun Mark I [30] and SEPTun Mark II [31] as a guideline, two tests were performed. TRex traffic generator was used to generate stateful traffic for the tests. The first test resulted in a maximum of 34.5 Gbps Suricata throughput using one NIC. The second test resulted in 60 Gbps using two NICs. No kernel drops were observed in neither of the cases.

A feature called *eXpress Data Path* (XDP) is described in SEPTun Mark II. However, the authors do not use XDP in their research, as it is most effective for handling elephant flows. The tests only consist of small flows.

Our thesis directly supports this work with a different testing approach explained in chapter 4. The guides SEPTun Mark I and II will also be used in our thesis as references for tuning Suricata.

3.2 Evaluating Network Intrusion Detection Systems for High-Speed Networks

In [32], Qinwen et al. evaluate three different IDSs for high-speed networks: Snort, Suricata and Bro. The testing environment consists of a sender, a receiver and a switch. The sender uses the protocol GridFTP to transfer 10 Gbps of test data.

Three different methods for sending data are used. The first is a single flow test case. The second is 20 flows running in parallel. The third is real time traffic from the campus, which provides a typical mix of flow types. Each scenario are evaluated with three different experiments. The first is default configuration which defines a baseline for evaluating performance. The second modifies the detection algorithm and configuration. The third modifies the data acquisition mechanisms to improve capturing performance.

From the results, they found that Suricata performed better with its multi-threaded architecture, while Snort and Bro are single-threaded. Snort and Suricata performed better using the *af-packet* data acquisition compared to the default *libpcap* data acquisition. Suricata dropped more packets when using the pattern matching algorithms *gfbs* and *b3g* than with *ac* and *b2g*. They also found that the IDSs seem to only use a small amount of memory resources.

Regarding the traffic, the IDSs handled 10 Gbps with a single flow, only dropping a few percent of the packages. For instance, 10 Gbps with 20 flows resulted in higher CPU utilizations and drop rates. Finally, the campus network resulted in even worse results. In other words, the number of flows can affect the performance of the IDS.

While our thesis does not explore pattern matching algorithms, it is interesting that the IDS performance was significantly affected by changing the algorithm. In addition, even if our thesis will only use constant network traffic, it is interesting that the number of flows can affect the performance of the IDS.

3.3 Kargus: A Highly-scalable Software-based Intrusion Detection System

In [33], Jamshed et al. are making their own IDS called Kargus based on the existing IDS Snort. In 2012 when this report was published, it was a need for an IDS that is better suited to profit from the multiple CPU cores and the modern commodity hardware advancements. No existing IDS could process more than 10 Gbps of traffic for minimum-sized packets. Additionally, the existing IDSs did not take advantage of the *graphics processing unit* (GPU) or high capacity NICs with RSS support.

Kargus solved these problems and were able to process 33 Gbps of data. Additionally, it managed to process 9-10 Gbps of traffic when all of the packets contained an attack signature. The IDS were significantly more efficient than the existing IDSs. Kargus is mainly using the CPUs for pattern matching, as the GPU has a higher latency and power usage. When the CPU gets overloaded, the GPU will offload the CPU and process some of the packets. An algorithm is developed for deciding the CPU processing threshold.

The report concludes that Kargus is an improved IDS compared to the alternative IDSs on the market. By combining the CPU for processing and the GPU for offloading, the computer resources can be better utilized. Also, the waste in process cycles are decreased. However, using the GPU to offload the CPU can be risky, as an inadequately tuned offloading will significantly hurt the performance. Additionally, if the cost of the context switching exceeds the CPU cycles required for the workload, the offloading is not beneficial. This may frequently occur for minimum-sized packets [33].

This work is not as relevant for our thesis as the previously introduced works. It is however interesting that the GPU can be used for offloading, even though Suricata does not support it.

Chapter 4

Approach

This chapter first introduces the system used for conducting this research. Next, the focus of this study is scoped by defining the behavior to be studied. Then, the process of developing the optimal configuration is presented. Lastly, the testing approach is explained.

4.1 System

This research will be performed on a system consisting of two Lenovo rack servers and a Ubiquiti switch. The first server is a *Lenovo Thinksystem SR630* which is used for running the Suricata IDS packet processing. This server will hereinafter be referred to as the sensor. The SR630 is equipped with an Intel x710 NIC that supports symmetric RSS hashing and the *workers* runmode. The sensor will also have the latest version of Suricata installed and configured. The hardware specifications can be seen in table 4.1.

Lenovo ThinkSystem SR630	
CPU	2x Intel(R) Xeon(R) Gold 5218T 2.1GHz cores=16 threads=32
CPU Cache	L1 Cache(64 KB), L2 Cache(1024 KB), L3 Cache (22MB)
Memory (RAM)	128GB 8x16GB DIMM DDR4 2933 MHz RAM
Ethernet(NIC)	x710 10 GbE SFP+
OS	Ubuntu 18.04.5 LTS
Kernel	Linux 5.4.0-70-generic
Suricata	v6.0.2
Driver	Intel i40e

Table 4.1: Lenovo SR 630 specification.

Lenovo x3550 M5	
CPU	2x Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz cores=12 threads=24
Memory (RAM)	128GB 8x16GB DIMM DDR4 2933 MHz RAM
OS	Ubuntu 18.04.5 LTS
Kernel	Linux 4.15.0-140-generic
TRex	v2.88 from github

Table 4.2: Lenovo x3550 M5 specification.

The other sensor is a *Lenovo x3550 M5* that runs the TRex traffic generator. This server will hereinafter be referred to as the traffic generator. TRex includes various *packet capture* (PCAP) files which can be used for simulating real-time traffic. TRex also includes a set of traffic configuration files in YAML-format. These traffic profiles combine a set of PCAP files as well as dynamically changing the IP address in both direction of the flows. The remaining specification can be seen in table 4.2. Both servers run Ubuntu 18.04 *Long Time Support* (LTS), as this is the same version Telenor is using.

Ubiquiti EdgeSwitch 16 XG is a 10 gigabit switch with 12 SFP+ ports. The traffic generator is connected to the switch with two SFP+ cables in a loopback configuration. This way, TRex can control the bidirectional flow by sending both the client and server side of the traffic. This traffic will then be mirrored to the sensor. The system setup is illustrated in figure 4.1.

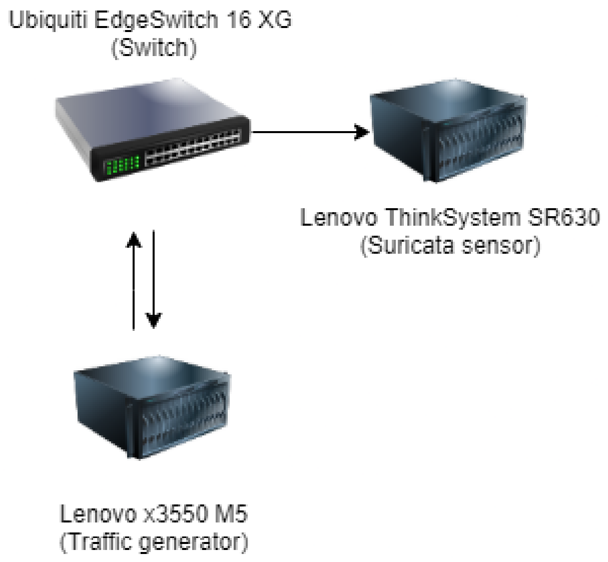

Figure 4.1: Illustration of the system setup. The sensor and traffic generator are connected through a switch.

A picture of the system are shown from the front in figure 4.2. The upper server is the traffic generator, and the lower server is the sensor.

The rest of the system is shown from behind in figure 4.3. The upper switch is used for port mirroring. The lower switch is only used to assign IP addresses to the servers in order to access the internet, and to enable remote access over SSH. It is not considered part of the system.

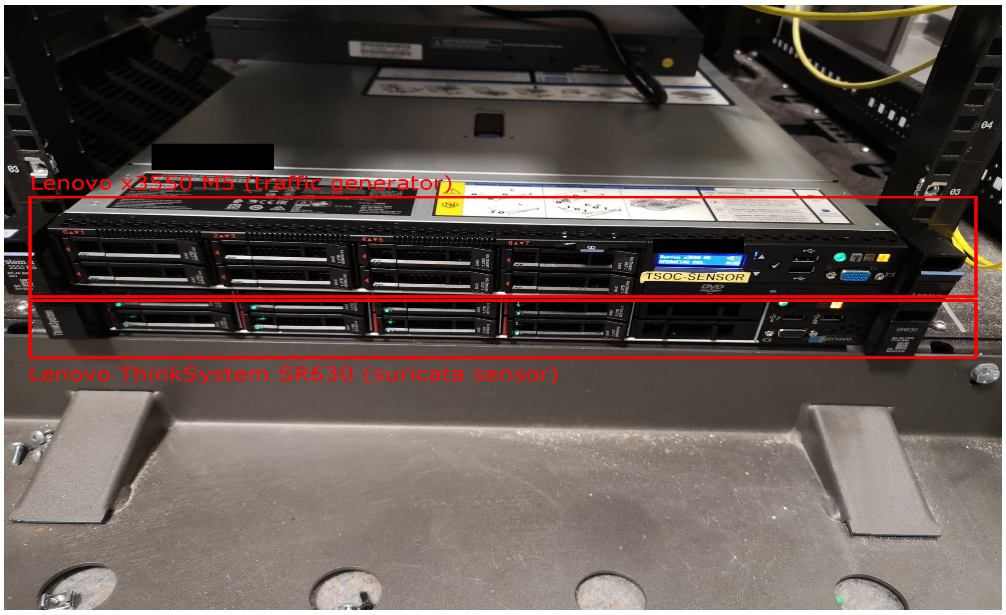
Lenovo x3550 M5 (traffic generator)
TSOC-SENSOR
Lenovo ThinkSystem SR630 (suricata sensor)

Figure 4.2: Picture of the system setup from the front.

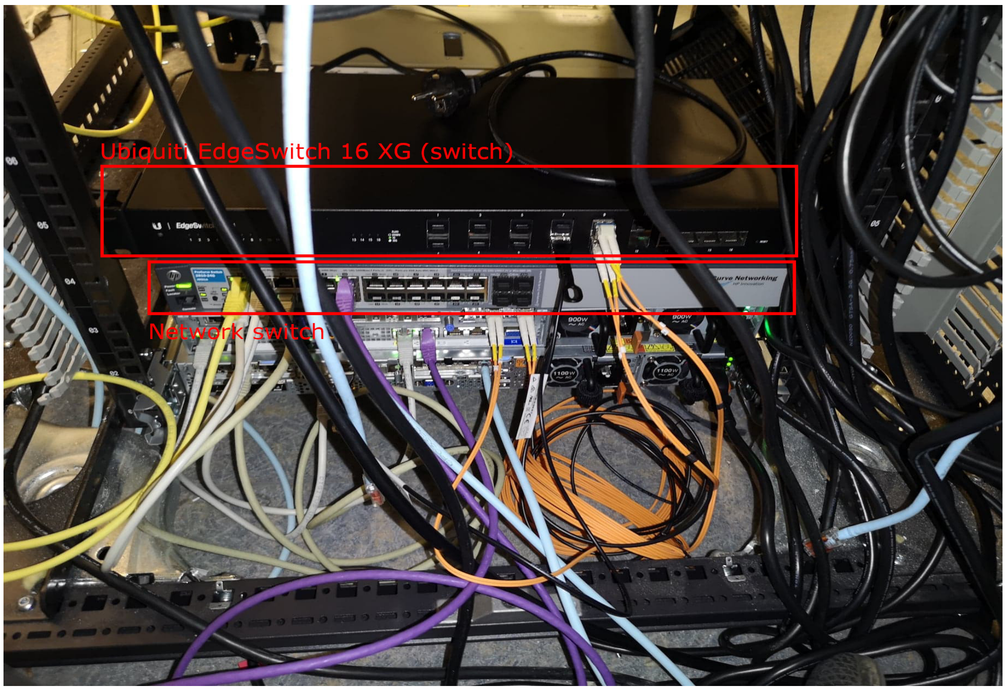

Figure 4.3: Picture of the system setup from behind, which includes two switches.

4.1.1 Mapping the Hierarchical Topology

The *hwloc* package is used to output an image of the hierarchical topology by using the following command:

```
$ lstopo --logical --output-format png > topo.png
```

The resulting image is shown in figure 4.4. In the figure one can also see which NUMA node each interface belongs to.

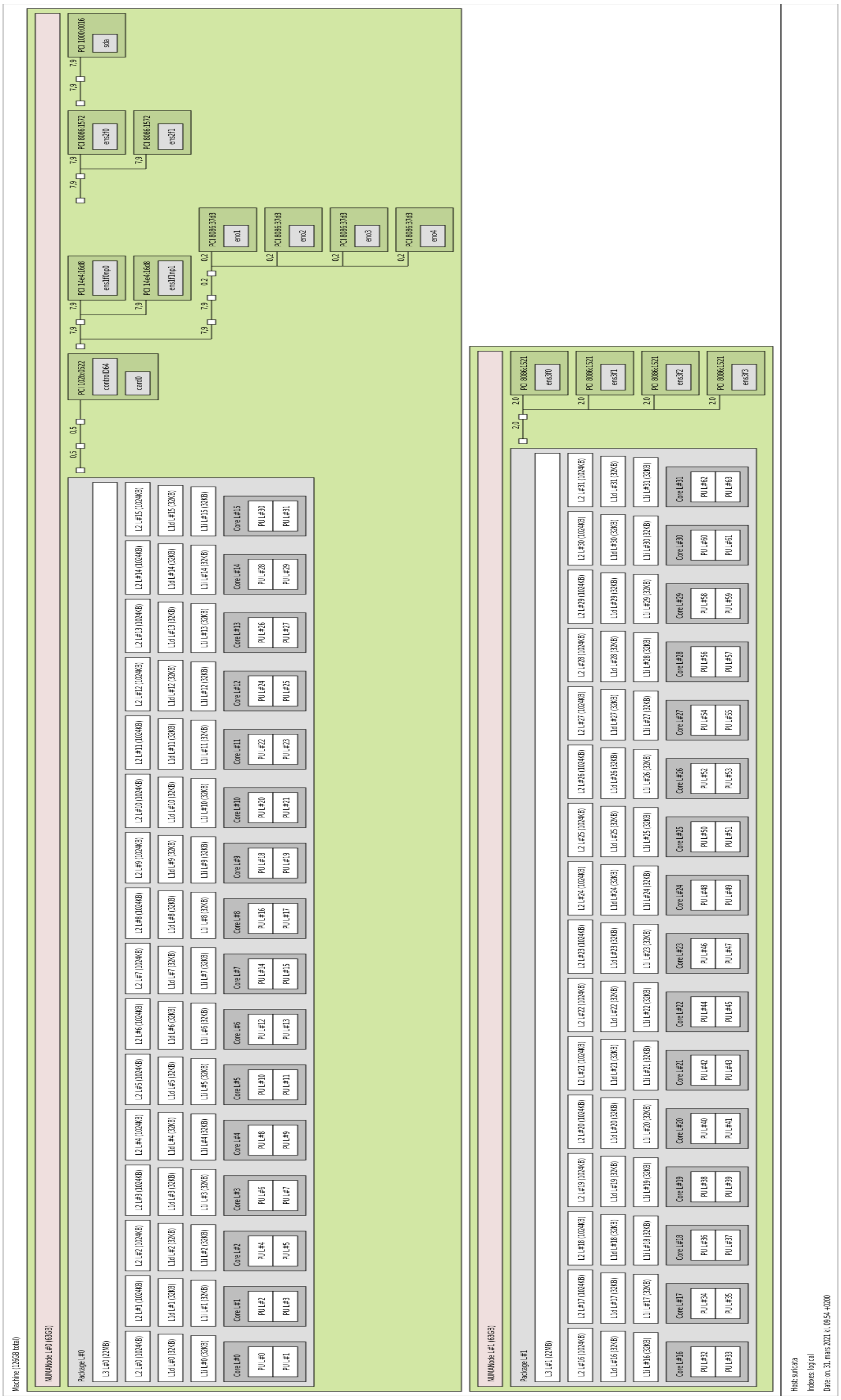

Figure 4.4: Output image from hwloc. Displays the hierarchical topology.

The main goal is to use the LLC as a bus and let the NIC push packets directly to the LLC, as illustrated in the simplified model in figure 4.5. This way, the worker threads will more often find packets in the LLC, effectively reducing cache misses. This is an Intel feature called DCA, an accelerated copy by a dedicated DMA engine [30, p. 11].

Figure 4.5 illustrates the packet pipeline in this system. Incoming packets are fetched by the NIC into multiple *first-in-first-out* (FIFO) receive queues, one per Suricata worker CPU. By using the symmetrical RSS hash value, packets of the same data flow are assigned to the same receive queue. The NIC driver allocates and maintains one ring buffer per receive queue within system memory. Af-packet, a high speed capture method built into Linux used to load balance the incoming packets from the NIC, then moves incoming packets from the receive queues to the corresponding ring buffers using DMA. DCA makes sure that the ring buffers are available in the LLC. The NIC will finally send interrupts to the Suricata worker CPU corresponding to the ring buffer. The CPU respond to the interrupt by starting to process packets. There is a interconnection between the two NUMA nodes in case they need to do a search in the corresponding nodes cache. NUMA The second node is idle in this experiment, but would run other processes in an operational environment.

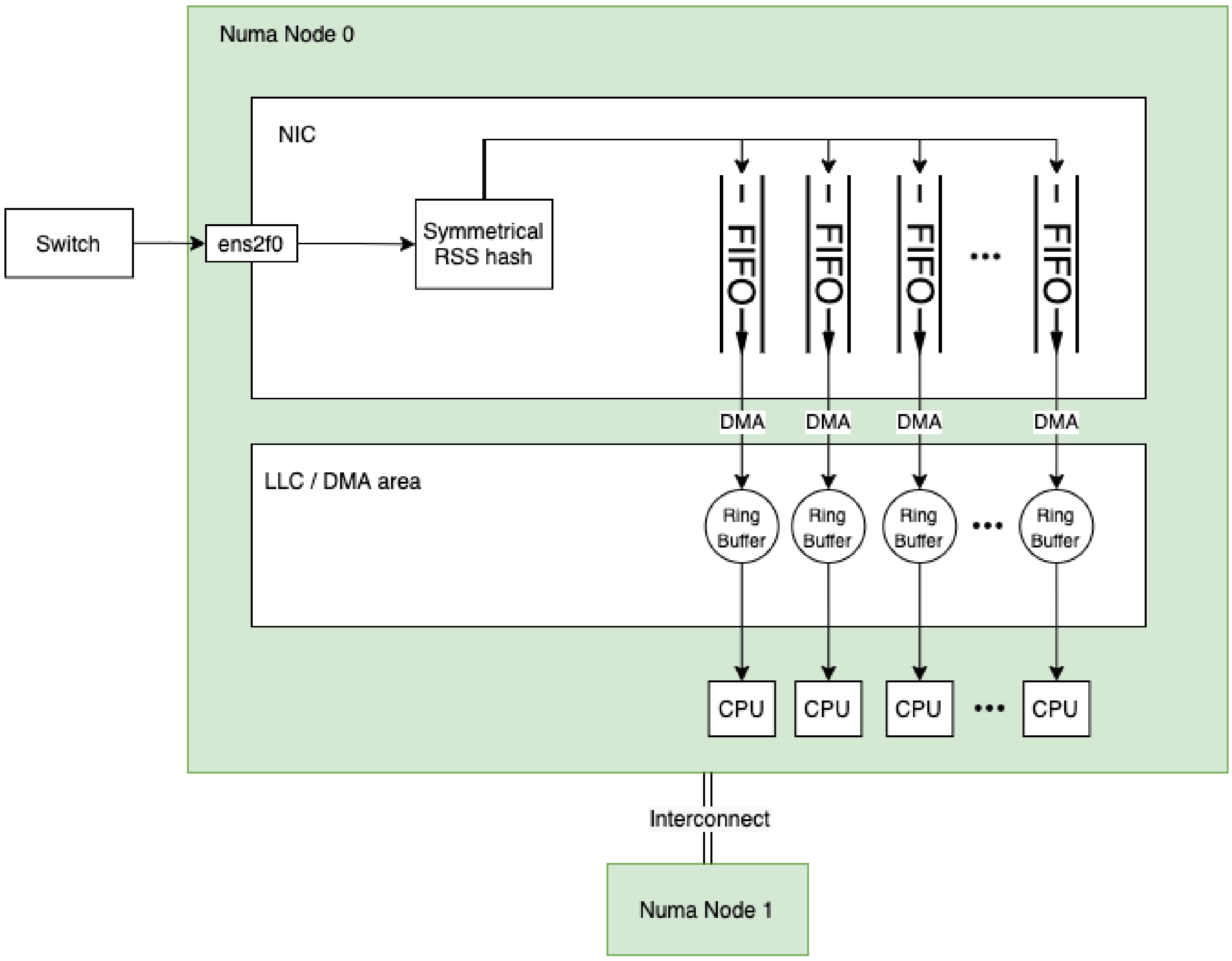

Figure 4.5: Illustration of the packet pipeline.

4.2 Behavior

The main behavior to study is the Suricata throughput, which is affected by several points in the packet pipeline. As stated in [30, p. 34-35], it is important to monitor packet drops in the following points:

Switch
As opposed to optical splitting, port mirroring can lead to packet drops. This can be mitigated by keeping the traffic under the supported throughput.

NIC
The NIC will drop packets which are malformed, or if the receive queue is full.

Softirq layer
The kernel will drop packets if it cannot pull packets fast enough from the receive queue and into the ring buffer.

Kernel drops
If the kernel cannot handle the throughput, it will discard packets instead of sending them to userspace.

Suricata
If Suricata cannot process the incoming packets fast enough, packets will be dropped.

4.2.1 Traffic Profile

The traffic generator is configured to simulate a client-side and a server-side with multiple concurrent connections. In order to simulate typical network traffic, a traffic profile included with TRex called *sfr_delay_10_1g.yaml* is used. The profile contains a combination of common protocols, and is defined by a French telecommunications enterprise called SFR. It is suitable for the stress testing in this research, and benchmarking in general. The traffic consists of the following protocols: HTTP/HTTPS, RTP, Exchange, Citrix, SMTP, DNS and Oracle. The test will produce 1 Gbps of traffic by default and can be adjusted up and down with a multiplier value.

The traffic profile introduces a delay of 10 ms from the client and server side, which is typical in a real-world scenario. Figure 4.6 illustrates the type of traffic transmitted, and the given time the packets are sent. The figure does not indicate how much data is sent per protocol, nor does it show in which direction each packet is going.

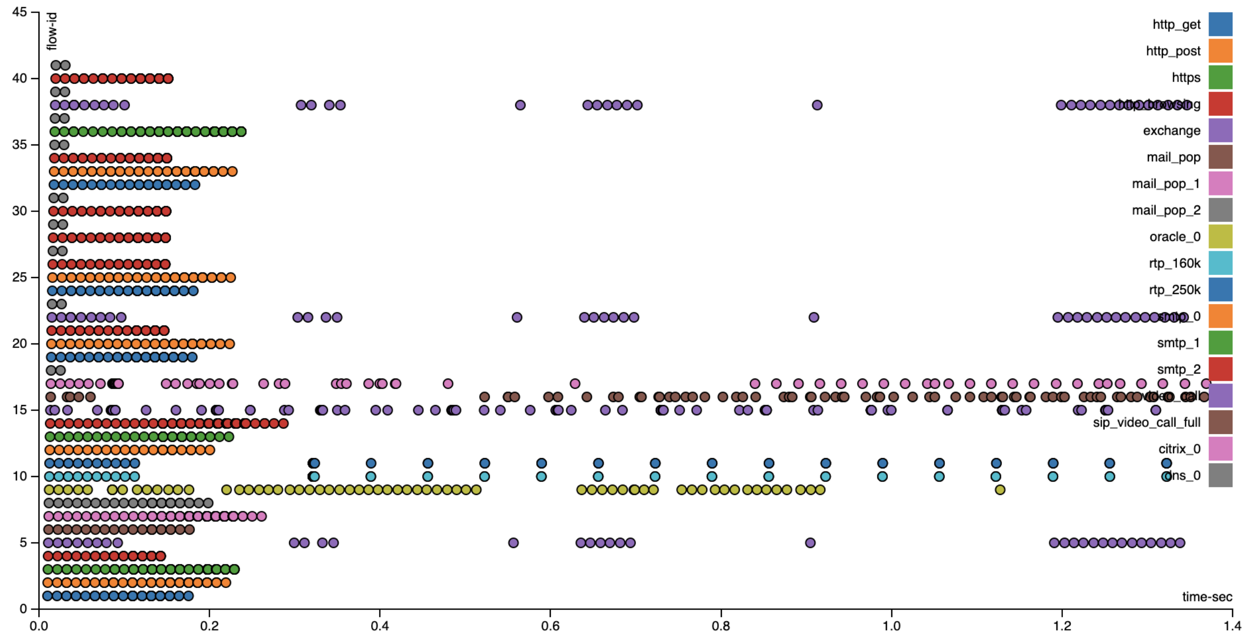

Figure 4.6: Illustration of the given time the packets are sent throughout the testing.

4.3 Testing Methodology

Ansible is used for automating the testing process and apply the configurations, as well as fetching the results from the sensor. This section first presents the Ansible task execution order, and then explains the testing approach step by step. Appendix A presents a guide for configuring Suricata manually.

4.3.1 Ansible

Ansible is used to automate the tests and to apply all the configurations on the sensor. This approach saves a considerable amount of time, as the testing process is repetitive and time consuming. The Ansible code of the tasks can be referred to in appendix B.1.2. Figure 4.7 illustrates the Ansible task execution order in a simplified manner. The tasks are divided into three groups: system, Suricata and testing.

The system group includes all tasks that affects the Linux system

Isolate CPUs
: All CPUs reserved for Suricata needs to be isolated from the Linux Scheduler, as the scheduling will be handled by Suricata itself. CPU 0 should never be isolated, as it is widely used by many Linux services.

Install dependencies
: Ansible will verify that the required Linux packages specified in the task are present. Otherwise, they will be installed.

Reload i40e NIC driver
: This task will reload the Intel i40e NIC driver to make sure the NIC is reset. This

is a good practice before making changes to the NIC settings. Additionally, all NIC counters will be reset, such as packets received and packets dropped. This is a desired side effect when performing testing.

Disable irqbalance

Irqbalance is a Linux service that balances interrupts among cores, which in turn shifts interrupt affinities frequently. The service will interfere with the careful setup. Therefore, it needs to be disabled.

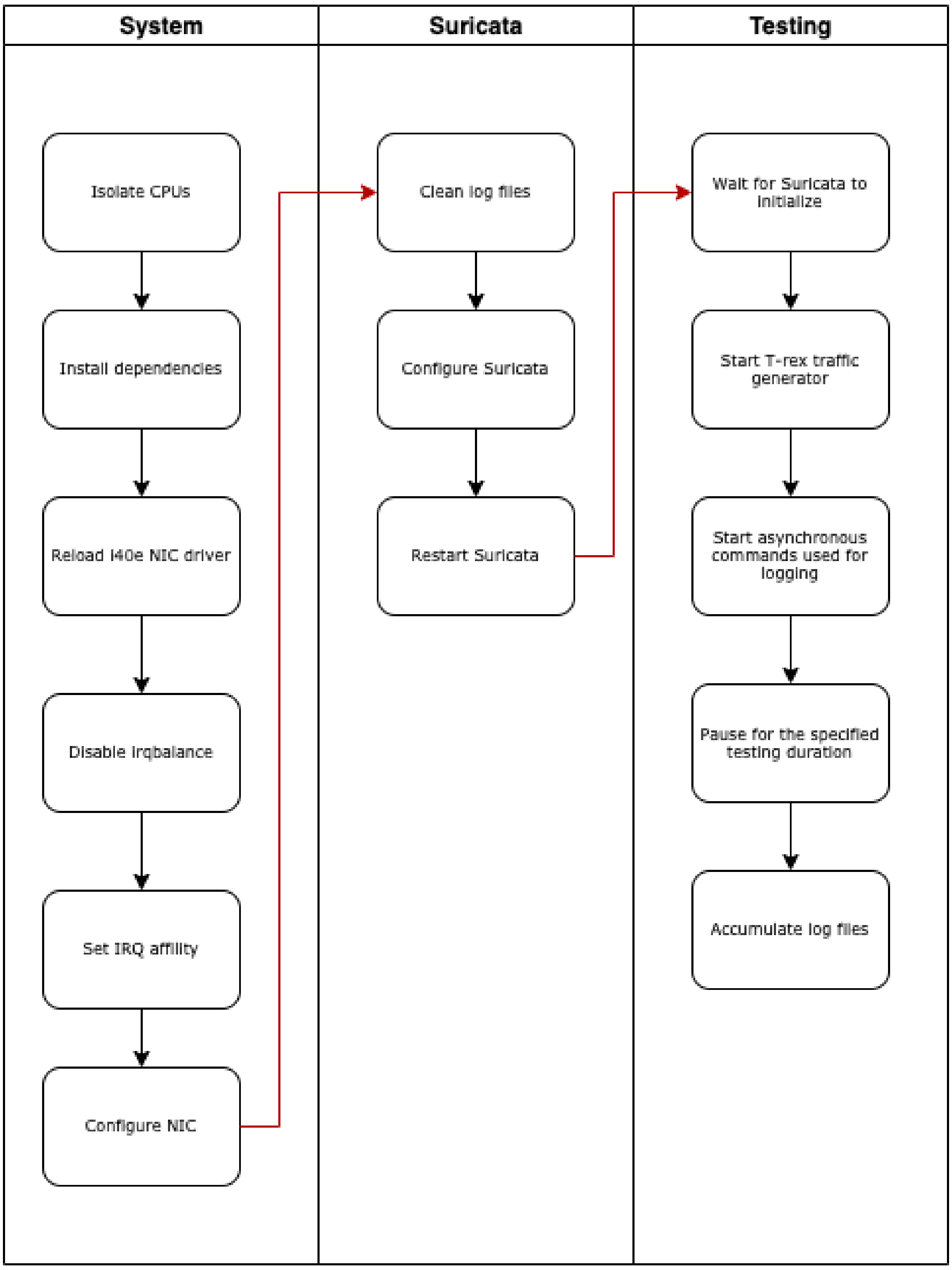

Figure 4.7: A simplified flowchart showing the Ansible task execution order.

Set IRQ affinity

With irqbalance disabled, it is possible to set an interrupt affinity for the NIC. By using the script *set_ irq_ affinity* mentioned in section 2.5.3, all interrupts from the NIC can be pinned to the Suricata worker threads. This way, the monitoring interface will not interfere with the rest of the cores.

Configure NIC

The NIC needs to be configured in order to cooperate with Suricata to handle the high throughput. The number of RSS queues are set to the number of Suricata worker threads so that every worker thread has a designated RSS queue. Load balancing is enabled for the NIC, which ensures that incoming traffic are distributed among the RSS queues. A symmetric RSS hash key are set to enable symmetric hashing.

The Suricata group includes all the modifications that affects the Suricata files

Clean log files

All Suricata log files should be cleaned before every test so only the relevant logs are gathered. Some of these log files must be manually deleted, while others can be configured to overwrite at startup.

Configure Suricata

The parameters in the Suricata configuration file found in appendix A.3 are used in the setup. The parameters of most importance is runmode and those under the af-packet and cpu-affinity sections. To better utilize the system resources, the runmode is set to workers. Here, each thread contains the full packet pipeline. In the af-packet section, the monitoring interface must be specified in order for Suricata to know which interface to listen to. *Cluster_ qm* is another parameter set in the af-packet section. This cluster type ensures that all packets are linked by the interface to a RSS queue and sent to the same socket. Also, a parameter that defines how many threads af-packet will use is set. Finally, the cpu affinity is set, determining which CPUs Suricata should use as worker threads. The management CPUs are also set, which will be used for housekeeping. They are not isolated from the Linux scheduler.

Restart Suricata

Suricata is restarted in order to apply the configuration.

The testing group includes all tasks that handles the testing. Each task is described below

Wait for Suricata to initialize

Suricata needs a few seconds to initialize and apply the configurations in order to be ready for processing packets.

Start TRex traffic generator

The TRex traffic generator is started and will send bidirectional, stateful traffic to the sensor.

Start asynchronous commands used for logging

A set of commands used for logging are executed in the background. All of the commands will run for the same duration as the test. The output of these commands are recorded by Ansible.

Pause for the specified testing duration

The Ansible playbook is paused for the specified testing duration. Suricata needs to process a sufficient amount of traffic for the statistics to stabilize.

Accumulate log files

Finally, Ansible will gather all of the recorded logs considered necessary for the results.

4.3.2 Testing Approach

A testing approach is established in order to stress test the sensor. The traffic load is set to 9 Gbps. This is 1 Gbps below the bandwidth capacity on the NIC and switch in order to mitigate packet loss. The default testing duration is 10 minutes, because Suricata seems to stabilize within this time frame. Each test is done twice to make sure the results are consistent. A ruleset of 29766 signatures from *Emerging Threats Open* are used throughout the testing. The ruleset is never updated in order to ensure consistency. The testing approach is described below:

Default configuration

This test will apply and test default configurations. The NIC is reset with default settings, and only the required settings are applied to the Suricata configuration. It is expected that the performance will be lower than the other tests. The result will be used as a baseline for comparison.

High performance configuration with default BIOS settings

This test will apply and test a high performance configuration using the workers runmode. In this setup, both the management and worker threads are located on NUMA node 0. Two management threads are used, pinned to the first CPU of both CPU groups. This should be sufficient for management and they should not be overloaded. The rest of the CPUs are pinned to the Suricata worker threads and isolated from the Linux scheduler. As the NIC is located on the same NUMA node, the latency should be minimal. Other noteworthy changes are larger buffer and memory allocations. Note that this test is performed with default BIOS settings. By limiting the processing power by unpinning one CPU at a time, the current Suricata throughput can be found. This is necessary in order to reveal any performance differences in the upcoming tests.

Apply recommended BIOS settings

The BIOS settings recommended in [30, p. 11-15] are applied for the remaining tests. Some noteworthy settings are mentioned below, while the rest of the settings are listed in appendix A.1.

- Enable DCA.
- Setting the max *Processor Idle Sleep States* (C-state) to 3.
- Enabling OS control over *Power Performance States* (P-states).
- Disable prefetchers, as DCA will send packets to LLC.

High performance configuration with recommended BIOS settings

The same high performance test is performed after the recommended BIOS settings are applied. The resulting Suricata throughput is used in the upcoming tests.

Additional testing

Five additional tests will be performed to examine the behaviour of the sensor with unconventional modifications applied, which is described in section 5.3. This includes different configurations of management and worker thread pinning, as well as different values for the buffer sizes and memory allocation sizes. Latency is introduced in one of the tests by pinning the Suricata worker threads to the CPUs on NUMA node one. This will cause the packets to traverse the NUMA interconnection link between node one and two. The result is designed to result in lower performance as a control.

Identifying the Suricata throughput

After the most performant Suricata configuration is found, the throughput limit needs to be identified. If Suricata handles the traffic load without packet loss, the processing power is lowered until a limit is found by unpinning CPUs. If Suricata does not handle the traffic load without packet loss, the traffic load is lowered until a limit is found.

Chapter 5

Results

Throughout this chapter, the terms packet loss and packet drop will have the same meaning and will be used interchangeably. To calculate the percentage of the packets being dropped throughout the packet pipeline, the drop percentages on the NIC, softirq and Suricata will be summarized. A distinction is made between packet loss at Suricata and in the kernel. Kernel drops are more severe, as it seems to trigger an increase in packet drops. The total packet loss tolerance limit in this research is set to 1%. If a result exceeds this tolerance, the Suricata throughput for that test is deemed invalid.

5.1 Default Configuration

The default configuration test is able to process 7 Gbps of data before exceeding the packet loss tolerance limit. In figure 5.1, the amount of packets captured by the kernel is presented as the y axis, and the testing duration in seconds as the x axis. The blue line counts the number of packets correctly sent to user space. There is a small drop of received packets when reaching 600 seconds. This is because the traffic generator is in a termination phase. The orange line counts the number of packets that have been discarded by the kernel instead of being sent to user space. The green line counts missing data packets in TCP streams, which is also affected by bad checksums and the Suricata engine running out of memory. If the value is positive, it could indicate that packets are being dropped. The values stabilize throughout the test.

Figure 5.2 shows the amount of packets received for the different traffic loads. When the traffic load is 4 Gbps and higher, packets captured by the kernel and the NIC are slightly lower than the total amount of packets received, indicating that packets are lost. The TCP reassembly gap statistics are also growing with the traffic load, but stabilizes from 6 Gbps and out. This indicates that the Suricata engine has problems with the high traffic loads related to memory capacity.

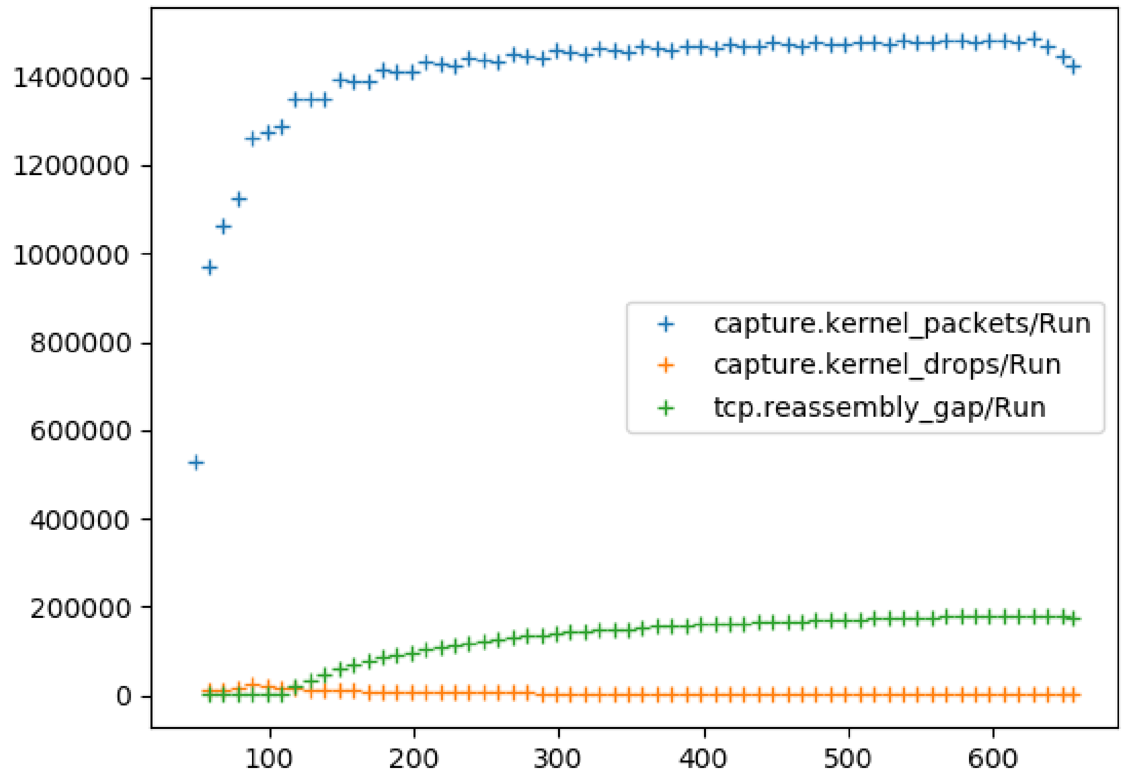

Figure 5.1: Statistics for the default test with 7 Gbps of data over a timespan of 600s.

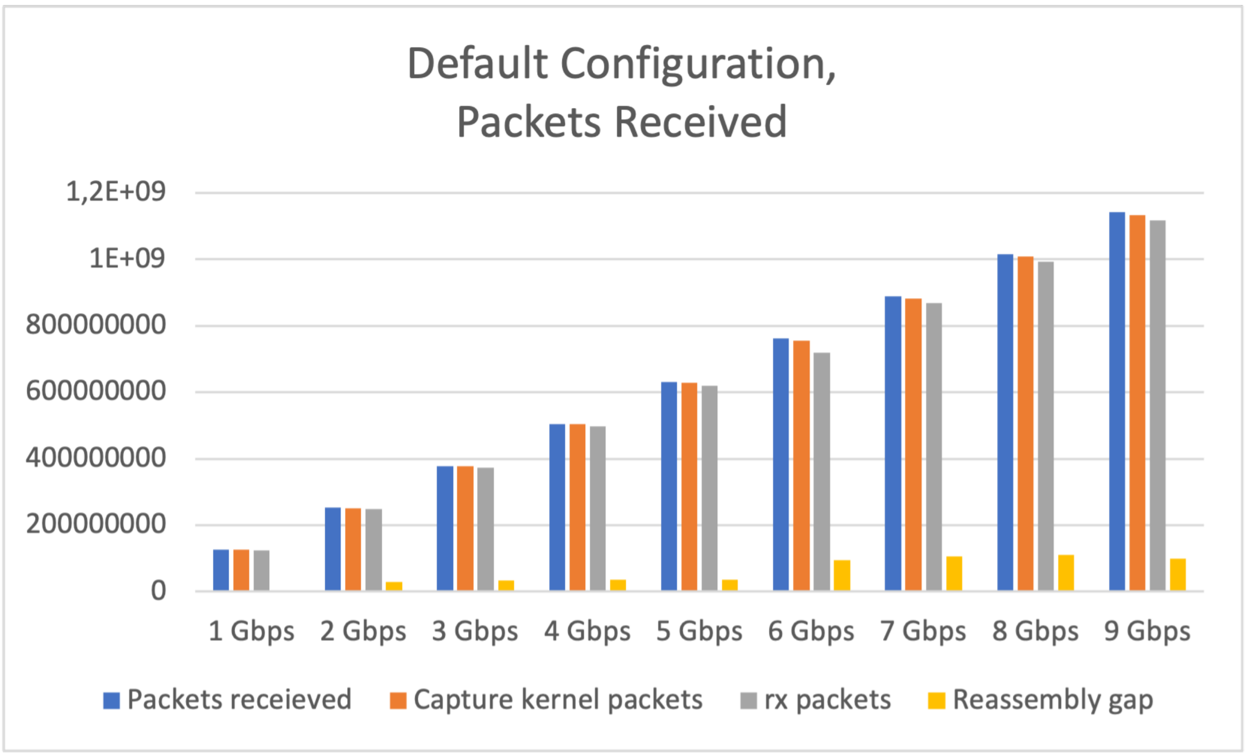

Figure 5.2: Packets received by Suricata with default configurations. Compared over a traffic range from 1 to 9 Gbps.

Figure 5.3 displays the packets being dropped for the different traffic loads. The figure illustrates where the packets are being dropped throughout the packet pipeline. At 6 Gbps, the kernel starts dropping packets. This causes the total drop rate to grow at a higher rate. At 7 Gpbs, the packet loss tolerance limit is still satisfied with a value of 0.95%. Note that some packets are dropped on the monitoring interface in accordance with the traffic load. This is not revealed in the figure, as the dropped packets only accounts for 0.01% of the received packets. However, the number is steadily growing, indicating that the monitoring interface encounters a problem with the increasing traffic load.

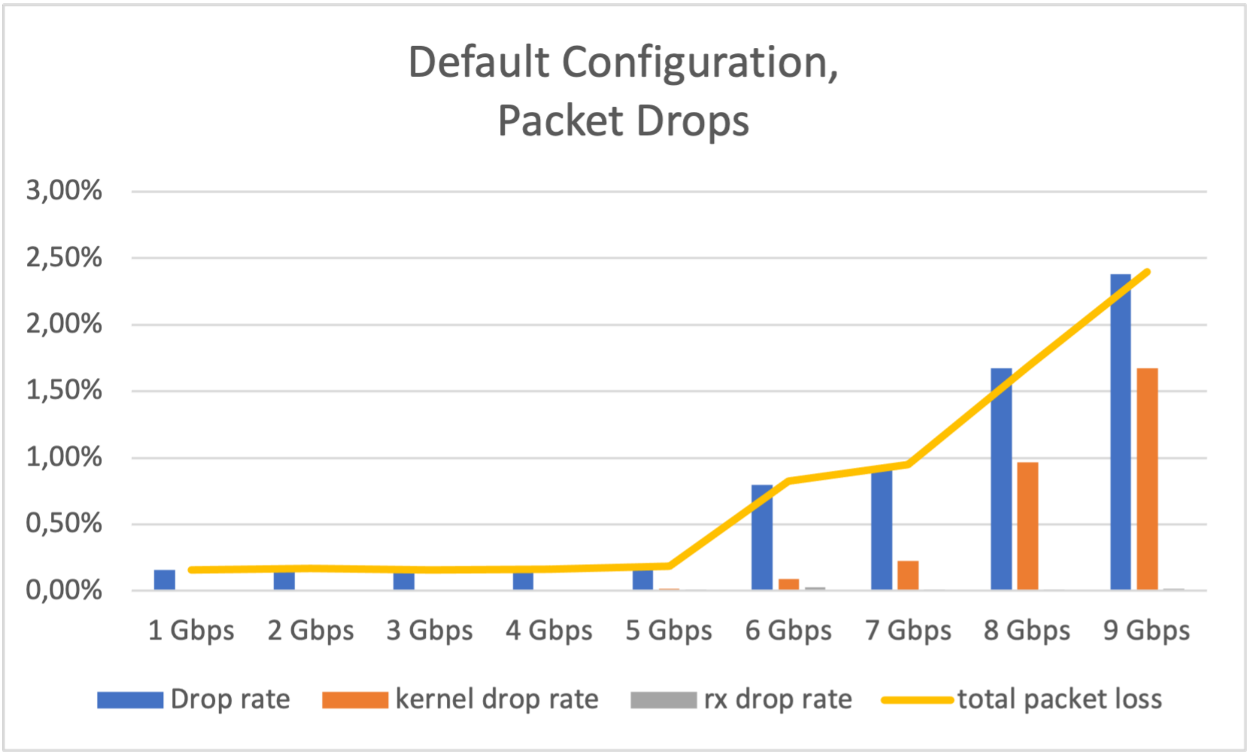

Figure 5.3: Packets dropped by Suricata with default configurations throughout the packet pipeline. Compared over a traffic range from 1 to 9 Gbps.

The cache misses are almost constant at 26% throughout the tests, as illustrated in figure 5.4. Because no system modifications has been done at this stage, this value is considered to be the standard baseline. The CPU utilization is on the other hand steadily increasing in accordance with the traffic load, a behavior that is logical. At 9 Gbps, none of the CPUs exceeds 30% of CPU utilization. This is because no CPUs are pinned to Suricata yet, which allows the IDS to utilize all 64 CPUs.

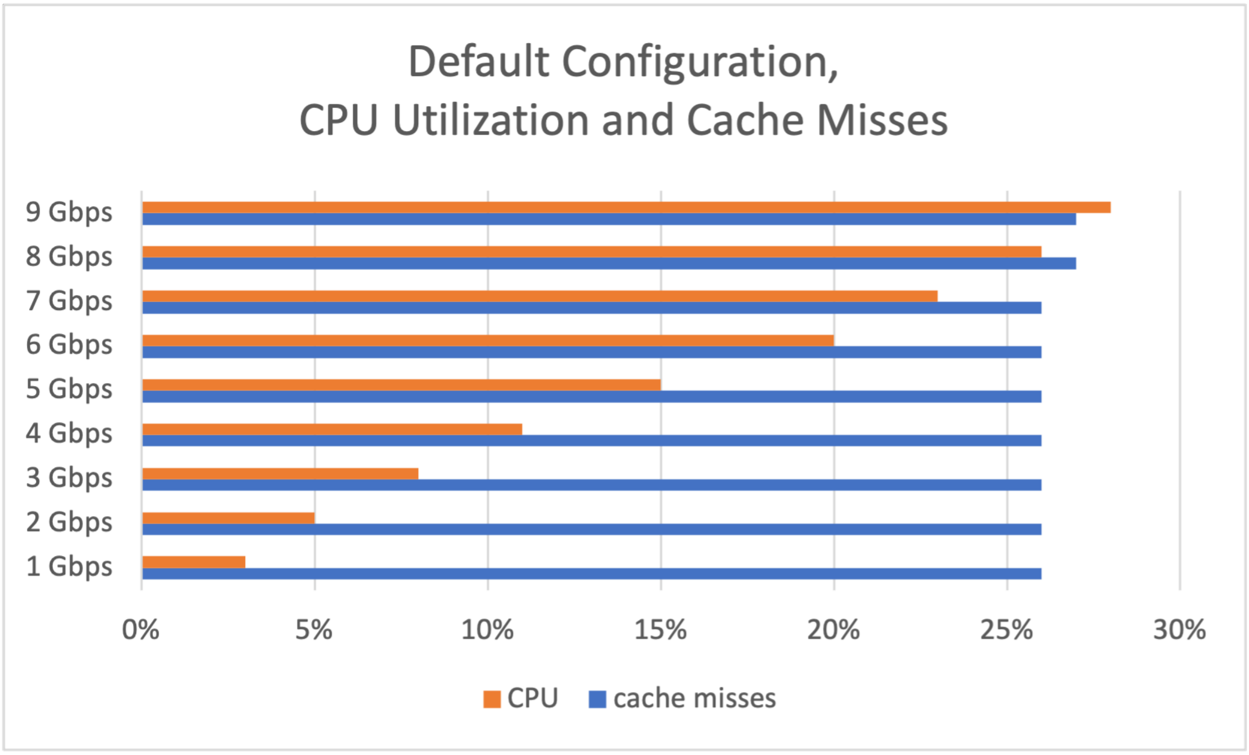

Figure 5.4: CPU utilization and cache misses for Suricata with default configurations. Compared over a traffic range from 1 to 9 Gbps.

5.2 High Performance Configuration

Figure 5.5 shows the total packet drop rate when running Suricata in a high performance configuration with default BIOS settings. The figure can be compared to figure 5.6, presenting the same test with recommended BIOS settings. In both cases, no packets are dropped on the monitoring interface, and an increase in kernel drops is evident when using 9 CPUs. The results in figure 5.5 show a slightly lower packet loss percentage, which is contrary to the expectation. Regardless of the BIOS settings, Suricata needs to use 10 CPUs in order to satisfy the packet drop tolerance limit. The TCP reassembly gap statistics are constant in these tests.

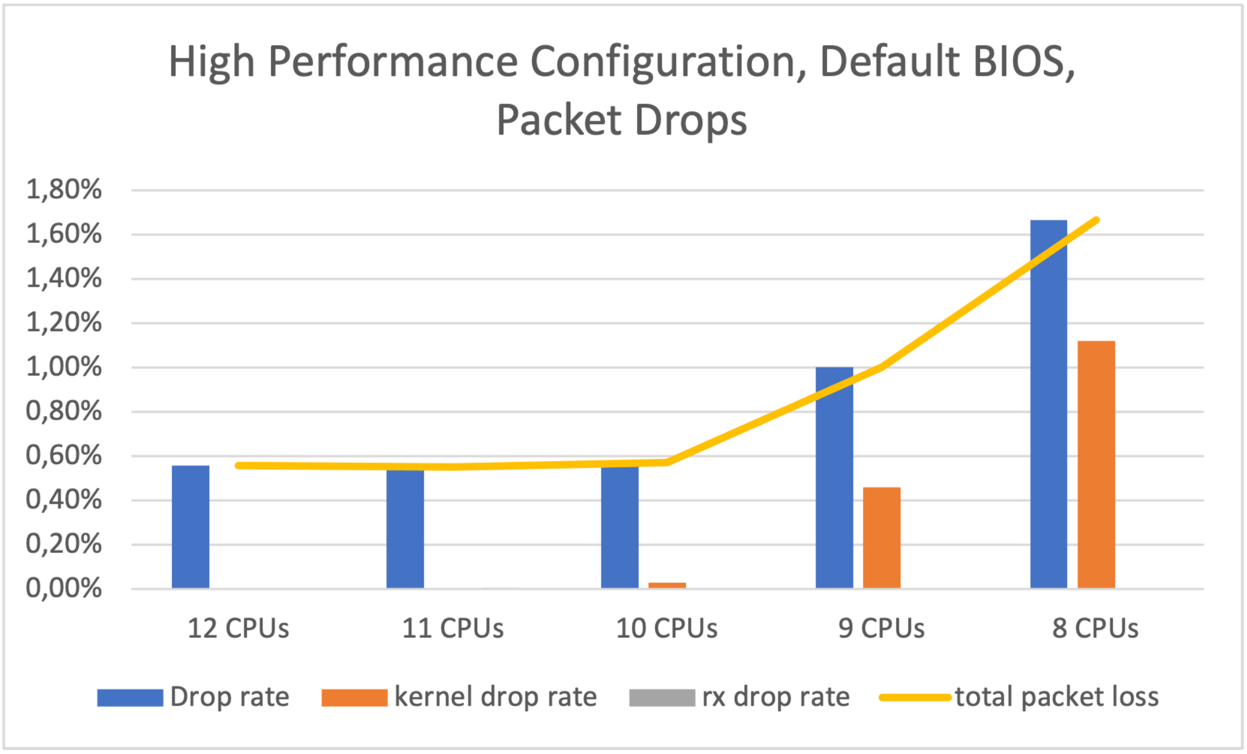

Figure 5.5: Packets dropped by Suricata in a high performance configuration with default BIOS settings. Compared from 8 to 12 CPUs.

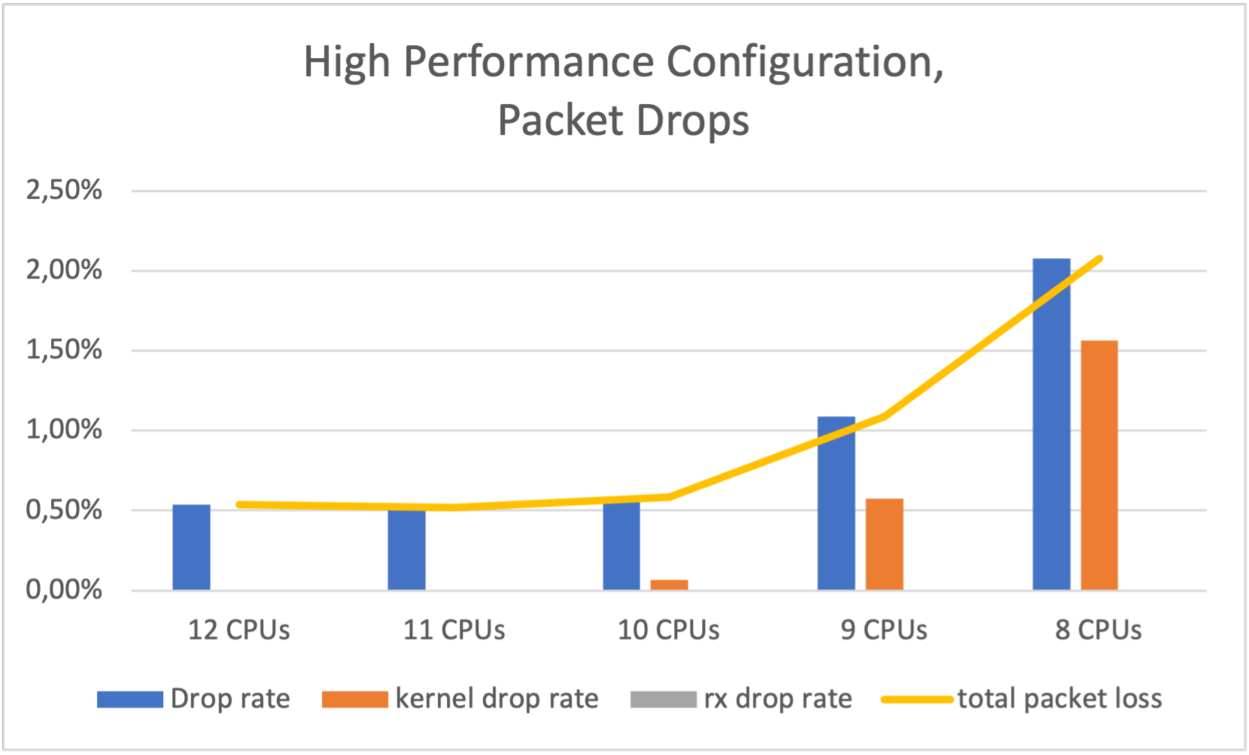

Figure 5.6: Packets dropped by Suricata in a high performance configuration with recommended BIOS settings. Compared from 8 to 12 CPUs.

Figure 5.7 shows the CPU utilization and cache misses for default BIOS settings. The figure can be compared to figure 5.8, presenting the same test with recommended BIOS settings. The differences in the results are insignificant. Just like figures 5.5 to 5.6, Suricata performs slightly better with default BIOS settings. The average cache miss statistics are constant on 35%, which is 14% more than the default test. This is a surprising results, as the BIOS settings were expected to greatly reduce the cache misses. In [30, p. 19-20], the resulting cache miss of a similar configuration is close to zero. When using fewer CPUs, the overall CPU utilization is higher, peaking at 82% for 8 CPUs. This is expected, since there are fewer CPUs to process the same amount of Suricata rules. Even though default BIOS settings yielded slightly better results, the recommended BIOS settings are used in the upcoming tests.

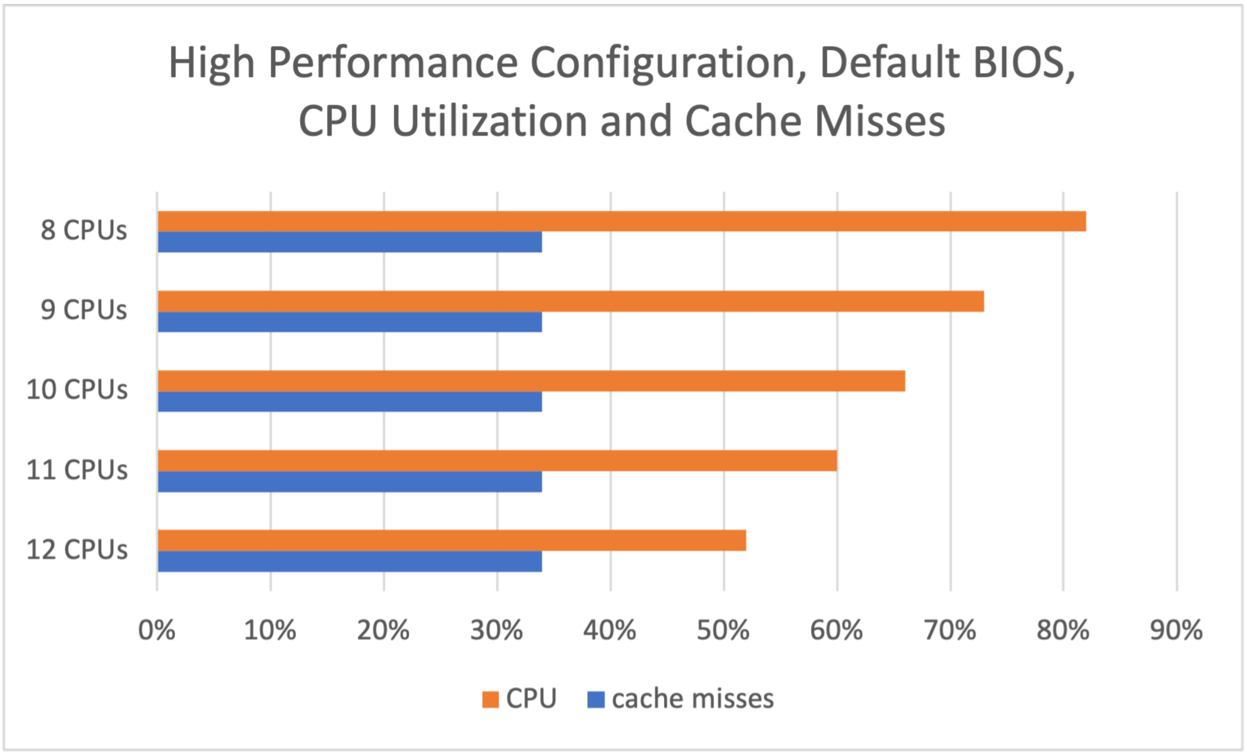

Figure 5.7: CPU utilization and cache misses for Suricata in a high performance configuration with default BIOS settings. Compared from 8 to 12 CPUs.

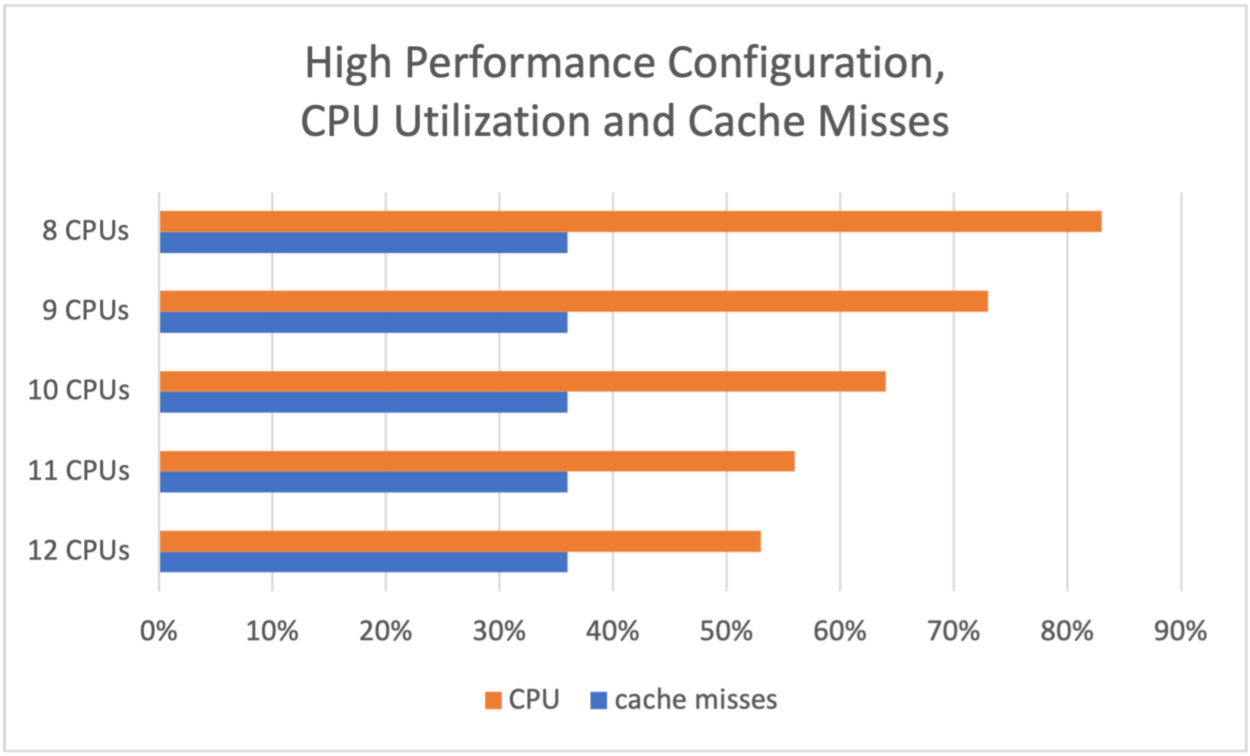

Figure 5.8: CPU utilization and cache misses for Suricata in a high performance configuration with recommended BIOS settings. Compared from 8 to 12 CPUs.

5.3 Additional Testing

In this section, five additional test are performed in order to observe how Suricata behaves when modifying the high performance configuration in unconventional ways. The first test is called *latency*, where the Suricata worker threads are pinned to the foreign NUMA node. This will add some latency which should decrease the performance. The second test is called *mgmt node 1*, where the two management threads are moved to the foreign node. If the performance is not impaired by this modification, it could be useful to have two additional disposable CPUs on the local node.

The third test is called *default buffer*, where the buffer values *afpacket_ ringsize*, *afpacket_ blocksize* and *max_ pending_ packets* are set to the default values. The modifies buffer values can be found in appendix A.3.3 and appendix A.3.1. The fourth test is called *default memory*, where stream memory values such as *memcap* are set to the default values. The modified memory values can be found in appendix A.3.3. The fifth test is called *default memory and buffer*, which is a combination of test three and four. The performance is expected to be impaired by the memory and buffer limitations, and the performance gain by using more memory resources will be evident.

Figure 5.9 shows the total packet drop rate when running the additional tests using 10 CPUs with the recommended BIOS settings. Next to no packets are dropped on the NIC. The tests with default buffer sizes show a higher drop rate of 1.83%, which includes a high number of kernel drops. This makes it clear that the default buffer sizes do not have enough capacity to contain the incoming packets. The latency test show a lower drop rate of 0.53%, slightly better than the high performance configuration. This is unexpected, and will be further discussed in chapter 6.

The two rightmost tests on the figure show next to no packet drops. At first glance, it seems that the default memory sizes improves performance. However, figure 5.10 shows that Suricata processes approximately half of the flows compared to the other tests. This is supported by the results in figure 5.11, where the reassembly gap statistics are in accordance to the amount of flows processed. Therefore, the packet loss for the default memory must be a misrepresentation.

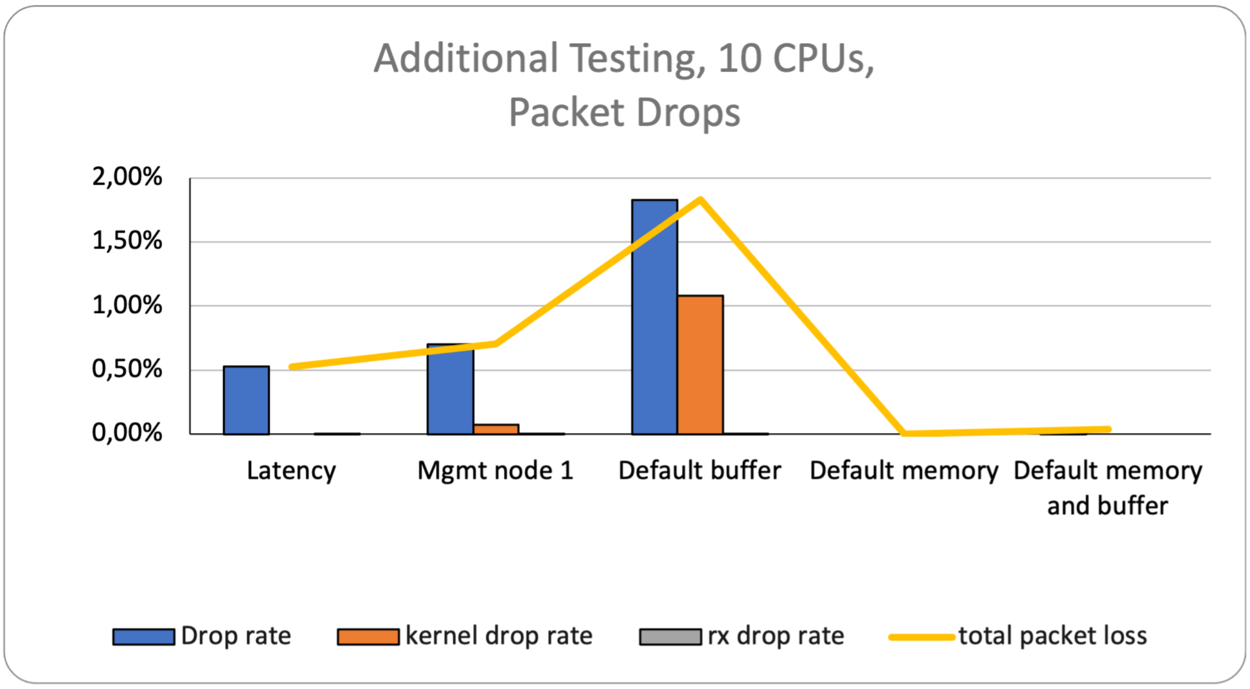

Figure 5.9: Packets dropped by Suricata for the additional tests. Suricata is using 10 CPUs with the recommended BIOS settings.

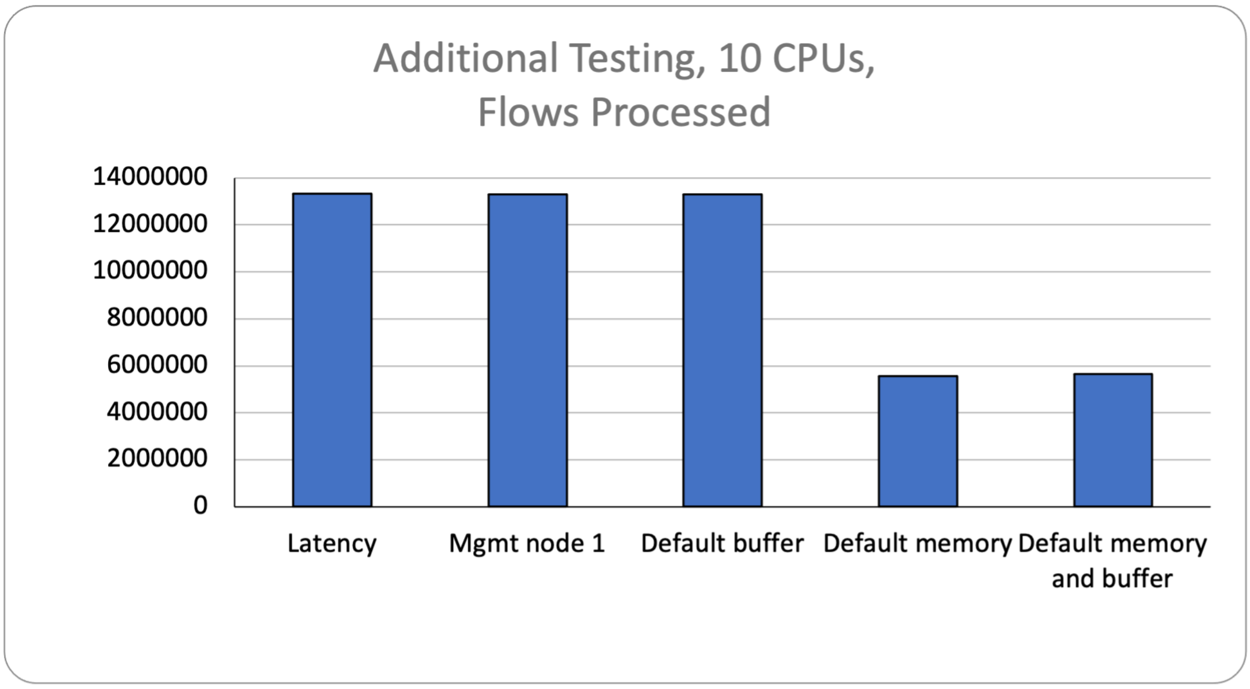

Figure 5.10: The amount of flows processed by Suricata for the additional tests. Suricata is using 10 CPUs with the recommended BIOS settings.

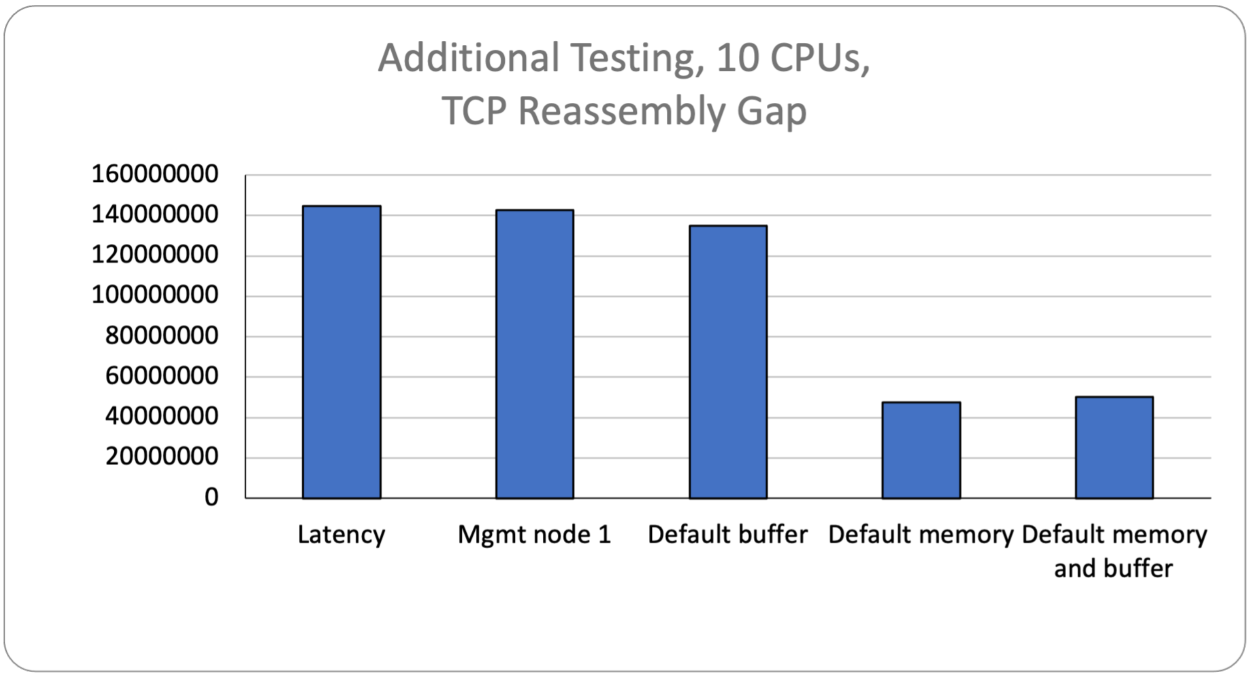

Figure 5.11: TCP reassembly statistics for the additional tests. Suricata is using 10 CPUs with the recommended BIOS settings.

Figure 5.12 shows the CPU utilization and cache misses with small variations for the additional tests. The results are similar to the high performance test when Suricata is using 10 CPUs presented in figure 5.8. However, the additional tests resulted in an average of 30% cache misses, which is 5% better than the high performance configuration.

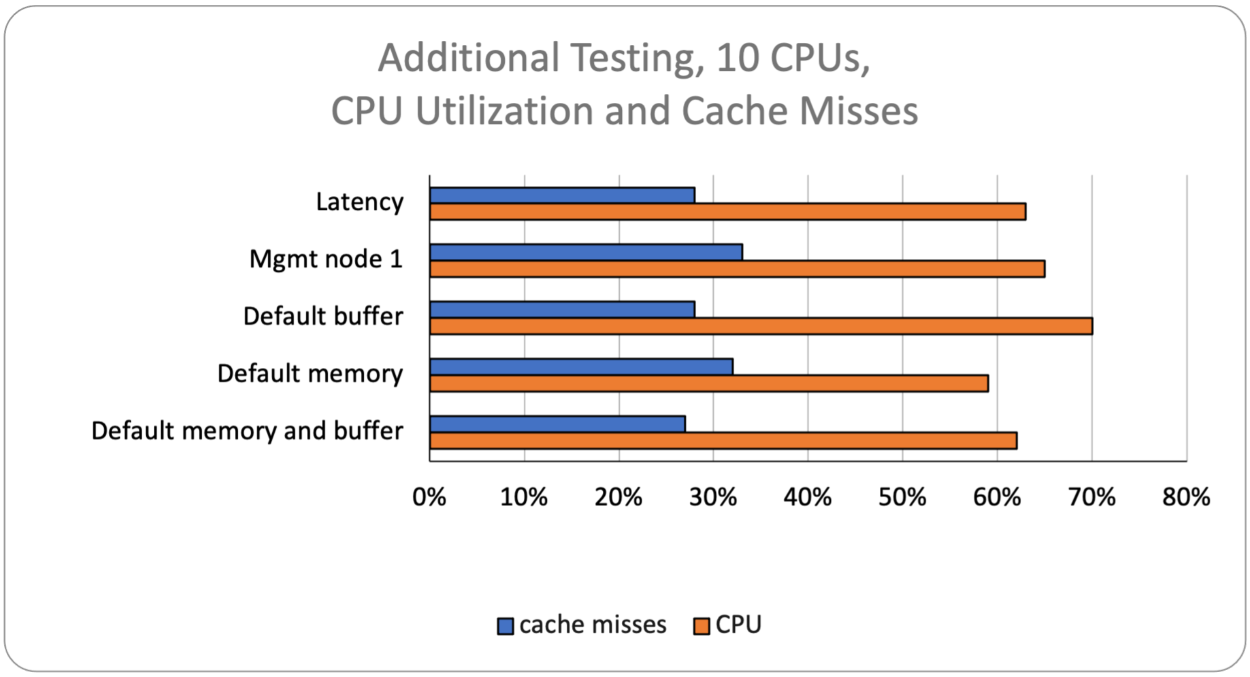

Figure 5.12: CPU utilization and cache misses for the additional tests. Suricata is using 10 CPUs with the recommended BIOS settings.

Chapter 6

Discussions

This chapter will further discuss the results by putting them into context. Then, the findings will be used to answer the problem statement. Finally, the most fundamental decisions that were taken in this research are discussed.

6.1 Test Results

Suricata never reports a lower packet loss than approximately 0.5%. However, it should be feasible to achieve a packet loss of zero. There is an indication that a significant packet loss occurs when the testing phase begins. An absolute amount of packets are being dropped regardless of the traffic load, testing duration and the processing power. It does not help to delay the traffic generation until Suricata is fully initialized. However, the initial packet loss does not include any kernel drops. Therefore, the kernel drop statistics are more trustworthy. Neither of the tests revealed any packet drops in the softirq layer. This indicates that the Linux kernel pulls packets fast enough from the RX queues and into the ring buffer. A sensor with different hardware might not produce the same result.

6.1.1 Default Configuration

The default configuration utilizes all 64 CPUs on the system. It is important to note that even though all CPUs are used, the CPU utilization is only 23% when reaching the Suricata throughput. This indicates that the processing power is not the limiting factor. The NUMA topology is not utilized, which could cause the NUMALINK to be saturated by the communication between threads on different nodes. Also, the capture threads might have run out of capacity caused by the increasing traffic load.

6.1.2 High Performance Configuration

The high performance configuration performed slightly better with default BIOS settings. Even though the results are almost identical, the recommended BIOS settings were expected to yield significantly better results. An explanation could be that the BIOS is not configured correctly, or that some functionality is missing. The high cache miss results of 35% also indicate that something is not working as intended in the packet pipeline. Another explanation is that the BIOS changes would only result in a better performance when the network speed reaches 50 Gbps and beyond, as stated in section 3.1. Troubleshooting this misbehavior is out of scope for this research.

6.1.3 Additional Testing

The additional tests include: *latency test*, *management on node one*, *default buffer*, *default memory* and *default memory and buffer*. None of the tests showed any improvements to the system. There were however some unexpected results. The purpose of the additional tests was to get a better understanding of how Suricata behaves when diverging from the recommended configurations. There are definitely more tests in this category that could have been conducted. In fact, a few additional tests were not included in the report, as they were not fruitful. Regardless, it is assumed that the included tests are most important.

Latency test

The latency test performs surprisingly well despite the fact that every packet traverses the interconnection between the NUMA nodes to reach the CPUs. A possible explanation is that there is no other communication on the NUMALINK as no other demanding processes are running on the system. The NUMALINK has a specific bandwidth capacity, and it seems that Suricata does not saturate the bandwidth by itself. By introducing more communication on the link, it is likely that the latency test would cause performance issues.

Management on Node One

With a packet loss of 0.7%, it seems that by moving the management threads to the foreign node, some performance is lost. The explanation could be that the management threads establish enough communication between the NUMA nodes to saturate the bandwidth. This is only a speculation, since there are not enough information available about the management thread behavior. The results are disappointing, as it would be practical to free up two CPUs from the local node. From a more realistic standpoint, the results only show a packet loss increase of 0.2%. Therefore, it should be possible to take advantage of this modification without suffering any significant performance degradation. On one hand, OISF recommends in [18] to pin the management threads to 10 CPUs on the foreign node. On the other hand, the management threads were pinned to two CPUs on the local node in [29].

Default Memory and Buffer

The results show that using the default memory values, zero packets are lost. Closer inspection of the results reveals that the flows processed is less than half of the normal value. It is uncertain why this is not reflected by the packet loss results. The TCP reassembly gap counter is also much lower than the normal value, which is likely because Suricata handles less flows.

The results when using default buffer values show a high packet loss of 1.83%, which does not satisfy the tolerance limit. This is expected, as the buffers do not have enough space for the incoming packets. When the buffer is full, the incoming packets will overwrite the already existing packets in the buffers. This will result in packets being dropped.

6.2 Identifying the Suricata Throughput

By analyzing the results, it seems that the latency configuration performs the best, slightly better than the high performance configuration. However, this test is designed to yield lower performance. Therefore, it is not used to determine the Suricata throughput. The high performance configuration with default BIOS settings results in the best performance. It is able to process 9 Gbps of traffic with 10 CPUs, only dropping 0.57% of the packets. This is 2 Gpbs more traffic than the default configuration. The final configuration is presented in appendix A.3.

During the stress testing of the sensor, no other processes were running besides Suricata. Concurrent processes on any of the nodes may impact the results in a negative way, but this is only speculations. It would probably affect the latency test the most, as the NUMALINK is the weakest link.

6.3 Decisions

A few decisions had to be made in this research. Ansible is an optional tool that is not strictly necessary to conduct the experiments. Pre-recorded traffic was used for the testing instead of real-time traffic. The TRex traffic generator is running on an independent server instead of replaying traffic on the sensor. Finally, Suricata requires one management thread per CPU group.

Ansible

As Telenor uses Ansible to deploy IDS configurations in the operational environment, the same tool is used in this research for convenience. It also turned out to be useful in the testing phase in order to automate repetitive tasks. A considerable amount of the research time frame was devoted to the Ansible project. The code is available on [34], and the most important parts is presented in appendix B.

6.3.1 Pre-Recorded Traffic

Pre-recorded traffic is used for the testing to enforce consistency. Another option is to use real-time traffic from Telenor's systems to make a more authentic test environment. However, this would complicate the testing as real-time traffic is an uncontrollable variable. In addition, this traffic is privacy sensitive. Finally, it would be challenging to get a traffic rate reaching 10 Gbps. A third option is to replay PCAPs on the sensor by using the tool *tcpreplay* or similar. This would be more convenient, but it could affect the sensor in a negative way.

Testing Phase

It is decided to run the TRex traffic generator on a separate server to not disturb the sensor during the testing phase. A switch is used to mirror the traffic to the sensor in contrary to a network tap in order to make the setup work as intended. With a tap, it is necessary to spilt each side of the bidirectional flow on two different ports. This does not work with the workers runmode in Suricata, as it is required to receive the bidirectional traffic on the same port.

6.3.2 Management Threads

The first CPU in each group is pinned to a management thread. These CPUs also handle Linux specific tasks. This is configured in the af-packet section of the Suricata configuration file. Suricata requires one af-packet section per CPU group. This is because of how each CPU group is distributed on the NUMA nodes. Each af-packet section require the same number of threads, which leads to an even number of total threads when two sections are declared. If a different number of threads are set in the af-packet sections, Suricata will pin the additional threads to the previously pinned CPUs. Even though one management thread should be sufficient for two af-packet sections, it is not possible in this case.

Chapter 7

Conclusions

The first main goal of the project was to set up a stand alone system used for generating traffic. This was done by configuring the TRex traffic generator on a separate system that can generate up to 10 Gbps of bidirectional traffic. This prevents the workload of generating traffic to affect the sensor. By connecting TRex to a switch in a loopback configuration, the traffic was mirrored from the switch to the sensor. To make sure that no packets are lost because of the bandwidth capacities, a maximum of 9 Gbps of traffic is sent. The second main goal was to configure Suricata for high performance on a single NUMA node. This comprised of making modifications on the NIC and Suricata, as well as making adjustments in Linux and in the BIOS. The CPU affinity of Suricata was restricted to the CPUs in NUMA node 0. The parameters in af-packet, cpu-affinity and runmode among others in the Suricata configuration file have been adjusted to utilize the system resources in a more efficient way. Several settings were adjusted on the NIC, where the most important change was to set a symmetric RSS key in order to enable symmetric hashing. This allows the use of multiple RSS queues. The CPUs pinned to Suricata have been isolated in the kernel. This is done to prevent the CPUs to be interrupted by the Linux scheduler while processing Suricata rules. The BIOS were adjusted as advised in [30, p. 11-15]. However, these changes had no performance impact on the results. A traffic rate beyond 50 Gbps might reveal different results, as discovered in section 3.1.

The third goal was to compare the high performance Suricata throughput with the default OOTB configuration of Suricata. The default configuration, using all 64 CPUs on the system, managed to process 7 Gbps of data before the Suricata throughput was reached. The high performance configuration reached a Suricata throughput of 9 Gbps using only 10 CPUs. This allows the remaining 54 CPUs to be utilized for other critical processes. The fourth and final goal was to use an automation tool in order to conveniently deploy the final configuration on any system that meets the hardware requirements. Ansible was used to develop a configuration management module used to both deploy the configuration and to automate the testing. The module can be used to deploy the high performance configuration on any sensor that satisfies the hardware requirements. However, the configuration needs to be adjusted in order to adhere to systems with different hardware specifications.

To conclude the research, a significantly higher Suricata throughput have been achieved by tuning Suricata for high performance on a single NUMA node, and the packet pipeline has been partially optimized. The solution is able to solve the issue Telenor is facing regarding packet loss. Nonetheless, several problems arose during the research. These are detailed in the next section.

7.1 Future work

The problem regarding the unexpected *packet loss at startup* should be addressed in the future. This may uncover several underlying issues in the packet pipeline. As mentioned in section 6.1, delaying the traffic generation until Suricata is finished initializing does not make any difference.

The TCP reassembly gap statistics related to the *suricata memory* settings are alarmingly high for all of the tests, and the value grows in accordance to the traffic load. This indicates that there is a problem. This seems to be related to another counter called *TCP segment memcap drop*, which is also showing high results. By fine-tuning the stream memory parameters in the Suricata configuration file presented in appendix A.3.3, these statistics could be improved.

Using the last layer cache as a bus did show some unexpected results. The high performance configuration results in higher cache misses than the default one. In [30, p. 19-20], the cache misses are close to zero percent. The problem could be anywhere on the sensor, but it is probably caused by the BIOS settings as they led to a higher cache miss percentage. This problem should be identified and corrected to fully take advantage of the configuration.

It would be interesting to test the high performance in a real-world environment using *real-time traffic*. This will verify whether the sensor is able to deliver a lossless detection in Telenor's environment. The number of concurrent flows may affect the performance of Suricata as mentioned in section 3.2, which will be evident by testing with real-time traffic.

Another interesting experiment would be to run *concurrent* processes on the sensor alongside Suricata. This may affect the performance of Suricata as well as disclosing any undesirable behaviours. No concurrent processes are present during the tests in order to isolate the area of research.

Express data path (XDP) is a Suricata feature explained in [31] that would be interesting to implement. This would allow the sensor to bypass *elephant flows*, a term for extremely large continuous flows, at the lowest point in the software stack. Most elephant flows like video streaming does not need to be analyzed, and can be safely bypassed. XDP depends on a filter called eBPF which is used to let specific elephant flows through to the IDS. However, bypassing flows would affect the network statistics, which could be a problem for some security companies.

Bibliography

[1] T. W. Edgar, D. O. Manz, *Research Methods for Cyber Security*. Todd Green, 2017, ISBN: 9780128053492.

[2] J. Hruska, *L2 vs. L3 cache: What's the Difference?* [Online]. Available: https://www.extremetech.com/computing/55662-top-tip-difference-between-l2-and-l3-cache, (accessed: 24.05.2021).

[3] E. Lai, *Direct memory access*. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/direct-memory-access, (accessed: 24.05.2021).

[4] Intel, *Intel I/O Acceleration Technology*. [Online]. Available: https://www.intel.com/content/www/us/en/wireless-network/accel-technology.html, (accessed: 24.05.2021).

[5] H. El-Rewini, M. Abd-El-Barr, "Advanced computer architecture and parallel processing," in. Wiley-Interscience, 2005, pp. 77–102.

[6] F. Denneman, *NUMA Deep Dive Part 1: From UMA to NUMA*. [Online]. Available: https://frankdenneman.nl/2016/07/07/numa-deep-dive-part-1-uma-numa/, (accessed: 03.11.2020).

[7] C. Lameter, "*An overview of non-uniform memory access*," New York: ACM, 2013. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2508834.2513149.

[8] N. Manchanda, K. Anand, *Non-Uniform Memory Access (NUMA)*. [Online]. Available: https://web.archive.org/web/20131228092942/http://www.cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf, (accessed: 03.11.2020).

[9] R. Bharadwaj, *Mastering Linux Kernel Development: A Kernel Developer's Reference Manual*. Packt Publishing, 2017, ISBN: 9781785883057. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=1615018&site=ehost-live.

[10] Red Hat, *Receive-Side Scaling (RSS)*. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rss, (accessed: 11.03.2021).

[11] OISF, *Packet Capture*. [Online]. Available: https://suricata.readthedocs.io/en/latest/performance/packet-capture.html, (accessed: 28.05.2021).

[12] S. Woo, K. Park, *Scalable TCP Session Monitoring with Symmetric Receive-side Scaling*. [Online]. Available: https://www.ndsl.kaist.edu/~kyoungsoo/papers/TR-symRSS.pdf, (accessed: 13.03.2021).

[13] OISF, *Suricata Home*. [Online]. Available: https://suricata-ids.org/, (accessed: 02.06.2021).

[14] ——, *Suricata Features*. [Online]. Available: https://suricata-ids.org/features/, (accessed: 02.06.2021).

[15] ——, *Rules Format*. [Online]. Available: https://suricata.readthedocs.io/en/suricata-6.0.1/rules/intro.html, (accessed: 30.03.2021).

[16] ——, *Runmodes*. [Online]. Available: https://suricata.readthedocs.io/en/suricata-6.0.1/performance/runmodes.html?highlight=runmodes#different-runmodes, (accessed: 27.05.2021).

[17] Red Hat, *IRQBALANCE.* [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/sect-red_hat_enterprise_linux-performance_tuning_guide-tool_reference-irqbalance, (accessed: 15.03.2021).

[18] OISF, *High Performance Configuration.* [Online]. Available: https://suricata.readthedocs.io/en/suricata-6.0.2/performance/high-performance-config.html?highlight=cpu%5C%20scheduling#cpu-affinity-and-numa, (accessed: 15.03.2021).

[19] Intel Corporation, *i40e Linux Base Driver for the Intel Ethernet Controller 700 Series.* [Online]. Available: https://downloadmirror.intel.com/24411/eng/readme.txt, (accessed: 15.03.2021).

[20] Red Hat, *What's IT automation?* [Online]. Available: https://www.redhat.com/en/topics/automation/whats-it-automation, (accessed: 16.03.2021).

[21] ——, *What is Infrastructure as Code (IaC)?* [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac, (accessed: 27.05.2021).

[22] ——, *How Ansible Works.* [Online]. Available: https://www.ansible.com/overview/how-ansible-works, (accessed: 16.03.2021).

[23] ——, *Ansible Documentation.* [Online]. Available: https://docs.ansible.com/ansible/latest/index.html, (accessed: 16.03.2021).

[24] ——, *Roles.* [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html#embedding-modules-and-plugins-in-roles, (accessed: 18.03.2021).

[25] Cisco, *TRex Low-Cost, High-Speed Stateful Traffic Generator.* [Online]. Available: https://github.com/cisco-system-traffic-generator/trex-core, (accessed: 24.05.2021).

[26] ——, *trex_manual.* [Online]. Available: https://trex-tgn.cisco.com/trex/doc/trex_manual.html, (accessed: 24.05.2021).

[27] ——, *trex_preso.* [Online]. Available: https://trex-tgn.cisco.com/trex/doc/trex_preso.html, (accessed: 24.05.2021).

[28] J. Johnson, *Reproducible Performance Testing of Suricata on a Budget Using TRex.* [Online]. Available: https://suricon.net/wp-content/uploads/2019/01/SuriCon2018_Johnson.pdf, (accessed: 24.05.2021).

[29] K. Jakimoski, N. V. Singhai, "Improvement of hardware firewall's data rates by optimizing suricata performances," *27th Telecommunications forum TELFOR 2019*, 2019.

[30] P. Manev, M. Purzynski, *Suricata Extreme Performance Tuning.* [Online]. Available: https://github.com/pevma/SEPTun, (accessed: 06.04.2021).

[31] ——, *SEPTun Mark II.* [Online]. Available: https://github.com/pevma/SEPTun-Mark-II, (accessed: 06.04.2021).

[32] Q. Hu, M. Rizwan Asghar, N. Brownlee, "Evaluating network intrusion detection systems for high-speed networks," *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, 2017.

[33] M. Jamshed, J. Lee, S. Moon, "*Kargus: A Highly Scalable Software-Based Intrusion Detection System*," *CCS '12: Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[34] V. Henriksen, R. Førde, *Ansible_numa.* [Online]. Available: https://github.com/vegah16/Ansible_numa, (accessed: 02.06.2021).

Appendix A

System Configuration

In order to optimize Suricata and the underlying system, SEPTun [30] (Suricata Extreme Performance Tuning) and SEPTun Mark II [31] will be used as guides. These guides are written by P. Manev from Suricata Core Team and M. Purzynski from Mozilla Threat Management.

A.1 BIOS Settings

The grub file needs to be modified in order to isolate the CPUs reserved for Suricata, disable Intel IOMMU and to set the max c-state to 3. In `/etc/default/grub`, we modified the line:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
```

to:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash isolcpus=1-10 intel_iommu=off ...
    intel_idle.max_cstate=3"
```

In the BIOS, we did the following:

- Disable
  - HW prefetcher
  - Adjacent sector prefetch
  - DCU stream prefetcher
- Enable
  - DCA
  - C-states and P-states
- Change operation mode from Maximum Performance to Custom

A.2 Linux Settings

A.2.1 Disable IRQbalance

IRQbalance is a Linux service used to balance interrupts among cores. Because interrupt affinities are carefully set up, we disabled the service like so:

```
$ systemctl stop irqbalance
$ systemctl disable irqbalance
```

A.2.2 Compile and install Suricata

As the system runs a recent version of Ubuntu, the official PPA (Personal Package Archive) can be used to install Suricata after updating the index. A command-line JSON processor called *jq* is also installed as it can be used to better display Suricata's EVE JSON output.

```
$ sudo add-apt-repository ppa:oisf/suricata-stable
$ sudo apt update
$ sudo apt install suricata jq
```

A.2.3 NIC driver and settings

In order to ensure that the latest i40e NIC driver on the system is installed, the i40e kernel module is removed and added. *Modprobe* is a command that will look for the correct driver in the module directory.

```
$ rmmod i40e && modprobe i40e
```

Then, a sequence of NIC settings are set. It is good practice to disable the NIC while setting some settings. The number of RSS queues is set to 15, which is equal to the number of Suricata worker threads. Receive hashing offload and RX ntuple filters and actions are enabled before activating the NIC again.

```
$ ifconfig eno1 down
$ ethtool -L eno1 combined 15
$ ethtool -K eno1 rxhash on
$ ethtool -K eno1 ntuple on
ifconfig eno1 up
```

The set_irq_affinity script is used to pin interrupts from a NIC to specific threads. The script is available in any Intel NIC source driver download. The number of RSS queues must match the number of pinned threads.

```
$ ./set_irq_affinity 1-15 eno1
```

Symmetric RSS can be enabled by using a low entropy toeplitz key. By doing this, AF_PACKET can be used with cluster_qm later on.

```
$ ethtool -X eno1 hkey 6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D...
   :5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A...
   :6D:5A:6D:5A:6D:5A:6D:5A equal 15
```

Receive and transmit offloading are disabled. Adaptive control is disabled for decreased latency. Since hardware interrupts only activates the NAPI processing loop so it starts processing packets, the maximum number of interrupts per second should be limited. The ring rx descriptor is set to 1024 in order for the DDIO to work better, which in turn should result in a lower L3 cache miss ratio.

```
$ ethtool -A eno1 rx off tx off
$ ethtool -C eno1 adaptive-rx off adaptive-tx off rx-usecs 125
$ ethtool -G eno1 rx 1024
```

Lastly, a bash script lets the NIC balance as much as possible.

```
$ for proto in tcp4 udp4 ah4 esp4 sctp4 tcp6 udp6 ah6 esp6 sctp6; do
    ethtool -N eno1 rx-flow-hash $proto sdfn
  done
```

A.3 Suricata Configuration

Suricata is configured in a file called *suricata.yaml*. This file is located in `/etc/suricata/suricata.yaml` by default. Below are the most important sections.

A.3.1 Af-packet

The af-packet section defines which NIC af-packet should use. It also defines that af-packet should use the *cluster_qm* cluster type as well as some other key parameters.

```
af-packet:
    - interface: ens2f0
      threads: 10
      cluster-id: 99
      cluster-type: cluster_qm
      defrag: no
      use-mmap: yes
      mmap-locked: yes
      tpacket-v3: yes
      ring-size: 200000
      block-size: 2097152

      - interface: default
```

A.3.2 Threading

The threading section defines the Suricata CPU affinity settings for management and worker threads.

```
threading:
    set-cpu-affinity: yes
    cpu-affinity:
        - management-cpu-set:
            cpu: ["0", "32"]
        - receive-cpu-set:
            cpu: ["0", "32"]
        - worker-cpu-set:
            cpu: ["1-10"]
            mode: "exclusive"
            prio:
                low: []
                medium: ["0", "32"]
                high: ["1-10"]
                default: "high"
    detect-thread-ratio: 1.0
```

A.3.3 Memory

The flow and stream sections define how much memory the system should allocate to Suricata in order to prevent resource starvation.

```
max-pending-packets: 65500

runmode: workers

flow:
  memcap: 14gb
  hash-size: 256072
  prealloc: 300000
  emergency-recovery: 30

stream:
  memcap: 12gb
  checksum-validation: no
  prealloc-sessions: 375000
  inline: auto
  bypass: yes
  reassembly:
    memcap: 20gb
    depth: 1mb
    toserver-chunk-size: 2560
    toclient-chunk-size: 2560
    randomize-chunk-size: yes
    segment-prealloc: 200000
```

Appendix B

Ansible

B.1 Common Role

There is only one role developed in Ansible, which is used for both configuring and testing the sensor. The system running Ansible is able to connect to both hosts, the sensor and the traffic generator, over SSH. The hosts are declared in the Ansible hosts file located in `/etc/ansible/hosts` by default like so:

```
[ubuntu]
user@IP

[trex]
user@IP
```

B.1.1 Default Variables

The following code declares the default variables used by the role. Variables declared in the playbook will override the defaults.

```
# monitor_if: "{{MONITOR_IF}}"
monitor_if: "ens2f0" # $ sudo lshw -class network -short

# Default configuration.
default: no

# Default memory allication and no local bypass.
default_memory: no

# High performance BIOS settings. Needs to be configured manually.
bios: yes

# NIC ring descriptor size
rx_ring: 1024 # Default 1024. 512 and 2048 yield worse results

```

```yaml
# All CPUs in a set, e.g. 0-63
cpu_set: "0-{{ansible_processor_vcpus - 1}}"

threading_set_cpu_affinity: 'yes'
threading_management_cpu: '0'
threading_receive_cpu: '{{threading_management_cpu}}' # This is not used for workers runmode, only autofp. For safe measure, we set equal to mgmt.
threading_worker_cpu: '"1-15", "33-47"'
threading_worker_cpu_plain: "{{threading_worker_cpu|regex_replace('[\x20\x22]', '')}}"
threading_worker_prio_low: ''
threading_worker_prio_medium: '{{threading_management_cpu}}'
threading_worker_prio_high: '{{threading_worker_cpu}}'
threading_worker_prio_default: "high"

afpacket_interface: "{{monitor_if}}"
afpacket_threads_1: 15 # Threads in CPU group 1
afpacket_threads_2: "{{afpacket_threads_1}}" # Threads in CPU group 2
afpacket_clustertype: "cluster_qm" # To bind Suricata to RSS queues
afpacket_defrag: "no" # TODO: test yes/no, trenger kun å teste med beste test?
afpacket_usemmap: "yes"
afpacket_mmaplocked: "yes"
afpacket_tpacketv3: "yes"

# Ring-size is another af-packet variable that can be considered for tuning
# and performance benefits. It basically means the buffer size for packets
# per thread.
# Default : 2048
# SEPTun setting : 200000
# SEPTun 2/redipranha setting : 300000
afpacket_ringsize: 200000

# SEPTun/default setting : 1048576
# SEPTun 2/redipranha setting : 2097152
afpacket_blocksize: 2097152

# To deal with high traffic, we set the max-pending-packets to maximum value 65500.
# Should not need any testing, leave at 65500.
# Default is 1024
max_pending_packets: 65500

runmode: "workers" # default is autofp

# Related to memory and emergency mode. See redpiranha under "Suricata Config".
# All values
flow_memcap: "{{'128mb' if default_memory else '14gb'}}"
flow_hash_size: "{{'65536' if default_memory else '256072'}}"
flow_prealloc: "{{'10000' if default_memory else '300000'}}"
stream_memcap: "{{'64mb' if default_memory else '12gb'}}"
stream_prealloc_sessions: "{{'2000' if default_memory else '375000'}}"
```

```
# Inspection will be skipped when stream.reassembly.depth is reached for a
↪  particular flow.
stream_bypass: "{{'no' if default_memory else 'yes'}}"
stream_reassembly_memcap: "{{'256mb' if default_memory else '20gb'}}"
stream_reassembly_segment_prealloc: "{{'2048' if default_memory else
↪  '200000'}}"

# Testing
counters: "capture.kernel_packets,capture.kernel_drops,tcp.reassembly_gap" #
↪  Comma separated. Which counters to plot with suri-stats.
test_seconds: 600 # How long to test in seconds.
test_name: "test1" # Which test is it? Remember to change this in each test.
↪  Used to make result dir.
traffic_gb: 9 # max 10.
trex_profile: "avl/sfr_delay_10_1g.yaml" # Which traffic profile to use for
↪  TRrex.
```

B.1.2 Tasks

The following code declares the tasks to be executed by the role.

```
---

# To prevent entering the password when ansible executes.
- name: Set passwordless sudo
  lineinfile:
    path: /etc/sudoers
    state: present
    regexp: '^%sudo'
    line: '%sudo ALL=(ALL) NOPASSWD: ALL'
    validate: 'visudo -cf %s'


# Suricata worker CPUs needs to be isolated from the Linux kernel
↪  scheduler.
- name: Isolate CPUs in grub file
  block:

    # Add isolcpus boot parameter to grub file. PS: "isolcpus=" key with no
    ↪  value is ok.
    - name: Modify grub file
      template:
        src: grub.j2
        dest: /etc/default/grub
      notify:
        - Reboot to apply isolcpus

    # Required after grub file is modified.
    - name: Update grub
      command: update-grub

```

```yaml
    # Force all handlers that have been notified to run now.
    # Thus, reboot now of necessary.
    - name: Flush handlers
      meta: flush_handlers


- name: Install packages
  package:
    name:
      - suricata # IDS
      - jq  # Command-line JSON processor
      - ethtool # Query or control network driver and hardware settings
      - build-essential # For compiling Intel driver
      - linux-tools-generic # Installing Perf for testing
      - linux-tools-5.4.0-72-generic # Installing Perf for testing
      - sysstat # For CPU utilization output
      - gawk # GNU awk needed to monitor network_softnet_stat
    state: latest


# We make sure that the Intel i40e driver is present to ensure max
↪   performance.
- name: Install Intel i40e kernel module
  block:

    - name: Make sure directory exists
      file:
        path: /opt/i40e
        state: directory
      notify:
        - Install i40e

    - name: Unarchive driver stored locally
      unarchive:
        src: ../files/i40e-2.14.13.tar.gz
        dest: /opt/i40e/
        creates: i40e-2.14.13

    # Force all handlers that have been notified to run now.
    # Thus, install i40e now of necessary.
    - name: Flush handlers
      meta: flush_handlers

- name: Reload i40e driver to reset NIC counters
  block:
    - name: Remove driver
      community.general.modprobe:
        name: i40e
        state: absent

    - name: Add driver
      community.general.modprobe:
        name: i40e
        state: present



- name: Stop irqbalance for high performance optimisation
  service:
    name: irqbalance
    state: stopped
  when: not default


# The set_irq_affinity script is included with the Intel i40e driver
↪  download.
# Pins interrupts from the monitor interface to the Suricata worker
↪  threads.
- name: Set irq affinity
  command: "/opt/i40e/i40e-2.14.13/scripts/set_irq_affinity
  ↪  {{threading_worker_cpu_plain}} {{monitor_if}}" # "local" means NUMA
  ↪  0.
  when: not default


# The NIC (Monitor IF) needs to be configured in order for better
↪  performance,
# and to support symmetric RSS.
- name: Configure NIC
  ignore_errors: yes
  when: not default
  block:

    - name: Disable NIC
      command: "ifconfig {{monitor_if}} down"

    - name: Change the number of RSS queues
      command: "ethtool -L {{monitor_if}} combined {{(afpacket_threads_1 +
      ↪  afpacket_threads_2) | int}}"

    - name: Enable receive hashing offload
      command: "ethtool -K {{monitor_if}} rxhash on"

    - name: Enable Rx ntuple filters and actions
      command: "ethtool -K {{monitor_if}} ntuple on"

    - name: Enable NIC
      command: "ifconfig {{monitor_if}} up"

    - name: Allow symmetric hashing by using low entropy toeplitz key
      command: "ethtool -X {{monitor_if}} hkey
      ↪  6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:5A:6D:
      ↪  equal {{(afpacket_threads_1 + afpacket_threads_2) | int}}"

    - name: Disable RX/TX pause.
      command: "ethtool -A {{monitor_if}} rx off tx off"

    - name: Adaptive control is disabled for lowest possible latency
      command: "ethtool -C {{monitor_if}} adaptive-rx off adaptive-tx off
       ↪  rx-usecs 125" # TODO: test e.g. 150

    - name: Changes the number of ring entries for the Rx ring. # TODO: test
      command: "ethtool -G {{monitor_if}} rx {{rx_ring}}" # buffer
       ↪  størrelse, finnes en maks størrelse men kan teste med mindre

    - name: Make sure the RSS hash function is Toeplitz.
      command: "ethtool -X {{monitor_if}} hfunc toeplitz"

    - name: Let the NIC balance as much as possible # TODO: test sdfn vs sd
      shell: |
        for proto in tcp4 udp4 tcp6 udp6; do
          ethtool -N {{monitor_if}} rx-flow-hash $proto sdfn
        done


# All tasks related to Suricata in this block.
- name: Suricata stuff
  block:

    # This file has no append parameter in suricata.yaml. Manually deletes
     ↪  file. New file will be generated by Suircata.
    - name: Clean suricata.log
      tags:
          - testing
      file:
        path: /var/log/suricata/suricata.log
        state: absent

    # This file has no append parameter in suricata.yaml. Manually deletes
     ↪  file. New file will be generated by Suircata.
    - name: Clean eve.json
      tags:
          - testing
      file:
        path: /var/log/suricata/eve.json
        state: absent

    # Uses the template to configure Suricata by inserting variables.
    - name: Configure suricata
      tags:
        - configure
      template:
        src: suricata.yaml.j2
        dest: /etc/suricata/suricata.yaml

    - name: Stop Suricata
      command: systemctl stop suricata.service

    - name: Start Suricata
```

```yaml
      command: systemctl start suricata.service


# ======================== Testing ======================== #


- name: Testing
  ignore_errors: yes
  tags:
    - testing
  block:

    # Suricata needs to start up before testing can begin.
    - name: Pause for Suricata initialization
      pause:
        seconds: 30

    # This task only runs on Trex machine.
    - name: Start Trex traffic generator
      shell: "./t-rex-64 -f {{trex_profile}} -c 4 -m {{traffic_gb}} -d {{(test_seconds - 10) | int}}"
      args:
        chdir: /opt/trex/v2.88
      async: "{{test_seconds * 2 | int}}" # Don't wait for result. Async value longer than command runtime.
      poll: 0 # Don't wait for result
      delegate_to: "{{groups['trex'][0]}}"
      tags:
        - trex

    # Example directory path for test1:
    # results/test1/2021-04-26T08:53:00Z/
    - name: Create timestamped dir for storing results
      tags:
          - test
      file:
        path: "{{role_path}}/../../results/{{test_name}}/{{ansible_date_time.iso8601}}"
        state: directory
        mode: "0755"
      register: newdir_res # Use this fact to store results in the right dir.
      delegate_to: localhost
      become: false

    - name: Copy softnet-stat.pl script to remote
      copy:
        src: softnet-stat.pl
        dest: /usr/local/sbin/softnet-stat.pl
        mode: '0755'

    # All Linux commands we use for logging stats are stated in the loop parameter.
    - name: Start commands used for logging stats
      shell: "{{ item }}"
      args:
        executable: /bin/bash
      async: "{{test_seconds * 2 | int}}" # Don't wait for result. Async value longer than command runtime.
      poll: 0 # Don't wait for result
      register: output
      loop:

        # Start mpstat for CPU utilization log
        - "mpstat -P {{cpu_set}} -N 0,1 10 {{(test_seconds / 10) | int}}"

        # Start mpstat for CPU interrupts log
        - "mpstat -P {{cpu_set}} -I SCPU -I SUM 10 {{(test_seconds / 10) | int}}"

        # Start perf for cache miss stats. PS: --log-fd 1 means to output to stdout and not stderr.
        - "perf stat --log-fd 1 -I 10000 -e LLC-loads,LLC-load-misses,LLC-stores,LLC-prefetches -C {{cpu_set}} sleep {{test_seconds}}"

        # Start pidstat for monitoring Suricata tasks
        - "pidstat -tuw -C 'Suricata|{{monitor_if}}|FM#|FR#|CW|CS|US' 10 {{(test_seconds / 10) | int}}"

        # Start ethtool for monitoring card's FIFO buffer drops/losses.
        - "for i in {1..{{(test_seconds / 10) | int}}}; do date; ethtool -S {{monitor_if}} | grep -E 'rx_dropped|rx_missed|rx_packets|errors'; echo ''; sleep 10; done"

        # Start softnet-stat.pl for monitoring packet drops at the softirq layer.
        - "perl /usr/local/sbin/softnet-stat.pl --count {{(test_seconds / 10) | int}} --sec 10"

    # Ansible will wait for the set time in order for Suricata to process some data.
    - name: Pause for testing
      pause:
        seconds: "{{test_seconds}}"

    # Because of some problems with printing the end of suricata.log,
    # we have to kill the service before stopping it.
    - name: Kill Suricata
      command: systemctl kill suricata.service

    # Wait for Suricata to summarize the run.
    - name: Pause for Suricata to shut down
      pause:
        seconds: 120

    # Stop Suricata.
    - name: Stop Suricata
      command: systemctl stop suricata.service

    # We have to wait for async tasks to finish in order to get the
    ↪  outputs.
    - name: Wait for async tasks to finish
      async_status:
        jid: "{{ item.ansible_job_id }}"
      register: output_results
      until: output_results.finished
      retries: 100
      delay: 10
      with_items: "{{ output.results }}"

    # All async tasks will create a log file with their command as the
    ↪  name.
    - name: Fetch command output logs
      copy:
        content: "{{item.stdout}}"
        dest: "{{newdir_res.path}}/{{item.cmd|regex_replace('[\x2F]',
        ↪  '-')}}.log"
      delegate_to: localhost
      become: no
      with_items: "{{output_results.results}}"
      no_log: True
      # debugger: always

    # suri-stats is a python module used to generate plots from the
    ↪  stats.log file.
    # The "counters" variable can be changed to plot different stats.
    - name: Generate suri-stats plot
      command: "suri-stats -p -c {{counters}} -S -o /tmp/plot.png
       ↪  /var/log/suricata/stats.log"

    - name: Fetch plot
      fetch:
        src: /tmp/plot.png
        dest: "{{newdir_res.path}}/plot.png"
        flat: yes # Override file path name

    - name: Fetch suricata.log
      fetch:
        src: /var/log/suricata/suricata.log
        dest: "{{newdir_res.path}}/suricata.log"
        flat: yes # Override file path name

    - name: Fetch stats.log
      fetch:
        src: /var/log/suricata/stats.log
        dest: "{{newdir_res.path}}/stats.log"
        flat: yes # Override file path name
```

```yaml

    - name: Fetch suricata.yaml
      fetch:
        src: /etc/suricata/suricata.yaml
        dest: "{{newdir_res.path}}/suricata.yaml"
        flat: yes # Override file path name

    - name: Fetch eve.json
      tags:
        - test
      fetch:
        src: /var/log/suricata/eve.json
        dest: "{{newdir_res.path}}/eve.json"
        flat: yes # Override file path name

    # -T for timestamp
    - name: Log dmesg
      block:

        - name: Register dmesg
          command: "dmesg -T"
          register: dmesg

        - name: Fetch dmesg
          copy:
            content: "{{dmesg.stdout}}"
            dest: "{{newdir_res.path}}/dmesg.log"
          delegate_to: localhost
          become: no

    - name: Dump all variables
      template:
        src: dumpall.j2
        dest: "{{newdir_res.path}}/dumpall.json"
      delegate_to: localhost
      become: false
```

B.1.3 Handlers

The following code declares the handlers used by the role. When a task notifies a handler, it will run at the end of the play.

```yaml
# Tasks in this file only runs when other tasks notify them.

# The Intel i40e driver is installed and added as a kernel module.
# Only when the driver is not present.
- name: Install i40e
  block:
    - name: Compile driver
      community.general.make:
        chdir: /opt/i40e/i40e-2.14.13/src
        target: install

    - name: Add the i40e module
      community.general.modprobe:
        name: i40e
        state: present


# Reboot is required in order to set the boot parameter from grub.
# We only reboot when the grub file is modified.
- name: Reboot to apply isolcpus
  reboot:
```

B.1.4 Suricata High Performance Playbook

The following code declares the playbook used to configure and test Suricata in a high performance configuration using 10 worker threads.

```
---

- name: cpu_10
  hosts: ubuntu
  become: yes
  roles:
    - role: ../roles/common
      vars:

        test_name: "cpu_10" # Which test is it? Remember to change this in
        ↪   each test. Used to make result dir.

        threading_management_cpu: '"0", "32"'
        threading_worker_cpu: '"1-10"'

        afpacket_threads_1: 10 # Threads in CPU group 1
        afpacket_threads_2: 0 # Threads in CPU group 2
```