CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Improving Intrusion Detection for IoT Networks

A Snort GPGPU Modification Using OpenCL

Master's thesis in Computer Systems and Networks

LINUS JOHANSSON
OSKAR OLSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

MASTER'S THESIS 2018

Improving Intrusion Detection for IoT Networks

A Snort GPGPU Modification Using OpenCL

LINUS JOHANSSON
OSKAR OLSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Improving Intrusion Detection for IoT Networks
A Snort GPGPU Modification Using OpenCL
LINUS JOHANSSON
OSKAR OLSSON

Supervisor: Magnus Almgren, Department of Computer Science and Engineering
Supervisor: Charalampos Stylianopoulos, Department of Computer Science and Engineering
Examiner: Olaf Landsiedel, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Improving Intrusion Detection for IoT Networks
A Snort GPGPU Modification Using OpenCL
Linus Johansson
Oskar Olsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Internet of Things, or IoT, is continuously growing with more devices being connected every day, adding new features and functionality to our personal and home devices by connecting them to the Internet. However, with the increase of devices and components, new security threats arise in previously offline systems that used to be immune to network attacks. This increase calls for better security options that can ensure protection of the data flowing in IoT networks by detecting and mitigating new threats. To contribute to the mentioned area, the goal of this thesis was to develop and evaluate a modified version of Snort, a widely used intrusion detection system. The idea was to improve the efficiency of computationally expensive pattern matching by extending Snort to use a graphical processing unit for such work. The Snort modification was tested by comparing it to that of an unmodified version of Snort in a closed environment with simulated network traffic. The tests were run on a single-board-computer to simulate the IoT context. The results show that the new functionality yields a speedup of 1.3 when analyzing captured traffic, a throughput increase of a factor of two when inspecting live traffic, and slightly less energy consumption, all when comparing to original Snort. With these results, it seems plausible to use the IoT devices as a means of strengthening their own security and protect them from network attacks.

Keywords: Computer, security, intrusion detection, pattern matching.

Acknowledgements

We would like to thank our two supervisors, Magnus Almgren and Charalampos Stylianopoulos, for guiding us throughout this project. Whenever we needed someone to bounce ideas off or get additional insight into the challenges we faced, they were always available to help us out. We also appreciate the thorough feedback on this thesis report that they have given us.

We would also like to thank our examiner Olaf Landsiedel as well as Marina Papatriantafilou for bringing an outside perspective and feedback on this report and the presentations of the work behind it.

Finally, we would like to thank our friends and families for the nonstop support during our work. This thesis would have not been possible without all of you!

Linus Johansson, Gothenburg, June 2018
Oskar Olsson, Gothenburg, June 2018

Contents

Acronyms **xi**

List of Figures **xiii**

1 Introduction **1**
- 1.1 Background and motivation . . . 1
- 1.2 Problem formulation . . . 2
- 1.3 Scope and limitations . . . 3
- 1.4 Challenges . . . 3
- 1.5 Disposition . . . 4

2 Technical Background **5**
- 2.1 Intrusion Detection Systems . . . 5
- 2.2 Snort . . . 7
- 2.3 Pattern matching . . . 7
  - 2.3.1 Early Snort - Boyer-Moore algorithm . . . 8
  - 2.3.2 Updated Snort - Aho-Corasick algorithm . . . 9
- 2.4 GPGPU and processor architecture . . . 11
  - 2.4.1 Aho-Corasick on GPU . . . 12
  - 2.4.2 PFAC - modified Aho-Corasick for GPU . . . 13
- 2.5 OpenCL . . . 13

3 Related Work **15**
- 3.1 Intrusion Detection Systems using GPGPU . . . 15
- 3.2 Portable IDS with Snort and Raspberry Pi . . . 17
- 3.3 Intrusion Detection System for embedded devices . . . 18
- 3.4 Our contribution . . . 19

4 Design **21**
- 4.1 Deciding the IDS . . . 21
- 4.2 Snort design . . . 21
- 4.3 Our model . . . 23
  - 4.3.1 Design tradeoffs . . . 24
- 4.4 Hardware . . . 25

5 Implementation **27**
- 5.1 Setup of the state machine . . . 27

5.2 Single packet to GPU 28
5.3 Pattern matching on GPU 30
5.4 Results to CPU 31
5.5 Multiple packets to the GPU 32
5.6 Gather results to CPU 33
5.7 Improving multiple packet solution 33

6 Evaluation **37**
6.1 Reference systems 37
6.2 Datasets 38
6.3 Reading pcap files 38
6.3.1 Single versus double buffer 39
6.3.2 GPU versus CPU 40
6.3.3 GPU optimal setting 42
6.3.4 Ruleset variations 43
6.4 Tcpreplay 45
6.5 Power consumption 48

7 Discussion **53**
7.1 Results 53
7.2 Tcpreplay issues 55
7.3 Keeping Snort functional 56
7.4 Security in GPUs 58
7.5 Ethics and sustainability 58
7.6 Future work 60

8 Conclusion **61**

Bibliography **63**

Acronyms

API	Application Programming Interface.
CCTV	Closed-Circuit Televisions.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
DDoS	Distributed Denial of Service.
DPI	Deep Packet Inspection.
GDPR	General Data Protection Regulation.
GPGPU	General-Purpose Computing on Graphics Processing Units.
GPU	Graphical Processing Unit.
IDPS	Intrusion Detection and Prevention System.
IDS	Intrusion Detection System.
IoT	Internet of Things.
IPS	Intrusion Prevention System.
NIDS	Network Intrusion Detection System.
OpenCL	Open Computing Language.
RAM	Random-Access Memory.
SBC	Single-Board Computer.
WSN	Wireless Sensor Network.

List of Figures

2.1 Simple state machine searching for pattern *XYZ* 10
2.2 Example image of a state machine in Snort looking for the patterns *YZX*, *XYZ*, and *WZ*. Output states are marked with an extra container. 11

4.1 Basic work flow of an analyzing thread in Snort as shown in the user manual [1]. 22
4.2 The execution flow of the modified detection step. It shows an overall picture on how the data is translated from the CPU and processed on the GPU. 24

5.1 How the two-dimensional array is serialized into a single array. 29
5.2 The execution flow when using two buffers to store data leading to higher utilization of the CPU. 34

6.1 Experiment 1: Execution time for the two GPU versions, one utilizing a single buffer and the other using two. 40
6.2 Experiment 2: Execution time for each of the Snort versions together with the dataset that was searched. 41
6.3 Experiment 3: Execution time for each of the Snort versions searching all the capture files from ISCX2012 12 of June. 42
6.4 Data from table 6.4 grouped by the buffer size for all thread amounts. 43
6.5 Execution time for Snort GPU and Snort ORG using the different rulesets for different datasets. 45
6.6 The percentage of unhandled packets, including both the disappeared as well as the dropped packets. 47
6.7 The percentage of dropped packets from those that were actually caught by the Snort version. 48
6.8 Power consumption measurements of the CPUs, GPU and RAM. . . 49
6.9 The combined power consumption of all four reference systems. . . . 50

1

Introduction

This chapter begins by first introducing the background and the motivations behind the thesis. After this, the main research questions are stated in the problem formulation. The chapter also contains the scope, the challenges of the project as well as a disposition of the rest of the thesis.

1.1 Background and motivation

The Internet of Things (IoT) is growing [2] and new devices are constantly being connected to the Internet to open up for and provide new features. Wireless Sensor Networks (WSNs), Single-Board Computers and Internet protocol (IP) cameras are all examples of products that are already on the market. Basically, having everything connected gives us a better understanding of how we live and how we use our available resources such as power consumption. Such knowledge might result in benefits to society as a whole. It would for instance be possible to predict when electrical power will be needed and when there is a surplus by using live monitoring of the power consumption and production.

The application area for IoT is enormous, and they are already being used in cars [3] and surveillance cameras [4], but also in more sensitive devices like medical equipment [5]. However, with this new functionality, new security threats arise that has not previously been encountered in such systems. One such vulnerability was discovered in 2016, when a malware called *Mirai* was found, which mainly focused on hijacking IoT devices in form of video surveillance cameras like closed-circuit televisions (CCTVs) and home routers [4]. In two weeks *Mirai* managed to hijack over 200 000 devices that were later used in so called botnets to perform Distributed Denial of Service (DDoS) attacks [4]. This is just one example of what could happen if IoT devices have insufficient security mechanisms. Imagine if the security breach instead was in some sort medical equipment, then the outcome of the attack could have been even more critical.

There are numerous different security alternatives on the market to protect desktops and laptops against malicious attacks and malformed packets. One such solution is an Intrusion Detection System (IDS), a real-time software tool that monitors a system by continuously scanning it, trying to identify potential malicious intrusions. IDSs that are deployed to inspect network packets are called Network Intrusion Detection System (NIDS). NIDSs are in comparison to common firewalls able to perform a deep level inspection of incoming network packets [6]. This means that NIDSs are not only examining the headers of the packets, but they are also able to examine the payload. This is one of the features that makes NIDSs a powerful tool when it comes to protecting computer systems, networks and connected devices. A more detailed description of Intrusion Detection Systems is given in section 2.1.

Even if these cyber security mechanisms exist, IDSs mainly developed for desktops and laptops may not be portable to fit resource limited IoT devices. One of the biggest issues is that IoT devices tend to have significantly less memory capacity than a regular desktop computer [7]. Besides limited memory, IoT devices are also very limited in terms of computational power and central processing unit (CPU) resources. This means that in order to protect these devices, an IDS has to be resource efficient. If this was not the case, the device would not be able to run anything else (maybe not even the IDS) and thus render the device useless.

However, one type of IoT device where recent research have managed to deploy Snort, a well known open-source IDS, mainly developed for desktops, is the Raspberry Pi which is a well known Single-Board Computer (SBC) [8]. Even though the research is showing that it is possible to run a widely used desktop IDS on a resource limited single circuit board computer, it also shows that the IDS is very resource demanding. This means that if the IDS was more optimized for the environment, it could for instance be installed on an SBC where the main purpose of the device is not just running it as an IDS. Even if the only purpose of the SBC is to act as an IDS, optimizing the IDS for the environment would increase the throughput, and thus increase the overall performance of the IDS.

One IDS optimization method that has been proven successful on desktops is to use general-purpose computing on graphics processing units (GPGPU) [9, 10]. The GPGPU technique is further described in section 2.4, but in short, it means using the graphic processing unit (GPU) for other purposes than producing computer graphics. In this context, the GPU is being used to process the pattern matching of the IDS. Results shows that by using this technique, it is possible to increase the performance significantly in terms of throughput. Nowadays, many Single-Board Computers are equipped with GPUs and has been used in other works to improve pattern matching. However, to the knowledge of this thesis the GPGPU technique has not been used to optimize an entire IDS deployed on an SBC.

1.2 Problem formulation

This thesis aims to extend the current research of Intrusion Detection Systems by investigating optimization alternatives for IDSs deployed on Single-Board Computers.

The main focus will be to examine if the proven desktop optimization technique of using GPGPU [9, 10] leads to any performance improvement on Single-Board Computers. To summarize, the research questions that this thesis aims to answer are the following:

1. Can GPGPU be used to offload the CPU and improve performance of an IDS deployed on a Single-Board Computer?

2. How well can Single-Board Computers run an optimized IDS?

These are the main questions that will be investigated in this thesis. The goals are to present an IDS optimized for Single-Board Computers or similarly devices with embedded computers and thereby supplying one additional alternative for security and protection in IoT networks.

1.3 Scope and limitations

The aim of this thesis is not to present a brand new Intrusion Detection System created from scratch. As the focus is on the optimization using GPGPU, the methods are applied to an already existing open-source IDS. The specific Intrusion Detection System used in this thesis is Snort Version 3 alpha 4. This is a Unix based version of Snort, which means that the optimized IDS presented is only applicable to Unix based devices. Moreover, the main optimization method investigated involves using the GPGPU technique, thus only Single-Board Computers equipped with GPUs are considered.

Even though there might be more parts of Snort that could be parallelized and executed on the GPU, the primary focus in this thesis is to move the execution of the pattern matching to the GPU. Further, Snort 3 uses the well known multi-pattern search algorithm Aho-Corasick [11, 12] and therefore this is the primary algorithm investigated in this thesis. In Snort version 3, there are also features that categorize packets into application specific groups. This feature gives the user of Snort the ability to write application specific rules, but it is not considered nor used in this thesis, since it both complicates the structure and creation of the state machines as well as the usage of the pattern matching algorithm significantly.

1.4 Challenges

There are certain obstacles that must be overcome in order to develop the modified IDS for IoT networks that this thesis aims for. Changing software and adding new features is something that is usually managed by entire development teams who are already familiar with the software in question. The challenges of the thesis will be based on how GPGPU can be implemented into an already working IDS and running it on an IoT device.

For example, it was previously presented that this thesis will be based upon Snort 3, and any modification we make in Snort will affect the existing functionality as well as the resources used. The resources used should be kept to a minimum to allow execution of other processes as well, not just the IDS as this would be pointless.

Also, Snort must be modified to enable GPGPU by introducing an application programming interface that provides functions for work on the GPU in order to offload the processing on the CPU. This can also affect the usage of resources on the device, but another challenge here is how it can be implemented in Snort while preserving the original functionality that the software provides.

1.5 Disposition

The report is structured in the following way:

- Chapter 1 (Introduction): Introduces the background and motivation together with the problem formulation, scope and limitations of the thesis.
- Chapter 2 (Technical background): Detailed descriptions of techniques, algorithms and systems used in this thesis.
- Chapter 3 (Related work): Presents research from other scientific work on the subject and how this thesis relates to them.
- Chapter 4 (Design): Gives an overall view of the system design. It also presents the different design choices together with explanations of what and why certain decisions have been made.
- Chapter 5 (Implementation): Details and descriptions of data structures as well as explanations of the work flow of the developed software.
- Chapter 6 (Evaluation): Presents results from the tests, how the developed software was tested and which test cases that were used.
- Chapter 7 (Discussion): Analyzes, discusses and compare the results to similar work. It also discusses different ethical and sustainability aspects of the thesis.
- Chapter 8 (Conclusion): Presents the main findings of the thesis, what could be concluded from the results and suggestions of potential future work.

2

Technical Background

This chapter introduces some technical background necessary to understand this thesis. First, Intrusion Detection Systems are explained in detail followed by a section on GPGPU together with highlighting the differences between CPUs and GPUs. The third and fourth section explains more about the IDS Snort and pattern matching in the IDS and how it can be ported to the GPU. Lastly, there is a short introduction to OpenCL and the functionality it enables.

2.1 Intrusion Detection Systems

An Intrusion Detection System is a security software and Debar et al. [13] describe it as a detector that scans a system, in order to identify potential intrusion attempts and open vulnerabilities that the system may be exposed to. There are different types of intrusion detection systems on the market today and a common way to classify them is to divide them into what kind of system they are developed to protect [14]. The four major types of IDSs are: Host-based Intrusion Detection system (HIDS), Network-based Intrusion Detection System (NIDS), Wireless-based Intrusion Detection System (WIDS) and Network Behavior Analysis (NBA) [14]. Each of these work differently and for various areas but they are placed there for the same reason, to increase the security of the host or network in which the IDS is deployed.

Even though the detection techniques described in the next paragraphs could be used by various types of IDS. The main point of view in this section will be from a NIDS perspective, as this is the focus of this thesis.

Intrusion Detection Systems could also be further categorized into three main subgroups, signature-based, anomaly-based and stateful protocol analysis-based (SPA), depending on what detection method they use [14]. The characteristics for signature-based IDSs, which are also referred to as knowledge-based IDSs, are that they use stored information from already known vulnerabilities [13]. The information contains signatures, usually patterns or strings, that could be found in the payload of packets of a certain malicious attack [14]. In order to identify a potential intrusion, the IDS uses a pattern matching algorithm that scans the packets looking for these specific signatures. A further description of how pattern matching is done is given section 2.3. Popular signature-based NIDSs such as Snort, combine these signatures with rules in order to specify the characteristics of the known attack even further. A rule can for instance specify in what type of protocol the signature is found, the origin and the destination of the packet as well as what type of action that should be taken if a malicious packet is found [14].

The main advantage of knowledge-based IDSs is their precision [13]. This means that if an alarm is raised by the IDS, the probability that it is actually an intrusion and not a false alarm, or false positive, is very high. However, the difficulties of extracting key signatures from known attacks, and the maintenance work of always keeping the IDS up-to-date with the most recent vulnerabilities are often considered to be their biggest weaknesses [13]. Since knowledge-based IDSs depend on information and key signatures from already known attacks in order to discover an attack, they are unable to detect so called zero-day attacks. This means that they cannot detect an intrusion attempt if the approach of the attack is unforeseen and never been used before.

Instead of using signatures, anomaly-based IDSs, also known as behavior-based systems, observe and record the normal and expected behavior of a system [14, 13]. This means that the IDS is, under a period of time, trained to recognize the normal behavior of a system. When the training period is done, this behavior is stored into a reference model in the IDS. To identify an intrusion or attack, the IDS compares if there are any differences between the system's current execution and the recorded reference model. If any anomalies are spotted in the execution, the IDS will alert the system by sending out an alarm [13].

Anomaly-based IDSs are, in comparison to knowledge-based IDSs, able to detect zero-day attacks [15] and unforeseen vulnerabilities, since they are not dependent on any information from previous attacks. This is also often considered as one of their biggest advantages [14, 13]. However, a major drawback that anomaly-based IDSs in general tend to suffer from is the high rate of false alarms [14, 13]. According to Debar et al. [13] this problem often correlates with the difficulties of capturing the entire scope of the normal behavior during the creation of the reference model. Moreover, it is also possible that the normal behavior of a system changes over time, which means that the reference model also needs to be updated to not generate any false alarms when the system is fully functional [13].

Stateful protocol analysis-based IDSs are similar to anomaly-based IDSs in the sense that they observe the behavior of the system. They are however observing the behavior of specific protocols rather than the system as whole [15]. Another difference is that the reference model that the system's behavior is comparing against is developed by well known and established vendors [16] instead of a system recorded one. These reference models are usually based on protocol standards developed by organizations like IETF [14].

The main focus of Intrusion Detection Systems is to identify attacks and generate alarms if the target system is exposed to a potential intrusion. Nowadays many commercial IDSs like McAfee M-1450 and Cisco IPS-4240, but also open source-solutions like Snort, have the possibility to actively block packets if they are considered malicious [15]. In this way they are not only acting passively and alerting, but they are also preventing malicious content to reach the system. Intrusion Detection Systems with this feature are also known as Intrusion Prevention Systems (IPS) or Intrusion Detection and Prevention Systems (IDPS).

2.2 Snort

Snort is an open source network intrusion detection as well as prevention system. Developed in the late 90s and presented in 1999 by the original author Martin Roesch, Snort was a new alternative to the growing area of network- and computer security [17]. Initially, at its first release, Snort was a very lightweight IDS in the sense that it was designed to be easy to deploy and manage while requiring little of the system available resources to run. The purpose of Snort was to fill the security gap of smaller networks in which it was not an option to deploy commercial IDSs due to reasons such as affordability or the complexity of such systems.

For a multitude of reasons, such as the open source aspect and the flexibility of Snort, the popularity of the IDS has steadily increased with time. Today, Snort is a fully fledged IPS as mentioned earlier, with multiple versions that have extended the support and functions of the IDS. The latest iteration of Snort, called Snort 3 or Snort++, changed many aspects of the previous workflow of Snort. New features such as multiple packets inspectors in one Snort process, a new way to specify the configuration with the Lua programming language and better cross platform support are some of the changes that have been made.

The only major downside of Snort has been its limited throughput that might relate to the fact that Snort has been single threaded. Due to this, Snort has gotten some competition in the area from newer IDSs such as Suricata. However, Snort's throughput is not a problem for the smaller networks it was initially designed for, such as the ones in homes or similar endpoints of networks. Also, as stated previously, the new release of Snort++ supports multi-threaded processing which could be used to improve the throughput of Snort.

2.3 Pattern matching

The purpose of NIDSs are to inspect the traffic in the network by examining the incoming data for malicious content and log or alert the system administrator when found. In the extension of not only detecting but also preventing attacks, this includes taking actions such as blocking individual packets or even sources of transmission. In knowledge-based NIDSs the traffic that is allowed to pass, and what is rejected or acted upon, is decided based on predetermined rules and patterns.

Patterns are the specific texts in a rule that is searched for by NIDS. By extracting the payload data from a network packet it can be inspected by some string search algorithm to find all the occurrences of the patterns. This thesis will use the IDS Snort which has by default used two well known pattern matching algorithms throughout its existence. These are the Boyer-Moore [18] as well as the Aho-Corasick [12] algorithms and both of these, especially the currently used Aho-Corasick algorithm, will be described later in this section.

A problem with pattern matching is that while numerical values are very fast to compare, such as the IP addresses of a network packet, patterns are a bit more complicated. When comparing a pattern, each character contained in it has to be compared one by one in order to determine if it is a match or not. This becomes a huge workload when the entire point of the software is to analyze an enormous number of network packets. The data of each packet must be matched to hundreds or even thousands of patterns, based on the rules of the NIDS, where works have shown the actual pattern matching to take up to 70% of NIDS overall processing [19, 20].

Today there exist both single- and multi-pattern search algorithms. The two types differ in the way that multi-pattern search can search for and find multiple patterns in a single iteration of the input, while the single only checks for one pattern and has to parse the input multiple times for different patterns. Even though searching for multiple patterns at once sounds better, it also requires more processing to set up and therefore it is not always beneficial to use. In the context of NIDS, multi-pattern search will most likely be beneficial since the NIDS contains so many patterns to identify a wide range of attacks.

In Snort, rules with the *content* keyword can specify what patterns Snort will look for in network packets. An example of a simple rule could be:

alert tcp any any -> any 80 (msg:"FTP ROOT"; content:"USER root"; nocase;)

This rule would alert the user if a tcp packet was found containing the data "USER root" in the content and if it was headed for port 80 on the receiving host. The content part is what the pattern matching algorithm will have as one of the patterns to look for when examining the data of a packet. These rules could obviously be more complicated, with for example a longer content string. This leads to the question, how can patterns like these be detected?

2.3.1 Early Snort - Boyer-Moore algorithm

In the first release of Snort, the pattern matching algorithm used was the Boyer-Moore string searching algorithm [17]. This is a well known single-pattern search algorithm first presented in 1975 [18] and has through the years been used in a wide array of applications, including Snort and the grep function in GNU operating systems. The basic idea of the algorithm is to look for a pattern in a given text beginning from the end of the pattern to enable jumping through the text when the last character is incorrect. By doing some preprocessing, the algorithm can gather information and build a jump table used to do a semi-linear parsing of the text.

The Boyer-Moore algorithm iterates over an input text and looks for a pattern by searching the pattern's last character. If it finds the pattern's last character, then the algorithm will backtrack the input to see if the other characters in front of the current are matching as well which could result in a full match. Otherwise the algorithm will use the so-called jump table to skip forward in the input based on the current character since this affects when the pattern could exist in the input text again. For example, if the current character is not part of the pattern, then the algorithm can perform a jump equal to length of the pattern, since no match could have been found at least until that point.

This algorithm works well for longer patterns as the jumping can skip large parts of the input and thus the algorithm's performance is less than linear time which is very efficient. This might also suit the lightweight aspect of Snort's first release that might have been designed for a couple of rules containing content to be searched for. However, with more and more attacks existing in the area of internet security, more rules would be required to keep a good level of protection which would lead to more and more patterns. The problem with more patterns is that Boyer-Moore only look for one pattern at a time. Checking for multiple patterns would require the algorithm to be run once for every pattern. This might be the reason why Snort 2 added the Aho-Corasick multi-pattern search algorithm, which is described in the next section.

2.3.2 Updated Snort - Aho-Corasick algorithm

In Snort 3, just as in Snort 2, the pattern matching is performed by the Aho-Corasick multi-pattern matching algorithm [12]. Compared to the previously described Boyer-Moore algorithm, the Aho-Corasick takes more preprocessing to enable the pattern matching. The reason for this is that the algorithm builds a finite state machine with the purpose of traversing the states to keep track of multiple pattern chains at once. Even though the initial setup takes longer, it is just a one time cost and enables the algorithm to search for multiple patterns at once throughout the process lifetime, something the Boyer-Moore algorithm can not. When the number of rules of a NIDS increases, which in turn gives more patterns to look for, searching for multiple patterns at once is preferable, if not even a requirement.

Aho-Corasick is based on state machines which is a fairly simple mathematical model that has a certain set of states, and based on input events the algorithm traverses between these states. Each character of a word or pattern will be represented as a transition from a current state to a next state. Note that the transition depends on the current state and the next input, therefore it is possible that two different states can lead to different next states, even when the input is the same. This is pretty obvious when given some thought but consider the following example. *Car* and *Bar* are two entirely different words that only differ in the first character. Thus the first character of each pattern would take the algorithm to different states and so the next state will be different as well even though the rest of the text is identical.

The generic pattern *xyz* would produce a state machine with four states and three transitions where each transition relates to a character. The algorithm always starts in a default initial state and if an x is read from this state, then the algorithm would do a transition to the next state. If a y is the next input in the current state, then the machine would go to the third. Finally, seeing an z leads to the last state and thus the machine would know that the word *xyz* is matched. If any other character would be the input in the previous steps, then the state machine would transition back to the initial state and search again. The state machine produced by the single pattern *xyz* can be seen in figure 2.1 which consists of the initial state and three additional ones for the character transitions.

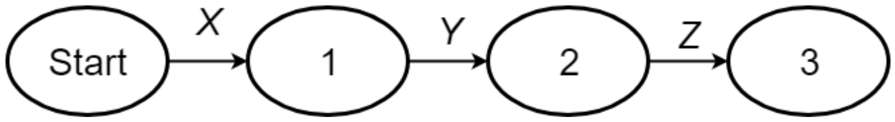


Figure 2.1: Simple state machine searching for pattern *XYZ*

A nice additional feature of the Aho-Corasick state machine is the introduction of failure transitions. As was just mentioned, an incorrect character could mean that the machine would have to go back to the initial state and search again, but in practice this is seldom the case, at least when searching for many patterns. Returning to the initial state can be seen as the default failure transition, but it can be used more effectively than this. If the previously seen input was correct up to a certain point in relation to some pattern, then the algorithm could return to this state instead of restarting from the initial state. The purpose of these transitions is to save time by skipping unnecessary computation, just like the jumping in Boyer-Moore that was explained in section 2.3.1.

For a simple example, consider the previous example of the state machine searching for *xyz*. Imagine that the state machine is in state two where it had just seen a character y, and then that the next input is a character x. Obviously, the state machine would not continue along the *xyz* pattern since it would require the next character to be z. In this case a failure transition could be performed to an earlier state, so instead of returning to the initial state, the state machine would go to the first. Since the first character in the *xyz* pattern is the character x, same as the next input, the transition would happen directly to the first state instead and thus bypassing the initial state.

Two additional examples of failure transitions can be seen as the dotted arrows in figure 2.2 that shows how the state machine can continue to another chain when the current search is broken. A failure transition is not handled differently from any other transition so the state machine can continue searching for patterns from the this state as normal. When matching longer texts for more patterns it becomes very beneficial to have failure transitions to minimize the amount for transitions of the algorithm.

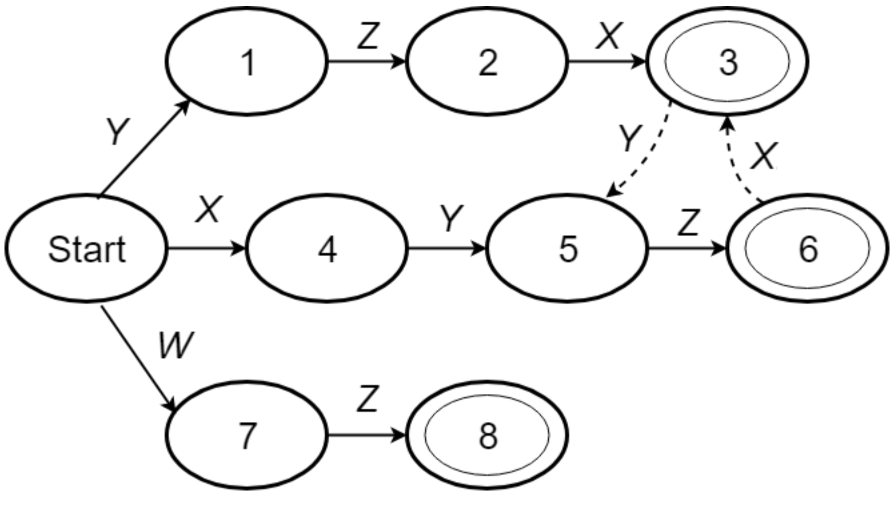


Figure 2.2: Example image of a state machine in Snort looking for the patterns *YZX*, *XYZ*, and *WZ*. Output states are marked with an extra container.

The failure transitions seen in figure 2.2 are just examples of the ones that would be present in this machine. There are additional ones, such as the case when it is not possible to continue it will go to the first state for that character. To exemplify this, state three would transition to state four if the next input would be an *X*, to state seven if the input was a *W* and so on.

The basic idea of Aho-Corasick is not necessarily very hard to understand. However, with more patterns, the state machine can grow enormous and thus take a long time to build, but more importantly, the machine will consume a significant amount of memory when holding all of these states. Luckily, these state machines only need to be built once at the initialization of Snort, and can thereafter be used throughout the rest of the state machine's lifetime.

2.4 GPGPU and processor architecture

General-purpose computing on graphics processing units, commonly known as GPGPU, is a computer terminology where the GPU is used for computational work that is usually done on the CPU side. Today there are many projects that use GPGPU to create new solutions, or improve upon existing ones, to problems in a multitude of areas. The rise of interest in GPGPU could be the reason that more sophisticated

APIs are being developed, such as CUDA and OpenCL that are easy to setup and use.

The reason that GPGPU exists is due to the architectural differences of the CPU and GPU [21]. CPUs in general are designed for a few threads doing efficient calculations with low latency by utilizing a large cache memory and operating at a high clock frequency. The CPU also contains a control unit containing additional and more complex instructions which allows it to do operations that the GPU can not. For this reason CPUs are sometimes referred to as latency devices.

As opposed to the CPUs, GPUs are designed to provide high throughput of data by running thousands of threads in parallel. Instead of having just a few cores as the CPU does, the GPUs can have thousands of cores to accommodate the parallel work it has been designed to achieve. This does not make them superior to the CPU since GPUs have smaller cache memory leading to more main memory accesses and operate at lower frequencies. Therefore GPUs, different to the CPUs, are sometimes referred to as throughput devices.

Due to their differences, CPUs can be more efficient in certain situations while the GPU might be a better candidate in others. Consider pattern matching, where large volumes of data have to be evaluated but with a fairly simple comparison. As such, this should be a job more suitable for the GPU. With the high parallelism that the GPU supply, the work can, at least in theory, be performed at much faster rates compared to the time spent on the CPU. However, due to the fact that the architecture is different and it requires additional work to enable the usage of a GPU, GPGPU is not always easy to make use of.

As mentioned earlier, today there exist APIs that enable easier ways to start using the GPU for computations not relating to graphics. Two of the most common APIs, OpenCL and CUDA, are both supporting the common languages C and C++ which make them a good choice many projects. However, in an IoT context with a multitude of different devices there will be different types of hardware, including what GPU a device comes equipped with. This could be a problem for CUDA as it is currently only supported by NVIDIA GPUs. Meanwhile OpenCL is supported by various GPU vendors and will be further described in section 2.5

2.4.1 Aho-Corasick on GPU

The Aho-Corasick algorithm is simple in the way that all it requires is a state and an input to decide where to go next. The simplicity makes it suitable to use it on a GPU since parallelism can be gained by simply partitioning the text to be searched and let different threads examine each part. Also, the threads only need to read from the input and the state machine so no race conditions occur between threads when executing concurrently.

The actual work of each thread is equal to the flow explained in section 2.3.2, just that each thread has less data to process since the input is divided between them. With less work, each thread will finish faster and thus performance can be gained. Due to the parallel characteristics of the GPU explained in section 2.4, the work can be done faster here compared to examining the entire text on the CPU.

The downside of this approach is that a pattern might be located on the same place where the text is split into two different threads. This means that neither thread can find the pattern since they only have one part each instead of the entire pattern that is searched for. One way to solve this is to let threads continue past their allocated length to a maximum of the longest pattern in the state machine, beyond this they will only find patterns that the other threads have already found.

2.4.2 PFAC - modified Aho-Corasick for GPU

There is a modified version of the previous Aho-Corasick called Parallel Failureless Aho-Corasick (PFAC) [22]. This algorithm is almost the same as Aho-Corasick on which it is based, with the difference that it does not include the failure states that was explained in section 2.3.2.

Instead of having many threads searching a specific length each, a single thread will be designated to each character of the input. This means that the algorithm will use as many threads as the total number of characters in the text. Every thread will then search the input from the character it was dealt until it returns to the initial state. Returning here means that the pattern it searched for was found or that the search was broken, i.e. the current pattern was not in the input. This makes the algorithm suitable for GPUs that can spawn many threads for every character in the data to be searched.

The downside of this algorithm is that the input might be longer than the number of threads the GPU can spawn. Of course it would still work by queuing the threads and letting each run when resources become available. In this case the GPU thread reinitialises and runs at a new index and performs the search anew from the initial state which is basically the behavior that failure transitions try to remove. Also, in this case many threads could start up just to terminate while other threads still work, instead of partitioning the work evenly to each available thread.

2.5 OpenCL

First developed by Apple Inc. in 2008, OpenCL is an API enabling heterogeneous computing over a multitude of different processing units such as CPUs, GPUs and more [23]. Development of the API has been handed over to the non-profit organization Khronos Group Inc. who are also responsible for other well known toolkits such as OpenGL, Vulkan and more.

OpenCL supplies functionality that can retrieve information about and allocating computational devices located on the running system. The CPU is known as the host in OpenCL since this is the default computational device in all systems and the one running the process with OpenCL. The CPU can thus find more devices in the system and then delegate work to these with OpenCL functionality. This could for example, like in this thesis, be used to allocate the GPU in order to do some work on such a device.

In this way, OpenCL enables high parallelism in a program, because work can be distributed over multiple devices. After the CPU distributes work to a device, it can continue with other work and return later to retrieve the results from the device. This is how OpenCL can be used to improve the performance of applications, but in order to do so, OpenCL requires additional resources on the system that is hosting the execution. An obvious example, if the hosting system has no GPU, there is obviously no way of allocating or use it.

OpenCL uses multiple work-items divided into work-groups to process the jobs assigned to a device. Each work-item can be visualized as a working thread since this is the object that executes a given function that has been loaded for processing on the device. Each work-item in the same work-group executes the same function and is also executed on the same compute unit in the device.

The memory model of OpenCL is divided into three levels: global, local and private. Each work-item has its own private memory, the work-items in the same work-group share local memory while all work-items in all of the work-group share the same global memory. In the memory model of OpenCL, the host memory is separated from the OpenCL memory so the host can not directly access it. Instead the host must use functions to read or write to device memory to send and retrieve data. This makes sense as most graphic cards have their own dedicated memory, but it is true even in cases where the host and device share memory such as with an integrated GPU.

However, OpenCL is not without limitations and it has some restrictions on what and how data can be copied to other devices. For example, it is not allowed to transfer structures that contain pointers and arrays of dynamic size are not supported either. These kinds of data are common in both C and C++ and for that reason such data must somehow be converted to a legal object before sending it to the GPU. This will be further described in section 5.1 as it was important for our work with Aho-Corasick.

3

Related Work

This chapter introduces multiple works that relate to this thesis in different ways. As mentioned previously, GPGPU is not a new concept and has been explored in academic work for years and there exist multiple projects that have used it to reach a multitude of goals, including improving IDSs. Each section below will also highlight how the works differentiate in requirements and tools, such as framework and hardware. Finally, the last section will compare this thesis to the work described here and highlight their differences.

3.1 Intrusion Detection Systems using GPGPU

The first project that tried to optimize Snort with GPGPU was PixelSnort, which was presented at the *Computer Security Applications Conference* by Jacob and Brodley in 2006 [24]. Using the nowadays deprecated language *Cg* in conjunction with Open Graphics Library (OpenGL), a platform-independent API for graphics, they modified Snort to perform a portion of the pattern matching on a GPU with the purpose of offloading work from the CPU. By using the resources in the GPU, such as pixel processing and image memory, it was possible to perform the pattern matching work of Snort on the GPU, although not exactly as done on the CPU.

PixelSnort achieved at best a 40% increase in performance when the CPU was under high load, but with no noticeable performance gain under normal load. The lack of performance improvements during normal load might be due to the fact that the language and hardware were too complicated to use for GPGPU at that time as later projects have proven plenty of good, promising results. Another reason could be that the creators of PixelSnort used the Boyer-Moore pattern matching algorithm since they could not develop the Aho-Corasick algorithm on the GPU due to the limitations in for example *Cg*. Interestingly enough however, they raise the question of how secure it is to use GPUs for this kind of work. Since the architecture of GPUs compared to CPUs are different, it could open up for attacks not previously seen. This is something that has not been discussed or mentioned in later works.

There are big differences between this thesis and PixelSnort, such as the framework used and the targeted devices. The language *Cg* used in PixelSnort was created by NVIDIA to shade pixels for games. Using this language, from such a different domain, to perform pattern matching is impressive on its own but it is also one of the limitations in PixelSnort. With modern hardware and frameworks such as OpenCL, it is possible to use more demanding algorithms and data structures. That is why the Boyer-Moore algorithm will not be used in this thesis, but instead Aho-Corasick is used which PixelSnort could not implement due to the limitations. Another difference is the fact that this thesis will run and evaluate the IDS on a single-board-computer and not a desktop computer.

Compared to PixelSnort, a more successful work at using GPGPU to offload the CPU is Gnort, a modification of Snort presented in 2008 by Vasiliadis et al. [9] One of the major differences to PixelSnort is the fact that Gnort uses the Compute Unified Device Architecture (CUDA) toolkit developed for NVIDIA GPUs, that had just been released at this time. This high level programming interface opened up for capabilities not available to the developers of PixelSnort, such as porting the multi-pattern matching Aho-Corasick algorithm to the GPU. Some graphical concepts were still utilized in Gnort, such as storing the Aho-Corasick state table and patterns to search in texture memory. However, by using the new and improved functionality, and also doing all the pattern matching on the GPU, the prototype of Gnort outperformed Snort by a factor of two in terms of pure throughput during their evaluation.

Gnort is a project more similar to this thesis in terms of what frameworks that is used to improve the performance of the IDS. The framework used for GPGPU programming in Gnort is CUDA, which is in many aspects very similar to OpenCL. The main difference between these two is that, as stated previously, CUDA is only supported by NVIDIA GPUs. Since it is expected that single-board-computers will have varying hardware components, and that NVIDIA GPU will not be that common in them, OpenCL is a much better fit for IoT in general. As in the case for PixelSnort, the differences come down to framework and target, even though the frameworks are similar in this case.

Kargus is another IDS employing a GPGPU approach for intrusion detection. However, Kargus is not a modification of Snort and differs from it since Kargus is designed and developed for high-bandwidth networks [10]. Still, Kargus does have similarities to other works in this area since, just like Gnort, it was developed with CUDA to enable the usage of GPU for pattern matching.

Contrary to what could be assumed for high-bandwidth networks, Kargus is actually designed to do most of the work on the CPU. By default in Kargus, the CPU handles all of the stages of the IDS such as preprocessing and logging, but also the pattern matching. The GPU only becomes involved when the CPU is getting overloaded with work, at which point the CPU will start using the GPU for some of the pattern matching to balance the workload. Kargus will only do this for larger packets though, since it takes time for the CPU to send the packets to the GPU. Therefore it can be beneficial to just spend that time to pattern match the string on the CPU instead. This design leads to a very high throughput compared to versions of Snort, but it also requires more powerful hardware to run it well.

The high-bandwidth target of Kargus separates the work from this project. The entire design of the Kargus IDS is directed away from IoT devices based on its high-speed target. To run Kargus efficiently the devices would require a significant performance increase that is not suitable for IoT devices in terms of affordability and size. Also, like Gnort, Kargus use the CUDA framework that only supported NVIDIA GPUs.

Recently, in June 2017, a work of using Snort with OpenCL was done by Xie et al. [25] The project is referred to as OpenCLSnort and the core idea of it was very similar to this thesis, to improve the performance of Snort using GPGPU with OpenCL. However, there are some important details missing in the paper. For example, there is no mention of what Snort version was used in the project, but we assumed it to be Snort 2.

In this work the authors use a modification of the Aho-Corasick algorithm to do the pattern matching, called Parallel Failureless Aho-Corasick (PFAC) [22]. As stated in the explanation of PFAC in section 2.4.2, this algorithm is designed to fit the architecture of GPUs. The work is parallelized by dropping the failure transitions of Aho-Corasick state machine and only running through one iteration per thread on the GPU. This means that once the algorithm returns to the initial state, the thread will stop and terminate. This requires more threads but less work for each, a design that does sound good for GPUs capable of running a huge amount of threads.

The results from OpenCL Snort achieved a similar gain to that of Gnort, namely two times the throughput of Snort. However, OpenCL Snort and Gnort are most likely running different version of Snort, so their exact data differs with Gnort having a throughput of 2,3 Gbit/s while OpenCL Snort achieves 6,758Gbit/s. This is of course also due to the more modern machine used in OpenCL Snort.

This thesis have many similarities to OpenCL Snort due to the fact that it uses Snort along with OpenCL. Nonetheless, this project will aim to use Snort's traditional way of pattern matching using the Aho-Corasick algorithm. Aho-Corasick and PFAC does not differentiate much from each other but the original is in the context of IoT considered to be a better fit since it distributes the work between the threads more evenly. Also, this thesis will use Snort 3 as the IDS to be modified and evaluated. This is different to the the assumed Snort 2 version used in OpenCL Snort.

3.2 Portable IDS with Snort and Raspberry Pi

A project more related to IDSs in IoT networks, and not so much GPGPU, is the RPiDS [8]. In this work a Raspberry Pi 2 running Snort to function as a portable IDS was thoroughly tested to evaluate the capacity of modern single-board-computers. Compared to the other projects mentioned, RPiDS is more of an experimental project testing Snort in an IoT context.

The evaluation was done by doing several tests with different rulesets of Snort and varying network traffic. By measuring the workload of the device during these test runs it could be seen how the Raspberry Pi performed. The measurements showed that the Raspberry Pi could run Snort without ever overwhelming the CPU or filling its entire memory capacity. These results strengthen the argument that single-board-computers are a reasonable choice for security in future IoT networks, especially since it is expected that hardware improves with time. The success of the project serves as a good background and encouragement of this thesis.

The results also motivate this thesis, because even if it is possible to run Snort on the Raspberry Pi, other processes on the device might be blocked due to it. Each IoT device will have to investigate the incoming traffic, unless the network has a central point IDS in which case it would have to communicate with it. Beyond this, the device must also be able to fulfill its main function that it was deployed to do, otherwise there would be no point of deploying it. Even though the Raspberry Pi was not overloaded in [8], the available resources might not have been enough to run additional processes which would have been a problem.

3.3 Intrusion Detection System for embedded devices

The current research in the IDS field is not only focusing on developing software to desktop computers and single-board computers. There are also solutions which aim to protect devices that have even less memory capacity than a single-board computer. One example of such a solution was presented in the paper Flexible Intrusion Detection Systems for Memory-Constrained Embedded Systems by Tabrizi et al. [7]. In this paper the authors are presenting a software tool that produces a customized IDS based on the memory capacity of the targeted device. This means, given the user-defined security coverage functions, the security properties of the system and memory requirements, the tool produces an IDS customized to operate on the specified system.

With this tool the authors were able to produce an IDS, tailored for an electrical smart meter, that operated on 4MB of memory. Even though 4MB is low memory usage, one has to take into account that an electrical smart meter only performs a few specific tasks. This narrows down the set of invariants that the system needs to preserve while the system is operational. The number of states in this IDS depends on these invariants and the correct behavior of the system. This means that a device with more functionality would have more states and would require more memory usage to provide the same level of security.

In comparison to this thesis, the IDSs proposed by Tabrizi et al. differ in several ways. Probably the most significant one is the detection method. This thesis is based on Snort which means that it uses a signature-based detection method, whereas the IDSs produced by the software tool instead uses an anomaly-based detection method. Moreover, even though performance in terms of throughput is tested and evaluated in the paper by Tabrizi et al., the main focus is on memory efficiency and especially to get an IDS to run on an embedded device with very high memory constraints. Even though this thesis also will consider memory efficiency, it will not be at the same level.

3.4 Our contribution

As have been stated throughout this chapter, there are similarities and differences with all the works that have been discussed. Even though the projects mentioned in this chapter are related to our thesis, they do not necessarily fill the same gap. One of the most notable differences with our project is the aim and evaluation of single board computers for IoT. The works mentioned here that used GPGPU have all targeted the more common desktop computer and do not reflect the IoT context. Another is the fact that to our knowledge, none of these projects used Snort 3 to fulfill their aims, and thus lose all of the benefits from it, such as the built in multithreaded packet capturing.

4
Design

This chapter aims to clarify some of the functionality of Snort and how it will be changed to enable the aim of this thesis. This chapter remains at a high level while the next chapter contains more details about the implementation. Following the section on Snort, the chapter will also further introduce the ODROID-XU4, the target device in this thesis, along with its components.

4.1 Deciding the IDS

Snort is an established and well known IDS widely used around the world and is sometimes referred to as the de facto standard IDS. While Snort might be well known, there are several other reasons to as why it was chosen for this thesis. Most importantly is the fact that Snort is open source, meaning that anyone can download and modify the source code. Since this was a requirement for us to enable GPGPU, it was one of the most important features.

Additionally, Snort has been around since its release in 1999. With the time and development of Snort it has a promise of better documentation and questions that have been answered previously, allowing us to focus on the main objectives of the thesis. While this might be less true for Snort 3, certain things in the new release are unchanged and so old knowledge can still be applied. Snort has also been used in similar projects and works previously, ensuring us that it would fit for this thesis.

For the more recently developed Suricata, another IDS that was considered, similar information seemed to be more sparse. While Suricata might do certain things better than Snort, it was considered beneficial to take the NIDS with more information available.

4.2 Snort design

Snort 3 is designed from a central process spawning additional threads handling the actual packet processing. This fits well for the multi-threaded design that is one of the key features of the new Snort 3 alpha release. The initial thread is the bridge between the user and the rest of the program, handling I/O functions and all the initial setups such as the configuration file that holds settings for Snort.

For every network interface that is to be inspected, the initial Snort thread will spawn more threads called analyzers which are the main workers of Snort. Each analyzer holds a connection to a network interface on the system to which it constantly checks for network packets. Once a batch of packets has been retrieved, each packet will be processed on its own. The steps that each packet goes through, as seen in the Snort documentation [1], can be seen in figure 4.1.

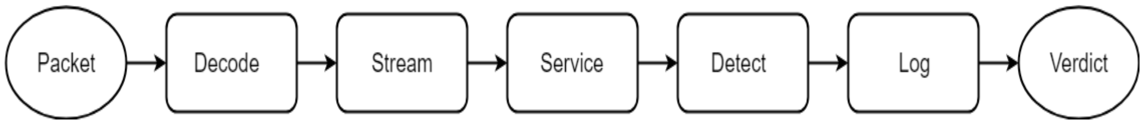


Figure 4.1: Basic work flow of an analyzing thread in Snort as shown in the user manual [1].

In the decode step, an analyzer finds common information about the packet such as the source and destination addresses and ports while examining the encapsulating protocols for anomalies. This is followed by TCP packet reassembly in the stream step and then service specific analysis in the next one. Once finished with the pre-processing steps, the analyzer thread then enters the detection phase which includes the pattern matching. The searching is basically split into two phases where the first of these does a fast pattern matching search to find any matches in the input. This is followed by a second phase which performs another match that considers packet information like the header, flags and additional data. This phase will be refereed to as the match phase from now on. Finally, the analyzer logs the findings of the current packet and then returns a verdict of what should be done with the package. This last step is not something that matters much when running Snort in IDS mode as this mode can not reject or block packets.

The previously explained flow is performed by each analyzer spawned in Snort. Earlier in section 2.2, it was mentioned that one of the main features of Snort 3 is that it allows multiple packet processing threads, that is multiple analyzers. However, this is not completely true since multiple threads can not listen on the same network interface. When they do, the traffic is replicated on each analyzer, so instead of sharing the work they will perform twice as much since each thread inspects the same packets. However with external programs, such as a network load balancer, it could be possible to redirect the traffic to multiple analyzers. This is not something that will be done in this thesis as it will only make use of a single analyzer.

When it comes to the pattern matching, Snort has a couple of different formats to handle the Aho-Corasick state machine, sacrificing speed for memory usage or the other way around. For example, Snort can use a sparse state machine that only holds a few states in memory for the normal transitions. However, due to the structure of the machine it has do to additional checks that impacts the performance. Snort can also use the full matrix format which is also the one to be used in this thesis. In this mode the matrix holds all the transitions in the machine, including the zero elements that points back to the initial state. This requires more memory but the actual matching can be performed faster due to the simple structure. There are additional formats described in the documentation [1] where each has both pros and cons in terms of memory and performance. As the other formats will not be used in this thesis, they will not be discussed any further.

To perform pattern matching, Snort creates multiple state machines based on the rules that it is configured to handle. The rules are grouped based on protocol, port and service and each group will create its own state machine during the initialization of Snort. Every pattern contained in these rules will therefore be compiled into the state machine which is then built accordingly once all the patterns are added. How this is built and traversed depends on the format of the state machine.

Snort relies on some common UNIX libraries to enable features essential to the system, such as libpcap [26] for packet capturing while from a hardware perspective, Snort only requires a CPU and memory to run. However, how well a device can run Snort obviously depends on the specifics of the available resources. The requirements make Snort compatible with a various set of devices since memory and a CPU are key components in any functional device. The Snort modification developed in this thesis will work differently and for the same reason requires additional resources, which will be further discussed in the following section.

4.3 Our model

To reach the goal of this project, we have to change the way that Snort works during the detection step seen in figure 4.1. More specifically, the payload of the network packets that Snort reads, the actual application data, must be sent to and then searched on the GPU. The assumption is that the cost of the data transfer to the GPU will be offset by the faster pattern matching on the GPU. This is one of the aspects that is later evaluated in chapter 6

The overall work flow will for two reasons be very much the same in the modified version of Snort. Firstly, this allows the main features of Snort to remain functional and supply the same service. Secondly, using the already implemented functionality allows the work in this thesis to focus on the pattern matching on the GPU.

The idea is to take the data that is normally searched on the CPU and send it together with the state machine to the GPU. Once the data is on the GPU it is possible to spawn more threads than on the CPU and in turn search the data in an equal fashion as to how the CPU does it. This will be done by splitting the data between the spawned threads and thus each thread does less work than the

CPU would have to do when searching the entire payload. This concept is further visualized by figure 4.2.

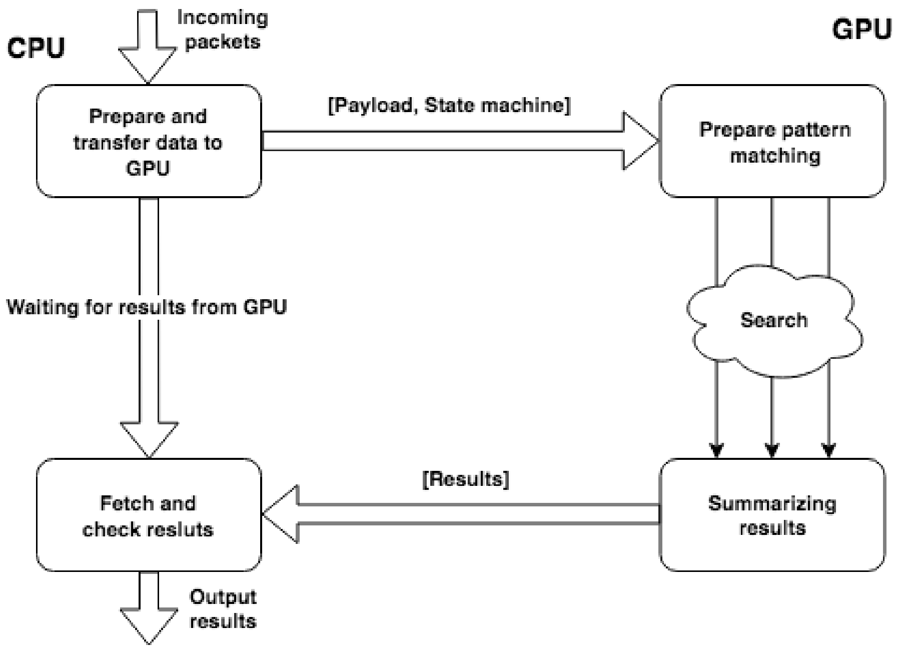


Figure 4.2: The execution flow of the modified detection step. It shows an overall picture on how the data is translated from the CPU and processed on the GPU.

4.3.1 Design tradeoffs

As was mentioned in section 2.4.1, splitting the data between threads is beneficial to the performance of the system, but it also introduces the problem not seen on the CPU. A certain pattern that is to be searched for could be split between two or more threads and thus be missed by these. For example, if the algorithm would search the data *xyzw*, two threads would split this by one searching *xy* and the other *zw*. If the pattern to look for was equal to the input *xyzw*, then both of these would return that zero matches were found. However, if a single thread would search the entire string, as it is on the CPU, then the algorithm would return that a pattern was found.

One way to not allow any misses to occur is to allow a thread to continue searching until the current pattern is broken. But then two threads might read the same data, since one continued past its own space, and thus finding a pattern that another thread has already found. This would result in more found patterns than what is actually present in the data. While this is a negative effect of this approach, it would still be expected that most of the matches found would be filtered out by Snort. This is because Snort validates the matched patterns in relation to the packet information after the payload has been searched. For example, even if the content of a Snort rule was matched in the payload, the source and destination addresses of that packet could still be different to the ones that the rule specified. In this case, even if we had a content match, Snort would not output an alert. Additionally, the issue could be further alleviated by keeping track on where the match occurred and how long the matched pattern was. If the starting position of the pattern was outside of the input that the thread had been assigned, then the next thread would be ensured to have found it as well. This way, duplicate matches could be filtered out earlier in order to not analyze them any further.

The decisions to overlap is based on that it is better, from a security perspective, that all patterns are found and reported multiple times, rather than not alerting for an attack at all. Performance-wise, it should have little impact of letting a thread execute a couple of extra characters. As a side note, this is not a problem for the PFAC algorithm as this does not contain any failure states. Therefore it is easy to break the execution of a thread once it returns to the initial state. Also, no threads can find the same patterns since all of the threads begin at different indexes. While this is a benefit of PFAC, the drawback is that PFAC is not compatible with Snort. The state machines in Snort are built to include failure transitions and thus PFAC would not be able to work as intended.

A bigger performance impact could be that, as the related projects also mentioned [9, 10], simply transferring a packet to the GPU is not very efficient since it takes time for the CPU to transfer data between the CPU and GPU. One of the goals is thus to investigate the cost of single packet processing solution on the GPU to measure the time it takes for a single packet to be evaluated on the GPU compared to evaluation on the CPU. This will be followed by an extension where network packets data is buffered and sent to the GPU as a batch to reduce the CPU time spent per packet.

When it comes to compatibility and hardware, the modification of Snort will require additional components to run. For one, the modified version will use the GPU to do pattern matching which requires such a device to be available on the system. Running our modification on a system without a GPU will not be able to use the modified search developed in this thesis as not having OpenCL support will make it unable to run at all.

A downside of this prototype is that it is expected to require more memory and setup time compared to original Snort. This is due to the fact that certain objects created by Snort must be converted and stored to fit the architecture of the GPU and the OpenCL API. On the other hand, this will not impact the real-time analysis of the network traffic.

4.4 Hardware

This thesis targets the ODROID-XU4 [27] to act as the IoT device that will run the modified IDS we develop in this project. The SBC comes equipped with a Samsung Exynos 5422 Cortex mobile processor which is the one also used in the smartphone Samsung Galaxy 5S. The processor includes two CPU cores, one quad-core ARM Cortex-A15 and a quad-core ARM Cortex-A7. Additionally, it has an integrated GPU, namely ARM Mali-T628. Further more it has 2GB RAM memory and also a Gigabit Ethernet port which will be useful when evaluating our Snort modification. With these features the ODROID-XU4 is a very suitable device for this thesis, having the required processing units while also being an alternative for powerful IoT devices.

Integrated GPUs usually use a portion of the RAM in the system since they lack dedicated memory. This is also true for ARM Mali-T628 GPU that the ODROID-XU4 uses. For this reason, the ARM Mali GPU has a different memory model compared to GPUs on traditional desktop PCs [28]. This model has no private or local memory, mentioned in section 2.5. This is a drawback since the faster memory types could have been used for some of the data in the pattern matching, given it would be large enough to hold it.

However, since the data is located in the same memory, there is no need for copying between different memory modules. Data can instead be mapped to the device, in this case the GPU, making data only available on the CPU to also be available to other devices. This can prove beneficial, since less memory will be occupied by data copies while also limiting the time spent on copying by instead translating the address spaces.

The decision to use the ODROID-XU4 is based upon on its relative powerful hardware specifications. While there are many alternatives on the market, such as the Raspberry Pi 3 that also was considered, they all have pros and cons. For example, the Raspberry is widely used by a large community and well documented. Further more, the Raspberry is cheaper than the ODROID, making it more suitable in a situation where a huge amount of devices had to be purchased for installation. The downside of this is that you get what you pay for, as the Raspberry has half of the ODROIDS RAM and less powerful computational units, GPU as well as CPU. For this reason, the ODROID would give us less limitations during the developments of this thesis.

5 Implementation

This chapter will further detail the actual implementations of the Snort modification developed during this thesis. This includes how data is made avilable to the GPU, how to perform pattern matching and how the result is transferred back. The details of this chapter are separated into two different sections, where the first one describes the work for a single packet while the second describes the case for multiple packet searching.

5.1 Setup of the state machine

Original Snort is, like many other softwares, a CPU only application. To be able to offload the CPU with GPU processing, the data used in pattern matching has to be available to the GPU in order to search for the patterns. The functionality in OpenCL enables the allocation of data so that it can be transferred to the GPU and then send back the results to the CPU.

In order to use the functionality that OpenCL provides, some required objects must first be initialized in the setup of Snort. These objects can then be used to allocate devices such as the GPU in this case, or queue up jobs that are to be executed by the GPU. Due to the fact that the work in this thesis only relates to the pattern matching, all of these objects are created when the state machines are created in the setup of Snort.

Snort might create multiple state machines depending on the configuration it has which is the specification of what settings and plugins to use. This means that each machine will set up its own context, device relation to the GPU and so on. For parallel work this is not a problem since OpenCL separates these processes from each other.

To enable functional OpenCL support, the following must be set up in the state machine at creation:

1. Context - identifies the environment such as system platform and device to use

2. CommandQueue - the object that can queue work that a computational device, in this case GPU, retrieves from to execute

3. Program - built from a source file, in this thesis the pattern matching code, and is executed on the GPU when work is delegated

4. Kernel - the object that holds the program and accesses the CommandQueue to retrieve work

After a new state machine structure has been created it is thus also prepared to use the GPU. Later in the detection phase of Snort, this state machine is passed as an argument to the search function that is responsible for the pattern matching. Therefore the state machine also supplies the GPU functionality to the function which enables it to use the GPU for pattern matching instead.

5.2 Single packet to GPU

In the detection phase, Snort will analyze a packet by passing the payload data and the state machine with the patterns, along with some other parameters. In this stage the state machine described in the previous section is ready to be used for the pattern matching. For example, it contains the OpenCL kernel that can be used to delegate jobs to the GPU.

However, before the work can be carried out, data has to be made available to the GPU for it to have something to process. By using OpenCL buffer functions, available in the state machine, the data can be directed to the GPU where it will be used to perform the pattern matching. In the current solution, the following data is sent to the GPU in each iteration:

1. The state table - sent as an array representation of all the states and the transitions belonging to them

2. The payload - the data that is to be searched for the patterns in the machine

3. The translation - an array of unsigned chars used to translate input characters to uppercase

4. The length - an integer that specifies the length of the payload to be searched

5. The result - an array where matches are saved when found by setting the state on the same index where it was found

Each of the data objects above are vital to perform the pattern matching on the GPU, but not all of them are compatible with OpenCL. This is especially true for the state machine that Snort creates since this is a structure object with pointers. The state machine structure holds the states with transitions, the list of matches for each state along with additional information that can be useful for the machine but not necessarily for the pattern matching. OpenCL can not copy or translate structures containing pointers to the GPU. Instead the needed data must be converted to one or multiple arrays to be sent to the GPU.

The data required from the state machine is the actual state table with the transitions. This thesis will use the full matrix format of the state machines due to its simple structure which works well with the limitations of OpenCL. The full matrix format uses a state table that can be visualized as a two-dimensional array of integers where each row is a state and each index in a row holds the next state for a possible input. Additionally, it holds two extra integers, one for the format and another to identify if the state has any matches. This format can easily be converted to a one dimensional array due to the fact that each row is a fixed length. Also, with a fixed amount of rows, one for each state in the state machine, the conversion only needs to happen once.

To fit the Snort state table to the OpenCL requirements, the state table is parsed and each value in every row is saved into a newly created one-dimensional array. The new array has the same size as all the rows together in order to hold the same information that the state table does. When retrieving from the one dimensional array, we first get an offset to pass the previous rows in the array by calculating the current row minus one times the row length. After this, the index of the current row is added to the offset which yields the position to read from.

$$newindex = (currentRow - 1) * rowLength + index$$

This means that the index for 3.1 in figure 5.1 get a new index of 6 in the converted array.

$$6 = (3 - 1) * 3 + 0$$

Basically, the Snorts table is serialized by merging the rows together in a new array which figure 5.1 illustrates.

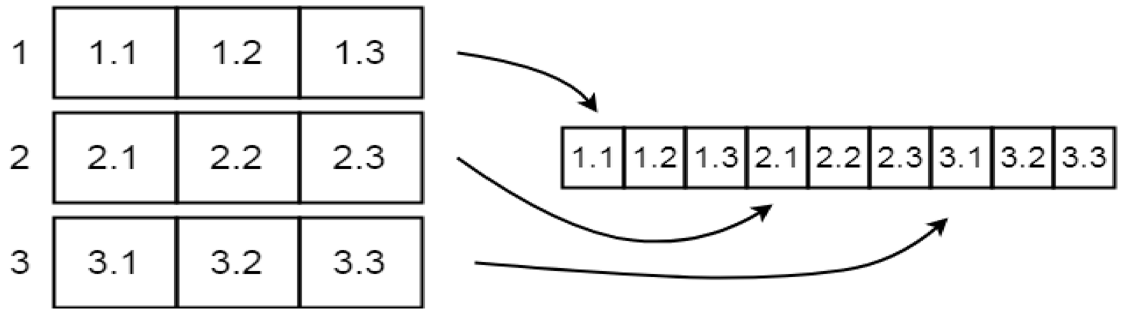


Figure 5.1: How the two-dimensional array is serialized into a single array.

The other data to be translated is already in an acceptable format and can thus be sent straight to the GPU without further adjustments. After translating the addresses of the data for a single packet to the GPU, it is ready to search that data for matches which will be explained in the next section.

5.3 Pattern matching on GPU

After the data is made available to the GPU, see figure 4.2, it is possible to begin the pattern search. To start the search, the function will use the OpenCL objects created with the state machine, such as the OpenCL program mentioned in section 5.1. The CPU will enqueue the program to the kernel that will eventually execute it and notify the CPU when the job is finished. The program in turn knows what to do with the data that has previously been handed to the GPU. The CPU process could go on with additional work after this but in the current version it blocks the process until the job is finished on the GPU side.

The OpenCL program object contains the source of our modified version of Snort's pattern matching. The objective of this function is to enable partitioning of the data to be searched by different threads. This function iterates over the input text and handles one character at a time but before using the character it must be translated via the translation array to set it to uppercase. The reason for this is that the numerical values for uppercase and lowercase characters are different, so when analyzing the characters they are always converted to uppercase beforehand. After conversion, the current state is first checked if it is an output state or not. If it is, the counter is incremented and the state is written to the result array and if not, the search simply continues. Before continuing the next iteration of the loop, the state transition happens by looking up the next state via using the index as offset in the state table which sets the next state. The pseudo code for this algorithm can be seen in algorithm 1 and is executed by every thread spawned in the GPU search.

Since threads do not rely on each others data, but only the input and previous state, it becomes easy to run the pattern matching over multiple threads at once. The input length is divided so that each thread spawned has equal amount of work compared to the other threads. Each thread will start searching the input at different indexes based on its own unique id. The idea of each thread is to set its own starting position beyond the part that the other threads, the ones with lower id than its own, will handle. From this position, it searches the specified length of the input, marks any matches it finds and then stops. Since the threads only reads from the state machine and the input text, while keeping their own current state in a local variable, there are never any race conditions between the threads. The only time these could occur is when a thread is writing results which is solved by using atomic functions in these cases.

Algorithm 1 Aho-Corasick GPU

```
1: $startPos \leftarrow threadId$
2: $limit \leftarrow startPos + (length/threads)$
3: for $j \leftarrow startPos; j \leq limit \ \& \ j \leq length; j++$ do
4:     $index \leftarrow translate[input[j]]$
5:     if $stateTable[(state * 258) + 1] = 1$ then
6:         $counter++$
7:         $ResultArray[state]++$
8:     end if
9:     $state = stateTable[(state * 258) + 2 + index]$
10: end for
11: $ResultArray[0] += counter$
```

5.4 Results to CPU

As mentioned in section 5.2, during the GPUs search the CPU thread is blocked to await the result of the pattern matching. When the GPU is finished with the job that is blocking, the CPU is alerted and can continue with the program. The NIDS has no important task to perform at this time, other than the pattern matching. Instead, the device can use the CPU for other work during the time that the analyzer thread is blocked. In this place it will use OpenCL functionality to read from the result array on the GPU.

During the GPU execution, every thread will mark a match in the result array. The result array is a one-dimensional array with length equal to that of the input text so that every index in the array relates to the same index in the text. So when a GPU thread finds a match on a certain index, that index is marked with the state that had the match. This is then read back into the result array on the CPU so that it can be used further.

Snort's pattern matching happens in two separate phases. The first of the two is the fast scanning search of the entire input, the same part we now perform on the GPU. Snort usually applies a callback function the moment a match is found on the CPU. This function is passed during execution as an argument and can point to different functions based on the current packet. However, it is not possible to do this on the GPU since it holds no link to the functions on the CPU and has no way of calling them. It is not possible to transfer this function to the GPU either, since such a function may in turn call additional functions on the CPU.

To provide the same functionality as the CPU, the callback function will still be used but after the search has finished on the GPU. This is the purpose of the result array that the CPU reads from after the GPU search is finished. The CPU will parse the result array, and for every non zero element, since state zero can not be an output state, it will call the callback function for the index where it was found together with the state and other information that the function needs. This way the second phase of the search, the match phase, is called for every found match.

Since the callbacks are the same that original Snort uses, the work from here will follow Snort's standard protocol and therefore supply the same functionality. Once all matches in the result array have been handled, the pattern matching is completed and Snort can go on with the next packet.

5.5 Multiple packets to the GPU

To improve the transfers between the CPU and the GPU in the single packet solution, we also implemented a multiple packet version. In this implementation the payload and length of each arriving packet is buffered into an array and sent to the GPU as a batch at a later point. In this mode, instead of searching every time a packet is handled, the function will store the data and then check the current length of the buffer.

When checking the length of the buffered data, if it has not passed the threshold length, the function will simply exit and continue normal execution of Snort. If the threshold has been passed however, the current packet is also buffered and then the search is launched on the GPU. This implementation means that the CPU has to spend less time waiting for the GPU compared to the single packet solution.

The same objects that was transferred in the single packet solution is also transferred in this one. This is handled the same way as in the single packet solution, such as the state table being converted into a one-dimensional array to support the OpenCL requirements. The data made available to the GPU was the state table, the translation array, the payload data and its length and lastly the result array. However, in this implementation, the result array will simply return what states got matched and the total count of these. The reason why is explained in the next section and has to do with Snort's sequential design.

Once everything has been made available to the GPU, the search will behave almost exactly as described in section 5.3. Still, there are two differences between this version and the single packet transfer worth mentioning. First, the results are outputted differently in a more simple manner, as was mentioned above. The second is that this solution contains a flush due to data being buffered instead of instantly being searched. When the modified Snort version is terminated there might still be data in the buffers that has not been searched. To handle this, each state machine will handle the stored data, if any, in a final search before being deleted.

5.6 Gather results to CPU

When the GPU has searched the buffered data the CPU can fetch the results from the GPU. However, as mentioned in the previous section, these results only contain which patterns and how many of them are found in total. The reason for this is because when we buffer packets we make the match phase of Snort impossible.

Snort relies on the Data Acquisition library [29], or DAQ for short, which is an abstraction layer to the libpcap calls used to capture packets on the network card. DAQ basically retrieves a batch of packets and then sends them to Snort one by one. This is where Snort will decode and search the packet and alert if something malicious is found. In this implementation, the CPU will buffer the data and then continue, thus DAQ will consider the packet to be handled. When the new packet is then handled by DAQ and sent to Snort, it will consider the previous to be finished and thus this packet might be overwritten with other data.

For this reason, while buffering the payload and length of the data was simple enough, buffering additional packet data was not. The packet structure is a fairly large one and every part of the data in the structure would have to be copied. This would take a lot of time to double copy every packet and it would also impact the memory considering that we would need to keep thousands of these in memory. Referencing to the already existing data is impossible since when enough data is gathered it is highly likely that the data is overwritten.

Since the data for the second search could not be made available, we decided to use a simpler result array in order to read less data from the GPU. The results include what states that were found and a total count of these. So while providing better performance it comes with the downside of not functioning as Snort originally intended.

5.7 Improving multiple packet solution

When doing some initial testing of the multiple packet solution, we found that it was not living up to expectations. While being better than the single packet solution, the performance was on par if not slightly worse than original Snort version. Searching the actual input on the GPU was two to three times faster compared to the CPU, but due to the overhead of data transferring between devices led it to become worse when looking at the total processing time.

When the GPU search was launched, the CPU would wait for the search to finish and the results made available. While the CPU could be spending this time on other processes, it might be that the CPU has nothing else to do as well. To increase the utilization of the CPU, we decided to implement a common technique in areas such as graphics, namely double buffering.

A double buffering scheme was for example used in Gnort [9] to improve parallelism. In computer games it can be used to avoid partially rendered screens to be drawn by showing the image data from the first buffer while writing new data into the second. The buffering in our solution is similar to how Gnort handles it. When the first buffer has been filled and the launch is searched on the GPU, the CPU will continue buffer data from new packets in the second buffer. When the second buffer is full, the CPU will check if the GPU has finished or wait for it to do, before checking results and launching the GPU search on the second buffer. The CPU will now fill the first buffer again with new data and so on. This way the CPU will spend less time waiting and instead work during the GPU pattern matching. The new solution will change the scheme seen in figure 4.2 to what can be seen in figure 5.2 below.

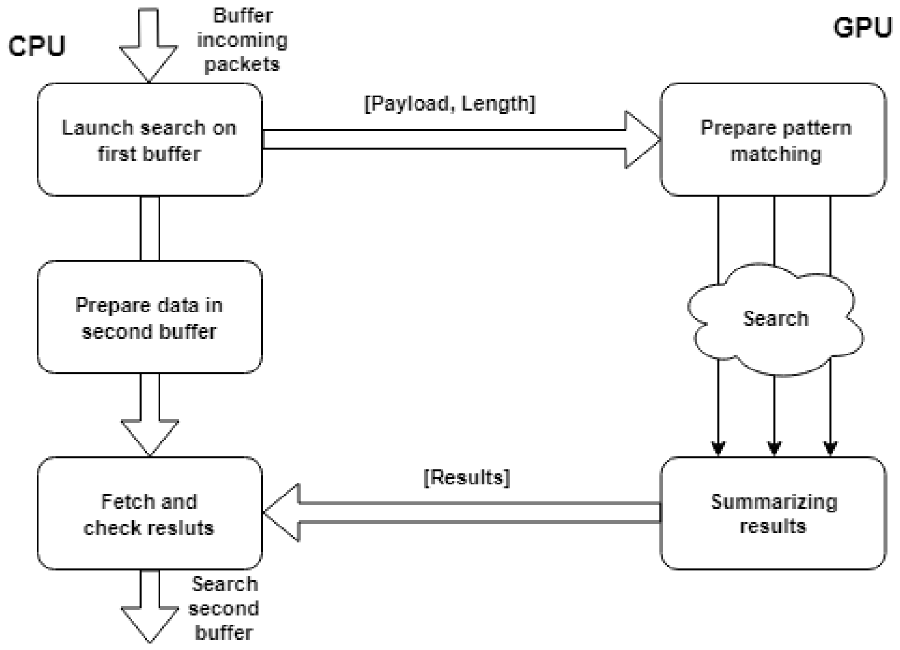


Figure 5.2: The execution flow when using two buffers to store data leading to higher utilization of the CPU.

When the last data from a file has been read into memory, or when the process is terminated, we still want to flush the buffers of the data, as was mentioned in section 5.5. In the flush search the CPU will have to wait for the GPU to finish since no more data will be stored and the machine will be destroyed afterwards to free memory.

Since the biggest gain from this solution is that the CPU spends less time waiting, the initial buffer size could be split in two to limit the memory consumption of the new scheme. This way it is still possible to run the new solution with the same memory requirements as the previous solutions.

Beyond the higher CPU utilization, the only difference in this solution is that the results from the GPU matching is returned later than in the previous ones. The

]

CPU will continue with the packet buffering in the empty buffer and will not check for the results from the GPU until the second buffer has been filled. When the CPU has filled the available buffer, it will check the results or wait if the GPU has not yet finished. After handling these results, the CPU will launch the new search and then continue with the previously searched buffer that is now again available.

6

Evaluation

In this chapter we will evaluate the modified version of Snort that has been developed during this thesis. By measuring the time to finish processing of a finite stream of network packets we can see how the original version stands versus the modified. This will be tested by simulating live traffic via Tcpreplay but also by parsing PCAP files that Snort handles by default. The experiments run will be performed on the ODROID-XU4 that has been described earlier in section 4.4.

6.1 Reference systems

To benchmark and evaluate our implemented optimization methods, we decided to use two reference systems, to which we compare our results. The first one being the original, unmodified version of Snort 3 and the second being a slightly modified version of the original. The decision to use the original version as one reference system was obvious since it is the basic system from which we started.

However, since the execution flow of the implemented multi-packet version, described in Section 5.5, differ significantly from the original version, we decided to use one additional reference system beyond original Snort. The main difference between the two systems is that the original version of Snort is performing a match phase if a pattern is found in the search. Due to the reasons explained Section 5.6, the match phase is totally ignored and excluded in the multi-packet version. Moreover, batching the payload into chunks before it is examined as done in the multi-packet version also changes the execution flow significantly. Thus if the original version of Snort was the only reference system it would be hard determine if a potential performance gain would come from skipping the match phase, buffering, or from having the execution on the GPU.

To get a more precise evaluation of the impact of outsourcing the pattern matching to the GPU, we developed a modified reference system. This modified reference system is similar to our multi-packet version in the sense that it stores data in a buffer and searches the entire buffer in one go. However, instead of using the multi-threaded GPU approach for the pattern matching this system is, likewise to original Snort, using a single-threaded CPU approach instead. Moreover, to measure the impact of the GPU approach, the modified system also excludes the match phase. Since the modified system also stores the payload data, it also uses the same flushing mechanism as described in Section 5.7. This basically means that the system examines all the remaining data that is stored in the buffers in a so called flush before the system terminates.

In the following Sections the original version of Snort is referred to as Snort ORG, and the modified reference system as Snort CPU. Furthermore, our double buffering version is referred to as Snort GPU (double) and the single buffering version as Snort GPU (single).

6.2 Datasets

It is possible to use network traffic that has been pre-recorded in capture files known by their extension packet capture (pcap). These files contain necessary information to recreate network packets. These files can be used by software to either analyze the recorded traffic or replay it as live traffic. Both of these features will be used to evaluate our modifications of Snort.

For the evaluations we decided to use two different sources for the datasets. The first of these are two smaller samples from Appneta, the current developers of Tcpreplay. The two capture files, named SmallFlows and BigFlows, contains some varied application data [30]. It is not specified if they contain any malicious traffic so the assumption is that they do not.

The other source are capture files from ISCXIDS2012 [31, 32]. These datasets are specifically designed to simulate real traffic in order to test and evaluate IDSs. These capture files are larger, ranging from just a few up to several gigabytes. The traffic were captured as a means of giving an alternative to older, static datasets that are outdated compared to current network trends such as the one for the DARPA [33] evaluation.

6.3 Reading pcap files

Snort comes with the feature of reading network packets from a capture file as if the traffic came from a live network. We used this to debug our modified Snort during development and also to do some of the evaluation tests. For these tests, the datasets described in section 6.2 were downloaded and stored locally on the ODROID-XU4.

Snort's DAQ, the library responsible for the packet capture, will use the information in the capture file to recreate the packets which Snort can analyze, just as when inspecting a real network interface. These packet are processed by the running reference system in the way that Snort has been modified to handle them. Since everything in this test is handled locally, there can be no packet losses occurring during these tests.

Since the ODROID-XU4 is a 32-bit system there is a limit on the sizes that Snort can read, which is set to two gigabytes. This affects the capture files from ISCXIDS2012 since each of the captures were larger than this. To be able to include them in our tests we had to make these smaller. We used Wireshark [34], a network analyzer software, to split the capture files into several files where each contained only a subset of the original file.

Another limitation is the available space on the ODROID-XU4. There was about 6GB of available space on the device before downloading the datasets. This means that even if the capture files were fragmented into smaller files, it would not be able to store much larger files on the device. For this reason, the largest dataset tested in this case has 4,22 GB of data.

In these tests, each version is completely silent from alerts, meaning that matches occur but they will not be notified back to the user. The idea is that the process will only be occupied searching the payloads and not spending additional time printing output. The tests are run using each Snort version to search a capture file and measuring the execution time. The datasets used for these tests are presented in table 6.1.

Name	Details
SmallFlows	Appneta sample on 9.4 MB data with 1209 flows over a 5 minute duration.
BigFlows	Appneta sample on 368 MB data with 40686 flows over a 5 minute duration.
ISCX12 121	The first part from ISCX2012 on 13 of June, 634 MB of data from dataset without malicious activity.
ISCX12 131	The first part from ISCX2012 on 12 of June, 1.01 GB of data from dataset that includes activity from network infiltration.
ISCX12 12 Full	The entire file from ISCX2012 on 12 of June, 4.22 GB of data from dataset that includes activity from network infiltration.

Table 6.1: The datasets used for testing Snort via reading prerecorded files.

6.3.1 Single versus double buffer

Before we started comparing the GPU version to the CPU versions, we did a test to ensure that our double buffer implementation performed better than the single buffer counterpart. The results from this experiment can be seen in figure 6.1 and shows that the double buffer version performs better than the single buffer version.

Additionally, the performance gain in these tests are increasing with time, meaning that searching more data will give a larger difference between the two results in favour of the double buffer version.

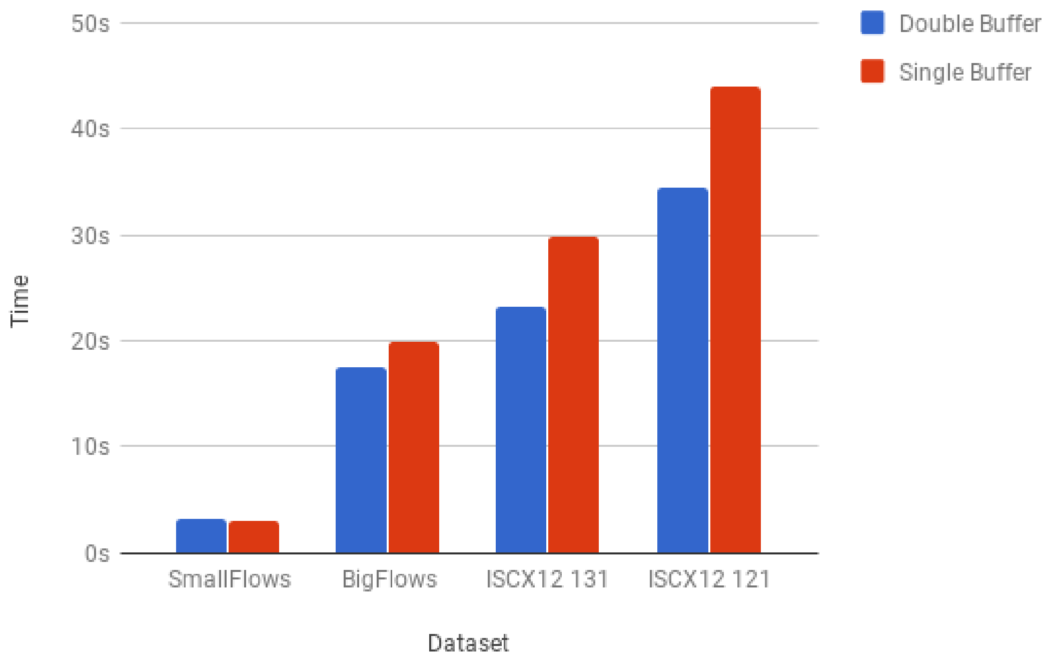


Figure 6.1: Experiment 1: Execution time for the two GPU versions, one utilizing a single buffer and the other using two.

6.3.2 GPU versus CPU

Knowing that the double buffering solution is the superior in terms of speed, we continue to present the results between this one, the Snort GPU (double) version, and the two CPU versions. The same tests were performed to see the difference between the two CPU versions and the GPU version. As can be seen in figure 6.2 and table 6.2, with the increase of data size to search, the GPU solution performs better compared to the two CPU versions. For *SmallFlows* the GPU performs worse than the two CPU solutions, while the GPU performs slightly better when searching *BigFlows*, however less significant than the two bigger ones.

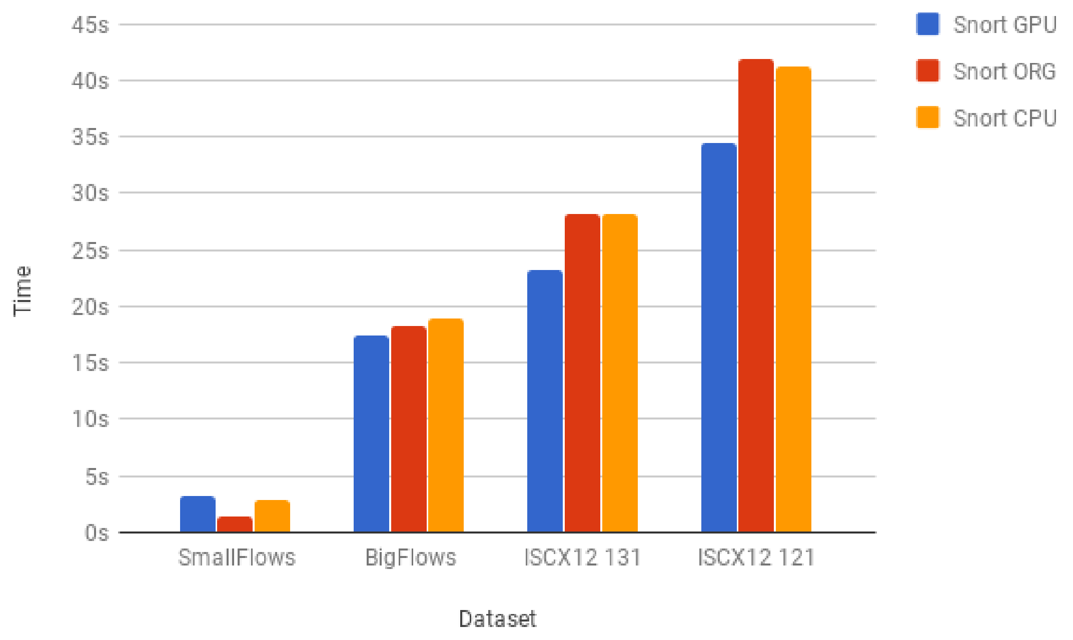


Figure 6.2: Experiment 2: Execution time for each of the Snort versions together with the dataset that was searched.

	SmallFlows	BigFlows	ISCX12 131	ISCX12 121
Snort GPU	0.412	1.041	1.208	1.215
Snort CPU	0.480	0.964	0.996	1.018

Table 6.2: Speedup based on the results presented in figure 6.2. Snort ORG is the reference model to which the other two is compared.

This was then followed by doing the same test on a larger capture file. We used the entirety of the captured traffic from June 12 of ISCX2012. Since this capture file totals on 4.22 GB, it still had to be split into smaller files so that Snort could handle the size. Snort reads the fragments of the original files instead of one large file. These are handled in succession without reinitializing, simulating a continuous run over the capture file.

The results from this test can be seen in figure 6.3 and table 6.3. As can be seen, they further supports that the GPU performance better, and also that the gains increase with time. In this case the GPU version performs the searching about 1.3 times faster compared to original Snort.

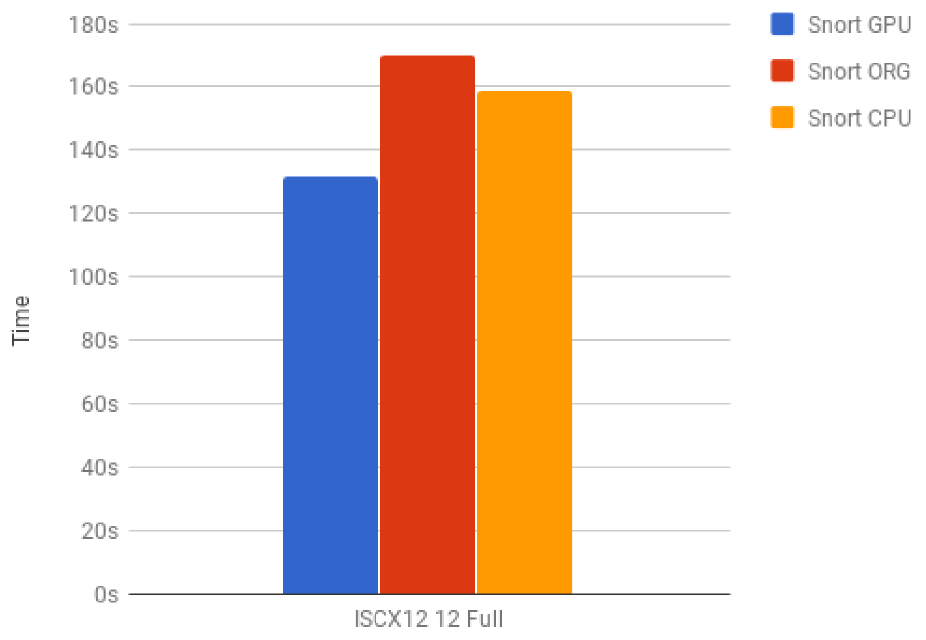


Figure 6.3: Experiment 3: Execution time for each of the Snort versions searching all the capture files from ISCX2012 12 of June.

	ISCX12 12 Full
Snort GPU	1.291
Snort CPU	1.069

Table 6.3: Speedup based on the results presented in figure 6.3. Snort ORG is the reference model to which the other two is compared.

6.3.3 GPU optimal setting

We followed the previous tests with multiple runs of our Snort modification to ensure that the initial test was executed with the optimal settings. There is a hardware sweet spot where the time it takes to search data on the GPU, prepare data on the CPU and the handling of data between the two takes approximately the same time. In this setting, will be waiting the least for each other and utilization will be highest. By varying the buffer size and the amount of threads searching the data, we got the results presented in table 6.4 which are also summarized in figure 6.4.

As can be seen in these results where the BigFlows dataset is being searched, the setting we used, 768 threads and 2M buffer, produces one of the best timings. It is only beaten in the test run using a buffer size of 1 million characters with 768 threads searching the buffers. However, the difference is only 20 ms which could be considered negligible and on par with the one previously used. We can also see that the gain is very large compared to using a single thread on the GPU which is an obvious observation since the point of the GPU is parallelism. Finally, we can see that the timings for different tests do not vary as much as one might think, such as when the thread count is increased on same buffer size, but this can be due to a multitude of different factors which will be discussed more later.

Threads	**Buffer size (chars)**					
	10k	**100k**	**500k**	**1M**	**2M**	**5M**
1	582	-	-	-	-	-
64	48,336	19,632	17,800	18,038	18,133	18,328
128	47,380	19,665	17,345	17,145	17,176	17,450
256	46,417	19,460	17,325	17,193	17,117	17,321
512	47,066	19,489	17,345	17,042	17,078	17,130
768	46,195	19,279	17,217	16,863	16,884	17,269
1024	47,036	19,460	17,240	17,147	17,040	17,177
1536	46,701	19,464	17,162	16,950	16,853	17,131

Table 6.4: Execution times when testing different threads and buffer sizes. The dataset used is BigFlows and the time data is given in seconds.

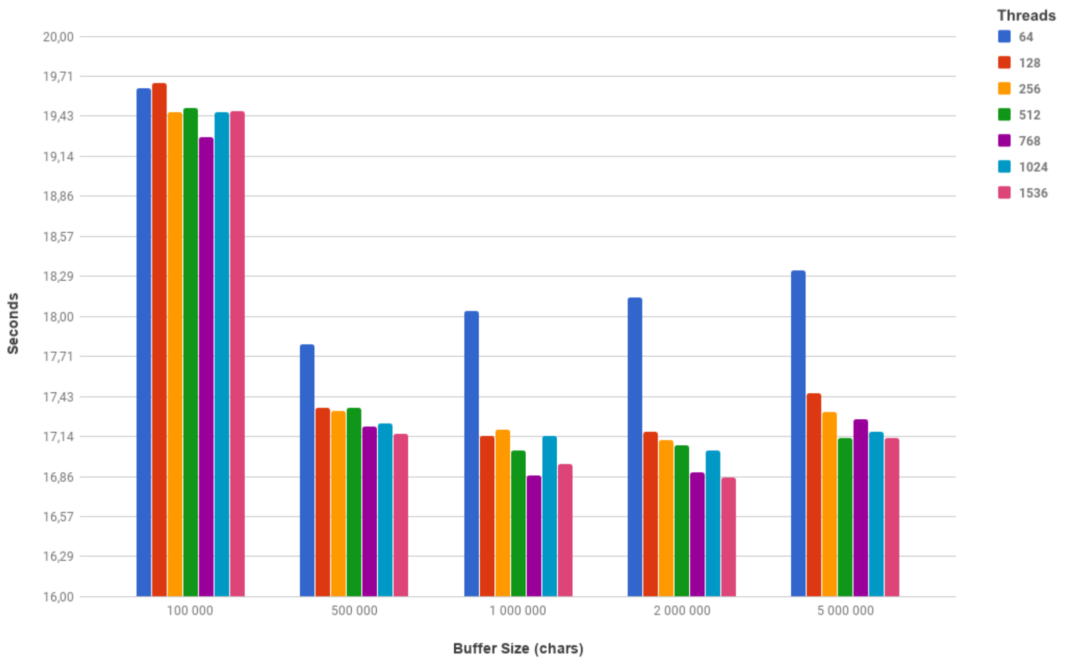


Figure 6.4: Data from table 6.4 grouped by the buffer size for all thread amounts.

6.3.4 Ruleset variations

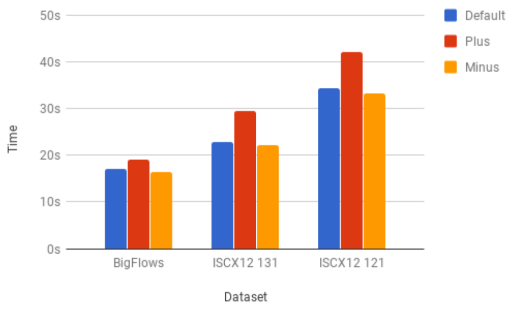
We also wanted to test how our Snort modification is affected when loading different rulesets. Up until this point, all tests has used the official Snort community ruleset that can be downloaded from Snort. By default, this set contains a lot more rules that are commented out. The reason that rules might not be in use is decided based on the severity of the vulnerability, if it has been patched, and more which is based on a couple of policies [35].

By editing the community ruleset by either commenting rules to use less, or uncommenting to use more, we modified the default ruleset into two additional rulesets. We call them Community+ and Community-, the first having 400 additional rules while the latter has 400 less compared to the original. This also affects the size of the state machines that Snort builds in order to search the payload data. What rules to enable or disable was random since the only purpose was to increase or decrease the size of the state machines. The difference between the three rulesets can be seen in table 6.5.

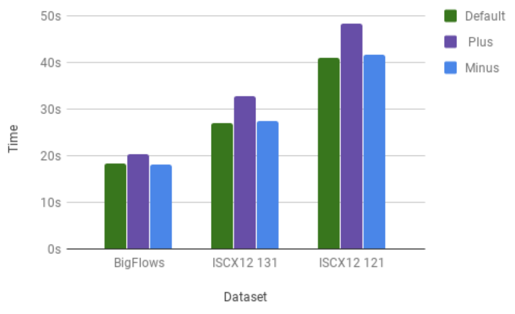
	Community	Community+	Community-
Total rules	829	1229	429
Patterns	1649	2413	857
Pattern chars	34054	45509	17360
States	26083	33949	13357
Transitions	487953	761917	162656
Number of machines	74	100	63

Table 6.5: The three rulesets used in the tests, all created from the community ruleset. The information given is the summary from all of the created state machines.

By running similar tests as in the previously mentioned but with the different rulesets, we measured how the different rules affected the execution time. The results from the tests can be seen in figure 6.5 where each version is run with the three rulesets. From a performance perspective, the GPU version is still faster than the CPU version for all the different rulesets. The larger ruleset, noted as Plus in the figure, has a significant increase of execution time for both versions. Interestingly enough, the smaller ruleset has very little impact compared to the default ruleset. All of the results seems to form a pattern that emerges for both the CPU and GPU versions where the increase is larger for the Community+ ruleset while little to no difference for the two other.



(a) GPU double buffer



(b) Original Snort

Figure 6.5: Execution time for Snort GPU and Snort ORG using the different rulesets for different datasets.

6.4 Tcpreplay

We wanted to evaluate our modified version of Snort in a different way than just reading local pcap files. It is possible to replay pcap files as live traffic by injecting them onto the network as real traffic which better simulates a real network environment. Tcpreplay is a software tool that allows the user to do just that, replay recorded network traffic back onto networks, hosts and more [36]. It is a widely used tool when it comes to testing IDSs, IPSs and firewalls and it is also used in projects like Gnort [9], which is closely related to this thesis.

The main idea of using Tcpreplay in our case was to test the throughput rate and to evaluate how well our modified version of Snort performed in a simulation of a real environment. This means that we wanted to test what receiving rate of our modified version managed to handle before it starts to drop packets while comparing these results to the reference systems stated in section 6.1. As stated in the introduction to this chapter, our modified version of Snort was implemented on an ODROID-XU4 and as transmitting unit we used a desktop computer equipped with a Gigabit Ethernet network card. To avoid any potential noise or traffic from other networks, the wifi was disabled on both devices and they were instead connected with a crossed over Ethernet cable creating a host to host connection. Additionally, to exclude any possible temporary performance peeks for a single run, we did run each test three times and calculated the average value of these tests.

The dataset we used for these tests was one single packet, a HTTP GET request packet, extracted from the BigFlows set that was presented in section 6.3. This packet was later looped for 1,500,000 times, which means that we sent the same packet 1.5 million times in every test. The reason for looping a single packet was due to the previously described datasets produced strange results when being replayed. These datasets got results with a varying amount of packets that mysteriously disappeared somewhere between the network card and the libpcap library on the receiving device. While the single packet dataset also had disappearing packets, they were more consistent and only occurred at higher transmission rate. The results from Tcpreplay can thus be considered somewhat uncertain and this problem will therefore be further explained and discussed in section 7.2.

In table 6.6 the amount of disappearing packets can be viewed. A couple of hundred will begin to disappear at the 500 Mbit/s mark while drastically increasing with higher transmission rate. The amount is also higher when using two buffering versions for Snort, where the GPU solution is the version where most disappear.

System	**Mbit/s**			
	400	**500**	**600**	**700**
Snort ORG	0	114	11845	341967
Snort GPU	0	346	278606	492201
Snort CPU	16	188	151835	411661

Table 6.6: The amount of packets disappearing when when replaying the single packet dataset at varying transmission rates.

Since we know how many packets that were received by Snort, it is not a critical issue that some packets might disappear when evaluating the software. When Snort is not capable of keeping up with the incoming traffic, it will instead drop this and also keep statistics on this data. This way we can replay the dataset and see how much traffic Snort will drop depending on the transmission rate. The amount of dropped packets for each version at different speeds can be seen in table 6.7. These results further validates the expectations from the previous tests. Since each search happens faster, Snort should be able to handle traffic at higher rates. The results show that the GPU solution is analyzing a higher amount of packets on all the transmission rates. While the results might be a bit unfair towards original Snort version that is running the match phase, the GPU version is still performing better compared to the Snort CPU version that does not.

The reason why higher transmission rate leads to less packet drops in Snort is because of the fact that more packets are disappearing at the higher transmission rates. Since less packets are actually caught by Snort, each version will have less packets to analyze and therefore less packets will be dropped. So while the numbers seems to be better in the table, in practice they are actually worse.

System	**Mbit/s**						
	150	**200**	**300**	**400**	**500**	**600**	**700**
Snort ORG	40323	409845	777297	958437	1067646	1125218	828593
Snort GPU	0	0	269	230105	479613	359785	267525
Snort CPU	0	0	182189	515033	700532	689525	486032

Table 6.7: Amount of dropped packets for each of the different tests.

To visualize the results, we also present these in graphs in figure 6.6 and figure 6.7. The first of these, figure 6.6, shows the percentage of packets that were not analyzed based on the packets sent from the transmitting device. This means that the drop percentage includes both the disappeared packets as well as the packets dropped by Snort. The figure shows that the Snort GPU version handles higher transmission rates without dropping packets and also performs better when the transmission rate is further increased.

The second figure, figure 6.7, shows the percentage of dropped packets based on the amount that was actually caught by the running Snort version. As previously mentioned, more packets are disappearing at higher transmission rate which leads to less packets to handle by Snort. This is why the amount of packets dropped are decreasing at the higher transmission rates. Still, the same results are presented here, where the GPU outperforms the other two CPU versions at all the different tests.

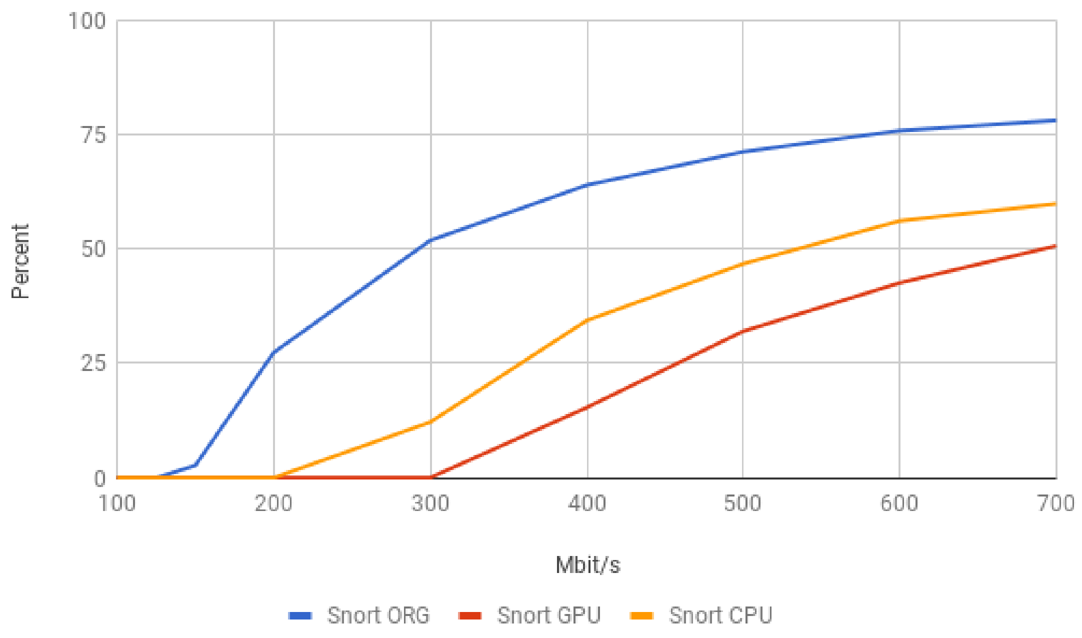


Figure 6.6: The percentage of unhandled packets, including both the disappeared as well as the dropped packets.

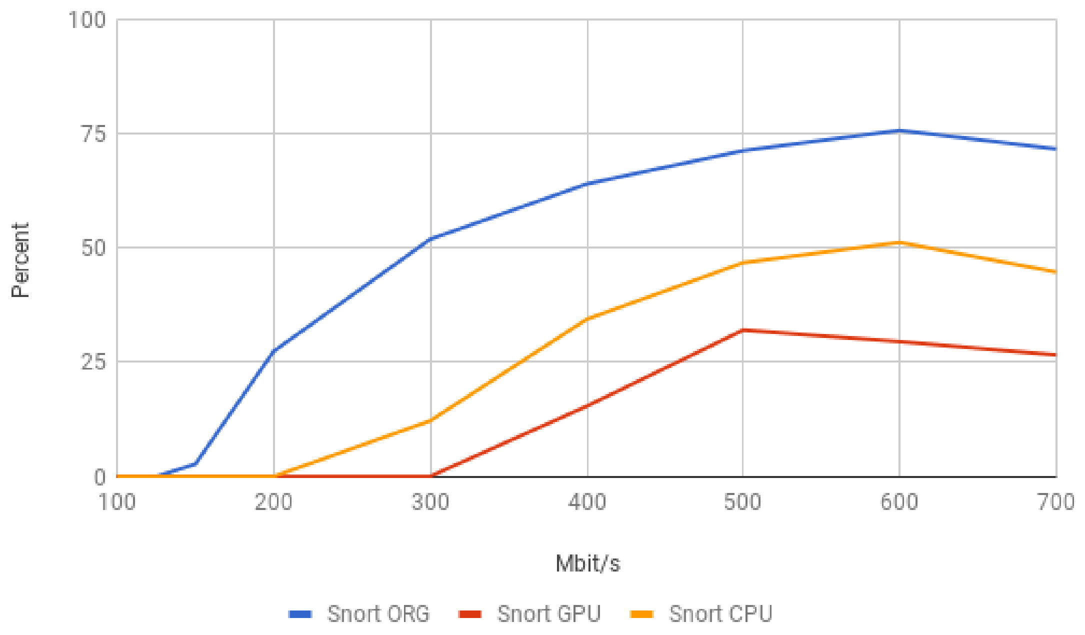
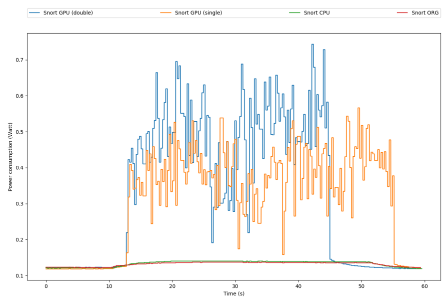


Figure 6.7: The percentage of dropped packets from those that were actually caught by the Snort version.

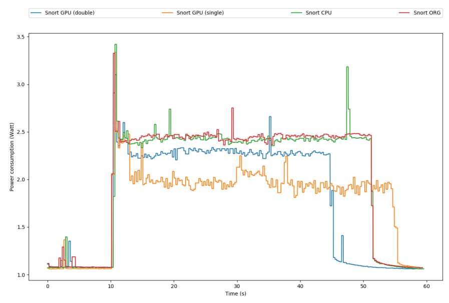
6.5 Power consumption

The final tests we did was to measure and compare the power consumption of all the reference systems. ODROID-XU4 is unfortunately not equipped with power measuring sensors, so the power consumption tests were conducted on an ODROID-XU3 [37] instead. ODROID-XU3 is equipped with the same processor setup as well as the same GPU as ODROID-XU4, which makes it a legitimate substitute to the ODROID-XU4. The only significant difference (for the power consumption tests) between these two hardware systems is the RAM speed and the memory bandwidth. The RAM speed of the ODROID-XU3 is 933Mhz and the memory bandwidth is 14.9GB/s, whereas the RAM speed of the ODROID-XU4 is 750Mhz and the memory bandwidth is 12GB/s.

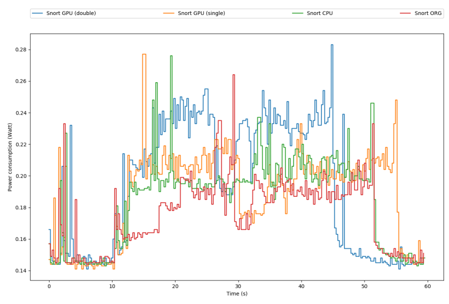
The tests were conducted in the same way as described in section 6.3, which means that we measured the power consumption when the system was processing a pcap file. The dataset used in these tests were the ISCX12 131(1.01 GB). The reason for choosing this dataset was that it was the dataset with the largest data size and the longest execution time and thus would even out potential power spikes in the measurements. The script we used for the measurements was developed by Simon Kindström for his master thesis paper [38]. We measured the power consumption of the following four components: CPU (a15), CPU (a7), GPU and RAM memory with a sample rate of 100 samples/second. To show the idle power consumption we started the execution of each system 10 seconds after the measurements were started. We stopped all measurements after 60 seconds which was enough to capture the entire execution time for each systems including some idle time after Snort terminates. All measurements are visualized in figure 6.8. Note that there are different scales on the power consumption axis (y-axis) in the different figures, this is to show the variation of the power consumption better.



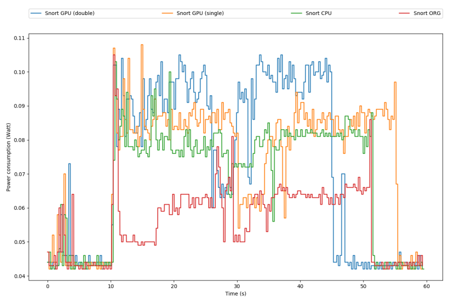
(a) Power consumption on GPU



(b) Power consumption on CPU(a15)



(c) Power consumption on CPU (a7)



(d) Power consumption on RAM

Figure 6.8: Power consumption measurements of the CPUs, GPU and RAM.

In (a), which visualize the GPU power consumption, it clearly shows that the GPU version with the double buffering scheme has a highest power consumption during the execution. The second highest power consumption has the GPU single buffering version and as expected both of the CPU versions have a GPU power consumption of close to zero Watts.

The power consumption of the CPU (a15) is presented in (b). It shows that the Snort CPU and Snort ORG version are almost equal during the the whole execution time and this is also the highest power consumption. The third highest power consumption has the double buffering version and the system that has the lowest power consumption is the single buffering version.

In (c), which shows the power consumption of the second CPU(a7), the power consumption is quite evenly distributed between all the systems. However, the double buffering has the highest power consumption during its execution time and Snort ORG has the lowest. A comparison of the power consumption of the two

CPUs, (b) and (c), clearly shows that the CPU (a15) is utilized more than the CPU(a7).

Finally (d) shows the power consumed by the RAM. Similar to (c), this figure also shows a quite even distribution of the systems with one exception, Snort ORG, which has a significantly less power consumption than the other three systems.

To evaluate the overall power consumption we also made a plot of the power consumption of all components combined, which means that for each sample, we summed the measured values for every component. The result of this is presented in figure 6.9. It shows that during execution time, the double buffering version has the highest power consumption and the single buffering has the lowest. In between these systems are Snort CPU and Snort ORG which almost has the same distribution.

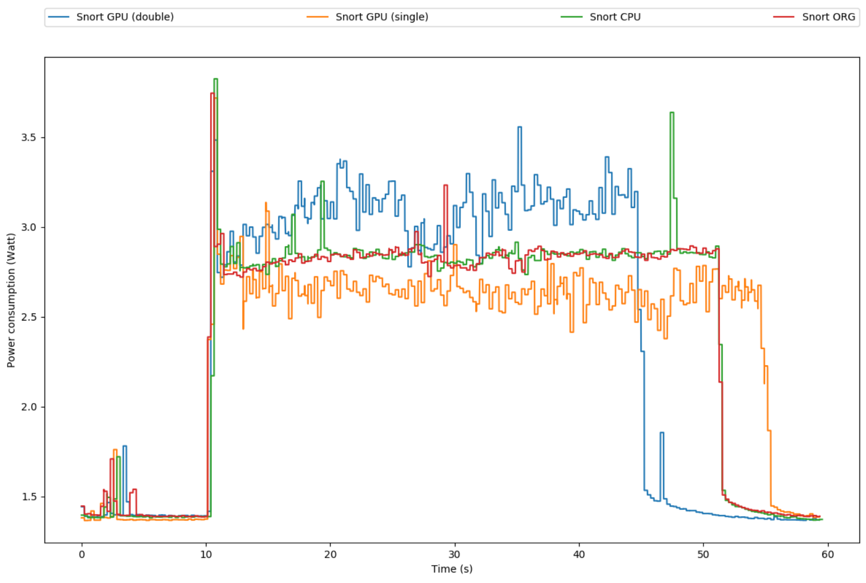


Figure 6.9: The combined power consumption of all four reference systems.

Note that the execution time for the double buffering version is the shortest, which can be visualized in figure 6.9 as well as in figure 6.8. To evaluate which system that was the most effective we also calculated the average power consumption and the numbers we got are presented in table 6.8. This table shows that the system with the lowest average power consumption is the GPU single buffering and the second lowest is the double buffering system.

Reference system	Power Consumption (W)
Snort GPU (double)	2.4106
Snort GPU (single)	2.3712
Snort CPU	2.4281
Snort ORG	2.4271

Table 6.8: Average power consumption for each of the Snort versions.

7

Discussion

In this chapter we will discuss the functionality and results of our Snort modifications. Things such as how the solution could be improved or extended, the impact of not working entirely as original Snort and potential issues with using GPUs for computation.

7.1 Results

From the capture file testing described in section 6.3 we could see that the Snort GPU (double) implementation searched the capture files faster compared to the two CPU versions of Snort used as baselines. We are certain that the modified version can be improved further with more sophisticated transferring of data and better understanding of the frameworks included.

Each search seems to be faster on the GPU compared to the searching on the CPU, which is the reason why the modified version outperforms the CPU versions. However, this is only true for the double buffering implementation that prepares the next buffer when the search is performed on the GPU. If the buffering and searching is run sequentially as in the original multiple packet implementation, the gain is little to none. This idea also supports why the GPU version is not as beneficial for the smaller datasets. As was mentioned in section 5.5, the flush version can not be made in parallel as there will be no more data to handle. Therefore the CPU will wait for the GPU to finish searching in this case, and continue first when all data is made available. This yields lower CPU utilization and takes longer time to finish compared to the parallel approach.

Some hardware limitations will also be a factor to why the improvements might be less than other works. This is mostly due to comparing our work in an IoT context to those evaluated on desktop computers with more powerful hardware. Nonetheless, one bottleneck for our modification could be the GPU using shared RAM memory since it has no dedicated memory of its own as was also mentioned in section 4.4. The Aho-Corasick algorithm would most likely benefit significantly from faster memory due to its highly random reads (the matches can not be predicted and are often varied) that creates many cache misses.

It is also worth mentioning why results in figure 6.4 sometimes prove better for lower parallelism. As mentioned in section 6.3 there is a sweet spot between the CPU and GPU that performs best when the utilization of both devices are the highest. Another thing is that the dataset also could impact these results. For example, if the buffer is very large when using smaller capture files, the traffic that is to be searched by a machine has a higher chance of fitting into a single buffer. In this case the data has to be searched by the flush function which is slower than the ordinary search since the CPU has to wait for the search to finish. When running more threads it is also possible to get an unnecessary overhead. There is no limit to the amount of threads that can be set so it might be way higher than the GPU in the system can handle. In this the GPU would reinitialize finished threads to do the work that has not yet been started. If that is the case, the Snort modification is ending an already running thread just to start it over for searching another part of the input which seems pointless. While this could be a factor, there is no way of determining if this was the case or not. The ruleset tests also further supports the same findings that the GPU version performs better than original Snort. We also see in figure 6.5 that the GPU and CPU measurements follow the same pattern. We think that the increase of state machine instances in the Community+ ruleset is the biggest impact on the performance. For one, this results in more setup time for all Snort versions since more machines needs to be created. Additionally, each state machine in the GPU versions holds a buffer, and with more machines there will be more buffers that the network traffic is separated into. With less traffic per buffer, it is a higher chance that some of these state machines has to use the slow flush function when Snort terminates since they never reached the threshold where the search is launched. Nonetheless, since the CPU and GPU version behave similarly for the same rulesets, we consider this effect to be reasonable.

While we had some errors using Tcpreplay, we still think that the simulation of a real network is something to include and discuss. The results from these tests validated the expectation that the GPU searching was faster since the GPU version of Snort analyzed more of the incoming packets. Since there were no issues when replaying the same packet on rates less than 500 Mbit/s, we believe that this was not dependent on Snort. For this reason we expect similar results when running these tests without any packets disappearing. The issues with the Tcpreplay tests and what we did to analyze them are further discussed in section 7.2.

Finally, the results of the power consumption measurements shows that the single buffering version has the lowest average power consumption, followed by the double buffering version. The differences between all versions are however very small and the difference between the single buffering version and the original version of Snort was 2.3%. One interesting thing to note in the power consumption results is that the double buffering version is using the most energy during the execution time of Snort but the execution time is shorter than the other versions. The fact that it has a shorter execution time makes the double buffering version more energy efficient than the original version of Snort for this specific dataset. However, even though we did not test the power consumption for other datasets, we assume that the energy efficiency for the double buffering version would increase for larger datasets and decrease for smaller ones. This assumption is based on the power consumption tests as well as the execution time tests which showed that the performance of the double buffering increased with the size of the datasets compared to the original version of Snort. Having this in mind we think that the double buffering version is more suitable for crowded networks with higher traffic load, whereas the single buffering version would be more efficient in networks with less traffic due to it being more energy efficient.

When considering all of our results, we believe that GPU based searching absolutely is a possible next step for more IDSs, even in IoT context. While the Tcpreplay tests might be considered uncertain due to packets disappearing, the local parsing of pcap files shows a performance gain with our solution. Additionally, with more time and insight, we are certain that the Snort GPU version could be improved further to reap even larger benefits.

7.2 Tcpreplay issues

When we were evaluating our system using Tcpreplay, we ran into some issues regarding mysteriously lost packets while using the prerecorded datasets described in section 6.2. To elaborate, we transmitted a dataset containing 14 261 packets (SmallFlows) with Tcpreplay in 100Mbit/s, and only roughly 12 700 of those packets was received by Snort and the libpcap library on the ODROID. This could of course be the case if the rest of the packets were displayed as dropped but they were not. In some suspicious way these packets were neither received nor dropped. Our first thought was that the lost data were blocked by the Linux built-in firewall, Iptables. However, this was not the case since the firewall policies were set to allow all traffic, leaving this option of the table.

To exclude the possibility that the issue was caused by the transmitting side, we recorded all the outgoing data traffic from the desktop with a packet capturing software called Wireshark [34] that was running on the same machine. However the results from Wireshark showed that all packets were transmitted. To further investigate this we installed another packet capturing software on the ODROID, namely Tcpdump [26]. The reason why we did not use Wireshark on the ODROID as well, was that the installation failed. This was probably due to limitations in Ubuntu-mate [39], which is the operating system installed on the ODROID. The tests with Tcpdump activated, likewise Snort, showed that we only received approximately 12 700 packets.

Since the source and destination IP and MAC addresses in the datasets were not matching the ODROIDs nor the transmitting computer, we also enabled promiscuous mode on the Ethernet network interface on the ODROID. Enabling this means that the network interface will allow all packets to be received and not discard any traffic even though it is not intended for that device. Enabling promiscuous did not change the outcome and we still lost the same amount of packets.

Furthermore, we rewrote all the source and destination addresses of all the packets in the dataset (SmallFlows) to match the devices used in the test. This was done with a tool called Tcprewrite, which is tool that is included in the Tcpreplay software. The rewritten dataset was stored in a new capture file (pcap) and the new addresses were checked and confirmed to be correct by reading the new pcap file with Wireshark. When this file was replayed the amount of traffic captured by Snort as well as Tcpdump increased to approximately 22 000 packets. This traffic is however including the outgoing traffic, which means that it records all the responses to the received packets. We recorded all the responses on the desktop computer, and most of them were reset messages to unidentified TCP connections. Subtracting the outgoing messages from the 22 000 showed that we now only received roughly 11 500 packets.

To investigate the issue on a lower level we installed a software called ethtool [40], which shows statistics from the network card driver instead of at the kernel level where Snort and Tcpdump record their traffic. Using ethtool showed that the network interface is receiving all the 14 261 packets transmitted and does not report any dropped packets. This means that the packets are lost when the network driver is transferring the packets to the libpcap library. The problem seems to be similar to one of the problems stated on the Tcpreplays FAQ page [41], where packets are lost but not counted as dropped by packet capturing software like Tcpdump. On the page they also state that the problem probably is related to either the OS kernel or the network driver and may or may not be solvable. In our case we were not able to fix it, so we instead choose to loop a single packet multiple times to represent the same amount of traffic.

7.3 Keeping Snort functional

Both the single and double buffer versions of the multiple packet solution were incompatible with the original work flow of Snort. This has to do with Snort's sequential design which was ultimately based on the DAQ library that feeds packet one by one to Snort from the network card. Snort is not saving packet information after a packet has been handled, which in our case is the buffering. Due to this it was not possible to use the match phase in Snort that relates a pattern match to the packet it was found in, the same function that the generation of alerts and handling of logs depends on. While no solution to this has been implemented, we believe that it can be solved in some way.

A naive approach would be to copy all of the data in the packet structure inside Snort and relate the match phase to this stored data. This would need a separate copy function for each of the structures contained within the packet structure since there is no supplied functionality of copying these. This solution is highly likely to be too inefficient to use. Packets arrive at high speeds and spending time to copy data that is already available will take more time than the GPU can save, at least with the hardware used in this thesis.

Another possible solution would be to extend the modified version with some software that stored the required information about the packets. When a match would be detected in the GPU search, it would share this data to the extension which in turn could alert the user. However, how the actual handling of data between these two parties has not been detailed any further.

A different approach would be to modify not only Snort but also the DAQ library. By changing DAQ we might have been able to keep the packets in memory and signal back when the search had finished to clear the buffered packets. This way Snort would still work as in the original version by performing the match phase, however it is unclear exactly how much memory this would require. This depends on the number of state machines that would need to buffer packets and the amount of data each would buffer. Considering that the standard community rules builds about 70 state machines it would have a considerable upper limit. For example, if each machine held the average size of just the buffers payloads in the current implementation, around 20 MB, this would still allocate 1.4 GB in memory.

The question is then if alerting matches, without relation to the packet, is enough to consider a system to be protected. This depends on the environment as it affects what kind of traffic to expect, the allowed time period between the arrival of a malicious packet and the alert and so on.

Since packets are buffered to be searched at a later point in time, there is already a latency effect in using the GPU approach. If this would be extended further by additional work it could reach a point where the damage of a network attack has happened before it is detected. This would be less of a problem in an prevention context, where the traffic was not allowed to pass before searched, but then you run into other problems with things such as connections timing out when waiting for a response and so on.

It would be interesting to know if and how the related works, such as Gnort [9] and OpenCL Snort [25], have solved this problem. In the case of Gnort, it seems that alerts are raised on content matches, basically skipping the match phase and raising alerts for every match the GPU has found, which is similar to what we do. This can also be that Snort handled the match phase differently in older version of Snort. For the second project mentioned, there is no specific information if and how this problem is handled, or if it more or less ignored the match phase.

Finally, while the results were better for the double buffering implementation described in section 5.7, it is still interesting to consider usage of the standard multiple packet solution as well. Since the latter implementation forced the CPU to wait during the GPU search, it could be that this implementation is useful in a situation where a system needed more CPU offloading for other tasks. We are unsure if the current OpenCL functionality is actively waiting or blocking, but the idea is to make the process blocked until the GPU search is finished. During this time, the CPU would work on relevant tasks, for example gathering or sending sensor data. In theory, this would allow the the device to supply the same service while also running an IDS at a lower CPU utilization. Of course, this would decrease the performance of the IDS and have to be balanced based on the amount of incoming traffic.

7.4 Security in GPUs

When new functionality is implemented into existing software, it is important to consider how it affects aspects of the end product, such as the security. While Snort in itself might not be an interesting attack target, it could still be attacked to limit detection of other attacks against the host system. If Snort, or any other NIDS, would be overloaded, then traffic could pass through without being inspected. This is something that could be exploited when trying to attack a system without getting detected.

There are multiple Common Vulnerabilities and Exposures (CVE) related to GPUs, for example there are hundreds CVEs related to NVIDIA GPUs that can be easily found by their CVE ID or keywords [42, 43]. Many of these are denial of service attacks that have a possibility for escalation of privileges, but there are other worse ones as well, for example unwanted code execution.

It is very hard to predict exactly how the GPU could be exploited when running our Snort modification. Beyond Snorts original features, we have added data transfers as well as code execution on the GPU enabled by OpenCL. The security of the modified Snort version therefore is as strong as the weakest part of OpenCL and Snort combined.

However, it still introduces the possibility that the GPU could somehow be used via the modifications made in Snort. It should therefore be considered when evaluating the usage of prototypes such as these, that make use of GPGPU.

7.5 Ethics and sustainability

Intrusion detection systems are a powerful software tool when it comes to detect malicious intrusions and cyber physical attacks, but from an ethical point of view they could be questionable. The ability to perform deep packet inspection (DPI) is probably the biggest advantage of NIDS from a security perspective. Using DPI means that they are not only examining packet headers of the network traffic but they also inspects the payload [44]. Looking at NIDS from a privacy-concerning aspect, the DPI is also the most questionable part of the system. If the network packets inspected by the NIDS is not encrypted they can store all kinds of sensitive information [44], such as personal information as well as business secrets could be stored.

The privacy issue is not a new topic in the field of IDSs, in fact there numerous of papers addressing the privacy issue in all kinds of IDSs [44, 45, 46]. Nonetheless, as the new privacy EU-law, General data protection regulation (GDPR) [47], enters to force as this thesis is written, it makes the topic even more present. The main purpose of GDPR is to put a higher pressure on how European companies store, use and in general treat data containing personal information.

What and how the information is stored differ from NIDS to NIDS but it also depends on which rules and policies that are applied [44]. In theory, a NIDS can store all packets that is routed through it. Consider the following Snort rule:

alert ip any any -> any any (msg:"match any ip packet")

This rule will alert and store information such as the header and payload of every incoming and outgoing packet that uses the IP protocol. Excluding protocol such as ICMP and ARP, it means that a NIDS is able to basically store information from every packet that passes through it with a single rule. To store every packet like this would not be practically in the long run, since it would fill up the hard drive rapidly if the NIDS is exposed to a crowded network with a large amount of traffic. There are also other rules that are not as generic and thus not as data-consuming as the previously mentioned, that can make the IDS store sensitive information [44].

However, the main point is that if the person that installs the NIDS and applies its rules and policies is not careful and has the privacy aspect in mind, the NIDS can store sensitive information that it is not suppose to. Therefore it is of highest concern that the ruleset applied has a high precision rate to only detect actual intrusion and to avoid false alarms that causes the NIDS to store information that it is not intended to do.

As described in section 5.6, our multiple packet solution does not use the match phase of Snort and thus only stores the total number of matches and the number of matches for each pattern. The reason for this is explained in section 7.3 and should not be accounted as a feature. However, from a privacy-concerning aspect, our multiple packet solution could be considered to have less of these concerns as it neither store headers nor payloads of the packets.

From a sustainability point of view, we think that IDSs developed for IoT probably will contribute to a more sustainable future. Imagine the consequences if an intruder, similar to *Mirai* [4], gets access to hundreds of thousands Internet connected devices with a wide array of functionality. Depending on this, there could be different consequences. For example, imagine that the controllers of a smart home that control heating and cooling were to be tampered with, or measurement devices being corrupted with incorrect data or even worse scenarios. Basically, this could be used to increase the power consumption which in turn could on lead to a less sustainable future for the environment. It would also lead to finical consequences for the affected households. An IDS developed for IoT could potentially detect such intrusions, giving the companies involved a chance to fix the problem before it affects more households and thus also contributing to the sustainability aspect.

7.6 Future work

This section will mention some possible future works for this projects as there are a couple of directions it could be extended or expanded upon. The mentioned works have not necessarily been investigated and might not be possible but would benefit the Snort GPU version developed in this work.

Our Snort modification is using a simple division of work for the Aho-Corasick algorithm. It was used since Snort already has a fully functioning construction of the Aho-Corasick state machines. We mentioned the modified Aho-Corasick for GPU, namely PFAC [22]. PFAC is not using failure states so the algorithm does not work with original Snort since failure states are already created in the Aho-Corasick state machine. Snort could therefore be modified to skip the building of failure states which would decrease the time for building state machines and also possibly saving memory. With these modified Aho-Corasick state machines it would be possible to use the PFAC algorithm on the GPU and it would be interesting to see the performance differences using this algorithm compared to our solution.

Another work would be to improve the Snort modification for it to handle packets more like original Snort. In the GPU modification we can only handle the payloads as was discussed in section 7.3. Possible ways to keep packets in memory via DAQ, transferring them to GPU memory or other ways to enable evaluation of a match with the packet data would be a nice improvement. This would allow the Snort modification to supply the same service as the original Snort.

It would also be interesting to see the Snort modification developed in this thesis in a real environment. While both pcap files and traffic playback has been used, this does not fully simulate a real environment. Testing the software in a distributed IoT network would be nice to also validate the usefulness of the software that has been developed in this thesis.

8 Conclusion

The number of IoT devices is increasing and becoming more common, and with new functionality comes new security threats to an area where security has not been the priority. To improve upon this issue, the goal of this thesis was to investigate Snort and develop a modified version to run on a single-board computer with better performance using GPGPU techniques. By considering how Snort works and using OpenCL to achieve GPU computing, a modification of Snort was developed to increase the performance of the pattern matching, one of the most computationally expensive tasks in most IDSs today.

Thorough testing of the software presented in this thesis shows a best-case speed up of 1.3 when analyzing pre-recorded traffic. Additionally, the software achieved an improvement of factor two in terms of throughput when the pattern matching of the IDS is performed on the GPU instead of the CPU. The tests also show that the modified version decreases the energy consumption slightly, even when the performance is increased compared to original Snort. Additionally, the software utilizes the CPU as well as GPU, and the specifications of the software can be tailored for the purpose of the system. This could be used to allow devices to run other processes while still having the IDS operating, thus not disrupting the core purpose of the device.

A possible future work could be to implement additional pattern matching algorithms and compare them to these results. Another project would be to evaluate the prototype in real networks compared to the isolated environments that this thesis made use of.

Bibliography

[1] Snort 3 User Manual. https://www.snort.org/downloads [Accessed: 2018-04-09].

[2] Ericsson. (2016, June) Ericsson mobility report. https://www.ericsson.com/assets/local/mobility-report/documents/2016/Ericsson-mobility-report-june-2016.pdf [Accessed: 2017-11-23].

[3] R. van der Meulen and J. Rivera, "Gartner Says By 2020, a Quarter Billion Connected Vehicles Will Enable New In-Vehicle Services and Automated Driving Capabilities," https://www.gartner.com/newsroom/id/2970017, 2015, [Accessed: 2018-03-26].

[4] K. Angrishi, "Turning Internet of Things (IoT) into Internet of Vulnerabilities (IoV): IoT Botnets," *CoRR*, vol. abs/1702.03681, jun 2017. [Online]. Available: http://arxiv.org/abs/1702.03681

[5] G. Shanmugasundaram and G. Sankarikaarguzhali, "An Investigation on IoT Healthcare Analytics," *International Journal of Information Engineering and Electronic Business*, vol. 9, no. 2, p. 11, 03 2017. [Online]. Available: http://proxy.lib.chalmers.se/login?url=https://search-proquest-com.proxy.lib.chalmers.se/docview/1886767215?accountid=10041

[6] Q. I. Ali and S. Lazim, "Design and implementation of an embedded intrusion detection system for wireless applications," *IET Information Security*, vol. 6, no. 3, pp. 171–182, Sept 2012.

[7] F. M. Tabrizi and K. Pattabiraman, "Flexible Intrusion Detection Systems for Memory-Constrained Embedded Systems," in *2015 11th European Dependable Computing Conference (EDCC)*. IEEE, Sept 2015, pp. 1–12.

[8] A. Sforzin, F. G. Mármol, M. Conti, and J. M. Bohli, "RPiDS: Raspberry Pi IDS - A Fruitful Intrusion Detection System for IoT," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE, July 2016, pp. 440–448.

[9] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings*, R. Lippmann, E. Kirda, and A. Trachtenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 116–134. [Online]. Available: https://doi.org/10.1007/978-3-540-87403-4_7

[10] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: A Highly-scalable Software-based Intrusion Detection System," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 317–328. [Online]. Available: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/2382196.2382232

[11] M. Norton, "Optimizing pattern matching for intrusion detection," *Sourcefire, Inc., Columbia, MD*, 2004.

[12] A. V. Aho and M. J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search,” *Commun. ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975. [Online]. Available: http://doi.acm.org/10.1145/360825.360855

[13] H. Debar, M. Dacier, and A. Wespi, “A revised taxonomy for intrusion-detection systems,” *Annales Des Télécommunications*, vol. 55, no. 7, pp. 361–378, Jul 2000. [Online]. Available: https://doi.org/10.1007/BF02994844

[14] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, “Intrusion detection system: A comprehensive review,” *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16 – 24, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804512001944

[15] M. Särelä, T. Kyöstilä, T. Kiravuo, and J. Manner, “Evaluating intrusion prevention systems with evasions,” *International Journal of Communication Systems*, vol. 30, no. 16, pp. e3339–n/a, 2017, e3339 dac.3339. [Online]. Available: http://dx.doi.org/10.1002/dac.3339

[16] D. Mudzingwa and R. Agrawal, “A study of methodologies used in intrusion detection and prevention systems (idps),” in *2012 Proceedings of IEEE Southeastcon*, March 2012, pp. 1–6.

[17] M. Roesch, “Snort – Lightweight Intrusion Detection for Networks,” in *Proceedings of LISA ’99: 13th Systems Administration Conference*, Nov 1999.

[18] R. S. Boyer and J. S. Moore, “A Fast String Searching Algorithm,” *Commun. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977. [Online]. Available: http://doi.acm.org/10.1145/359842.359859

[19] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, “Generating realistic workloads for network intrusion detection systems,” *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 1, p. 207, 2004.

[20] J. B. D. Cabrera, J. Gosar, W. Lee, and R. K. Mehra, “On the statistical distribution of processing times in network intrusion detection,” in *2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601)*, vol. 1, Dec 2004, pp. 75–80 Vol.1.

[21] K. Krewell, “What’s the Difference Between a CPU and a GPU?” https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/, 2009, accessed: 2018-01-21.

[22] C. H. Lin, C. H. Liu, L. S. Chien, and S. C. Chang, “Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs,” *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 1906–1916, Oct 2013.

[23] OpenCL - The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/ [Accessed: 2018-05-14].

[24] N. Jacob and C. Brodley, “Offloading IDS Computation to the GPU,” in *2006 22nd Annual Computer Security Applications Conference (ACSAC’06)*, Dec 2006, pp. 371–380.

[25] H. Xie, Y. Xiang, and C. Chen. (2017, July) Parallel Design and Performance Optimization based on OpenCL Snort. https://www.atlantis-press.com/proceedings/jimec-17/25880643 [Accessed: 2018-06-05].

[26] Tcpdump and Libpcap. http://www.tcpdump.org/ [Accessed: 2018-04-09].

[27] ODROID-XU4 User Manual. https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf [Accessed: 2018-03-28].

[28] ARM Mali GPU OpenCL Developer Guide. https://static.docs.arm.com/100614/0300/arm_mali_gpu_opencl_developer_guide_100614_0300_00_en.pdf [Accessed: 2018-03-28].

[29] Snort FAQ README.daq. https://www.snort.org/faq/readme-daq [Accessed: 2018-04-20].

[30] Tcpreplay sample captures. http://tcpreplay.appneta.com/wiki/captures.html [Accessed: 2018-05-08].

[31] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, “Toward developing a systematic approach to generate benchmark datasets for intrusion detection,” *Computers & Security*, vol. 31, no. 3, pp. 357 – 374, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404811001672

[32] Intrusion detection evaluation dataset (ISCXIDS2012). http://www.unb.ca/cic/datasets/ids.html [Accessed: 2018-05-08].

[33] DARPA Datasets. https://www.ll.mit.edu/ideval/data/ [Accessed: 2018-05-08].

[34] Wireshark. https://www.wireshark.org/ [Accessed: 2018-06-04].

[35] Snort FAQ - Why are rules commented out by default? https://www.snort.org/faq/why-are-rules-commented-out-by-default [Accessed: 2018-06-05].

[36] Tcpreplay Overview. https://tcpreplay.appneta.com/wiki/overview.html [Accessed: 2018-05-11].

[37] ODROID-XU3. http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127 [Accessed: 2018-06-08].

[38] S. Kindström, “Network Intrusion Detection in Embedded/IoT Devices using GPGPU,” Master’s thesis, Chalmers University of Technology, Sweden, June 2018.

[39] Ubuntu MATE For a retrospective future. https://ubuntu-mate.org/ [Accessed: 2018-06-04].

[40] ethtool - utility for controlling network drivers and hardware. https://mirrors.edge.kernel.org/pub/software/network/ethtool/ [Accessed: 2018-06-04].

[41] Tcpreplay - Frequently Asked Questions. http://tcpreplay.synfin.net/wiki/FAQ [Accessed: 2018-05-24].

[42] CVE Details - Nvidia: Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-5264/Nvidia.html [Accessed: 2018-05-14].

[43] CVE Nvidia. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Nvidia [Accessed: 2018-05-14].

[44] N. Ulltveit-Moe, “A roadmap towards improving managed security services from a privacy perspective,” *Ethics and Information Technology*, vol. 16, no. 3, pp. 227–240, Sep 2014. [Online]. Available: https://doi.org/10.1007/s10676-014-9348-3

[45] S. Niksefat, P. Kaghazgaran, and B. Sadeghiyan, “Privacy issues in intrusion detection systems: A taxonomy, survey and future directions,” *Computer Science Review*, vol. 25, pp. 69 – 78, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574013716301204

[46] E. Lundin and E. Jonsson, “Anomaly-based intrusion detection: privacy concerns and other problems,” *Computer Networks*, vol. 34, no. 4, pp. 623 – 640, 2000, recent Advances in Intrusion Detection Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128600001341

[47] Council of European Union, “REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) ,” 2016, https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679&from=EN [Accessed: 2018-06-04].