Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Magnus Lien Lilja

Efficient representation of data in intrusion detection systems

Master's thesis in Information Security
Supervisor: Slobodan Petrovic
June 2021

NTNU
Kunnskap for en bedre verden

Magnus Lien Lilja

Efficient representation of data in intrusion detection systems

Master's thesis in Information Security
Supervisor: Slobodan Petrovic
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Abstract

Businesses are very dependent on their information technology (IT) systems, which can be large and complex to secure. Network intrusion detection systems (NIDS) are often used to monitor central network locations and alert administrators when a possible attack is detected. Signature based NIDS search for patterns of known attacks in network traffic, where the amount of signatures are growing as new attacks emerge. Speed is critical in such systems in order to detect the possible threat as fast as possible. Thus, memory is key in order to provide good performance, which is often limited. Efficient representation of data in NIDS can reduce the amount of required memory in existing implementations or possibly extend the usage to devices with limited memory, whilst not impacting the performance. The work in this project looks at a measure of implementation efficiency, how efficient existing implementations are in two of the most used open source intrusion detection systems (IDS) and suggests improvements. Two central elements in an IDS are investigated, namely the fast pattern matcher and representation of signatures using a quantitative method. A technical analysis with a code review presents up to date models of relevant data structures. The models are then compared and analysed on a theoretical level in order to see if there is room for improvement. The suggested improvements are based around applying and customising elements of compact data structures from previous research in order to fit characteristics and the purpose of the existing representation. Furthermore, two algorithms to perform operations on the resulting improvements are developed. In addition to the suggested improvements, a proof of concept compares the existing implementation to the alternative ones, using a developed metric for efficiency and a variation of widely used signature collections. The findings throughout the thesis are data structures which are more efficient than the existing implementations.

Contents

Figures

Tables

Acronyms

BP balanced parentheses. 19

DFA deterministic finite state automaton. 15, 28, 49, 51, 53, 55, 72

DFUDS depth-first unary degree sequence. 19

FPGA field-programmable gate array. 3

HIDS host-based intrusion detection system. 7, 72

HTML hypertext markup language. 27

IDS intrusion detection system. 1, 2, 3, 7, 8, 16, 20

IMRaD introduction, methods, results and discussion. 4

LOUDS level-order unary degree sequence. 19

NFA non-deterministic finite state automaton. 15, 16, 28

NIDS network intrusion detection system. 1, 2, 7, 72

PoC proof of concept. 4, 5, 24, 26, 28, 29, 31, 35, 47, 49, 58, 60, 62, 65, 72, 73, 75, 77

TCP transmission control protocol. 8, 36, 38, 42, 60, 73

UDP user datagram protocol. 8, 36, 38, 42, 60, 73

UML unified modelling language. 28

Glossary

ASCII American Standard Code for Information Interchange is a set of 256 different characters covering eighth bits. 49, 50, 72

children When talking about the abstract data-structure “tree”, inheritance between vertices is described with family relations. Children in this case are vertices inheriting from a specific vertex. 18, 19, 20

computer word A fixed size unit of data. The word size, determines the amount of bits in the word. 11, 40

dense graph A graph where the amount of edges is close to maximum. Each node has a high amount of connections. Opposite of a sparse graph. 20

exact search In exact search all the entries in the search pattern have to match. 8

operation An operation on a compact data structure is a query on the abstract representation with the purpose of manipulating an element from the uncompressed set. For example, a read operation on an sequence. 3, 10, 17, 43

ordinal tree A tree where the vertices can have arbitrary amount of children, contrary to a cardinal tree, where the vertices have a set amount. 39

planar graph A undirected graph which can be drawn such that no edges cross each-other. 20

regular expression An encoded search pattern describing a set of strings. 15

signature A description and identification of an anomaly. Realised in plain-text format and contains a header and options. 1, 2, 7

Chapter 1

Introduction

This chapter presents the problem area and motivation behind the research. Next, research questions are presented together with the planned contributions and the outline of the thesis.

1.1 Topic covered by the project

Misuse or signature based intrusion detection system (IDS) is essentially to search for patterns in a search text. Particularly, a network intrusion detection system (NIDS) tries to find known indicators of suspicious activity in network traffic. Such an indicator can be a sequence of characters which is unique to a specific attack. The search pattern indicates suspicious activity and is contained within a signature. As the amount of attacks grow, more search patterns are required in order to detect new attacks. For a NIDS situated in a large cooperate network, the amount of signatures can be tremendous. This requires fast storage (memory), which is often available in limited amounts. Although memory is relatively cheap and can be extended in large systems, some systems are constrained by the amount of build in memory, such as for example Internet of things (IoT) or certain types of firewall devices. By reducing the amount of required memory by NIDS, their application can be extended to such devices with limited memory. Compact data structures aims to do this, whilst maintaining a high performance. This project explores central data structures in two of the most used open source NIDS today, with focus on efficiency.

1.2 Keywords

Relevant keywords for this thesis are chosen according to the 2021 IEEE Taxonomy [1]. The keywords are: Data structures, Intrusion detection, Algorithms, Encoding, Codes, Data compression

1.3 Problem description

The two most popular open source NIDS, Snort and Suricata are somewhat old systems. Snort had its first release in 1998, while Suricata was released approximately a decade later, in 2010 [2, 3]. In fact, they have become so popular and widely used that they are considered the de-facto standard for anomaly based intrusion detection [4].

A main component in both of the systems are signatures. Several signatures are grouped together in what is called a ruleset, which can become quite large. It then requires more fast storage, or memory in order to store all the signatures in the ruleset. There are no official hardware requirements for either Snort or Suricata, but it is recommended to add main memory (RAM) in order to increase the system performance [5]. For example, Snort and Suricata stores incoming network traffic in memory as a temporary storage (buffering). Running out of memory can then lead to network traffic not being analysed (dropped). Memory is therefore important for the proper functionality and performance of an IDS [6]. Performance is also increased by reducing the amount of memory instructions that are loaded from disk (swapping). Accessing instructions in main memory is about 10^5 times faster than from disk, and naturally has a great impact on the functionality of the application [7]. Thus, if the reduced memory space impacts the access times, it has an overall negative impact on the performance as well.

The book *"Snort IDS and IPS toolkit"*, which features members of the development team behind Snort - Sourcefire, suggest using two to four gigabytes of memory for best performance, with one gigabyte as minimum [5]. Some systems, such as embedded devices and certain firewalls, have limited built-in memory. Meaning that in some cases, an IDS can be too memory consuming in order to perform ideally in a system where performance is critical. When adding memory is not a possibility, an option is to reduce the amount of required memory by the application whilst keeping fast access times, using compact data structures.

Being open source, Suricata and Snort have been improved upon by different people with varying theoretical backgrounds. It is possible that some of the original code from the first release is still present in the current version. Because developments in compact data structures have been made in the last 10-20 years, it can be the case that the data structures implemented in Snort or Suricata are not optimal with respect to efficiency. This project aims to investigate this issue closer.

1.4 Justification, motivation and benefits

As mentioned, limiting the amount of memory required by an IDS can extend the systems usage to function on systems without possibilities to add memory. One benefit with this research can be a smaller memory footprint, which can reduce the required memory in existing implementations and thus save money. Improvements to these specific systems can also be applied to other systems with similar

characteristics.

An example illustrates the problem area further. Zhao et al. [8] researched a way to achieve 100Gbps throughput in an IDS, with the help of a field-programmable gate array (FPGA) which has limited embedded memory. In the process, they tested the space consumption of a multi pattern search algorithm (Aho-Corasick NFA), which is used in Snort. They measured that it consumed 23MB of memory (Block RAM) on Snort version 3, using the snort registered rule-set. This made the current version of the algorithm consume too much memory for the FPGA, and a different algorithm was developed as a result of that.

Another aspect is sustainability. ATEA calculated that if each Norwegian deleted four to five emails (2000-2500KB) each day for a year, it could power one thousand average Norwegian households for a year [9, 10]. Thus there is also a sustainability aspect to limiting the memory consumption of devices, because it requires power to run the hardware. This can be transferred to IDS as well. By reducing the amount of space required per implementation, it can save a lot of memory when applied in multiple instances and consequently the required power to run those devices.

However, it is not enough to only reduce the amount of memory or space consumption of the IDS, if it is limits the performance as a result. Compact data structures aims to reduce the space required to represent an abstract object, whilst maintaining fast operations on the data.

1.5 Research questions

In order to suggest improvements, it is of interest to have a closer look at the existing representations first. Hence, the logical approach to the problem is to get an overview of the data structures and how they are applied in the code, before they are analysed and improved. The research questions reflect this approach. To scope the project down, certain central data structures have been chosen in this project, compared to explore every bit of data structures. As mentioned previously, the signatures are space consuming and have to be kept in memory in order for the search to be efficient. Thus, it is of interest to have a closer look at. Another important part of an IDS is the fast pattern matching algorithm, which is looked into as well. The three research questions are as follows:

1. What are the efficiency criteria for signature representation and the implementation of the fast pattern matching algorithm?
2. How efficient is the representation of signatures and the implementation of the fast pattern matching algorithm regarding the criteria from question one?
3. How can these data structures be improved in order to meet optimal values of the criteria defined in question one?

First, a definition of efficiency is determined, which is used as a foundation for comparison later. The second research question is about creating a model from the

data structures in the code, with enough abstractions to only represent elements of importance. This makes it easier to compare the representations to optimal values of the efficiency criteria. They will also be investigated to see if compact representations of data are utilised and to what degree they are efficient. Suggesting improvements to these data structures is the third research question, as well as creating a proof of concept (PoC) for a practical comparison. The first research question is answered in Chapter 5, while the two others are answered in Chapters 6 to 8.

1.6 Contributions

The main contributions are suggested improvements to the representation of data and a metric to measure the efficiency. Firstly, a measure of implementation efficiency is defined together with a developed efficiency metric. Then, a technical analysis with a code review of recent versions of Snort and Suricata provides understanding and knowledge about central data structures in the two systems. This is further analysed in theoretical means, in order to investigate if any measures are undertaken to improve the efficiency of the data structures and to see if there are room for improvement. On that basis, improvements are presented and compared with the existing implementations on a theoretical level. Fundamental data structures from existing research are customised and combined in order to improve the representations, based on their characteristics, purpose and abstract models from the theoretical analysis. Two new algorithms are also developed, to perform relevant operations on the developed data structures. Lastly, the efficiency metric is used in a PoC, in order to see how the improvements compare to the existing implementations.

1.7 Outline

The thesis follows the introduction, methods, results and discussion (IMRaD) structure and is divided into the following chapters: Chapter 2 presents some initial theoretical background about compact data structures and intrusion detection systems, which is required knowledge for the rest of the thesis. Chapter 3 looks at the state of the art research regarding the three research questions. The literature presented here is a basis for the results in this thesis. Chapter 4 presents the research method used to answer the research questions. Chapters 5, 6, 7 and 8 constitute the main part of the thesis, where the research contributions and results are presented. Firstly the efficiency criteria are presented in Chapter 5 in order to determine what an efficient data structure is. In Chapter 6 and Chapter 7 the signature representation and fast pattern matcher are analysed in technical and theoretical measures before alternative representations are suggested. Technical analysis includes a code review which is used for the theoretical analysis of the data structures. Then alternative improvements for the signature representation

and fast pattern matcher are suggested based on the abstraction. A PoC implementation of selected alternatives is described in Chapter 8, where the results are presented and compared. Chapter 9 discusses the obtained results and alternative improvements. Lastly, Chapter 10 concludes the thesis and suggests some future work.

Chapter 2

Theoretical background

This section presents the theoretical background for intrusion detection system (IDS), compact data structures and mathematics, which is required for further reading.

2.1 Intrusion detection systems

Background knowledge of central elements in an IDS, specifically the signatures and detection engine is presented here.

An IDS is essentially a pattern matching system. Its functionality is to search for patterns of misuse (signatures) in a search text and create an alert if a match is found. It is based on the fact that there is a difference between normal and malicious activity and that indicators of such activity can be recognised. An IDS can be categorised based on the data-units it searches in. A host-based intrusion detection system (HIDS) searches for anomalies in logs or files, while a network intrusion detection system (NIDS) searches for anomalies in network packets. Snort and Suricata are both NIDS [11].

A signature or rule in an IDS is used to identify malicious activity. Following is an example signature in Snort, where the highlighted part represents the signature header and the options following the header in parenthesis. It should be noted that only a select set of options are presented here and that several others exist.

```
alert tcp $SRC_NET any -> $DST_NET 80 (content: "Some ma-
licious pattern"; msg:"Pattern detected"; sid:1000000; rev:1;
classtype:misc-activity;)
```

The header classifies the signature based on the following fields.

- Rule type
- Protocol
- Source address
- Source port
- Direction

- Destination address
- Destination port

The most important field amongst the options is the *content*, which contains the pattern to be searched for in the data packets described by the header. The *msg* is the message to be shown to an operator when a match is found. Other important options are the *sid*, which is a unique signature identifier.

A large database of signatures means that a central part of the detection engine in an IDS is a fast search algorithm. The IDS considered in this thesis, Snort and Suricata both use exact search, where internal variations of an attack requires a new signature. Processing and searching in linear time of the amount of rules in the database would take too long and depend on the amount of rules. A better approach is searching in linear time for the search string (network packet), which is called multi-pattern search. The fast pattern matcher will then find all the matching signatures in a single pass over the search text, which is much quicker than searching through all the signatures one by one for a possible match, assuming a large database [11].

In addition to the fast pattern matcher improving the performance of the IDS, the signatures are represented in a grouped form, using the information in the header. This limits the amount of signatures which needs to be searched through. For example, transmission control protocol (TCP) signatures are irrelevant for user datagram protocol (UDP) traffic.

2.2 Measurement

It is of interest to compare different representations and select the best fitting, using a common unit of measurement. This section presents some basic concepts, which are required to understand the rest of this thesis. The binary logarithm, or the logarithm to the base 2 is used during this thesis and expressed as log unless stated otherwise.

2.2.1 O-notation

One widely used concept to describe the cost of an algorithm as a function of the input size is asymptotic growth, described by O-notation. $O(g(n))$ can be formalised as the following [12].

$$O(g(n)) = \{f(n) : a, b \in \mathbb{N} \mid 0 \leq f(n) \leq ag(n),\ \forall n \geq b.\} \tag{2.1}$$

$f(n)$ cannot be greater than $ag(n)$, considering a large n [13]. It is an upper bound or worst-case performance of an algorithm for a large input size. For example, we can say that $f(n) = an^2 + bn - c$ is $O(n^2)$, but not $O(n)$, because $f(n)$ will eventually grow larger than $O(n)$, independent of the coefficients [7]. There is always a constant u which will make $un^2 > f(n)$, thus f is $O(g)$ meaning that f does not grow faster than g [12]. When talking about algorithms in general,

large growth is unwanted and thus noted as the worst-case performance. The same concept can be converted to describe the best case of an algorithm, or slow growth noted as $\Omega(g(n))$. We can also say that $f(n) = o(n^3)$, describing that a quadratic function is always slower than a cubic, because their ratio goes to zero for large values of n, explained by Rawlins [12]:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{an^2 + bn + c}{n^3} = \frac{a + b + c}{n} = 0 \,, \forall a, b, c \tag{2.2}$$

Figure 2.1 shows an illustrative example of asymptotic growth for small values of n and using the binary logarithm.

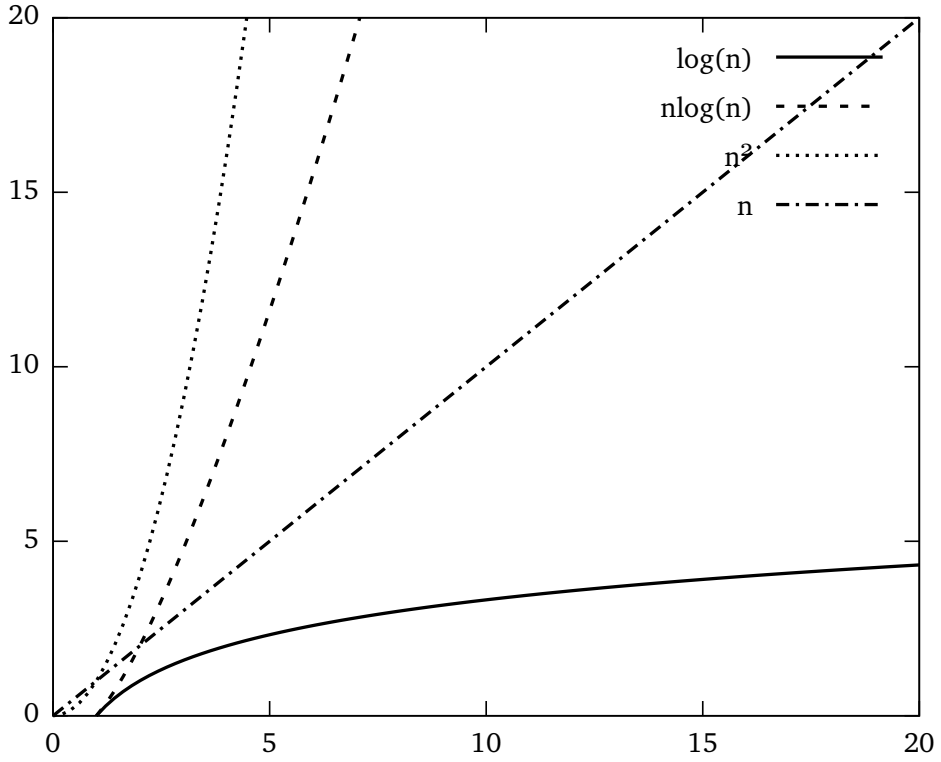


Figure 2.1: Illustration of asymptotic complexity

2.2.2 Entropy

Entropy is commonly referred to as a measurement of randomness or uncertainty [14]. The notion of entropy is used in this thesis to understand how much space is required to encode different symbols [7]. Entropy says something about how many bits it minimally takes to represent the information without any loss [12]. With more knowledge about variables in a set, fewer bits are needed to carry the information in it. For example, English text contains a lot of redundancy,

which can be taken advantage of to represent the information in a smaller space. This results in a lower entropy. Thus, entropy also has a relation with the amount of surprise or uncertainty. In compressed or random data, the entropy will be high because there are fewer patterns to take advantage of. Only knowing the number of elements in a set, leads to the *worst-case entropy* [7].

$$\mathcal{H}_{wc}(X) = log(|X|) \tag{2.3}$$

Having a probability model of the elements in the set can lower the entropy. However, it cannot get any worse than the worst-case entropy. This is done by assigning elements with a higher probability a shorter code, which is done in for example, morse code [14]. *Shannon entropy* estimates the number of bits in each element [7].

$$\mathcal{H}(X) = \sum_{x_i \in X} Pr(x_i) log \frac{1}{Pr(x_i)} \tag{2.4}$$

When talking about the worst-case entropy of a set, it is interpreted as the minimum amount of bits required to distinguish an element [7].

2.3 Compact data structures

Gonzalo Navarro [7] introduced the term *compact data structures* in his book *"Compact Data Structures: A Practical Approach"* as the following.

> "Compact data structures help represent data in a reduced space, while allowing querying, navigating and operating it in compressed form" [7, pg.i].

It is different from compression, because in data compression the data is represented in less space but has to be decompressed each time it is operated on. On the contrary, compact data structures aim to be able to represent the data in a compact format *while it can be operated on*, saving both space and time. This section covers briefly some central elements in compact data structures.

2.3.1 Bitvectors

Bitvectors are a fundamental component of compact data structures, because it is used in almost all compact representations. The name is descriptive, because a bitvector is a sequence of bits represented as an array $B[1,n]$ and structured to support some operations. Especially two operations are of interest to get as efficient as possible. $Rank_1(B,i)$ provides the number of unities in B to position i and $select_1(B,i)$ returns the position of the i'th unity in B. Each of the queries can be performed in constant time. Jacobson [15] proposed a constant time rank query. A constant time solution for the select query was obtained by Clark [16] a few years later. Clever use of combinations of *rank* and *select* allows for constant time queries on complex data structures. However, it is by default static after

initialisation and does not support *write* operations. As a bit sequence contains 0s and 1s, Shannon entropy can be used to calculate the empirical entropy of a specific bit sequence B with length n, containing m unities [7, 17].

$$\mathscr{H}_0(B) = \frac{n}{m}\ log\ \frac{m}{n} + \frac{n-m}{n}\ log\ \frac{n}{n-m} \tag{2.5}$$

The worst-case entropy of B is n if the bit sequence is balanced [7]. However, this calculation only includes the sparsity of the bitvector. Other factors which can be taken advantage of to compress a sequence, such as runs is not included [7]. A lower entropy is a result of m being closer to either zero or n than $n/2$, which is a sparse bitvector [17].

Constant time for the *rank* operation can be solved by using a few additional precomputed data structures, which is described briefly here. One fundamental insight is that $rank_1(B,i) = sum(B,i)$ [7]. This is simplified by the *popcount* operation which counts the number of ones in a computer word. Population count (popcount) is an operation that is implemented directly into modern architectures or can be performed very efficiently. B is divided into k blocks of w bits, where for each multiple of $\frac{i}{kw}$ a partial sum of the positions up to that point is stored in an array $R[0, \frac{n}{kw}]$. However, constant times cannot be achieved with $o(n)$ overhead. For that, another precomputed array is needed. $R'[0, \frac{n}{w}]$, stores a partial sum of ones relative to R, where all the values are between 0 and $(kw)-w$. Thus, they are smaller than those in R and can be stored in fewer bits [7]. The operation is conducted by summing the intermediate values from R and R' together a popcount of the rest of the bits up to i. A space consumption of $n\mathscr{H}_0(B) + o(n)$ is achieved when $k = w$.

The *select* operation can be solved in $O(log\ n)$ time with no additional space on top of what is needed for rank. A variant is using binary search on R, in order to find the location which is just before exceeding i. The same process is done for R' where at last, the next word is processed bit by bit to find the location of the i'th unity in B. For a constant time select with $o(n)$ overhead, a more advanced representation is required [7]. However, it is not described in detail here.

If the bitvector is sparse ($m < n/2$), the overhead can be $O(m)$ bits, supporting constant time select and access/rank in time $O(log\frac{n}{m})$ [7, pg. 83, 98]. This visualise the compromise between space and time, where a reduction in space leads to a higher time consumption.

Chapter 3

Related Work

This chapter looks at what existing literature can provide of knowledge related to the research questions. First, it looks at what information exists for data structures in Snort and Suricata. Next, it looks at existing comparisons between the two systems, as well as other related comparisons. Lastly, a brief overview of some relevant compact data structures based on existing literature is presented.

3.1 Internals

This section investigates what existing literature has to say about signature representation and implementation of the fast pattern matcher in Snort and Suricata. Relevant details and concepts are explained as well.

3.1.1 Signature representation

Snort

Snort is possibly the most used IDS, due to its age and increasing popularity [11]. Because of this, there exists some literature of the detailed functionality and internals of Snort. One book which explores the details of Snort, and is contributed to by the members of the Snort team, is *"Snort IDS and IPS Toolkit"* from 2006 [5]. Because Snort is a piece of software, which of course changes over time, updated literature is preferred to cover the details. However, the basic functionality and structures do not change that much on a general level, which means that the book can still be used to learn about the fundamentals. Chapter five covers details about packet processing and the detection engine, where the structure of the signature grouping is especially interesting. The signatures are grouped together in a two-dimensional linked list. As the signatures are parsed at initialisation, they are put in this structure, where the signature options to signatures with matching headers are grouped under the respective headers. The nodes representing the headers are called rule tree nodes (RTN) and the nodes representing the signature options are called option tree nodes (OTN). Several RTNs are then represented

in a linked list, with an arbitrary number of OTNs in a linked list referenced from the respective RTNs.

Another book which clarifies the inner workings of Snort is the book *"Nessus, Snort, & Ethereal Power Tools - Chapter 7"* by Archibald et al [18]. This book mentions the RTNs and OTNs in the same way as "Snort IDS and IPS Toolkit" where signatures that share a header are grouped together using RTN and OTN objects. It also explains some of the program flow between the classes, which is helpful when reading the code and modelling the data structures and providing a better understanding of the system. The book is from 2005, so most likely some of the source code and data structures have changed since then.

Suricata

While quite a bit of literature is written about Snort, way less exists for Suricata. To the author's knowledge, there is no literature which explains the inner workings of Suricata, like it does with Snort. The main resource for internal details about Suricata, except the source code, is the manual [19]. However, it does not cover logical data structures and overall functionality as it does in some literature for Snort. Based on the amount of available information from the manual and website etc, it seems that Snort and Suricata are two somewhat similar systems. For example, they can both accept the same Signature format, which can indicate that the internals are also like to a certain degree.

3.1.2 Pattern matching

The pattern matching algorithm is the main part of a signature-based intrusion detection system (as explained in Section 2.1). Some recommend, from discussing with system administrators, that an IDS needs to support at least 10000 rules [8]. This means that the requirements for the pattern matching algorithm in a high speed network is quite strict. It is of interest to spend as little time on processing each network packet as possible in order to make the system perform well. The pattern matching algorithm achieves this by filtering out the rules which can be a possible match [5, 20]. In Sort version > 2.0, the fast pattern matching algorithm is a part of the detection engine, and compiled by grouping the rules together by their destination port, as they already are in the rule tree (RTN). Then the longest content string of each rule is added to the set of keywords, which is compiled in the algorithm. The pattern matching algorithm is then able to find a set of possible matching signatures in linear time of the data unit $O(n)$ [5]. One widely used multi pattern searching algorithm is Aho-Corasick [21].

Aho-Corasick

Both Snort and Suricata support Aho-Corasick, which is a software realised finite state machine approach to fast pattern matching [19, 22].

A finite state machine consists of, as the name implies, finite amount of states. A state machine is firstly built to an initial state and then input events transition from one current state to another [23]. Unless parallelism is used, one state is active all the time and transitions to another state based on some input. Figures 3.1 and 3.2 illustrates finite state machines to detect the regular expression "A[B]+C", with one A followed by one or more B's and ending with one C. The unlabelled edges illustrates an epsilon transition or transition without input symbol. They consists of an initial state *S0* which is initially active. In order to move to *S1* and A is required. When the final state is active, the pattern is accepted.

Several different versions of Aho-Corasick have been developed since its original release in 1975, where there is generally a trade-off between memory consumption and performance [23–26]. However, they are mostly based on two fundamental concepts. The deterministic finite state automaton (DFA) implementation uses more memory than the non-deterministic finite state automaton (NFA), but on the contrary have a performance gain due to fewer state transitions required [25].

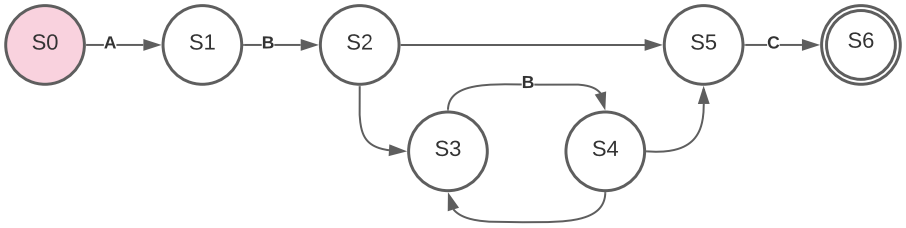


Figure 3.1: Example of a Nondeterministic Finite Automaton (NFA)

Essentially, the NFA version can require more than one state transition for an element in the search string. When an input event is not found in the current state, the failure pointer must be consolidated in order to look for another possible next state. This results in multiple possible states. Using the example from Figure 3.1 can illustrate this. If *S2* is active and the input is B, both *S3* and *S5* must be checked for the next active state.

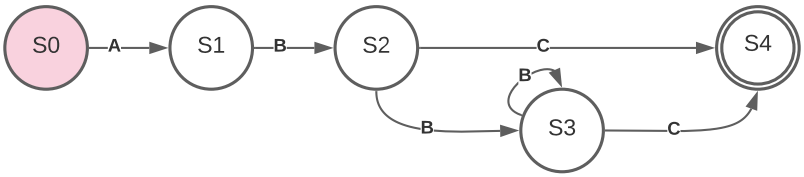


Figure 3.2: Example of a Deterministic Finite Automaton (DFA)

The DFA on the other hand, only has one state transition per element, which makes it faster [25]. This is done by traversing the states, resolving the failure pointers (epsilon transition) and adding them to the machine. The example in

Figure 3.2 illustrates this. This automaton is a conversion from the NFA in Figure 3.1. When *S2* is active and a B arrives as input, the next active state is determined immediately. However, this results in more non-empty transitions and consuming more space.

A finite state machine can be represented in several ways, but the most trivial one is a matrix, showing the relation between states and transition. More transitions make the transition table not as sparse, and thus more memory consuming. In the original Aho-Corasick paper, it is suggested that the DFA implementation can reduce the amount of state transitions by up to 50% [21]. However, realistically the performance gain is not that great in the intrusion detection system (IDS) environment. Because most of the traffic is normal, the search algorithm will spend most of its time in the "zero state", waiting for an initial match [21]. Hence, the NFA implementation is viable because it will reduce the memory consumption, whilst only being slightly slower in practice.

Realisations

Several types of representations can be used to store the state transitions in Aho-Corasick, such as a table or matrix, linked list or graph [27]. In general, the state transitions can be thought of as a matrix where the rows are states and the columns are elements in the alphabet (possible input events). The main factor contributing to the size of the representation is the amount of states, because the alphabet size (σ) is finite and often set to 2^8 bits (1 Byte) [25].

A plain representation of the state transition table is very space consuming, due to the $O(n^2)$ space complexity. Several strategies have been taken to reduce the space consumption of the data structure. Tuck et al. [23] proposed the use of a bitmap to reduce the space of the transition table in the non-optimised (NFA) version of Aho-Corasick. Instead of having σ pointers for each state, where some of them are invalid, a bitvector is utilised. If a bit is set, the transition is valid.

Kumar et al. [28] suggested to reduce the amount of transitions in the optimised version, by grouping similar transitions into an algorithm called D^2FA. If the state transition table is modelled as a graph where the states are vertices and directed edges are transitions, the algorithm aims to reduce the number of edges by taking advantage of similar transitions [29]. Results showed that the number of transitions was reduced by 95%, which again can reduce the number of bits required to represent the DFA. However, the increased cost in time this comes with when searching, is not described in detail.

Another variation, called "Split-AC" was developed by Dimopoulos et al [30]. It aims to reduce the space of the transition table by splitting the table into smaller ones, where the most frequent transitions are kept in faster memory, closer to the CPU for faster access. Results from a practical experiment showed that the suggestion was decreased the space usage from 28-75% compared to the existing best known approach. The paper does not go into details regarding the time usage, but concludes that it is not the fastest alternative.

Dharmapurikar et al. [31] suggested using substrings in combination with a Bloom filter to optimise the multi pattern search. A Bloom filter is an efficient data structure to find out if an input string is contained in a set. The algorithm they name "JACK-NFA", takes advantage of the fact that there is more normal traffic than malicious. Short patterns below 16 Bytes can then be stored directly in the Bloom filter for fast query, which is most of the patterns in the tested ruleset.

None of the existing literature discusses the compromise between space and time usage, and it seems that from the result in the papers that the space consumption is in focus. Few details are provided in literature about the internal representation of the state transitions in either Snort or Suricata. Although, some details are provided in the manual for the two systems. According to the Snort manual, the default implementation is Aho-Corasick Binary NFA (ac-bnfa-nq), which supposedly uses a compression of the transition table [22]. The default implementation in Suricata is the DFA version of Aho-Corasick (ac) [19].

3.2 Compact representation of data

This section presents research regarding some commonly used abstract methods to represent data, such as trees, arrays and graphs.

3.2.1 Arrays and sequences

Both an array and a sequence can be used to store elements of variable or fixed size, such as integers. The differences between them are the supported operations on the data structures, which makes them better suited in various scenarios.

An array A[1,n] supports the basic operations read or access and write for any position $1 \geq i \leq n$. If a computer word of the size of 32 bits is used to represent the number 10 (0b1010), it leaves $32-4=28$ bits unused, which is wasteful. An array, when talking about compact data structures, aims to reduce this amount of wasted space and store the elements in a compact manner, whilst allowing for fast read and write operations. This type of representation is better suited where only simple operations are required and will provide a representation which is closer to the optimal space usage than a sequence. Compact data structures aim to use as close to the amount of bits needed to represent the elements in the array, whilst allow for fast read and write operations. Examples of arrays are sampled pointers, Elias-Fano Codes and Direct Access Codes [7].

Elias-Fano codes is a good alternative to represent an increasing sequence of numbers in an efficient manner. If u is the highest number in an array of length n, Elias-Fano codes uses $2n + n\ log\ (\frac{u}{n})$ bits at most [32]. This is close to the information theoretical bounds, or a "quasi-succinct representation" [33]. In Elias-Fano codes, there are two bitvectors in use. One storing a concatenation of the lower $log\ (\frac{u}{n})$ bits for each element, while the other stores a concatenation of the encoded carnality of the upper bits [32, 33]. Depending on the implementation of select, Elias-Fano codes can support constant time access queries [34].

A sequence on the other hand supports more advanced operations such as rank, select and read on the elements. This type of data structure is used when these operations are required in the implementation. Wavelet trees are a typical example of a sequence, and often used to represent strings, grids and more. The tree is of depth $log\ \sigma$, and has a logical bitvector for each node. The bitvector on the root level has the length n, and is either 0 or 1 if the character is in the first or last half of the alphabet. This is continued recursively down each level of the tree [35]. Select operation on an array would require a linear ($O(n)$) search, but a wavelet tree supports $O(log\ \sigma)$ rank and select, with a plain implementation of bitvectors. In the same implementation, the wavelet tree requires $n\ log\ \sigma + o(n\ log\ \sigma)$ bits [35].

3.2.2 Linked List

A Linked list is an abstract data structure which is commonly used to represent a list of objects. In a singly linked list, each object has a relation to the next object in the list, while in a doubly linked list, each object has a relation to the next and previous objects [36]. The traditional implementation uses pointers to show the relations, which is not ideal when it comes to space consumption ($O(n\ log\ n)$). Research has been conducted on a compact representation of linked lists. Sinha [37] suggested to base the relation of a linked list on the differences between the pointers. The improved representation reduced the space consumption by 33%. Whilst no accurate measurement was given on the traversal time it is presumed to be quite similar.

3.2.3 Trees

A *tree* is a mathematical abstraction which is often used to represent concepts in everyday life, such as family tree or organisation of a cooperation. More formally, it is a set of edges and vertices, where it is only one edge connecting two vertices [36]. The edges can be labelled or unlabelled.

A typical representation of a tree contains vertices or nodes v, which has an edge e or pointer to some children c. Using pointers to a node child, the representation requires $O(n\ log\ n)$ bits, which is quite wasteful considering that only $2n - \theta(log\ n)$ bits is enough to distinguish any tree of n nodes [38]. This is an information theoretical lower bound space usage for unlabelled trees, and the gap between a pointer-based representation and the worst-case entropy has been a foundation for research in the area of compact data structures [15]. For labelled trees with labels from an alphabet σ it adds $n\ log\ \sigma$ to the worst-case entropy [39].

The required operations on the tree determine the size and efficiency of the representation. Often the application determines which operations are of interest. For example, in the GIS and LiDAR applications (Sections 3.3.2 and 3.3.3) a range query is of main interest, while calculating route length between two nodes is irrelevant. Thus, a simpler representation can be used instead which only supports

the required operations. The following paragraphs present some of these structures, which offer a compromise between space/time and amount of supported operations.

A level-order unary degree sequence (LOUDS) representation is simple in the sense that it does not support many operations, but on the other hand it is compact while offering fast access times [7]. It uses a bitvector $B[1, 2n+1]$, where n is the amount of nodes. This reduces the space from n pointers to $2n$ bits [7]. In order to construct B, the nodes v are traversed in level-order, starting from the root and 1^c0 are appended to B, where c is the amount of children nodes. The representation is almost balanced, which makes it difficult to compress it further [38]. Depending on the implementation of *rank* and *select* operations on the bitvector, the operations can be performed in constant time. However, the data structure is limited in the sense that queries such as finding the depth of a node cannot be solved in constant time.

A data structure, which supports more queries is balanced parentheses (BP). The tree is still represented with a bitvector and the operations are limited to *rank* and *select*. However, the difference from LOUDS is the structure of the bitvector, where each bit represents a closed (0) or open (1) parentheses. When representing a tree in the balanced parenthesis structure, it is traversed depth first and a 1 is written to the bitvector when arriving at a node, and a 0 when it is left [38]. This allows for some extended queries, such as finding the depth of a node because the sub-tree of a node is mapped onto the bitvector. However, it does not offer constant time within the same space usage as LOUDS [7].

Lastly, the depth-first unary degree sequence (DFUDS) representation combines the LOUDS and BP [7]. The main reduction in space comes from representing trees as bitvectors instead of pointers which typically use wn bits in total.

3.2.4 Graphs

Unlike a tree, which is a special case of a graph, a general graph is a set of vertices and connecting edges, without any further restrictions. The edges can be directional, undirectional, weighted or labelled [36]. In order to see how space efficient the compact representations are, a lower bound has must be defined. Farzan and Munro [40] presented a theorem which describes the worst-case entropy of binary relations from an information theoretical point of view, seen in Theorem 1.

Theorem 1 *Any representation of n^2 0–1 matrices containing e ones requires at least $log\binom{n^2}{e}$ bits for some matrices.*

Because there are $\binom{n^2}{e}$ different ways of selecting e pairs of edges from n^2 vertices, the counting argument is also the worst-case entropy for directed graphs, presented and simplified by Navarro [7] in Equation (3.1).

$$\mathscr{H}_{wc}(n,e) = log\binom{n^2}{e} = e\ log\ \frac{n^2}{e} + O(e) \tag{3.1}$$

For directed graphs, one of the most compact representations is an *adjacency list* storing the neighbours to a node [7]. This can for example be done by concatenating all the nodes neighbours $N(v)$ into a list $N[1,e] = N(1)\dots N(n)$, as such a two-dimensional array is not required [7]. Although, a helper array is needed in order to separate the nodes. This type of representation requires $n\ log\ e + e\ log\ n$ bits and is not far from optimal [7]. Another commonly used representation is the adjacency matrix $M[1,n][1,n]$, where $M[i][j] = 1$ if node i is adjacent to node j. Due to the n^2 space requirement, it is best used for dense graphs [7].

Although such a representation is quite compact, it supports only a limited set of operations in constant time: number of neighbours to a node and list of neighbours [7]. In order to find the best representation, it can be beneficial to know which types of operations that are needed. For example, in a planar graph it can be of interest to find the shortest path from one point to another. However, this is not something which is relevant in an IDS, and will cause unnecessary complexity.

A label on the edges adds $e\ log\ \sigma$ to the worst-case entropy, if the labels are taken from an alphabet with size σ. For labelled graphs, adjacency lists or matrices cannot be used. Compact representations of binary relations with labels were first studied by Barbay et al. [41] in 2007. It was later improved upon, adding more operations on the data structure [35, 42]. Navarro [7] presented an alternative improvement of a compact representation of a labelled directed graph, based on these improvements. In this case, a limited set of operations from the suggestions by Barbay [35] was implemented. Navarro [7] uses two sequences and bitvectors to represent a labelled graph. The first sequence L contains the edge labels in the order as a concatenation of the adjacency list. A bitvector B separates the nodes edges $B[1,e+n] = 10^{e_1}\dots 10^{e_n}$. Another sequence N contains the node identifiers stored in a different order than the adjacency lists, where instead they are grouped by labels. A corresponding bitvector B_L, similar to B, separates the area of each label belonging to a label or a node.

If the adjacency matrix (M) is sparse or clustered, then a compact alternative can be a k^2-tree [7]. A clustered graph is where some vertices have higher edge connectivity than the rest. The root of the k^2-tree is the full adjacency matrix, where the child nodes are sub-matrices. This leads to a cardinal tree where the adjacency matrix is divided into k^2 equal sub-matrices, often using a value of $k = 2$. If a sub-matrix only has 0's it is represented as a node without children (shallow node), thus consuming little space for each such area [43]. Each node in the tree has k bits and is realised in a bitvector T ordered in a level-order traversal of the tree. The process is repeated recursively for each sub-matrix containing ones, until the final depth of $log_k\ n$ which represents single cells of M. Another bitvector L contains the nodes in the final depth [43]. Space usage in the worst-case is $k^2 e(log_{k^2}\frac{n^2}{e} + O(1))$, while the worst-case time usage for neighbour queries (list all adjacent nodes) are linear $O(n)$ [43].

3.3 Existing comparisons

This section looks at some comparisons between Snort and Suricata, as well as analysis of compact data structures in other systems. The intention is to describe some of the existing work, its benefits and limitations.

3.3.1 IDS

Comparisons of the memory consumption in Snort and Suricata, have been addressed several times in literature. Many of the studies keep the configuration of the two systems to default [4, 6]. This means that the fast pattern matching algorithm (Aho-Corasick) mode of operation is different (see Section 3.1.2). Thus, the two systems are incomparable in the sense of memory consumption. However, in 2013, an analysis between Snort and Suricata was done by White et al. [44], which changed the default configuration of Snort by applying a DFA version of Aho-Corasick. This change kept the algorithms equal on both systems, and thus makes them comparable. The research concluded with Suricata consuming less memory than Snort on average. However, the comparison was done with an increasing amount of cores for Suricata and Snort in "multi instance mode". From the graph presented in the paper, the memory consumption is rather similar up to eight cores or instances, which can indicate a flaw in Snort's software due to the sudden jump in memory usage. Even still, Suricata has a slight advantage in memory consumption up to that point. It would be interesting for the purpose of this project to see further details, regarding which data structures consumed the most memory or why the memory usage was high. A comparison without Snort in multi instance mode would be interesting as well.

In general, the current approaches to comparison between Snort and Suricata have a technical and practical perspective, instead of a theoretical and detailed one. An analysis of compact data structures specifically, is to the author knowledge, not researched for Snort or Suricata, but addressed for other systems. The following sections look at analysis with the intent of improving the size and efficiency of data structures in other systems, which can be related to data structures in an IDS.

3.3.2 LiDAR

Ladra et al. [45] investigated the compact storage of LiDAR point clouds. Due to the amount of space required by the number of three-dimensional data points and some additional metadata, a compact representation is required. Firstly, an analysis of the existing methods to store data is done, where drawbacks are identified. They suggest using k^3-trees to improve upon the existing LAS and LAZ format, which was currently in use. K^3-trees are in this case used to store coordinates in a grid, in which case a sparse matrix can be represented in a compact manner. A static value of $k = 2$ is used, but can be adapted in order to impact the space consumption and query times of the data structure. The three dimensional space

(x,y,z) is divided into 2^3 cells, where the empty cells use few bits. Two bitvectors are then used instead, to represent the tree structure, as an "output" of the k^3-tree.

Although the types of information in LiDAR and IDS environments are different, the tree structure is on a general level useful for representing several types of data in a compact manner. It is difficult to tell whether the research is directly applicable to improve data structures in Snort or Suricata, before doing more research on the two systems, but it can be a possible candidate. The benefits with this research are that the lower level data structures, which are used to represent the conceptual trees are described, as well as an experimental evaluation with several datasets are presented.

3.3.3 GIS

Brisaboa et al. [46] researched a compact representation of spatial indexes in Geographic Information Systems (GIS). Spatial indexes are often used in GIS, due to the capability of efficient search in overlapping geographical objects, such as different layers on a map etc. A spatial index is a maximal and minimal x, y coordinate of the objects minimum bounding rectangle (MBR), instead of all the coordinates the object itself [46]. Brisaboa et al. [46] suggest a wavelet tree to store the spatial indexes in a compact manner, whilst allowing for queries in compact form. By extending the x, y grid to contain one coordinate per column, the x coordinates (row) can be projected onto the y coordinates (columns) in a grid. By sorting the coordinates from the x-axis, the ranks of the coordinates (rank space) are worked on, instead of the coordinate space. The wavelet tree is used to store this data structure by utilising the projection of the nodes order in the x-axis to the y-axis. Two bitmaps are used in each node of the tree, to keep track of the spatial indexes lower and upper bound (B_l^x, B_u^x). A *range* query is performed on the wavelet tree to select objects with coordinates in a MBR.

The research is an example of a creative way to customise and apply fundamental data structures to this type of data, and develop queries to suite the purpose of the system. It also shows a thorough methodology in testing and comparing the suggested improvements space and time. It is difficult to say how applicable it is before understanding how the data is represented in Snort and Suricata.

3.3.4 Graph databases

Álvarez-García et al. [47] proposed a compact representation for labelled attributed graphs, which is used in for example modelling data in social networks or web graphs. Special in this case is that the nodes and edges contain additional information, which the existing compact proposals do not include. They suggest using a modified k^2-tree to represent such a graph, which they name *Att*K^2*-tree*. It is not a single tree as the naming might suggest, but the graph is divided into several connected representations. This is done partially for it to be a modular system, where parts can be changed without affecting the whole. The first part

is the Schema, which contains basic information about the nodes and edges of the graph and serves as a foundation for indexing more data. Data is the second part, which contains values for edges and nodes of the graph. Attribute values are represented differently depending whether they are sparse or dense. Dense attributes are defined as values which are shared amongst nodes or edges, and are stored in k^2-trees. The spare attributes on the other hand are stored in a list, because they are unique to a certain edge or node. The third part is the relationship, which generally is information about which nodes are connected to each-other via which edges. A k^2-tree with support for multiple edges between nodes is used to store this representation. A dynamic version of the data structure is also presented, which has an ability to add states after the initial construction. The paper concludes with improved space and query-times compared with other graph engines. Attributed graphs seem initially somewhat like the rule-tree in Snort. The "layered approach" with custom optimisations is also an interesting idea.

3.4 Summary

As far as we know, a systematic research on the data structures in intrusion detection systems have not been done in previous literature. However, there has been research looking into compressing the signature set, by searching for similar rules and making sure that the amount of unnecessary matching is reduced [48]. The signatures are created by human experts, which can lead to errors such as one rule being covered by another, identical or very similar signatures etc. However, this is not the same as minimising the signatures complexity when represented in memory.

Research in compact data structures for other systems does exist. However, it is difficult to directly transfer implementations in one system to another, because the types of data and operations on them determine which representation is best suited. Existing literature does not provide relevant details about the representation of signatures and the representation of the fast pattern matching algorithm, especially in Suricata. There is some information about Snort, but it is not detailed enough to get an overview of the current representation of data. Thus, some more research is required in order to understand the current data structures in Snort and Suricata, as well as to suggest relevant improvements. From the existing literature, we have not been able to identify research in compact data structures in intrusion detection systems. Another observation is that improved data structures for other systems, customises and applies fundamental compact data structures from previous research. For example, a k^3-tree is used to suggest an improved data structure for LiDAR data, because the nature of the data and purpose of the queries on the data is suitable.

Aho-Corasick on the other hand have been optimised several times, but there is lack of a defined metric to compare the results efficiency. From the results it seems like the space consumption is the focus, instead of seeing time and space as a whole. In some of the literature, the suggested improvement is compared to

the existing using a proof of concept (PoC). Others compare the proposed representation to existing improvements. Asymptotic growth and entropy are not used consistently either.

Compact data structures for general graphs, trees, arrays and sequences have been optimised to approach their worst case entropy, whilst also achieving fast access times. Thus, it is challenging to optimise these any further.

Chapter 4

Methodology

In order to answer the research questions in the best way possible, a structured and defined approach to the problem area must be discussed prior to starting the research. This research presents the general research method and the select approach to answer the research questions.

4.1 Research method

Because of the specificity of the research questions, the research type can be categorised as applied research. Furthermore, a quantitative method is conducted with a theoretical comparison and proof of concept.

4.2 Related work

Several sources of literature were used to get an overview of the state of the art research. Microsoft Academic proved to have a good collection of publications with options to filter topics and publication year and various ranking possibilities which was valuable [49]. A publication's references can also be sorted and filtered. Different sources for literature were used during the process in order to gather the most up to date research regarding the research questions. Other sources for literature is the NTNU university library - Oria [50]. A combination of relevant phrases using the boolean operators "AND"/"OR" was used in the collection process. Some of the phrases used were the following:

- data structure, data structure, data-structure
- snort
- suricata
- compact
- succinct
- pattern matcher, aho-corasick, aho corasick
- ruleset, rule-set, rule-structure, rule tree

4.3 Efficiency metric

Recall, the first research question is about defining some efficiency criteria to evaluate the existing data structures and propose improved representations. Criteria helps to make a conscious choice, when finding the most efficient data structure. This section presents a couple of methods which can help find relevant criteria for efficiency in this context. One method is to see what recognises a good solution of the problem, or what factors make an implementation good, bad or successful on a general level. Another method is to use criteria which are standardised or widely used by others. Both approaches are used when defining criteria.

4.4 Improvement

A general strategy when suggesting improvements is to understand the problem and the existing solutions, in order to see how it was solved previously and drawbacks or limitations to that approach. Background theory and related work give a basis to understand the problem area and an overview of the existing solutions. A technical analysis provides more up-to-date information of the systems and further understanding and detailing the problem formulation. Then, a theoretical analysis formulates and describes the problem in order to find out what and where to improve. This information makes it easier to see characteristics, possible areas that can be changed and where existing research can be applied or modified, in order to achieve a better solution. Such a method and way of thinking is used to suggest improvements in both the fast pattern matcher and the signature representation.

4.5 Analysis

The analysis method follows the improvement methodology, where firstly a technical and theoretical analysis are required to suggest improvements. After the improvements are suggested, a proof of concept (PoC) is presented with two of the most promising alternatives. Binary logarithm, or the logarithm to the base 2 is used during this thesis and expressed as "log" unless stated otherwise. This section presents the technical and theoretical analysis approach.

4.5.1 Technical

Prior to modelling the data structures in the two systems, information about them must be collected. One method of extracting relevant information about the data structures are through analysing the code in the two systems. In order to improve the efficiency in analysing the code, prior information such as manuals and changelogs, can give valuable indications on where to look for the specific functionalities in the code. Unless very up-to-date information exists regarding internal details of the two systems, a code analysis provides the most recent information. Another benefit is that the details can be abstracted away as desired.

Code analysis or code review in this case, is about getting context to the variables in order to see how they are represented and related on an abstract level. One method of doing this, which is also done in this thesis, is using Doxygen [51]. It is a static code analysis tool which can generate documentation in hypertext markup language (HTML) format which shows relation between variables, classes and functions. The benefit is that it can create call-graphs that makes it easier to navigate and understand the code.

Another method to collect information about the data structures is through existing literature. This can greatly improve the efficiency, in which the code does not have to be analysed over again. But it can have drawbacks such as not being detailed enough information or simply outdated with respect to the current version of the code base. Existing literature can also be used next to analysing the code, where some of the main ideas behind the data structures are kept in mind when analysing the code for details. This of course assumes that the main structure of the code has not changed during that time, which is very often the case on a general level.

Bacchelli and Bird [52] explored, amongst other things, code review challenges. When they asked how developers start code reviews, the developers answer was to start with familiar points in the code. This made it easier to see what was going on. One way of doing this starting to look for what is described in existing literature in the code. Furthermore, 91% responded that it takes longer to review unfamiliar files.

Höst and Johansson [53] tested out two code review methods, where one was a detailed approach and the other a more general one. The research concluded with no significant differences between the tested methods. However, it does not mean that the approach has nothing to say when analysing code, but can indicate that the results do not depend on the approach.

Based on this, a combination of methods has been selected and applied in this thesis. Existing literature is used together with a general approach code review, where Doxygen is used as well as a assisting tool. Another benefit with a code review is that the level of detail can be selected as required.

4.5.2 Theoretical

Rawlins [12] describes five steps to analysing an algorithm in *"Compared to what? : an introduction to the analysis of algorithms"*. After recognising the problem, an abstract model must be made to describe the problem. Next, an algorithm is designed according to the model, which is then analysed. Lastly, the result of the analysis is reviewed. This approach can be translated to analysing data structures. Rawlins [12] defines a model as a combination of the allowed operations (environment) and a set of operations to reduce (goal). In other words, the model can be thought of as constraints, inputs and wanted outcome to the problem. An algorithm is then a solution to that problem, which in this case is the efficient data structure itself. The modelling step will in this definition also include the

algorithm, which is already designed in the two systems, Snort and Suricata.

The data structures found in the technical analysis are modelled on an abstract level to capture central elements which describes the problem. This model is then analysed in order to determine the efficiency and room for improvement.

4.5.3 Versions and formalities

This section presents some technical considerations and choices which where made during the analysis.

Several versions exist of Suricata and Snort, but the latest stable release is considered in this thesis. For Snort that is version 2.9.17, while version 6.0.1 is the latest stable release for Suricata as of 8th December 2020

Although the mentioned versions of Snort and Suricata are written in C and do not use objects but rather structs, the term object is used synonymous throughout this text. Furthermore, the unified modelling language (UML) is used to create a model of the data structure from the code and show the inheritance between the objects.

When comparing the implementation of the fast pattern matching algorithm, it is important to model and analyse comparable versions in Snort and Suricata. If not, the deterministic finite state automaton (DFA) representation will consume more memory by intention, in order to achieve somewhat better performance in theory. In this thesis, the DFA variant of the Aho-Corasick algorithm is used over the non-deterministic finite state automaton (NFA) version. It is chosen because Suricata does not support the NFA variant, as of Apr 4, 2016 [1]

4.6 Experimental result

For a PoC implementation, it is beneficial to use existing mature code libraries if possible. Using a library which is well tested, takes advantage of low-level optimisations such as for example cache efficiency, ensures that we do not invest additional effort in development of these parts. It can also remove error in the result, stemming from low-level operating systems specific implementations and optimisations, which is easy to ignore.

Quite a few open source implementations exist, where a few have been recommended in previous literature [54–57]. Most of the libraries are written in C++, and include for example bitvectors with support for constant rank and select operations, which is widely used in more complex representations of graphs and trees. And a basis for most compact representation of data. This also means that the PoC is implemented in C++. Other reasons for choosing C++ as a programming language is that Snort and Suricata are written in C, which makes it easy to reuse certain components. This makes the implementation of the existing representations more accurate. C++ is also a fast programming language, thus the

[1] https://github.com/OISF/suricata/commit/4f8e1f59a6c3d76f49863ddaafb97e04bfecc092

overhead in the code language itself are low. In addition, some of the libraries also include functions to measure the space usage, which is valuable and provides more accurate results.

The two libraries used are the "Succinct Data Structure Library 2.0" by Gog et al. [55] and "succinct" by Ottaviano et al. [56]. These libraries are chosen because of recommendations from existing literature and that they both included testing. They also have decent documentation and together provide a wide set of functions and classes. The following versions of the libraries are used.

- SDSL Lite - version 2.1.0 [55]
- succinct - commit id: 69eebbdcaa0562028a22cb7c877e512e4f1210b [56]

The output of the PoC captures differences in efficiency between the existing and the suggested implementation. More about this is given in Chapter 5. Further details regarding the implementation environment, considerations and descriptions of the implementations are given in Chapter 8. In order to provide verifiability to the result, the code is uploaded to an open repository on GitHub [58].

4.6.1 Dataset

Experiments in the PoC are conducted using different datasets or rulesets, to see the performance in various scenarios. The rulesets contain signatures in a plaintext format, where a signature usually takes one line. However, this is not a requirement. Table 4.1 contains the used datasets. The emerging threats datasets are taken from the Proofpoint Emerging Threats Rules [59]. While the Snort open and registered rulesets are taken from the Snort website [60]. The "-all" datasets contain a collection of signatures for various detection objectives, while the other ones are aimed towards detecting suspicious traffic in specific network traffic. The "date" column indicates when the dataset was downloaded. These datasets are selected because they are used within the industry and cover a lot of different types of traffic. This will provide realistic results to what is expected in a production scenario. Some targeted datasets are also included in order to see the performance on smaller datasets. Smaller and targeted datasets contain signatures for certain environments and types of traffic, which means that they have different properties than a diverse signature set. Results of this can be only a few ports, and possibly create different results.

Dataset	Lines	Source	Date	Description
emerging-threats-all-snort.rules	59968	[59]	11.Mar.21	All rules from Emerging threats Snort 2.9
emerging-threats-all-suricata.rules	61736	[59]	11.Mar.21	All rules from Emerging threats Suricata 5.0
emerging-threats-policy.rules	2312	[59]	11.Mar.21	Emerging threat rules specific for applications that can be disallowed in a company based on the policy (Steam, Torrenting)
emerging-threats-scan.rules	724	[59]	11.Mar.21	Emerging threat rules specific to recognize scanning / probing (Nessus, Nmap)
emerging-threats-web_server.rules	1470	[59]	11.Mar.21	Emerging threat rules specific to recognize malicious activity on webservers
snort-community-all.rules	3975	[60]	04.Feb.21	All rules from Snort 2.9 community rules. All commented lines have been uncommented.
snort-registred-all-3.0.rules	50555	[60]	25.Mar.21	All rules from Snort 3.0 registered ruleset
snort-registred-all-2.9.rules	56611	[60]	25.Mar.21	All rules from Snort 2.9 registered ruleset
all.rules	237351	-	-	A concatenation of all the rulesets above

Table 4.1: Overview of datasets

Chapter 5

Efficiency Metric

This section describes the term *efficiency* used in the context of measuring the performance of data structures, and presents a developed metric to measure efficiency.

From the definition of compact data structures in Section 2.3 and related work about compact data structures in Section 3.2, it is apparent that space and time are two central metrics to compare data structures.

One method to find criteria is to see what identifies a good solution to the problem, as mentioned in the methodology, (see Chapter 4). By looking at a good solution for the improved representation, the following factors are of interest.

- Low space consumption
- Fast indexing or querying for data
- Straightforward and easy to implement
- Amount of gained space is greater than loss of time

Both space and time, again come up as relevant metrics. When talking about the term *efficiency* in the context of compact data structures, both space and time are key concepts to its performance. Another aspect is the compromise between space and time, which can make a good solution.

Chapter 3 presented the worst-case entropy of graphs and trees, which is the optimal space usage. The worst-case entropy is limited by the number of elements in a set, which is an information theoretical lower bound. Time is measured using asymptotic growth, which cannot be faster than constant $O(1)$. A challenge in compact data structures is to improve both space and time to be closest to its optimal values. Usually, space must be compromised on behalf of time or the opposite.

5.1 Practical evaluation

The proof of concept (PoC) implementation in Chapter 8 aims to capture the differences in efficiency between the suggested improvement and the existing implementation. Because of this, a measure of the compromise between space and

time have to be constructed. The metric is presented here.

Figure 5.1 illustrates the compromise between space and time. A reduced space consumption is of course desired. An optimal result would be a reduced time usage as well, which is an optimal solution. On the contrary, both increased space and time consumption is a bad result. On the contrary, a system with high memory usage can be valuable in certain situations and environments where memory restrictions are not considered important, if it is fast. Therefore, a measurement which captures the compromise between space and time is needed. More specifically, a measurement which captures the time lost versus the space gained can determine if it is worth using. Such a metric was developed in this thesis and is formalised as the following:

We call the space and time usage for the suggested improvement w_i and x_i respectively, while the space and time usage for the existing implementation is w_e and x_e, respectively. The formula described in Equation (5.1) can be used to determine if the space difference is greater than the time loss.

$$z = (\frac{w_e}{w_i})/(\frac{x_i}{x_e}) = \frac{w_e x_e}{w_i x_i} \tag{5.1}$$

The first part of the equation shows the space difference, while the second part is the time difference. If the time difference is greater than the space difference, it will result in $z < 1$, which is undesired. In the opposite case $z > 1.0$, it means that the space saved outweighs the time lost, which is an efficient solution, according to this definition. A higher value is better, or a solution with higher efficiency.

As a summary, space and time determines an efficient implementation and the difference between them can decide if the solution is worth implementing.

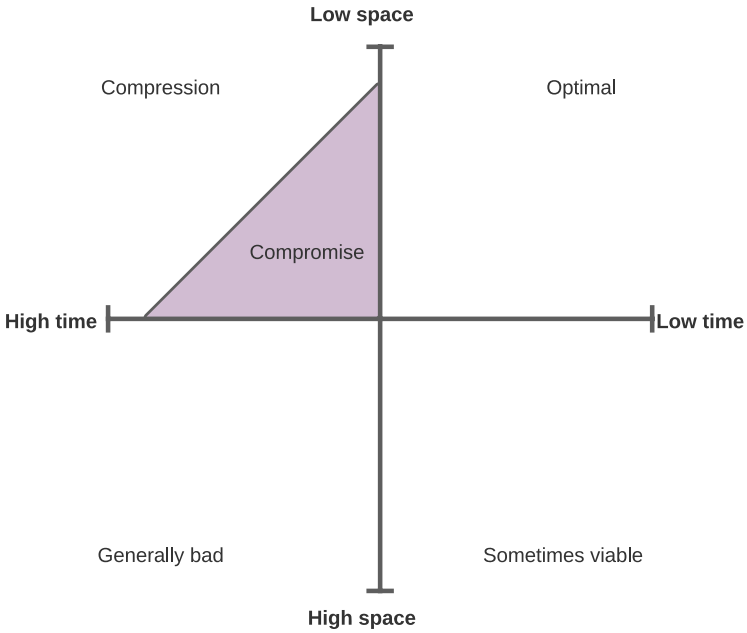


Figure 5.1: Illustration of space and time compromise

Chapter 6

Signature representation

This chapter presents the technical and theoretical analysis of the signature representation in Snort and Suricata. The improvements are developed from characteristics of the theoretical models. Customisation and combination of the best suited compact data structures from existing research, results in two suggested alternative improvements. An algorithm to perform operations on the data structure is also developed and presented. Lastly, an evaluation selects the best suited alternative representation for a proof of concept (PoC) comparison.

6.1 Technical analysis

This section presents a technical description of the signature representation in Snort and Suricata. It works as a basis for further analysis and modelling. The signature representation in Snort and Suricata are described in technical terms, as it is in the code. The description is simplified to only contain the rough layout of the signature representation and not details in the code during initialisation and similar. A technical description of the growth rate is also investigated. Analysing what happens in technical terms when input is added to the data structure will make it easier to create an abstract model later.

6.1.1 Snort

This section describes technical details about the signature structure in Snort. Figure 6.1 is the result from the code analysis and describes the internal signature representation in Snort. An explanation and description follows next. Names from this figure are taken from the code, will be used throughout this section. Knowledge from existing literature are considered when analysing the source code. As mentioned in the related work (Section 3.1.1), it described a linked list of unique signature headers containing another linked list with the corresponding signature options. When analysing the source code, it came apparent that this representation is deprecated. It can still be seen traces of it, where a _OptTreeNode (OTN)

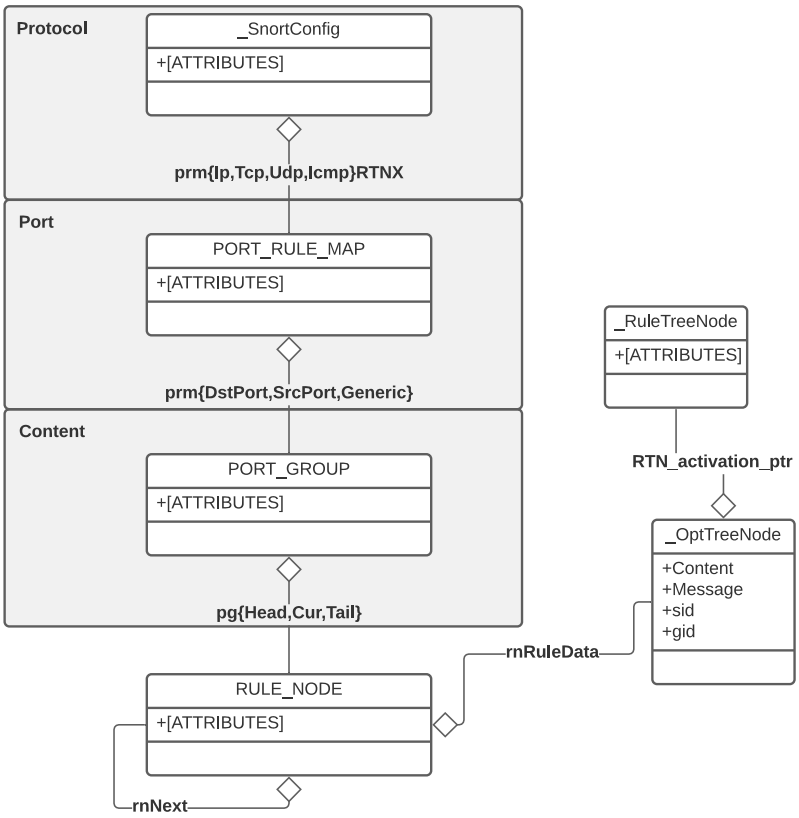


Figure 6.1: Internal signature representation in Snort

references the according _RuleTreeNode (RTN), representing the signature options and header. The old representation would create many groups where each new change in a single signature header field (source/destination IP/port etc.) would make a new RTN node. Traversal of this data structure is linear in time, which is not optimal.

The current implementation of the signature representation in Snort is based on the fact that the source and destination ports can limit the relevant signatures when searching for patterns in network traffic. Not using every field in the signature header, like before. Signatures are also put into a tree structure for faster traversal of the data structure.

Firstly, the object _SnortConfig ties all the information in the whole system together and functions as the root in the tree. The content of the signature is put into the OTN or other objects referred to from the OTN. Based on the protocol (transmission control protocol (TCP), user datagram protocol (UDP), IP and ICMP) a corresponding PORT_RULE_MAP is selected. It then limits the number of possible matching rules further by using the source and destination ports. prmDstPort, prmSrcPort and prmGeneric are arrays of pointers to a PORT_GROUP, where each object is indexed on a port. Each array is the size of MAX_PORTS (65536). Se-

lection of the correct PORT_GROUPs is based on the port specification in the rule header:

1. **prmDstPort**:
   Specified source and destination port
   Any source port and specified destination port
2. **prmSrcPort**
   Specified source and destination port
   Specified source port and *any* destination port
3. **prmGeneric**
   Any source and destination port

The PORT_GROUP contains several attributes, where some of them are pointers to linked lists of _rule_nodes. The correct pointer is selected depending on the content-type in the rule (uri content, no content or regular content).

Comparing the possible options in the rule header to what's used by Snort, the protocol, source and destination port are used together with the content type to group the relevant rules. The flow direction is also indirectly considered by the source and destination port.

If we look at the growth rate for the signature representation in technical terms, the input is a signature or rule. For any signature, a _RuleTreeNode (RTN) and _OptTreeNode (OTN) object have to be created, which is referenced from at least one instance of a RULE_NODE object. The representation can be simplified to imagine a RULE_NODE containing the signature header and options (OTN and RTN). A signature can have two specified ports, which will create two PORT_GROUP objects with each its own RULE_NODE referencing the same OTN. A best-case scenario is that all signatures have the same protocol and *any* source and destination port in the signature header, which is unlikely. This would put all the signatures in one port group, which would minimise the amount of structural overhead. The worst-case scenario will be signatures with different protocols and specified source and destination ports, which creates multiple references to the same signature. The initial size of the data structure without any signatures added, is quite high because each prm{DstPort,SrcPort,Generic} array is initialised to 65536 ports (including 0).

6.1.2 Suricata

This section describes technical details about the signature structure in Suricata. Several details have been omitted, such as temporary data structures used during the initialisation, in order to focus on the main representation.

Suricata represents the signatures a bit different from what Snort does. However, the basic structure is similar. Figure 6.2 is the result from the code analysis and describes the internal signature representation in Suricata. An explanation and description follows next. Names from this figure are taken from the code and will be used in the description and as an example throughout the section.

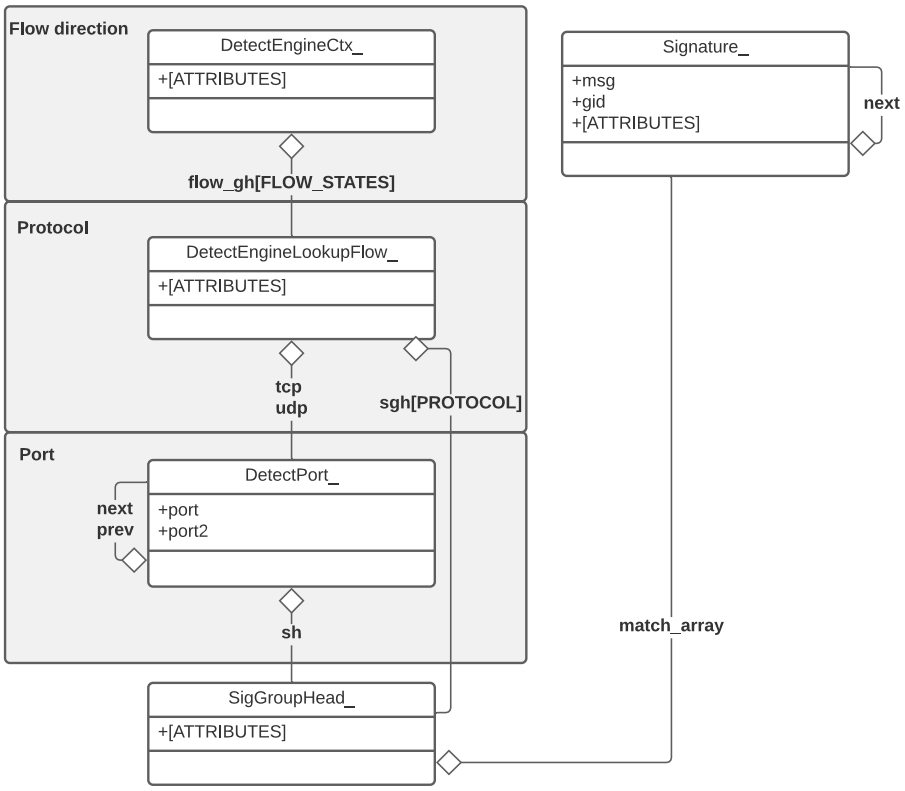


Figure 6.2: Internal signature representation in Suricata

Internally, Suricata creates a tree structure. The object DetectEngineCtx_ is the main detection engine context, and the root in the tree representation. It separates the signatures by the flow direction. Flow_gh is an array of flow directions, which is either to-server or to-client and references a DetectEngineLookupFlow_ object. For TCP and UDP signatures, the protocol and destination port groups the signatures. Other protocols are not separated by destination port. The signatures are then separated by the protocol and port. The pointers “tcp” and “udp” in the DetectEngineLookupFlow_ object, are references to a DetectPort_ object (port object), which describes a port or port-range (port:port2). Port objects are contained in a doubly linked list, where the protocol points to the head of the list. For other protocols, the “sgh” array is used instead, which points directly to a SigGroupHead_ avoiding the port object. Each linked list of Signature_ objects is referenced to by a SigGroupHead_ object, acting as a pointer to the head of the list. Lastly, the Signature_ object contains, or references the data in a signature. The representation is built by traversing the data structure, adding the signatures to the correct branch depending on the flow direction, protocol and port.

In order to conserve memory, Suricata has a feature to limit the amount of port groups, which makes the representation more adjustable than Snort. Fewer port groups will lead to lower performance, but lower memory usage and contrary,

more port groups will lead to higher performance but higher memory usage. With less port groups, more signatures will be grouped together causing irrelevant signatures to be merged with relevant ones for the current traffic. On the contrary, a high number of port groups can cause only relevant signatures to be in a port group, but results in more memory usage to represent all the extra objects and pointers this causes. It works by firstly sorting the signatures based on some criteria, for example number of signatures in a port group etc. Each port object in the list is then iterated and added until a specified limit is reached. When the limit is reached, the rest of the port objects are merged into one group in the linked list. This means that, for example, port objects with few signatures will be merged, and not have their own port object in order to conserve memory. Comparing the possible options in the header that classifies a rule, Suricata uses the destination port, flow direction and protocol.

Even though there are differences between the signature representation in Snort and Suricata, they are quite similar on a general level. A simplified model of the signature representation in Suricata is quite like Snort, in the sense that it separates the relevant signatures using a tree and linked lists. Each leaf on the tree contains a SigGroupHead_ with a linked list of Signature_ objects, which is a merged version of the RTN and OTN nodes in Snort. For each added signature with a unique port, a SigGroupHead_ is created and referenced to by a port object in the case of an UDP or TCP rule. The doubly linked list of port objects utilises more memory per node than a singly linked list, but also allows for traversal in both directions and more efficient insertions and deletions. Compared to Snort, this data structure does not allow for multiple references to a single signature, because the combination of a destination port range and flow direction are unique. If a rule has the *any* keyword as a port, the port range will simply be 0 to 65535.

6.2 Theoretical analysis

This section builds on the findings from the technical analysis (Section 6.1), and analyses the data structures on an abstract and theoretical level in order to find fundamental elements which can be changed in order to improve the representations efficiency.

6.2.1 Abstraction

A decision tree is a type of data structure which is used in machine learning to model a decision [61]. It consists of vertices representing a decision and edges representing mutually exclusive alternatives. For example, a protocol can either be TCP or UDP, but not both at the same time (not considering encapsulation).

If we look at the technical analysis and consider the objects as vertices and their inheritance as edges, the signature representation in Snort and Suricata fits the description of a tree (see Section 3.2.3). More specifically, an ordinal tree where one node is considered root (rooted) [36]. Following the definition of a

decision tree, an abstraction of the signature representation resembles a decision tree, where the edges are labelled according to some criteria, for example port, flow direction etc. depending on the depth. A node's depth is set to the time when the decision happens related to the other decisions. Leaf-nodes represent consequences for that decision path or branch in the tree. We call the vertices representing the signatures *data*, while the vertices representing the path to the "data nodes" the *skeleton*.

6.2.2 Analysis

A tree can be realised in different ways, but most used are pointers to show the relation between vertices. Although the signature representations in Suricata and Snort are different on a technical level (see Section 6.1), a similarity is that they both use pointers.

Space

In order to measure the space requirements of a pointer-based tree, it is of interest to know how many bits are required to represent a number of vertices. A pointer based tree with m vertices and n pointers, use a minimum of $n \log m$ bits. This is because each pointer has to differentiate between m unique vertices [7]. In practice however, one computer word (w) is used per pointer which makes it consume wm bits instead. Depending on the type of system, the size can either be 64 or 32 bits, but a 64-bit word size is most common. Difference in space consumption between Snort and Suricata is reduced to the number of vertices and pointers in the representations.

Time

Recall, the purpose with the signature representation in Snort and Suricata, is to select a group of relevant signatures based on the current network traffic (see Section 3.1). This can be seen as traversing a decision tree based on some criteria. It is of interest to perform this traversal as efficient as possible. Different operations on a tree exist and are of varying degree of relevance, depending on the purpose and goal of the representation. Navarro [7, pg.213] lists operations on ordinal trees in "Compact data structures", where relevant queries on a tree when traversing it are the following:

- *root():* find the root node
- *children():* number of children to a node
- *child(t): t*'th child of a node
- *nsibling():* next sibling of a node
- *childlabelled(l, t): t*'th child of a node with the label l

Supported operations in $O(1)$ on a classical representation of a tree based on pointers are limited to the first child, next sibling and t'th child [38]. The same

can be considered for the *root* operation. Arroyuelo et al. [38] implemented a pointer based tree and realised that the *child(t)* operation required one memory access and took around 18-31 nanoseconds (ns). Another realisation is that such an implementation can be viable in systems which have enough available memory, because it offers fast access to certain operations. Limiting the required operations of a representation can help to find a data structure, which offers better efficiency.

6.2.3 Characteristics

The characteristics of the tree can help to select an improved representation. Maximum amount of children to a node is 2^{16}, when considering one vertex for each port like in Snort. Secondly, the vertices contain attributes, especially in the leaf nodes, which represent the signature content. After the tree is built, it is static until more signatures are added. The main structure of the decision tree (skeleton) is built during the initialisation and kept static during the runtime. As mentioned during the technical analysis, the signatures are grouped into different levels (see Section 6.1). In Suricata, the number of nodes in the skeleton depends on the number of unique ports in the signature headers. The number of bits compared to the amount of edges are quite high and wasteful, considering the *wm* space consumption. Thus, the relation between the vertices consumes space. Another characteristic is that the leaf nodes have attributes which needs to be considered in the suggested improvement.

6.3 Suggested improvement

This section presents two alternative improvements for the representation of signatures in Suricata and Snort. The characteristics from the abstract version of the representation (see Section 6.2) and the abstractions information theoretical lower bound space usage and time consumption determine the new improvements.

By reducing the number of bits required to represent the edges in the tree, a more compact representation can be used. In this case, moving away from a pointer based representation to a bitvector based one, can reduce the space consumption of the tree closer to optimal. However, the traversal time depends on the required operations on the data structure. Using techniques from compact data structures, the time it takes to navigate the representation can still be low. Thus, ensuring an efficient representation.

6.3.1 Alternative one

This section presents the first alternative representation of the signature representation. Section 6.2 modelled the signature representation in both Snort and Suricata as a decision tree, where each leaf node is a set of signatures. This section splits the alternative suggestion into two parts, where one is the skeleton and

the other connects the signatures to the skeleton. The first part presents a data structure which is closer to the information theoretical minimum for a labelled tree, using the best fitting compact data structures from previous research. The second part connects a set of signatures to a certain branch in the tree, using a developed algorithm and according data structures.

Figure 6.3 shows an illustration of the suggested improvement, and is used during the section as an example and reference. The edges are labelled according to fields in the signature header. In this example, the nodes in the second level are divided by the flow direction. The label c means flow direction "to client" while s represents "to server". Then, the nodes are further divided by the protocol, where T means TCP, while U means UDP. Below that, the ports used by the signatures (P_i) decides where the signature decides in the tree. The alternative representation consists of two bitvectors (B, B_v), one sequence (S) and one array (S_v), which is further explained in the following sections.

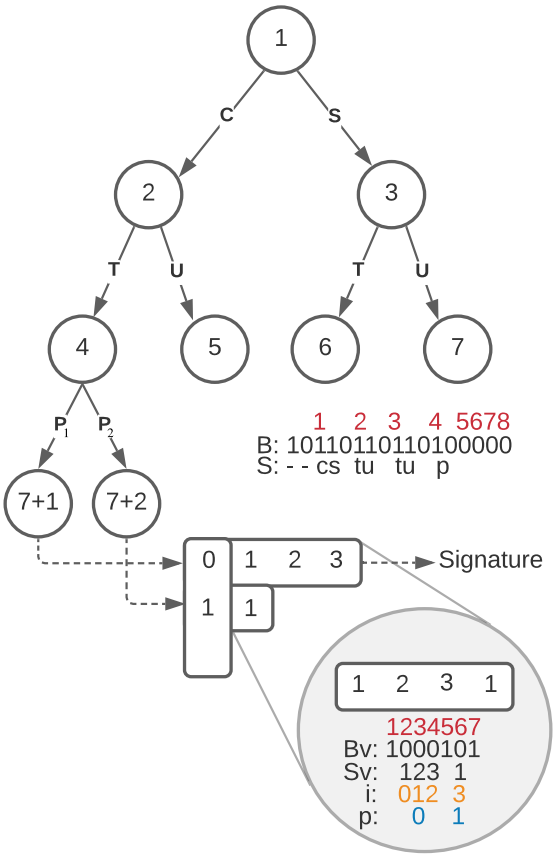


Figure 6.3: Signature representation improvement - alternative one

Structure

In order to suggest a compact representation for the labelled tree (first part), it is valuable to know which operations are needed in the abstract data structure. Recall, from the theoretical analysis of the abstract signature representation, important operations on the data structures is to find a child node with a specific label. For a labelled tree, Navarro [7, pg.220, 244] suggests the *LABELLEDCHILD* algorithm seen in Algorithm 1, to find a node with a certain label in a LOUDS representation. The operations shown here are of interest to perform in as little time as possible in order to achieve an efficient improvement.

Algorithm 1: Operations on a labelled tree using the LOUDS representation of a compact tree [7, pg.220, 244]

Input : labelled tree T (seen as the sequence S and bitvector B) label l, current state v and index j
Output: The destination vertice

1 **Procedure** `NODEMAP(`i`)`**:**
2 ⌊ **return** $rank_0(B,i)$
3 **Procedure** `NODESELECT(`v`)`**:**
4 ⌊ **return** $select_0(B,v)+1$
5 **Procedure** `CHILD(`i,j`)`**:**
6 ⌊ **return** `NODEMAP` $(select_0(B,rank_1(B,i+j))+1)$
7 **Procedure** `LABELLEDCHILD(`l,v,j`)`**:**
8 $i \leftarrow$ `NODESELECT` (v)
9 $s \leftarrow rank_1(B,i)-1$
10 **return** `CHILD` $(i,select_l(S,rank_l(s)+j)-s)$

Two different indexes are used, which can cause confusion and must be explained. One is the node identifier (ID) v, which is the node ID shown in Figure 6.3 as numbers inside the circles. The other one is the nodes index in B, which is called i. *Nodeselect* converts the node ID to an index in B, and opposite for *nodemap*. The procedure on Line 7 takes a node ID and a label as arguments and returns the j'th child with that label.

From these operations, it is apparent that rank and select operations on the bitvector B have an impact on the time usage. Another observation is that only simple queries on the tree B are required, such as *child*, which can be solved using rank and select. The LOUDS structure has the option to support constant time rank and select in $2n$ bits, which is optimal for simple queries. However, the representation is limited by its operations, as mentioned in Section 3.2.3. DFUDS supports more operations, but cannot perform them in constant time with the same space as LOUDS. Because advanced operations are not required, it is more efficient and simpler to use LOUDS. Such a representation has asymptotically optimal space usage, with possibilities for constant time queries, as explained in Section 3.2.3.

Thus, using the fundamentals of data structures from previous research are best suited for this part of the representation.

A LOUDS structure represented as a bitvector, cannot support labels in a labelled tree. Thus, a sequence S is needed as well containing the labels. A potential time bottleneck in Algorithm 1 is select and rank operations on the sequence S. Using a plain array would require a linear search, which is too slow. Wavelet trees, however, can perform these operations in logarithmic time $O(log\ \sigma)$, and are chosen instead.

Because one signature can be referenced by multiple ports, references to signatures from a path in the tree will be space consuming. This is the background for the development of the array S_v, representing the connection between the decision tree and signatures (second part). When this array is constructed, it is of interest to read the elements within it, which means an array would offer better compression rates and access times compared to a sequence. The bitvector B_v separates the elements between the port groups, instead of using a sequence per group, which would require a reference (pointer) to each one. For example, the first portgroup contains three signature references, which means that the bits 1000 is inserted into B_v. Because the number of vertices in the skeleton is static, it can be used to get an index into S_v with the algorithm seen in Algorithm 2, which was developed.

Algorithm 2: Find signatures in a portgroup, based on the node identifier

Input : The sequence Sv and bitvector Bv with a node identifier v
Output: Relevant signatures for a node ID (port group)

```
1 Procedure GETSIGNATURES(v):
2     i ← rank_0(Bv, select_1(Bv, v + 1))
3     j ← rank_0(Bv, select_1(Bv, v + 2))
4     for i to j by 1 do
          /* Do action on Sv[i]                    */
```

For example, if we use Figure 6.3 and portgroup 1 ($v = 1$) which contains the signature reference 1, the following operations would take place. $select_1(Bv, v + 1)$) would return 5, because the second 1 in Bv is located at position five. i would then be 3 because there are three zeroes before position five in Bv. The same process happens with j, where the position is incremented by one in order to get the start of the next index, which is the end of the current.

Construction

The LOUDS tree is created by a level order insertion of nodes. For each node, the bits 1^c0 are inserted into B, where c is the number of children to a node. If a node is a leafnode (no children), 0 is inserted into the bitvector. In addition, labels are inserted into the sequence S when adding a nodes child. The tree itself, meaning

the LOUDS structure B and S can be built prior to adding the signatures. When the signatures are added, the tree is traversed according to the header information in the signature. A reference (pointer) to a signature is added to the sequence Sv, representing the connection between the decision tree and signature for a particular branch. 10^n is inserted into B_v for the n signatures tied to a particular branch. Drawbacks here is that the port groups must be predefined before the signatures are added in order to preserve the leafrank index into Sv.

Time and space

If we compare this alternative representation's time and space usage to the existing implementation, it is slower but offers a more compact representation. Traversal of the LOUDS representation of a labelled tree is limited to the $O(log\ n)$ operation times on the wavelet tree containing the labels (S). However, it is compact with $O(n)+n\ log\ \sigma+o(n\ log\ \sigma)$ bits. If we use Elias-Fano codes to represent the array S_v we can achieve a low space usage, as well as fast access times (See, Section 3.2.1).

6.3.2 Alternative two

This alternative is based on the same thought as the alternative one, where the representation is divided into two parts. It uses a LOUDS labelled tree for the first part (skeleton), like in the alternative one. The difference is in the second part, which is using the leaf nodes index to connect a signature to a port. Figure 6.4 shows an illustration of the suggested improvement. The figure use the same labels as explained in the first alternative.

Structure

In this alternative, the whole tree is represented using the LOUDS structure. When adding a signature to the structure, the tree is traversed according to the flow direction, protocol and port. A new child node without a labelled edge, marked as x, is then added to the vertex pointed to by the port edge. The node ID of the leaf nodes, index the relevant signatures directly instead of a helper array like alternative one. This would lead to a space consumption of $2n$ bits which is less than the bits used with Elias-Fano codes in S_v plus the size of B_v in alternative one (see Section 6.3.1). A drawback is that the index of the other nodes is shifted when inserting a signature. Because the bitvector B_v must be built by a level-order traversal of the nodes, adding nodes different from this traversal would make the operations incorrect. This means that some temporary storage is required to get the order of the nodes correctly, before inserting them in the bitvector.

LOUDS representation supports a *nextchild* operation in constant ($O(1)$) time, which is helpful when iterating the relevant leaf nodes [7]. Another drawback with this representation is that the original references to the signatures in *sig_list* would must be kept in its original form, with pointers. This would require at least

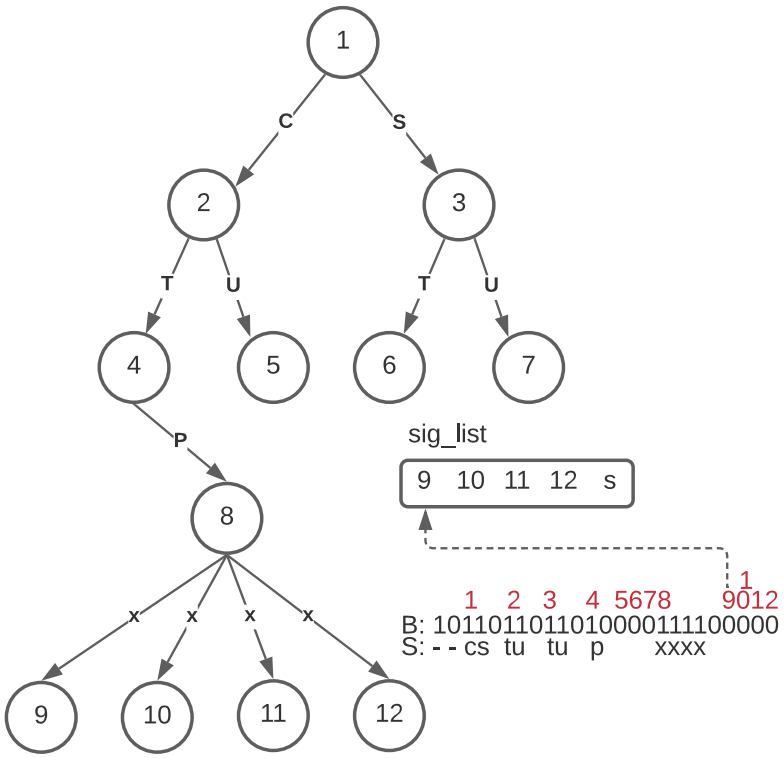


Figure 6.4: Signature representation improvement - alternative two

$n \log n$ bits, which is in practice wn, to store the n signatures and pointers, which is a waste of space. In the alternative one (see Section 6.3.1), the *sig_list* array is not needed because the references is stored directly in the Sv array, which requires fewer bits. Because of this, the space consumption would be higher than with the alternative one, with a quadratic space complexity in practice, which is like the existing implementation.

6.3.3 Evaluation

From the technical analysis of the signature representation in Snort and Suricata, we see that the existing implementations can be seen as a labelled tree. Based on the worst-case entropy of a labelled tree, we see that there is a gap between the existing implementation and the theoretical optimal space usage. From the characteristics of the signature representation, we see that moving away from a pointer-based representation to the one based on bitvectors can bring us closer to this lower bound space usage. Two alternatives were presented, where a LOUDS version of a labelled tree was used because the required operations on the tree were quite simple. Using a compressed array for the signature references, as in alternative one, allows for using pointers directly to the signatures and therefore

avoiding the dominating $n\ log\ n$ term, as in the alternative two.

Because of this, alternative one are chosen as the best alternative representation and is implemented in a PoC practical experiment (see Chapter 8). It is compared to the existing representation using the efficiency metric from Equation (5.1).

Chapter 7

Fast pattern matcher

This chapter presents the technical and theoretical analysis of the fast pattern matcher - Aho-Corasick deterministic finite state automaton (DFA) in Snort and Suricata. The improvements are developed from characteristics of the theoretical models. Customisation and combination of the best suited compact data structures from existing research, results in two alternative improvements. An algorithm to perform operations on the data structure is also developed and presented. Lastly, an evaluation selects the best suited alternative representation for a proof of concept (PoC) comparison.

7.1 Technical analysis

This section presents a technical description of the fast pattern matcher in Snort and Suricata. It works as a basis for further analysis and modelling.

7.1.1 Snort

Figure 7.1 shows the internal representation of the Aho-Corasick DFA implementation (Version 1). Names used in this figure are taken from the code in Snort, and will be used during this description.

Several implementations of Aho-Corasick exist in Snort, but this version is considered because of comparability with Suricata. Existing literature have done the same when conducting a quantitative comparison of the two intrusion detection systems [44].

It is quite a straightforward implementation of the original Aho-Corasick algorithm. A two-dimensional array is used to store the state transition table (the goto function). The ACSM_STRUCT object is the main reference to the data used by the Aho-Corasick algorithm. It contains an array of pointers to ACSM_STATETABLE objects, one for each state. This represents one dimension of the state table. The NextState array represents the other dimension, covering the alphabet length which is 256 (ASCII). It contains the next state in the automaton.

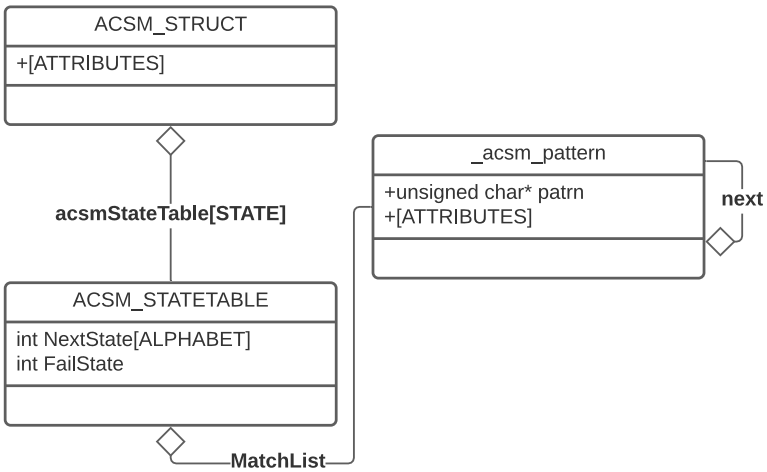


Figure 7.1: Snort fast pattern matcher representation

Each state also has a MatchList, which is a singly linked list of patterns terminated at this state. During the construction of the data structure, the NFA is built first using the FailState as a temporary pointer representing fail state transitions for each valid state. However, from the code, it does not seem like this pointer is freed when converting the NFA to DFA, which leads to unnecessary use of memory. Next, the DFA is built by resolving the fail state transitions and inserting their destination into the state transition table.

7.1.2 Suricata

Figure 7.2 shows the internal representation of the Aho-Corasick DFA implementation (Version 1) in Suricata. Names used in this figure are taken from the code, and will be used during this description.

The internal representation of the fast pattern matching algorithm, Aho-Corasick is a bit more advanced and optimised than with Snort. However, on a general level they are still quite similar. A two-dimensional array (t_state_table_u{16,32}) is also used in Suricata to represent the transition state table. One dimension is the current state, while the other is the alphabet size (ASCII). Suricata optimises the amount of memory used depending on the number of states. If there are more than 2^{16} states, 32-bit are used in the transition table instead of 16-bit, in order to limit the unused bits (overhead). This feature makes Suricata consume less memory than Snort in certain scenarios. Some additional data structures are used during construction of the state table, such as a goto- and failure-table and a linked list of patterns. However, these are all freed after construction of the DFA, and consequently not considered as a part of the final data structure. The object SCACOutputTable_ contains a list of pattern identifiers terminating in a given state. A pattern ID refers to a SCACPatternList_ object, containing a signature id

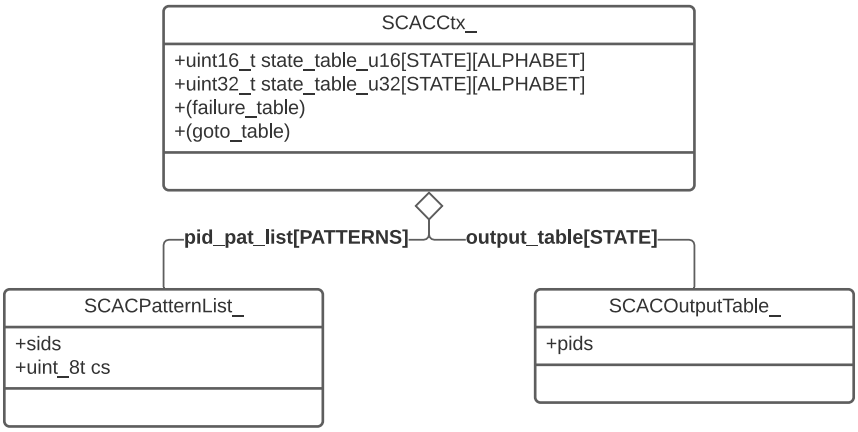


Figure 7.2: Suricata fast pattern matcher representation

(sid) as well as character pointer to the case sensitive pattern (cs).

Suricata's and Snort's internal representation of the Aho-Corasick algorithm are quite similar. Suricata has the advantage that it limits the pointer size, to fit better when there are fewer states. The representation is very space consuming because each state has initialised as many pointers as the amount of symbols in the alphabet. The main difference is how they reference the output function, to find possible matching signatures. However, it is not relevant in this case where the focus is on the efficiency of the DFA state transition table.

7.2 Theoretical analysis

This section is an extension of the technical description of data structures in Suricata and Snort. It aims to analyse the existing implementation of the fast pattern matcher in the two systems.

7.2.1 Abstraction

The implementations of Aho Corasick DFA in Snort and Suricata is quite similar. A two-dimensional table is used to store the state transition table (the goto function), which is the main data structure in the algorithm. This results in a matrix, $M[1,n][1,\sigma]$ where n is the amount of states and σ number of characters in the alphabet. An entry in the table is a reference to the next state from a particular state and character. The state transition matrix can be modelled as a labelled graph where an edge from a source to a destination vertex, plus a label on the edge from the set of states indicates the next vertex (state). A transition is uniquely identified by a tuple (s_i, s_j, σ_k), where σ_k is a label and s_i and s_j is a source and destination vertex.

7.2.2 Analysis

The matrix used to represent the state transition table in Snort and Suricata is based on pointers. This section aims to analyse this implementation with regards to space and time.

Space

A two-dimensional matrix, especially based on pointers, is quite space consuming. It is not practical unless the graph is very dense or in other words, the nodes have a high degree. An adjacency matrix $M[1,n][1,\sigma]$ requires $n\sigma$ bits [7]. However, that indicates that each entry in the matrix uses one bit, and typically means that $M[i][j] = 1$ if node i is adjacent to node j. This is not the case with the state transition table in Snort and Suricata, because two nodes can be adjacent only with certain labels. The labels are chosen from the number of states n, which means that the amount of bits needed to represent the matrix is at least $\sigma n\, log\, \sigma n$, because σn pointers have to separate between σn different vertices or entries in the matrix. When comparing with the lower bound space usage of a directed labelled graph (see Section 3.2.4), there is a huge gap between the current space usage and the worst-case entropy. In other words, there is room for improvement. A challenge is that the number of edges in the graph e is not a part of the equation, because it is assumed that $e = n\sigma$ in the two-dimensional matrix.

Time

By modelling the state transition table as a labelled graph, the goal is to traverse the graph. Based on input from the search text, a new vertex is chosen. In the two-dimensional state transition matrix based on pointers, the access times are constant, because it requires one memory access for an input state and character to get the next state. It has presumably similar access times as the signature representation, in nanoseconds.

7.2.3 Characteristics

A characteristic when modelling the Aho-Corasick DFA state transition matrix as a labelled directed graph is that each state has an equal amount of edges. Each state also has no duplicate labels, meaning that each edge has only unique labels for a certain state. The matrix is also supposed to be quite sparse, with more labelled edges to state zero than any others. These characteristics allow us to further simplify the representation, in order to save both space and time, making it more efficient.

7.3 Suggested improvement

This section presents suggested improvements for the fast pattern matcher. Characteristics from the abstract version of the representation (see Section 7.2) and the abstractions information theoretical lower bound space usage and time compromise determine the new improvements.

7.3.1 Alternative one

This section presents the first alternative representation for the fast pattern matcher or Aho-Corasick DFA state transition table. It is based on clustering in the state transition table, when only a few characters in the alphabet are used more than others.

Structure

Recall, the DFA state transition table can be modelled as a labelled graph, which has a worst-case entropy described in Section 3.2.4. This alternative is based on a hypothesis that only a few of the labels in the alphabet are used in practice. If only a few labels or bytes are used by the rules in practice, this can result in a clustered graph which can be taken advantage of to create a compact representation.

In order to test out this hypothesis, a program was used to parse the content field in all the datasets used in this thesis, (see Code listing A.10) to see if some characters are used more than others. The result can be seen in Figure 7.3, and shows that this is indeed the case.

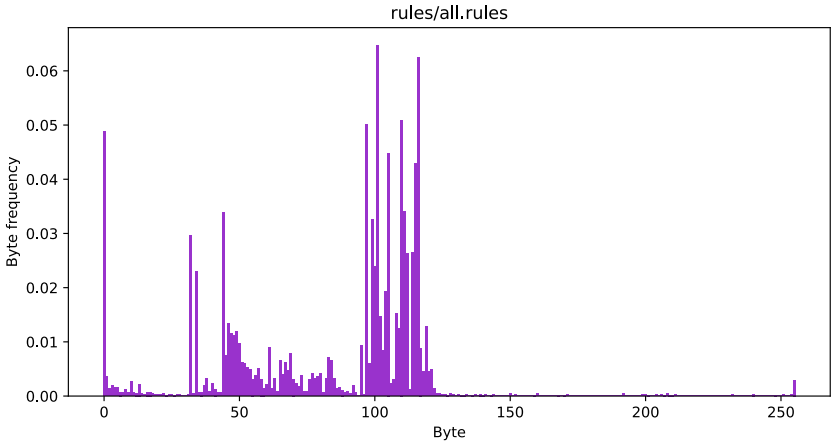


Figure 7.3: Byte frequency in all the datasets

Only 21 bytes out of 2^8 have a frequency greater than 0.01. From the figure we can see that they cluster around the upper and lowercase letters of the English alphabet.

The first alternative representation builds on this idea and is based on the edge triple (s_i, s_j, σ_k), which identifies a transition in the state transition table (see Section 7.2). It can then be represented as a point in the three-dimensional plane. This introduced an idea of using a k^3-tree to represent the triple, or a k^2-tree with weighted points, A benefit with a k^n-tree is that it is well suited for clustered graph, because an area without nodes consumes few bits in the data structure (see Section 3.2.4).

Recall, a binary relation between two states can be modelled as an adjacency matrix. The idea is then to add a third dimension to the adjacency matrix, which contains the labels for each state. Large areas without a relation can be represented using a few bits, using a k^3-tree. Figure 7.4 shows an illustrative example, using the state transition table from Figure 7.5.

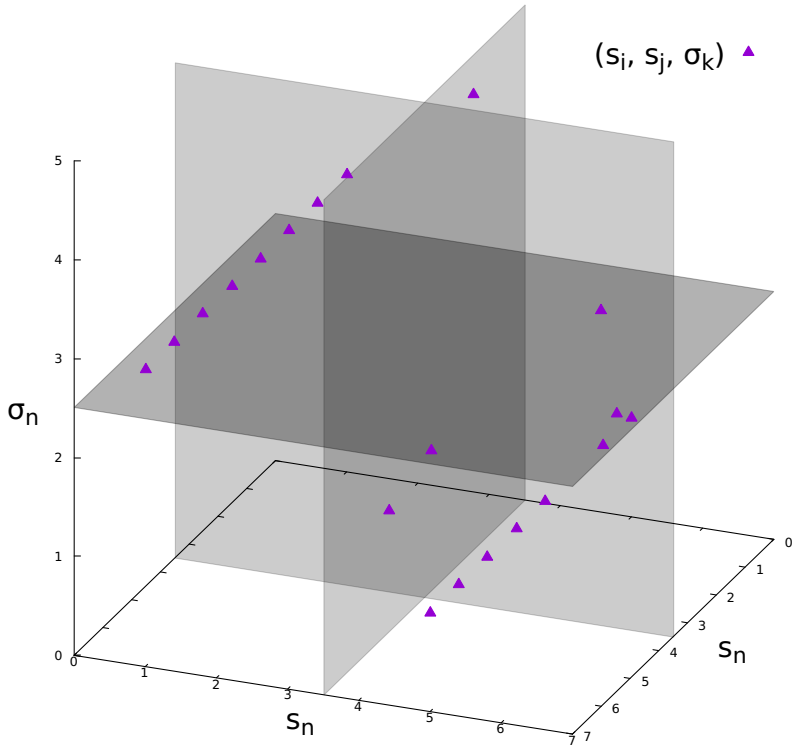


Figure 7.4: Fast pattern matcher improvement - alternative one

The three-dimensional space is divided into 2^3 areas. In this example, there is

a tendency for the points to cluster together and three areas contain none or very few points.

From the existing literature, a k^3-tree was used in compact representation of LiDAR points (see Section 3.3.2) [45]. Relevant operations on LiDAR point clouds are to get objects within a region, which is not relevant or transferable to this scenario. Existing literature about representing a labelled directed graph as a k^3-tree and relevant operations were not to be found. However, Navarro [7, p.293] states that the procedure *adjacent*, which tells if there is an edge from one node to another, can be completed in $O(log_k\ n)$ time on a k^2 tree. It is not certain whether this is transferable to a k^3 representation.

Other literature points out that the k^n-tree generalisation of a k^2-tree is not suitable for $n > 2$ [43]. This is because it is hard to find sparsity and clustering in three-dimensional or higher datasets. Instead they suggest using a layered approach where each Z-dimension is partitioned into a separate k^2-tree, which is named *Interleaved K^2-tree (IK^2-tree)*. The root has then k children with $|Z|$ bits each instead of k bits for each node. A similar structure to the k^2-tree allows for similar time guarantees and space usage. Experimental results were obtained by testing the IK^2-tree on Resource Description Framework (RDF) dataset which contains the tuple (subject, predicate, object). By considering the predicate as the label in the state transition table, we get the query *(s,p,?)*. This corresponds to the "*neighbour*" query, which finds the next state based on a label and a state. From the experimental results in the paper for that query, the IK^2-tree was slower than the existing representations. However, it is quite compact.

7.3.2 Alternative two

While the first alternative was based around clustering in the state transition matrix, this alternative suggests an improvement based on its abstraction and characteristics. Recall, Section 7.2 modelled the state transition matrix as a labelled directed graph. An issue with the current implementation, is the quadratic size of the matrix which makes it space consuming. Characterises from the matrix are used to find a more efficient improvement. An example of a DFA state transition matrix and the suggested alternative representation is shown in Figure 7.5, using the patterns "snort", "on" and "snow". This figure is used throughout the section, as a reference and example.

Structure

First, we give some background theory of a labelled graph other than what was explained in Section 3.2.4. Navarro [7, p.290] presents an algorithm to find the neighbour of a node with a certain label, which is shown here in Algorithm 3. It utilises two sequences and two bitvectors to represent a labelled graph. The sequence N contains a concatenation of unique node identifiers, where the bitvector B_L separates the identifiers by their labels. L contains the edge labels in the order of the adjacency list, where a similar bitvector B groups the labels by the source

-	N	O	R	S	T	W
0		5		1		
1	2	5		1		
2		3		1		
3		5	6	1		4
4		5		1		
5		5		1		
6		5		1	7	
7		5		1		

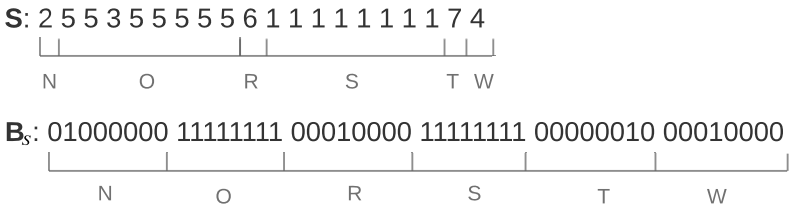


Figure 7.5: Fast pattern matcher improvement - alternative two

node, as explained in Section 3.2.4. Thus, p and r are calculated using the node identifier v and label l. The two bitvectors are required, because a node can have an arbitrary amount of edges to other nodes. Next, the offset q is calculated by counting how many occurrences of the label l happens up to the index p in L. Since N is grouped by labels, the correct label group r and offset into that group q is summed up to find the next state.

Algorithm 3: Operations on a labelled directed graph, where a node can have arbitrary amount of edges with any label [7, p.290]

Input : labelled directed graph G (seen as the sequences N and L, bitvectors B_L and B), label l, current state v and index j
Output: The next state

1 **Procedure** `NEIGH`(G, l, n, v)**:**
2 $\quad$ p $\leftarrow$ select$(B, v) - v$
3 $\quad$ r $\leftarrow$ select$(B_L, l) - l$
4 $\quad$ q $\leftarrow$ rank$_l(L, p)$
5 $\quad$ **return** $N[r + q + j]$

Due to the characteristics of the state transition table in Aho-Corasick DFA modelled as a labelled graph, the Algorithm 3 can be improved in order to fit this model. Based on these observations, the following improvements are suggested.

Because all states have the same amount of edges, there is no need for a

bitvector to separate the labels. Thus, p can be calculated using the state v and alphabet size σ.

$$p \leftarrow (v\sigma) \tag{7.1}$$

Because each state has the same amount of edges in the same order, L can be omitted as well. The calculation of q can then be simplified.

$$\begin{aligned} q &\leftarrow p/\sigma \\ q &\leftarrow v \end{aligned} \tag{7.2}$$

Thus, there is no need to calculate p either. Since, B_L serves the same functionality as B, it can be removed and replaced as well using the label (character in the search text - l) and number of states (n).

$$r \leftarrow ln \tag{7.3}$$

This results in a simpler calculation of the index in N, compared to the original one. However, it assumes that array N is of quadratic size, which is space consuming. Because most of the indexes in the state transition matrix are presumed to be zero, the array can be made shorter using the help of a bitvector $B_s[1, n\sigma]$. Here $B_s[i] = 1$ if the entry in the matrix is not zero. N can be used to only store the non-empty entries in the state transition matrix, called S. This is shown in Figure 7.5. Algorithm 4 was then developed to select the next state based on the labelled edge from a current state. The suggested improved representation is a compact representation of a concatenated and ordered adjacency list. It utilises one array S and bitvector B_s in order to represent the data.

Algorithm 4: Operations on a labelled directed graph, where each node has the same amount of unique edges

Input : labelled directed graph G (seen as the array S and bitvector B_s), label l, number of states/nodes n, current state v
Output: The next state

```
1 Procedure NEIGH(G, l, n, v):
2     p ← (ln) + v
3     if B_s[p] then
4         return S[rank(B_s, p)]
5     else
6         return 0
```

Space and time

In order to achieve a compact representation, the array S can be compressed using Elias-Fano codes. It is chosen because of a close to optimal space usage, and fast

access times (See Section 3.2.1). The space consumption of S and B_s depends on several input variables. Alphabet size (σ) and number of states (n) makes the size of the table, which equals the number of bits in $B_s[1, n\sigma]$. The size of S on the other hand, depends on the non-zero states in the table. In the worst-case, there are no zero-states in the state transition matrix which makes $S[1, n\sigma]$ as well. By looking at the byte frequency in all the tested datasets (see Figure 7.3) this is not the situation.

7.3.3 Evaluation

The first alternative improvement for the fast pattern matcher is about using a k^3-tree to represent the three-dimensional state transition matrix of a directed labelled graph. By using the IK^2-tree variant, it is possible that a compact representation can be used. However, the worst-case time guarantees and results from experiments described and performed by existing literature can make the alternative quite slow. It is not certain whether the three-dimensional state transition table is equally sparse and clustered for all datasets.

Because of this, alternative two was selected amongst the suggested improvements to the fast pattern matcher, and will be implemented in a PoC comparison.

Chapter 8

Experimental Result

This chapter implements the selected alternatives from Section 6.3 in C++, as mentioned in Chapter 4 and compares their efficiency using the developed metric from Equation (5.1). Two programs are made to compare the alternatives to the original implementations, one comparing the fast pattern matcher and the other, the signature representation. The results are presented after describing the setup and programs used to get the results. For reproducibility and transparency reasons, the programs have been uploaded to GitHub [58].

8.1 Environment

This section describes the testing environment which is used to get the results.

8.1.1 Physical setup

The experiment is conducted on a Lenovo ThinkPad T460P with the following specifications and environment:

- OS: Manjaro Linux 20.2.1 Nibia
- Kernel: x86_64 Linux 5.10.18-1-MANJARO
- Disk: 500G
- CPU: Intel Core i5-6300HQ @ 4x 3.2GHz
- GPU: Mesa Intel(R) HD Graphics 530 (SKL GT2)
- RAM: 15876MiB

8.1.2 Logical setup

The compiler g++ version 10.2.0, with the flags "-std=c++17 -O3 -ffast-math" is used to compile the programs for this experiment. Libraries mentioned in Section 4.6 are used with the following requirements and versions:

- Boost - version: 1.75.0
- CMake - version: 3.20.2

8.2 Description

This section describes the programs used to compare the alternatives and their existing representations.

8.2.1 Signature representation

To test the original signature representation as it is in Snort and Suricata against the improvement, it is of interest to measure space consumption and time usage. Space consumption is measured when the data structures are built. Time usage on the other hand requires a bit more explaining. Recall, the objective with the signature representation is to select a relevant group of signatures for the incoming network traffic. For example, transmission control protocol (TCP) signatures are not relevant for network traffic using the user datagram protocol (UDP) protocol. Therefore, the time usage is measured by the time it takes to traverse one path of the signature tree. To capture the essence of the original implementation, as it is in Snort and Suricata, a pointer-based tree representation are created by using flow direction, port and protocol to separate the signatures into groups.

The space consumed by the signatures themselves is not included, because it does not differ between the original and the improved representation. The complete program used in the proof of concept (PoC) can be found in Code listing A.1, but a rough explanation of the program flow follows and is shown in Algorithm 5. Firstly, a replica of the original implementation is made using parts of the code from Snort to make it similar to the existing representation. The signatures are parsed and added to a linked list at the correct branch in the tree. Next, the alternative representation is made by first building the skeleton according to certain fields in the header. Depending on the header information in the signature, the skeleton is traversed. A pointer to the signature is then added to a temporary array as an unsigned integer, which is later used to create the array S_v and bitvector B_v grouping the array by ports.

The last signature in the branch containing most signatures is chosen as a search object. Data from this object (flow direction, protocol and destination port) is used to search for a matching group of signatures and record the time in both implementations. The correctness of the alternative implementation is ensured by counting the number of signatures in the branch of the search object and comparing them to the original. For example, if 15 signatures have been categorised as flow direction to server, protocol TCP and port 80 in both the original implementation and the alternative one, we can assume that the rest have been categorised correctly as well.

8.2.2 Fast pattern matcher

The fast pattern matcher PoC is based around comparing the space and time for the original implementation against the suggested improvement. Space consumption is also measured when the data structures are built. References to a match

Algorithm 5: Signature representation - proof of concept overview

```
1  Parse the signature file
2  for number of runs do
       /* Build the original representation                    */
3      Initialise objects in the static part of the tree (skeleton)
4      for signature in signatures do
5          Traverse the tree based on the signature protocol and flow
             direction
6          for port in signature.ports do
7              Add a pointer to the signature in a linked list to the correct
                 branch, based on the port
       /* Build the improved representation                    */
8      Find all the ports in use by the signatures O(n)
9      Allocate a labelled tree and add the static nodes (skeleton)
10     for signature in signatures do
11         Traverse the tree based on the signature
12         for port in signature.ports do
13             Add a signature pointer to a temporary two-dimensional array,
                 indexed on the port
14     Build the signature reference array S_v and bitvector B_v from the
         temporary array
15     Record space usage from both of the structures
       /* Searching                                            */
16     Find a reference to a signature of which to search after
17     Traverse both the original and the improved representation using
         Algorithm 2, and record the time it takes to find the signature
18     Assert that the suggested improvement finds as many signatures as
         the original
19     Save time and space information
20     Delete and free the allocated objects which are not overwritten
21 Calculate the efficiency using the collected info and Equation (5.1)
22 Write collected information to I/O
```

(MatchList) is not accounted for, because they are the same in both the improved and existing representation. Temporary data structures used during initialisation are not considered as a part of the space consumption either. To compare the time usage of the fast pattern matcher, it is of interest to compare the time spent searching through a search text. Various search texts have been tested. In order not to introduce any biases to the algorithm, it is important to have close to a uniform distribution of bytes in the search text. The PoC is meant to serve as a general test of the performance, and thus every possible case of bytes in the search text has to be accounted for. If the algorithm are run with only numbers and letters in upper- and lowercase, it could be performing differently with different input characters. Various search texts with the size of 10KB are generated with the following UNIX commands:

```
$ head -c 10K </dev/urandom > mixed.txt
$ head -c 10K </dev/urandom | base64 | head -c 10K > common.txt
$ head -c 10M </dev/urandom | grep -P -a -o "[^\x00-\x7F]" | tr -d "\n" | head -c
      10K > less-common.txt
```

Figure 8.1 presents the byte frequency for the generated search texts used in the tests. It shows that the entropy of the search text in Figure 8.1a, containing a mix of all the 2^8 bytes, is quite high. The two other search texts, Figures 8.1b and 8.1c represent common and less common letters in the rulesets, according to the test in Figure 7.3.

The complete program can be found in Code listing A.2, but a rough program flow is as follows and is shown in Algorithm 6. Firstly, an input file of signatures is parsed and the longest content string of each signature added to the fast pattern matcher. An almost direct port of the Aho-corasick DFA algorithm from snort (from the file “acsmx.cpp”) is used to represent the original implementation, and build the DFA state transition table. The modifications made do not impact the space usage or time consumption of the algorithm, because they do not affect the algorithm’s general functionality. Time it takes for the original representation to search through the search text is registered. Next, the alternative representation is made by iterating the state transition matrix and adding the states which are not zero to the array S, as explained in Section 7.3.2. A “0” is added to the bitvector B_s for each zero state and a “1” for the rest.

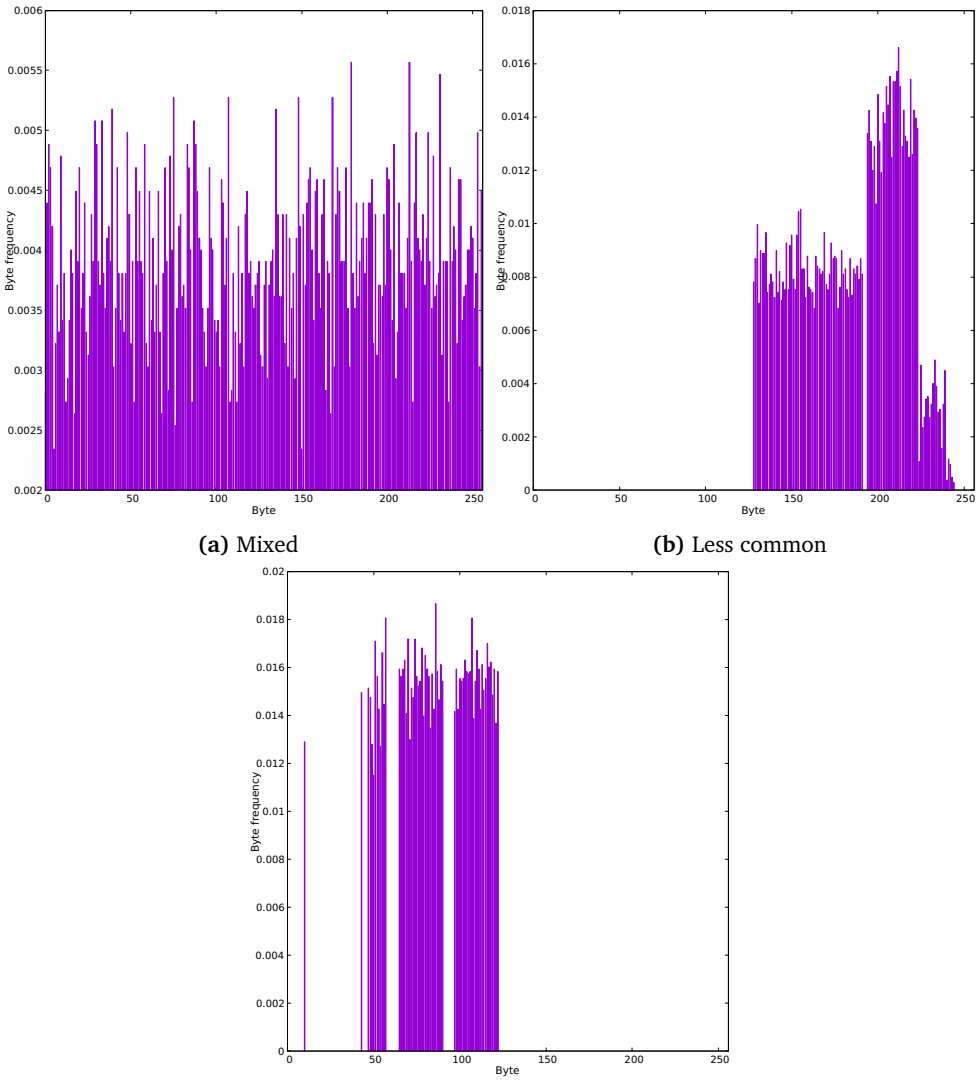
8.3 Results

After the PoCs are implemented and tested, results for various data-sets are collected and analysed. Table 8.1 shows the results for the signature representation, while Table 8.2 shows results from the fast pattern matcher.

100 runs are conducted in order to capture the variation in runtime, which depends on other tasks the CPU has as well as other miscellaneous outside factors. All input/output (I/O) operations and printing statements are disabled during the experiments to prevent unexpected waiting for I/O time to impact the results. In

Algorithm 6: Fast pattern matcher representation - proof of concept overview

```
1  Parse the signature file
2  Read the input search text as bytes
   /* Build the original representation                          */
3  for signature in signatures do
4      Add signature.content to the Aho-corasick NFA
5  Build the DFA state transition table from the NFA
6  for number of runs do
       /* Searching                                              */
7      Search in the original representation using the DFA state transition
         table, and record the time
       /* Build the suggested improvement based on the DFA state
          transition table                                       */
8      B_s ← bitvector[1,ALPHABET_SIZE ∗ MaxStates] = 0
9      c ← 0
10     for i in ALPHABET_SIZE do
11         for j in MaxStates do
12             state ← DFA.stateTable[i][j]
13             if state then
14                 add state to sequence S
15                 B_s[c] ← 1
16             c++
17     Search with suggested improvement using Algorithm 4
18     Assert that the suggested improvement finds as many matches as the
         original
19     Save time and space information
20     Delete and free the allocated objects which are not overwritten
21 Calculate the efficiency using the collected info and Equation (5.1)
22 Write collected information to I/O
```



(a) Mixed

(b) Less common

(c) Common

Figure 8.1: Fast pattern matcher - search text byte frequency

order to prevent caching between the runs from intervening, the used objects and variables are freed and re-declared between each run, if they are not overwritten.

The efficiency metric formula from Equation (5.1), explained in Chapter 5, is used to determine the efficiency, shown as the rightmost column. In these tables, the space is measured in Bytes (B), while the time is measured in nanoseconds (ns). Recall, the existing representations space and time is labelled W_e and X_e, while the improvements space and time is W_i and X_i, respectively. A measurement of central tendency of the runtime is used in the efficiency metric formula. Because of the possibility for extreme values to impact the mean, such as CPU scheduling, median are used to capture the difference in runtime instead of the population

mean. The space and time difference columns represent the first and the last part of the efficiency metric, where the space usage of the original implementation is divided by the improvements in space usage. For the time-ratio it is similar, where the median time usage of the improvement is divided by the median time usage of the original implementation.

As this is meant to serve as a PoC and not a direct integration, some limitations are made in order to simplify the process. The port field in the signature header can be written in many different ways. To simplify the implementation only a select few ways are implemented in the program. Below is an example of some of the port fields in the Snort community rules [60]. The variants highlighted are accepted in the current implementation.

- $HTTP_PORTS
- !21:23
- [547,8080,133,117,189,159]
- [$HTTP_PORTS,139,445]
- 6666:6669
- 80
- !80
- any

In other words, single ports, *any* keyword, list of ports or port variables are accepted. For the others, the whole signature is rejected. The values of port variables, such as `$HTTP_PORTS`, are taken from the variable's values in Snort and Suricata. A limited set of variables are considered, which can be seen in Code listing A.5. The selection is done on a basis of the most used port fields in the rulesets. A simple bash command can find the most used port fields amongst all the signatures used in the testing. The output can be seen in code listing 8.3 below.

```
$ grep ^alert all.rules |awk '{print $4 "\n" $7}' | sort | uniq -c | sort -nr| head
128632 any
 36089 $HTTP_PORTS
 12604 $FILE_DATA_PORTS
 12453 25
  5974 53
  3233 445
  2946 139
  1565 443
   670 $ORACLE_PORTS
   658 [139,445]
```

Some signatures are of course rejected because of this. However it is not assumed to impact the overall results. The flow direction is taken from the signature header. Bidirectional signatures (<>) are not considered, which again is easy to see that is not most of the rules with a bash command that can be seen in code listing 8.3 below. Limitations made in this PoC can cause an unbalanced tree, because of the flow direction. It can advantageously be changed to something which separates the signatures better in a production implementation.

```
$ grep ^alert all.rules | awk '{print $5}' | sort | uniq -c | sort -nr| head
```

```
108000 ->
   131 <>
    10 Microsoft
     3 OpenOffice
     1 WordPerfect
     1 Winamp
     1 PDF
     1 Oracle
     1 F5
     1 CUPS
```

In the PoC, only signatures using the TCP and UDP protocol are considered, even if the rulesets contain others, such as for example ICMP. Code listing 8.3 below count the different protocols in all the datasets.

```
$ grep ^alert all.rules | awk '{print $2}' | sort | uniq -c | sort -nr| head
81722 tcp
13729 http
 8438 udp
 2438 dns
 1172 tls
   73 icmp
  163 ip
   90 smtp
   58 ftp
   39 smb
```

Dataset	No. signatures	No. unique ports	No. Signatures in search branch	W_e (B)	W_i (B)	X_e (ns)	X_i (ns)	Space difference (W_e/W_i)	Time difference (X_i/X_e)	Result
emerging-threats-all snort 2.9	2310	109	1183	4280768	491310	89537	42774	8.71297	0.47772	18.23849
emerging-threats-all suricata 2.9	6701	105	6233	3263872	54242	34669	190211	60.17241	5.48649	10.96739
emerging-threats-policy	121	69	76	3218352	33558	5467	13158	95.90417	2.40680	39.84710
emerging-threats-scan	61	64	42	3185712	19432	2970	11961	163.94154	4.02727	40.70783
emerging-threats-web_server	261	60	180	3316976	75720	9134	15146	43.80581	1.65820	26.41768
snort-community-all	1460	129	1023	4116656	421302	80867	43387	9.77127	0.53652	18.21221
snort-registered-all-3	8779	347	3585	6589072	1490104	273764	107603	4.42189	0.39305	11.25019
snort-registered-all-2.9	1590	103	937	4040848	388054	69977	37269	10.41311	0.53259	19.55185

Table 8.1: Proof of concept result - signature representation

Dataset	No. Patterns	No. states	Percent zero states	W_e (B)	W_i (B)	Search text	X_e (ns)	X_i (ns)	Space difference (W_e/W_i)	Time difference (X_i/X_e)	Result
emerging-threats-all snort 2.9	2310	34038	66.40	47027272	7505771	mixed	110550	573627	6.26548	5.18885	1.20749
						less common	39753	22430	6.26548	0.56423	11.10440
						common	184426	1494329	6.26548	8.10259	0.77327
emerging-threats-all suricata 5.0	6701	80579	66.41	125574312	18460688	mixed	116451	642549	6.80226	5.51776	1.23279
						less common	39862	23439	6.80226	0.58800	11.56839
						common	272538	1812072	6.80226	6.64888	1.02307
emerging-threats-policy	121	1495	85.62	1828872	138381	mixed	44882	123873	13.21621	2.75997	4.78853
						less common	39260	20136	13.21621	0.51289	25.76819
						common	44109	373979	13.21621	8.47852	1.55879
emerging-threats-scan	61	1141	89.48	1336952	86347	mixed	43924	96632	15.48348	2.19998	7.03800
						less common	39249	20120	15.48348	0.51262	30.20433
						common	42905	279438	15.48348	6.51295	2.37734
emerging-threats-web_server	261	3736	77.40	4830312	546781	mixed	59563	213248	8.83409	3.58021	2.46748
						less common	39470	20530	8.83409	0.52014	16.98400
						common	63305	582689	8.83409	9.20447	0.95976
snort-community-all	1460	13502	71.57	20144312	2481255	mixed	90449	345265	8.11860	3.81723	2.12683
						less common	44784	23517	8.11860	0.52512	15.46045
						common	121317	1072987	8.11860	8.84449	0.91793
snort-registered-all-3	8779	237078	72.36	332523904	53706548	mixed	129683	688278	6.19150	5.30739	1.16658
						less common	38276	22817	6.19150	0.59612	10.38637
						common	266468	1825281	6.19150	6.84991	0.90388
snort-registered-all-2.9	1590	18605	18605	30514152	3570683	mixed	89339	388858	8.54575	4.35261	1.96336
						less common	38267	20244	8.54575	0.52902	16.15393
						common	118816	1053319	8.54575	8.86513	0.96397

Table 8.2: Proof of concept result - fast pattern matcher

8.3.1 Evaluation

The results from Tables 8.1 and 8.2, show that the space gained outweighs the time lost in all the tested datasets for both the signature representation and the fast pattern matcher considering the "mixed" search text. Looking at the results, the improved representations are more efficient than the original one, using the metric from Equation (5.1).

Signature representation

Comparing the space consumption of the original implementation to the alternative one, the decrease in bytes are between 77.39 and 99.39%. Smallest amount of space savings can be found in the "snort-registred-all-3" dataset, while a 99.39% decrease is seen in the "emerging-threats-scan" dataset. Number of used signatures of the datasets are between 61 and 8779, due to the limitations previously described. The ports used by the signatures is between 60 and 347. In four of the datasets, the median traversal time is higher for the improved representation (X_i) than the existing one (X_e). However, the four others have a faster traversal time as well as using less space, which is the best scenario.

Fast pattern matcher

When looking at the space consumption for the suggested improvement compared to original, the decrease in bytes are between 83.85 and 93.54%. The dataset with the highest and lowest space saving ratio in the fast pattern matcher improvement is the same as with the signature representation. The amount of zero states is 66.4% at minimum and goes as high as 89.48%, which matches the presumptions of the state transition table being sparse. By looking at the zero states compared to the efficiency, the dataset with the highest amount of zero states also has the highest value from the efficiency metric. Datasets which have a zero state percentage between 66 and 72, have somewhat similar efficiency. As presumed, the search text containing bytes which is less common in the signature pattern (*less common*), have larger time difference and thus higher efficiency. On the contrary, the search text containing bytes with higher frequency in the search pattern takes longer time and thus the efficiency is lower. The efficiency for the fast pattern matcher is in general lower than with the signature representation.

Chapter 9

Discussion

9.1 Experimental results

This section discusses the results obtained in the proof of concept implementation of the selected alternatives (see Section 8.3).

9.1.1 Signature representation

One benefit with the alternative signature representation is that it has a low "base memory usage". The original implementation allocates $2^{16} * 4$ pointers to linked lists of signature objects from a specific protocol and flow direction. This results in a quite high space consumption, even if no signatures are added to the structure - "base memory". On the other hand, this allows for fast access and can be viable to use if the number of signatures is high. The selected alternative representation only allocates memory for the ports in use by the signatures. This difference can impact the results, but also be a feature. It is wasteful to allocate space for a port which is not used, and the alternative implementation allows for easier implementation of this option. However, this leads to a higher initialisation time, which is discussed in more detail later.

Based on the results, it seems like the datasets with the lowest amount of ports were more efficient in the improved representation, than the ones with more ports. Fewer ports will lead to faster *rank* and *select* operations on the sequence containing the labels for the edges in the tree. However, it is difficult to be certain about this because other factors can also impact the efficiency. For example, if a signature includes many ports, it is referenced several times in the array connecting ports to a signature. Thus, it consumes more memory.

An interesting observation is that in four of the datasets, the improved representation was faster than the original. This could be down to programming error, leading to fewer signature references tied to a port in the improved representation than in the existing one. However, this was thought of as a possibility during development and ensured by asserting that the amount of signatures tied to a port is the same when traversing both representations (see Code listing A.1).

Comparing the results from the proof of concept (PoC) to the ones obtained in the theoretical analysis (see Section 6.3.1), it matches fairly good. Based on the theoretical analysis, the alternative one was slower but required less space, which is what can be observed in general from the results in Table 8.1. However, the metric captures the compromise between space and time in various situations, which is difficult to see from the theoretical analysis.

9.1.2 Fast pattern matcher

The efficiency from the PoC implementation of the improved fast pattern matcher is not as good as with the signature representation. Space difference can be one of the reasons. As mentioned in the related work (Chapter 3), the worst-case entropy of a labelled tree is $2n - \theta(log\ n) + n\ log\ \sigma$, where there are n nodes with labels drawn from an alphabet of size σ. For a labelled directed graph on the other hand, the worst-case entropy is $e\ log\ \frac{n^2}{e} + O(e) + e\ log\ \sigma$. Which means that the worst-case entropy is higher for a labelled directed graph than for a labelled tree. Amount of zero states can also have an impact, due to the design of the improved alternative representation.

Different search texts impacts the efficiency. In a host-based intrusion detection system (HIDS), the search text will contain mostly characters from the English alphabet and symbols, while a network intrusion detection system (NIDS) can contain all kinds of bytes. Thus, the number of "zero state hits" in the state transition matrix can be very different from a NIDS and impact the time usage. As mentioned previously, a search text with a uniform distribution amongst the ASCII characters is used (mixed), as well as two others containing less common and common bytes in the signature pattern. Looking at the results, the efficiency is in consistently lower for the *common* search text, because the time usage is higher for the improved representation, and opposite for the *less common* search text. This indicates that the alternative representation is working as intended by consuming less time for entries in the state transition matrix with the value 0. Four of the data sets have an efficiency result of less than one using the *common* search text, meaning that the improved representation is in general not worth it over the existing one. Thus, using the suggested improvement in a HIDS might not be a good option, but more testing is required to determine it fully.

Existing literature, proved that research has already been done on Aho-Corasick deterministic finite state automaton (DFA) version. An interesting question in that sense, is how the suggested implemented alternative improvement compare to previous improvements? Kumar et al. [28] achieved a compression ratio of more than 95% in an algorithm called D^2FA. This is better than the 83.85% minimum space savings, suggested in this thesis. However, Kumar mentions very little about time differences and thus it is hard to compare the efficiency. Dimopoulos et al. [30] compressed the state transition table, and achieved between 28.8 to 75.5% space savings per character compared to previous compression, and not the plain representation. Another concern is that it is not certain whether

the DFA or NFA version of the state transition table was used in the research. This makes it hard to compare the effectiveness of the improvement in this thesis against the suggestion by Dimopoulos et al [30].

9.2 General

Only transmission control protocol (TCP) and user datagram protocol (UDP) signatures have been selected as a part of the PoC. This limitation was set in order to make a simple and comparable implementation without considering all scenarios. Thus, two of the most used protocols in the rulesets were chosen. Also, both Suricata and Snort handle other protocols than TCP and UDP differently, by grouping them together. Adding more protocols to the signature representation would of course increase the base memory usage of the existing representation, but also the improvements in space consumption would increase. It is presumed that the signature representation would not be affected by adding more protocols.

Implementation of the fast pattern matcher, on the other hand, could be affected by the limited protocol selection. In this implementation, only the content of TCP and UDP rules have been considered. If the content of rules which are not considered in this implementation have very different characters, the byte frequency will be different and maybe the amount of zero states or efficiency. Recall, the byte frequency from Figure 7.3, contains the byte frequency for all the content fields in all the signatures in all the rulesets. A set limitation in the PoC can impact the byte frequency. This is something which is hard to estimate or say for certain, but it is a possibility.

Another point of failure is the documentation and understanding of the used libraries in the PoC, described in Chapter 4. Misunderstanding the functionality in the libraries to for example measure the space consumption of the data structure, can lead to wrong results.

One topic which has been briefly touched so far is the initialisation time. A fast initialisation time is desirable because, theoretically it enables the operator of the IDS to detect anomalies faster. However, in practice, it does not have that much to say whether the IDS take 5 or 15 minutes to initialise. Thus, it has been intentionally left out of the comparison between the existing and alternative representations.

Chapter 10

Conclusion

This chapter presents a short summary of the thesis and answers the research questions. Practical considerations are also presented together with future work.

10.1 Summary

During the thesis, three research questions were answered.

> Research question 1: What are the efficiency criteria for signature representation and the implementation of the fast pattern matching algorithm?

The term efficiency, in the context of compact data structures was defined in order to measure and compare the performance of the existing and improved representation. Firstly, by looking at existing literature that suggested improvements to data structures in other systems and what defines a good solution to an improvement, it came apparent that both space and time are important. Time is measured using asymptotic growth, while space consumption is measured in bits. In that context, the optimal values for the signature representation and fast pattern matcher were also defined by using theoretic lower bounds.

Equally important is also the compromise between space and time. A data structure, which allows for fast access times and higher space consumption can be used in systems with strict time requirements. On the other hand, a compact data structure with low space consumption and slow traversal time or time usage, is generally not viable as a compact data structure. However, it can be usable if the space gained outweighs the time loss. Because of that, a metric in Equation (5.1) was developed to capture the compromise between the loss in time versus the increase in space, in order to measure in a proof of concept (PoC) if the data structure is efficient.

> Research question 2: How efficient is the representation of signatures and the implementation of the fast pattern matching algorithm regarding the criteria from question one?

When the efficiency criteria are decided, the signature representation and the fast pattern matcher can be analysed to determine if there is room for improvement. Firstly, a code analysis of Snort and Suricata determined that the data structures are similar on an abstract level, but there are technical differences in both the signature representation and the fast pattern matcher. Another discovery was that existing literature was outdated when it comes to the signature representation and that the systems have been improved. Regarding the fast pattern matcher, the Aho-Corasick algorithm was used in both systems and implemented in a straightforward way as in the literature.

Next, a theoretical analysis was done based on the technical results, in order to see if there is room for improvement. Both the implementation of the fast pattern matcher and the signature representation were based on pointers, which are generally fast, with one memory access per operation, but there is room for improvement when it comes to space consumption. The fast pattern matcher was modelled as a labelled directed graph, while the signature representation was modelled as a labelled tree. A gap was found between the current space consumption and the models worst-case entropy.

Another discovery was that the signature representation in Snort and Suricata was either not modelled or changed from previous literature. The updated models of the signature representation in Snort and Suricata is a foundation for further research and optimisation.

> Research question 3: How can these data structures be improved in order to meet optimal values of the criteria defined in question one?

Characteristics from the theoretical analysis together with the abstract models was the foundation for two suggested improved alternative representations.

For the signature representation, two alternatives were suggested. The purpose of the data structure is to traverse the tree and select a relevant group of signature. Because of this, the suggested improvements were divided into two parts. One, which contains the decision tree to select a group of signatures based on the protocol, port and flow direction. Because the traversal of the tree requires simple operations on the abstract model, a LOUDS representation of a labelled, presented in existing literature, showed the best time/space trade off and was chosen to represent the first part. The other part contains the references between a branch in the decision tree and a set of signatures. An algorithm which selects a set of signatures based on a leaf-node identifier from the tree was developed in order to connect the two parts together with low space overhead.

With the fast pattern matcher, characteristics such as byte frequency and sparsity were used as a foundation for the suggested improvements. One alternative took advantage of clustering in a three-dimensional state transition matrix, which can possibly happen if only a few states are used. The other suggested alternative improved an algorithm from existing literature to select a node in a labelled graph. Recall, all nodes in a state transition table, modelled as a graph, have the same number of vertices. In this thesis, that was the basis for simplifying

the algorithm in order to save time. A new algorithm was then developed which takes advantage of a sparse transition table in order to save space and time, thus be more efficient.

Lastly, two selected alternatives were compared against the existing implementation in a PoC using C++. The results suggested that alternatives are more efficient than the existing implementations, as seen theoretically. Concerning the fast pattern matcher, the alternative representations efficiency depends on the search text, regarding the search text containing a uniform distribution of the alphabet it is more efficient. However, they are somewhat less compact than what is suggested by the existing literature.

10.2 Future work

An element which is not considered in this thesis is the content of the signatures themselves. In the signature representation, they are considered as nodes without going further into the attributes. The fast pattern matcher does use the content of each signature, but nothing further than that. Future work can be to investigate compact storage of the contents within the signatures. For example, the *msg* option in the signature consists of printable ASCII characters and can be stored in a compact manner.

A limitation with the nature of the signature representation is that there are multiple references to the same signature from a branch in the skeleton. For example, when a signature has the port variable $HTTP_PORTS a signature reference is added to all ports in the port variable. It is possible that taking advantage of this can increase the efficiency further. Future work can be to investigate this closer.

It was also some uncertainty regarding the k^3-tree representation of a labelled graph. Future work can be to investigate if there are clustering in the datasets and research if an algorithm can be developed to traverse the graph in reasonable time.

Bibliography

[1] *2021 IEEE Taxonomy*, 2021. [Online]. Available: https://www.ieee.org/content/dam/ieee-org/ieee/web/org/pubs/ieee-taxonomy.pdf (visited on 29/03/2021).

[2] Open Information Security Foundation, *Suricata IDS - downloads*, 2010. [Online]. Available: https://www.openinfosecfoundation.org/downloads/?C=M;O=A (visited on 08/05/2021).

[3] L. Greenemeier, *Sourcefire Has Big Plans For Open-Source Snort*, en, Apr. 2006. [Online]. Available: https://www.informationweek.com/sourcefire-has-big-plans-for-open-source/186700788 (visited on 08/05/2021).

[4] E. Albin and N. C. Rowe, 'A Realistic Experimental Comparison of the Suricata and Snort Intrusion-Detection Systems,' in *2012 26th International Conference on Advanced Information Networking and Applications Workshops*, Mar. 2012, pp. 122–127. DOI: 10.1109/WAINA.2012.29.

[5] 'Chapter 5 - Inner Workings,' in *Snort Intrusion Detection and Prevention Toolkit*, A. R. Baker and J. Esler, Eds., Rockland: Syngress, 2007, pp. 175–223, ISBN: 978-1-59749-099-3. DOI: 10.1016/B978-159749099-3/50010-0. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9781597490993500100.

[6] A. Gupta and L. S. Sharma, 'Performance Evaluation of Snort and Suricata Intrusion Detection Systems on Ubuntu Server,' en, in *Proceedings of ICRIC 2019*, P. K. Singh, A. K. Kar, Y. Singh, M. H. Kolekar and S. Tanwar, Eds., ser. Lecture Notes in Electrical Engineering, Cham: Springer International Publishing, 2020, pp. 811–821, ISBN: 978-3-030-29407-6. DOI: 10.1007/978-3-030-29407-6_58.

[7] G. Navarro, *Compact Data Structures: A Practical Approach*, 1st. USA: Cambridge University Press, 2016, ISBN: 1-107-15238-0.

[8] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar and J. Sherry, 'Achieving 100Gbps Intrusion Prevention on a Single Server,' in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 1083–1100, ISBN: 978-1-939133-19-9. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng.

[9] S. Sørenes, *Ber folk rydde i innboksen: Bra for miljøet å slette e-post*, nn-NO, Apr. 2021. [Online]. Available: https://www.nrk.no/vestland/ber-folk-rydde-i-innboksen_-_-bra-for-miljoet-a-slette-e-post-1.15449462 (visited on 27/04/2021).

[10] ATEA, *Lurer du på hvordan vi har gjort beregningene? - Bærekraft i ATEA*, no, 2021. [Online]. Available: https://www.atea.no/baerekraft-i-atea/vare-beregninger/ (visited on 27/04/2021).

[11] M. Roesch *et al.*, 'Snort: Lightweight intrusion detection for networks.,' in *Lisa*, Issue: 1, vol. 99, 1999, pp. 229–238.

[12] Rawlins, *Compared to what? : an introduction to the analysis of algorithms*, eng, ser. Principles of computer science series. New York: Computer Science Press, 1992, ISBN: 0-7167-8243-X.

[13] S. S. Skiena, *The Algorithm Design Manual*, eng, 3rd ed. 2020, ser. Texts in Computer Science. Cham: Springer International Publishing AG, 2020, ISSN: 1868-0941, Publication Title: The Algorithm Design Manual, ISBN: 978-3-030-54255-9.

[14] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, eng. Hoboken: John Wiley & Sons, Incorporated, 2006, Publication Title: Elements of Information Theory, ISBN: 978-0-471-24195-9.

[15] G. Jacobson, 'Space-efficient static trees and graphs,' in *30th annual symposium on foundations of computer science*, IEEE Computer Society, 1989, pp. 549–554.

[16] D. R. Clark, 'Compact Pat Trees,' PhD Thesis, UWSpace, CAN, 1997. [Online]. Available: http://hdl.handle.net/10012/64.

[17] J. Cordova and G. Navarro, 'Practical Dynamic Entropy-Compressed Bitvectors with Applications,' in *Proceedings of the 15th International Symposium on Experimental Algorithms - Volume 9685*, ser. SEA 2016, Berlin, Heidelberg: Springer-Verlag, Jun. 2016, pp. 105–117, ISBN: 978-3-319-38850-2. DOI: 10.1007/978-3-319-38851-9_8. [Online]. Available: https://doi.org/10.1007/978-3-319-38851-9_8 (visited on 25/05/2021).

[18] N. Archibald, G. Ramirez, N. Rathaus, J. Burke, B. Caswell and R. Deraison, 'Chapter 7 - The Inner Workings of Snort,' en, in *Nessus, Snort, & Ethereal Power Tools*, N. Archibald, G. Ramirez, N. Rathaus, J. Burke, B. Caswell and R. Deraison, Eds., Burlington: Syngress, Jan. 2005, pp. 151–179, ISBN: 978-1-59749-020-7. DOI: 10.1016/B978-159749020-7/50012-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9781597490207500124 (visited on 12/11/2020).

[19] *Suricata User Guide — Suricata 6.0.0 documentation*. [Online]. Available: https://suricata.readthedocs.io/en/suricata-6.0.0/ (visited on 13/10/2020).

[20] F. Zhang, J. Zhai, X. Shen, O. Mutlu and X. Du, 'Enabling Efficient Random Access to Hierarchically-Compressed Data,' in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, ISSN: 2375-026X, Apr. 2020, pp. 1069–1080. DOI: 10.1109/ICDE48307.2020.00097.

[21] A. V. Aho and M. J. Corasick, 'Efficient string matching: An aid to bibliographic search,' *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975, ISSN: 0001-0782. DOI: 10.1145/360825.360855. [Online]. Available: https://doi.org/10.1145/360825.360855 (visited on 03/11/2020).

[22] *SNORT Users Manual 2.9.16 - The Snort Project*, Apr. 2020. [Online]. Available: https://snort-org-site.s3.amazonaws.com/production/document_files/files/000/000/249/original/snort_manual.pdf (visited on 13/10/2020).

[23] N. Tuck, T. Sherwood, B. Calder and G. Varghese, 'Deterministic memory-efficient string matching algorithms for intrusion detection,' in *IEEE INFOCOM 2004*, ISSN: 0743-166X, vol. 4, Mar. 2004, 2628–2639 vol.4. DOI: 10.1109/INFCOM.2004.1354682.

[24] D. Pao, W. Lin and B. Liu, 'A memory-efficient pipelined implementation of the aho-corasick string-matching algorithm,' en, *ACM Transactions on Architecture and Code Optimization*, vol. 7, no. 2, pp. 1–27, Sep. 2010, ISSN: 1544-3566, 1544-3973. DOI: 10.1145/1839667.1839672. [Online]. Available: https://dl.acm.org/doi/10.1145/1839667.1839672 (visited on 03/11/2020).

[25] M. Norton, 'Optimizing pattern matching for intrusion detection,' en, *Sourcefire, Inc., Columbia, MD*, p. 11, 2004, publiser:Citeseer.

[26] T.-H. Lee and N.-L. Huang, 'An Efficient and Scalable Pattern Matching Scheme for Network Security Applications,' in *2008 Proceedings of 17th International Conference on Computer Communications and Networks*, ISSN: 1095-2055, Aug. 2008, pp. 1–7. DOI: 10.1109/ICCCN.2008.ECP.176.

[27] J. Holub, 'Simulation of nondeterministic finite automata in pattern matching.,' en, Publisher: Praha (Czech Republic) : Ceske vysoke uceni technicke v Praze, U - Thesis ; (Thesis (Dr.)) 2000. [Online]. Available: http://hdl.handle.net/10068/336087.

[28] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley and J. Turner, 'Algorithms to accelerate multiple regular expressions matching for deep packet inspection,' *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 339–350, Aug. 2006, ISSN: 0146-4833. DOI: 10.1145/1151659.1159952. [Online]. Available: https://doi.org/10.1145/1151659.1159952 (visited on 27/11/2020).

[29] M. Becchi, 'Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation,' ISBN: 9781109254013, PhD Thesis, Washington University, USA, 2009.

[30] V. Dimopoulos, I. Papaefstathiou and D. Pnevmatikatos, 'A Memory-Efficient Reconfigurable Aho-Corasick FSM Implementation for Intrusion Detection Systems,' in *Modeling and Simulation 2007 International Conference on Embedded Computer Systems: Architectures*, Jul. 2007, pp. 186–193. DOI: 10.1109/ICSAMOS.2007.4285750.

[31] S. Dharmapurikar and J. W. Lockwood, 'Fast and Scalable Pattern Matching for Network Intrusion Detection Systems,' *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, pp. 1781–1792, Oct. 2006, Conference Name: IEEE Journal on Selected Areas in Communications, ISSN: 1558-0008. DOI: 10.1109/JSAC.2006.877131.

[32] G. Ottaviano and R. Venturini, 'Partitioned Elias-Fano indexes,' in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, ser. SIGIR '14, New York, NY, USA: Association for Computing Machinery, Jul. 2014, pp. 273–282, ISBN: 978-1-4503-2257-7. DOI: 10.1145/2600428.2609615. [Online]. Available: https://doi.org/10.1145/2600428.2609615 (visited on 20/05/2021).

[33] S. Vigna, 'Quasi-succinct indices,' in *Proceedings of the sixth ACM international conference on Web search and data mining*, ser. WSDM '13, New York, NY, USA: Association for Computing Machinery, Feb. 2013, pp. 83–92, ISBN: 978-1-4503-1869-3. DOI: 10.1145/2433396.2433409. [Online]. Available: https://doi.org/10.1145/2433396.2433409 (visited on 20/05/2021).

[34] S. Vigna, 'Broadword implementation of rank/select queries,' in *Proceedings of the 7th international conference on Experimental algorithms*, ser. WEA'08, Berlin, Heidelberg: Springer-Verlag, May 2008, pp. 154–168, ISBN: 978-3-540-68548-7. (visited on 20/05/2021).

[35] J. Barbay, F. Claude and G. Navarro, 'Compact binary relation representations with rich functionality,' en, *Information and Computation*, vol. 232, pp. 19–37, Nov. 2013, ISSN: 0890-5401. DOI: 10.1016/j.ic.2013.10.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540113001144 (visited on 17/03/2021).

[36] R. Sedgewick, *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*, Third. Addison-Wesley Professional, 1998, ISBN: 978-0-7686-8533-6.

[37] P. Sinha, 'A memory-efficient doubly linked list,' *Linux Journal*, vol. 2005, no. 129, p. 10, Jan. 2005, ISSN: 1075-3583. [Online]. Available: https://www.linuxjournal.com/article/6828 (visited on 12/04/2021).

[38] D. Arroyuelo, R. Cánovas, G. Navarro and K. Sadakane, 'Succinct trees in practice,' in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, ser. ALENEX '10, USA: Society for Industrial and Applied Mathematics, Jan. 2010, pp. 84–97. (visited on 12/04/2021).

[39] P. Ferragina, F. Luccio, G. Manzini and S. Muthukrishnan, 'Structuring labeled trees for optimal succinctness, and beyond,' in *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, ISSN: 0272-5428, Oct. 2005, pp. 184–193. DOI: 10.1109/SFCS.2005.69.

[40] A. Farzan and J. I. Munro, 'Succinct encoding of arbitrary graphs,' en, *Theoretical Computer Science*, vol. 513, pp. 38–52, Nov. 2013, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2013.09.031. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397513007238 (visited on 18/04/2021).

[41] J. Barbay, A. Golynski, J. Ian Munro and S. Srinivasa Rao, 'Adaptive searching in succinctly encoded binary relations and tree-structured documents,' en, *Theoretical Computer Science*, The Burrows-Wheeler Transform, vol. 387, no. 3, pp. 284–297, Nov. 2007, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2007.07.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397507005270 (visited on 17/03/2021).

[42] J. Barbay, M. He, J. I. Munro and S. R. Satti, 'Succinct Indexes for Strings, Binary Relations and Multilabeled Trees,' *ACM Trans. Algorithms*, vol. 7, no. 4, Sep. 2011, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 1549-6325. DOI: 10.1145/2000807.2000820. [Online]. Available: https://doi.org/10.1145/2000807.2000820.

[43] S. Alvarez-Garcia, G. de Bernardo, N. R. Brisaboa and G. Navarro, 'A succinct data structure for self-indexing ternary relations,' en, *Journal of Discrete Algorithms*, vol. 43, pp. 38–53, Mar. 2017, ISSN: 1570-8667. DOI: 10.1016/j.jda.2016.10.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570866716300351 (visited on 22/04/2021).

[44] J. S. White, T. Fitzsimmons and J. N. Matthews, 'Quantitative analysis of intrusion detection systems: Snort and Suricata,' in *Cyber Sensing 2013*, vol. 8757, International Society for Optics and Photonics, May 2013, p. 875 704. DOI: 10.1117/12.2015616. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/8757/875704/Quantitative-analysis-of-intrusion-detection-systems-Snort-and-Suricata/10.1117/12.2015616.short (visited on 13/10/2020).

[45] S. Ladra, M. R. Luaces, J. R. Paramá and F. Silva-Coira, 'Space- and Time-Efficient Storage of LiDAR Point Clouds,' *arXiv:1912.11859 [cs]*, vol. 11811, pp. 513–527, 2019, arXiv: 1912.11859. DOI: 10.1007/978-3-030-32686-9_36. [Online]. Available: http://arxiv.org/abs/1912.11859 (visited on 09/10/2020).

[46] N. R. Brisaboa, M. R. Luaces, G. Navarro and D. Seco, 'A Fun Application of Compact Data Structures to Indexing Geographic Data,' en, in *Fun with Algorithms*, P. Boldi and L. Gargano, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2010, pp. 77–88, ISBN: 978-3-642-13122-6. DOI: 10.1007/978-3-642-13122-6_10.

[47] S. Álvarez-García, B. Freire, S. Ladra and Ó. Pedreira, 'Compact and efficient representation of general graph databases,' en, *Knowledge and Information Systems*, vol. 60, no. 3, pp. 1479–1510, Sep. 2019, ISSN: 0219-3116. DOI: 10.1007/s10115-018-1275-x. [Online]. Available: https://doi.org/10.1007/s10115-018-1275-x (visited on 04/10/2020).

[48] I. Lee, P. Do, L. Do and S.-R. Kim, 'A System for Finding the Compact Set of Intrusion Detection Rules on the MapReduce Environment,' English, *International Information Institute (Tokyo). Information; Koganei*, vol. 18, no. 9, pp. 3971–3978, Sep. 2015, Num Pages: 8 Place: Koganei, Japan, Koganei Publisher: International Information Institute, ISSN: 13434500. [Online]. Available: https://search.proquest.com/docview/1736913218/abstract/9ED005AF00084238PQ/1 (visited on 01/10/2020).

[49] *Home | Microsoft Academic*. [Online]. Available: https://academic.microsoft.com/home (visited on 07/04/2021).

[50] *Primo by Ex Libris*, en. [Online]. Available: https://bibsys-almaprimo.hosted.exlibrisgroup.com (visited on 07/04/2021).

[51] *Doxygen: Doxygen*. [Online]. Available: https://www.doxygen.nl/index.html (visited on 24/04/2021).

[52] A. Bacchelli and C. Bird, 'Expectations, outcomes, and challenges of modern code review,' in *2013 35th International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, May 2013, pp. 712–721. DOI: 10.1109/ICSE.2013.6606617.

[53] M. Höst and C. Johansson, 'Evaluation of code review methods through interviews and experimentation,' en, *Journal of Systems and Software*, vol. 52, no. 2, pp. 113–120, Jun. 2000, ISSN: 0164-1212. DOI: 10.1016/S0164-1212(99)00137-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121299001375 (visited on 30/04/2021).

[54] F. Claude, *LIBCDS 2 - A Compressed Data Structure Library*, original-date: 2011-05-13T01:57:53Z, Jul. 2020. [Online]. Available: https://github.com/fclaude/libcds2 (visited on 07/11/2020).

[55] S. Gog, T. Beller, A. Moffat and M. Petri, 'From Theory to Practice: Plug and Play with Succinct Data Structures,' in *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, pp. 326–337. [Online]. Available: https://github.com/simongog/sdsl-lite.

[56] G. Ottaviano, *Succinct*, original-date: 2011-10-16T14:59:19Z, Nov. 2020. [Online]. Available: `https://github.com/ot/succinct` (visited on 07/11/2020).

[57] *DYNAMIC: A succinct and compressed fully-dynamic data structures library*, original-date: 2015-10-30T16:20:39Z, Jul. 2020. [Online]. Available: `https://github.com/xxsds/DYNAMIC` (visited on 07/11/2020).

[58] Magnus, *StoneSwine/msc.thesis.code*, original-date: 2021-04-08T10:00:35Z, Apr. 2021. [Online]. Available: `https://github.com/StoneSwine/msc.thesis.code` (visited on 08/04/2021).

[59] *Proofpoint Emerging Threats Rules*. [Online]. Available: `https://rules.emergingthreats.net//open/` (visited on 30/03/2021).

[60] Snort, *Snort Rules and IDS Software Download*, 2020. [Online]. Available: `https://www.snort.org/downloads#rules` (visited on 12/11/2020).

[61] R. T. Clemen, *Making hard decisions : an introduction to decision analysis*, eng, ser. The Duxbury series in statistics and decision sciences. South-Western College Publ, 1997, ISBN: 0-534-26034-9.

Appendix A

Code listings

Code listing A.1: sigrep.cpp

```
/*
 * Author: Magnus Lien Lilja
 * Proof of concept (PoC) comparison of the signature representation, as it is in Snort and Suricata, and a suggested
 * improvement. The program takes a signature file as argument, parses the signatures and builds both of the
 * representations, before they are traversed and the time taken registered. Finally a developed metric for efficiency
 * is used in order to see which of them is the better one.
 */

#include <cstdio>

#include <elias_fano_compressed_list.hpp>
#include <mapper.hpp>

#include "parser.h"
#include "ptree.h"
#include "ltree.h"
#include "signature.h"
#include <ostream>
#include "defs.h"

using namespace sdsl;
using namespace std;
using namespace std::chrono;
using timer = std::chrono::high_resolution_clock;

int main(int argc, char *argv[]) {
  int no_samples = 2;
  vector<uint> l_total;
  vector<uint> l_select_unique;
  vector<uint> p_total;
  char *infile;

  int unique_ports = 0;
  int signo_sb_orig = 0;
  int signo_sb_alt = 0;
  int alt_b_siz = 0;
  int alt_s_siz = 0;
```

```
38
39
40  if (argc < 3) {
41    fprintf(stderr, "Usage: ./program -i infile -s sampleno\n");
42    exit(0);
43  }
44  for (int i = 1; i < argc; i++) {
45    if (strcmp(argv[i], "-i") == 0) {
46      infile = argv[i + 1];
47    } else if (strcmp(argv[i], "-s") == 0) {
48      no_samples = atoi(argv[i + 1]);
49    }
50  }
51
52  uint64_t p_node_size_b, sig_size_b, l_vlcp_size_b, l_vlcc_size_b, p_ct_ms,
        l_ct_ms;
53
54  std::vector<Signature *> sig_list;
55  ParseSigFile(sig_list, infile);
56  //               size of the pointers               +  the size of the objects in
        the list
57  sig_size_b += sizeof(Signature *) * sig_list.size() + (sizeof(Signature) *
        sig_list.size());
58  sig_size_b += sizeof(sig_list);
59
60  if (sig_list.size() <= 50) {
61    exit(0);
62  }
63
64  for (size_t x = 0; x < no_samples; x++) {
65    // Size variables in bytes (b)
66    p_node_size_b = l_vlcp_size_b = l_vlcc_size_b = p_ct_ms = l_ct_ms = 0;
67
68    // Initialise the original representation, which is based on pointers
69    Rootnode *root = new Rootnode();
70    p_node_size_b += sizeof(Rootnode *);
71    p_node_size_b += sizeof(Rootnode);
72    Flownode **cn_f;
73    Portnode **cn_p, *cn_p_tmp;
74
75    for (size_t i = 0; i < MAXFLOW; i++) {
76      root->flow_gh[i] = new Flownode();
77      p_node_size_b += sizeof(Flownode);
78      for (size_t j = 0; j < MAX_PORTS; j++) {
79        root->flow_gh[i]->tcp[j] = nullptr;
80        root->flow_gh[i]->udp[j] = nullptr;
81      }
82    }
83
84    // Parse all the signatures, traverse the tree and add them to a linked-list at
          the appropriate branch in the
85    // original representation.
86    for (Signature *sig : sig_list) {
87      if (sig->flowdir == FLOWDIR_TOCLIENT) {
88        cn_f = &root->flow_gh[FLOWDIR_TOCLIENT];
89      } else // TOSERVER
90      {
91        cn_f = &root->flow_gh[FLOWDIR_TOSERVER];
92      }
93
```

```
94        for (auto sp : sig->dstport) {
95          if (sig->protocol == IPPROTO_TCP) {
96            cn_p = &(*cn_f)->tcp[sp];
97          } else {
98            cn_p = &(*cn_f)->udp[sp];
99          }
100
101         Portnode *temp_pn = new Portnode();
102         temp_pn->sn = sig;
103
104         // list is empty
105         if (!(*cn_p)) {
106           (*cn_p) = temp_pn;
107           p_node_size_b += sizeof(Portnode);
108         } else { // traverse list until last node is reached and insert
109
110           Portnode *last = (*cn_p);
111
112           while (last->next) {
113             last = last->next;
114           }
115
116           last->next = temp_pn;
117           p_node_size_b += sizeof(Portnode);
118         }
119       }
120     }
121
122     // Build the alternative representation
123     std::set<uint32_t> up;
124     for (auto sig : sig_list) {
125       for (auto sp : sig->dstport) {
126         up.insert(sp);
127       }
128     }
129     std::vector<uint32_t> up_it(up.begin(), up.end());
130
131     if (up_it.size() <= 1) {
132       exit(0);
133     }
134
135     ltree<bit_vector, select_support_mcl<1>, select_support_mcl<0>, rank_support_v5
           <1>, rank_support_v<0>> l;
136     // Build the "static" part of the tree level order
137     l.append(2, {FLOWDIR_TOSERVER, FLOWDIR_TOCLIENT});
138     for (size_t i = 0; i < MAXFLOW; i++) {
139       l.append(2, {IPPROTO_TCP, IPPROTO_UDP});
140     }
141     for (size_t i = 0; i < MAXFLOW * IPPROTO_MAX; i++) {
142       l.append(up_it.size(), up_it);
143     }
144     for (size_t i = 0; i < MAXFLOW + IPPROTO_MAX; i++) {
145       for (auto p : up) {
146         l.append(0);
147       }
148     }
149     l.finalize();
150
151     // Insert the signature ids
152     // number of nodes in the static part of the tree. Hardcoded for now
```

```
153     int sc = 8;
154
155     const int no_protport = (MAXFLOW + IPPROTO_MAX) * up_it.size();
156     std::vector<uintptr_t> tmp_vlc[no_protport];
157
158     // Traverse all the signatures, and add them to the correct branch in the
            alternative representation.
159     for (auto sig : sig_list) {
160       int nodeid = 1;
161       if (sig->flowdir == FLOWDIR_TOCLIENT) {
162         nodeid = l.labeledchild(nodeid, FLOWDIR_TOCLIENT);
163       } else {
164         nodeid = l.labeledchild(nodeid, FLOWDIR_TOSERVER);
165       }
166
167       // Depth 2
168       if (sig->protocol == IPPROTO_TCP) {
169         nodeid = l.labeledchild(nodeid, IPPROTO_TCP);
170       } else {
171         nodeid = l.labeledchild(nodeid, IPPROTO_UDP);
172       }
173
174       // Depth 3
175       int pnid = nodeid;
176       for (auto sp : sig->dstport) {
177         nodeid = l.labeledchild(pnid, sp) - sc;
178         if (nodeid >= 0) {
179           tmp_vlc[nodeid].push_back(reinterpret_cast<std::uintptr_t>(sig));
180         }
181       }
182     }
183
184     // Build the array containing references between a branch and the signatures.
            Also clear the temporary vector from
185     // previously. The last signature from the branch with the most amount of
            signatures are also kept, as an object to
186     // search for later on.
187     std::vector<uintptr_t> tmp;
188     bit_vector vlc_bv(0, 0);
189
190     int maxpos_count = 0;
191     int sigid = 0;
192
193     for (size_t i = 0; i < no_protport; i++) {
194       int noc = tmp_vlc[i].size();
195       if (noc > maxpos_count) {
196         maxpos_count = noc;
197         sigid = ((Signature *) tmp_vlc[i].back())->id;
198       }
199       bit_vector tmp_bv = vlc_bv;
200       tmp_bv.resize(vlc_bv.size() + noc + 1);
201       int pos = vlc_bv.size();
202       tmp_bv[pos++] = 1;
203
204       for (size_t j = 0; j < noc; j++) {
205         tmp_bv[pos++] = 0;
206         tmp.push_back(tmp_vlc[i][j]);
207       }
208
209       tmp_bv.resize(pos);
```

```
      vlc_bv = bit_vector(std::move(tmp_bv));

      tmp_vlc[i].clear();
      tmp_vlc[i].shrink_to_fit();
    }

    succinct::elias_fano_compressed_list vlcarray(tmp);

    tmp.clear();
    tmp.shrink_to_fit();

    select_support_mcl<1> vlc_bv_s1(&vlc_bv);
    rank_support_v<0> vlc_bv_r0(&vlc_bv);

    l_vlcc_size_b += (float) succinct::mapper::size_tree_of(vlcarray)->size;
    l_vlcp_size_b += size_in_bytes(vlc_bv);

    // Search for the signature ID (sigid) in the alternative representation
    Signature *findme = sig_list[sigid];
    auto start = timer::now();
    int nodeid = 1;
    if (findme->flowdir == FLOWDIR_TOCLIENT) {
      nodeid = l.labeledchild(nodeid, FLOWDIR_TOCLIENT);
    } else {
      nodeid = l.labeledchild(nodeid, FLOWDIR_TOSERVER);
    }

    // Depth 2
    if (findme->protocol == IPPROTO_TCP) {
      nodeid = l.labeledchild(nodeid, IPPROTO_TCP);
    } else {
      nodeid = l.labeledchild(nodeid, IPPROTO_UDP);
    }

    // Depth 3
    int pnid = nodeid;
    int lfound, pfound;
    lfound = 0;
    pfound = 1;
    nodeid = l.labeledchild(pnid, findme->dstport[0]) - sc;
    if (nodeid >= 0) {

      int i = vlc_bv_r0(vlc_bv_s1(nodeid + 1));
      int stop = vlc_bv_r0(vlc_bv_s1(nodeid + 2));

      for (i; i < stop; i++) {
        lfound++;
        if (((Signature *) vlcarray[i]) == findme) {
          printf("[LOUDS]:FOUND THE SIGNATURE!\n");
        }
      }
    }

    // Search for signature in the original representation
    auto stop = timer::now();
    l_ct_ms = duration_cast<TIMEUNIT>(stop - start).count();

    start = timer::now();
    if (findme->flowdir == FLOWDIR_TOCLIENT) {
      cn_f = &root->flow_gh[FLOWDIR_TOCLIENT];
```

```
270     } else // TOSERVER
271     {
272       cn_f = &root->flow_gh[FLOWDIR_TOSERVER];
273     }
274
275     if (findme->protocol == IPPROTO_TCP) {
276       cn_p = &(*cn_f)->tcp[findme->dstport[0]];
277     } else {
278       cn_p = &(*cn_f)->udp[findme->dstport[0]];
279     }
280
281     while ((*cn_p)->next != nullptr) {
282       pfound++;
283       *cn_p = (*cn_p)->next;
284       if ((*cn_p)->sn == findme) {
285         printf("[POINTER]:FOUND THE SIGNATURE!\n");
286       }
287     }
288
289     // make sure that the branch where the signature is located, contains an equal
           amount of signatures.
290     assert(pfound == lfound);
291     stop = timer::now();
292     p_ct_ms = duration_cast<TIMEUNIT>(stop - start).count();
293
294     // Register time and space
295     unique_ports = up_it.size();
296     signo_sb_alt = lfound;
297     signo_sb_orig = pfound;
298     alt_b_siz = size_in_bytes(l.bv);
299     alt_s_siz = size_in_bytes(l.s);
300
301     p_total.push_back(p_ct_ms);
302     l_total.push_back(l_ct_ms);
303     l_select_unique.push_back(l.su_t);
304
305
306     // Delete and deconstruct
307     for (size_t i = 1; i < MAXFLOW; i++) {
308       cn_f = &root->flow_gh[i - 1];
309       for (size_t p = 1; p < MAXFLOW; p++) {
310         for (size_t j = 1; j < MAX_PORTS; j++) {
311           if (p - 1) {
312             cn_p = &(*cn_f)->tcp[j - 1];
313           } else {
314             cn_p = &(*cn_f)->udp[j - 1];
315           }
316
317           if ((*cn_p) != nullptr) {
318             if ((*cn_p)->next == nullptr) {
319               delete ((*cn_p));
320             } else {
321               while ((*cn_p)->next != nullptr) {
322                 cn_p_tmp = *cn_p;
323                 *cn_p = (*cn_p)->next;
324                 delete (cn_p_tmp);
325               }
326               delete (*cn_p);
327             }
328           }
```

```
329         }
330       }
331       delete (*cn_f);
332     }
333     delete (root);
334   }
335
336   // Find median: https://en.cppreference.com/w/cpp/algorithm/nth_element
337   const auto Xi_m = (l_total.begin() + l_total.size() / 2);
338   const auto Xe_m = (p_total.begin() + p_total.size() / 2);
339
340   std::nth_element(l_total.begin(), Xi_m, l_total.end());
341   std::nth_element(p_total.begin(), Xe_m, p_total.end());
342
343   // Original space and time
344   auto We = (float) p_node_size_b;
345   auto Xe = (float) *Xe_m;
346   // Alternative space and time
347   auto Wi = (float) alt_b_siz + alt_s_siz + l_vlcp_size_b + l_vlcc_size_b;
348   auto Xi = (float) *Xi_m;
349
350   printf("+--[Summary]------------------------------------------------\n");
351   printf("| Samples/runs                          : %d\n", no_samples);
352   printf("| No. Signatures in search branch       : %d-%d\n", signo_sb_alt,
          signo_sb_orig);
353   printf("| Filename                              : %s\n", infile);
354   printf("| Total No. Signatures                  : %lu \n", sig_list.size());
355   printf("| Signature list size                   : %lu\tB\n", sig_size_b);
356   printf("| No of unique ports                    : %d\n", unique_ports);
357   printf("+--[ORIGINAL representation]--------------------------\n");
358   printf("| Size  (We)                            : %.5f KB\n", (float) (We /
          1024.0f));
359   printf("| Size  (We)                            : %.1f B\n", We);
360   printf("| Median search time in ns (Xe)         : %.1f\n", Xe);
361   printf("+--[ALTERNATIVE representation]-----------------------------------\n");
362   printf("| Node relations (B)                    : %d\tB\n", alt_b_siz);
363   printf("| Labels (S)                            : %d\tB\n", alt_s_siz);
364   printf("| Sig.ref. - VLC vector (bitvector)     : %lu\tB\n", l_vlcp_size_b);
365   printf("| Sig.ref. - VLC vector (content)       : %lu\tB\n", l_vlcc_size_b);
366   printf("| Total  (Wi)                           : %.5f KB\n", (float) (Wi) /
          1024.0f);
367   printf("| Total  (Wi)                           : %.1f\n", (Wi));
368   printf("| Median search time in ns (Xi)         : %.1f\n", Xi);
369   printf("+-[EFFICIENCY]----------------------------------------------------\n");
370   printf("| Space difference (We / Wi)            : %.5f\n", (We / Wi));
371   printf("| Time difference  (Xi / Xe)            : %.5f\n", (Xi / Xe));
372   printf("| Result     (We * Xe) / (Wi * Xi)      : %.5f\n", (We * Xe) / (Wi * Xi))
          ;
373   printf("+-----------------------------------------------------------------\n");
374
375
376   for (auto sig : sig_list) {
377     free(sig->content);
378     free(sig->msg);
379
380     delete (sig); // use delete, because of "new"
381   }
382   return 0;
383 }
```

Code listing A.2: fpm.cpp

```cpp
/*
 * Author: Magnus Lien Lilja
 * Proof of concept (PoC) comparison of the Aho-Corasick deterministic finite automaton (DFA) algorithm, which is in
 * Snort and Suricata, and the suggested improvement.
 * The program takes a signature file as input, together with a search text and outputs the efficiency, according to a
 * criteria defined in my master thesis.
 */

#include <iostream>
#include <sdsl/bit_vectors.hpp>
#include <sdsl/wavelet_trees.hpp>
#include <string>
#include <mapper.hpp>
#include <cstdlib>
#include <elias_fano_compressed_list.hpp>
#include "snort-funcs/acsmx.h"
#include "parser.h"
#include "signature.h"

using namespace sdsl;
using namespace std::chrono;
using timer = std::chrono::high_resolution_clock;

ACSM_STRUCT *acsm = acsmNew();

int main(int argc, char *argv[]) {

  char *infile, *searchtext;
  searchtext = nullptr;
  int no_samples = 1;

  float c_states = 0.0f;
  int alt_n_siz = 0;
  int alt_b_siz = 0;

  // Parse arguments
  if (argc < 5) {
    fprintf(stderr, "Usage: ./program -i infile -s sampleno -st searchtext\n");
    exit(0);
  }

  for (int i = 1; i < argc; i++) {
    if (strcmp(argv[i], "-i") == 0) {
      infile = argv[i + 1];
    } else if (strcmp(argv[i], "-s") == 0) {
      no_samples = atoi(argv[i + 1]);
    } else if (strcmp(argv[i], "-st") == 0) {
      searchtext = argv[i + 1];
    }
  }

  // Parse signatures from the input file
  std::vector<Signature *> sig_list;
  ParseSigFile(sig_list, infile);

  // Read search text as bytes
```

```
57  std::ifstream input(searchtext, std::ios::in | std::ios::binary);
58  std::vector<uint8_t> bytes((std::istreambuf_iterator<char>(input)), (std::
        istreambuf_iterator<char>()));
59  input.close();
60
61  // Use the existing functionality from Snort to add patterns and compile the DFA
62  for (size_t i = 0; i < sig_list.size(); i++) {
63    acsmAddPattern(acsm, sig_list[i]->content, sig_list[i]->clen, 0, i);
64  }
65  acsmCompile(acsm);
66
67  uint r;
68  std::vector<uint> lg_l_total, ac_l_total;
69  std::vector<uint> tmp_n; // temporary
70  float lgtm;
71  int ac_nfound;
72  float notzerosates;
73
74  // Exit if few patterns or states
75  if (acsm->acsmMaxStates <= 1 || acsm->numPatterns <= 50) {
76    exit(0);
77  }
78
79
80  for (size_t i = 0; i < no_samples; i++) {
81    notzerosates = 0.0f;
82    // Aho-corasick search and time it
83    auto start = timer::now();
84    ac_nfound = acsmSearch(acsm, reinterpret_cast<unsigned char *>(bytes.data()),
          bytes.size(), (void *) 0, 0);
85    auto stop = timer::now();
86    int ac_st = duration_cast<TIMEUNIT>(stop - start).count();
87
88    // Build the alternative representation
89    int next;
90    uint c = 0;
91    uint state = 0;
92    bit_vector nb(ALPHABET_SIZE * acsm->acsmMaxStates + 1);
93
94    for (int k = 0; k < ALPHABET_SIZE; k++) {
95      for (int i = 0; i < acsm->acsmMaxStates; i++) {
96        next = acsm->acsmStateTable[i].NextState[k];
97        if (next > 0) {
98          notzerosates++;
99          tmp_n.push_back(next);
100         nb[c] = 1;
101       }
102       if (next >= 0) {
103         c++;
104       }
105
106     }
107   }
108
109   nb.resize(c);
110   succinct::elias_fano_compressed_list N(tmp_n);
111
112   tmp_n.clear();
113   tmp_n.shrink_to_fit();
114   int lg_nfound = 0;
```

```
115
116     uint s = (acsm->acsmNumStates + 1);
117     rank_support_v<1> nb_r1(&nb);
118
119     // Use the developed algorithm to search for matches in the improved
            representation.
120     // The existing matchList from Aho-Corasick in Snort is used here, but not
            accounted for memory-wise in either of
121     // the two algorithms
122     start = timer::now();
123     for (auto i : bytes) {
124       r = i * s;
125       if (nb[r + state]) {
126         state = N[nb_r1(r + state)];
127         if (acsm->acsmStateTable[state].MatchList != NULL) {
128           printf("[LG]:Match for %s\n", acsm->acsmStateTable[state].MatchList->
                  casepatrn);
129           lg_nfound++;
130         }
131       } else {
132         state = 0;
133       }
134     }
135
136     stop = timer::now();
137
138     // Assert that the two versions match equally as many patterns.
139     assert(lg_nfound == ac_nfound);
140
141     // Register time and space
142     int lg_st = duration_cast<TIMEUNIT>(stop - start).count();
143     lgtm = (float) succinct::mapper::size_tree_of(N)->size + (float) size_in_bytes(
            nb);
144
145     alt_b_siz = size_in_bytes(nb);
146     alt_n_siz = succinct::mapper::size_tree_of(N)->size;
147
148     ac_l_total.push_back(ac_st);
149     lg_l_total.push_back(lg_st);
150     c_states = c;
151   }
152
153   // Calculate median: https://en.cppreference.com/w/cpp/algorithm/nth_element
154   const auto Xi_m = (lg_l_total.begin() + lg_l_total.size() / 2);
155   const auto Xe_m = (ac_l_total.begin() + ac_l_total.size() / 2);
156
157   std::nth_element(lg_l_total.begin(), Xi_m, lg_l_total.end());
158   std::nth_element(ac_l_total.begin(), Xe_m, ac_l_total.end());
159
160   // Original space and time
161   auto We = (float) getMem(acsm);
162   auto Xe = (float) *Xe_m;
163   // Alternative space and time
164   auto Wi = (float) lgtm;
165   auto Xi = (float) *Xi_m;
166
167   printf("\n+--[Summary]------------------------------------------------\n");
168   printf("| Samples:                              : %d\n", no_samples);
169   printf("| Filename                              : %s\n", infile);
170   printf("| Percent zero states                   : %.2f%%\n", ((c_states -
```

```
      notzerosates) / c_states) * 100.0f);
171   printf("| No. Signatures                        : %lu \n", sig_list.size());
172   printf("| Number of Matches                     : %d\n", ac_nfound);
173   printf("+--[ORIGINAL representation]----------------------\n");
174   acsmPrintSummaryInfo(acsm);
175   printf("| Size (We)                             : %.1f B\n", We);
176   printf("| Median search time in ns (Xe)         : %.1f\n", Xe);
177   printf("+--[ALTERNATIVE representation]----------------------\n");
178   printf("| N                                     : %d B\n", alt_n_siz);
179   printf("| B                                     : %d B\n", alt_b_siz);
180   if (lgtm < 1024 * 1024)
181     printf("| Size (Wi)                             : %.5f KB\n", Wi / 1024.0f);
182   else
183     printf("| Size (Wi)                             : %.5f MB\n", Wi / (1024.0f *
          1024.0f));
184   printf("| Size (Wi)                             : %.1f B\n", Wi);
185   printf("| Median search time in ns (Xi)         : %.1f\n", Xi);
186   printf("+-[EFFICIENCY]-----------------------------------------------------\n");
187   printf("| Space difference (We / Wi)            : %.5f\n", (We / Wi));
188   printf("| Time difference  (Xi / Xe)            : %.5f\n", (Xi / Xe));
189   printf("| Result     (We * Xe) / (Wi * Xi)      : %.5f\n", (We * Xe) / (Wi * Xi))
          ;
190   printf("+-------------------------------------------------------------------\n");
191
192
193   acsmFree(acsm);
194
195   for (auto sig : sig_list) {
196     free(sig->content);
197     free(sig->msg);
198
199     delete (sig); // use delete, because of "new"
200   }
201
202   return 0;
203 }
```

Code listing A.3: lgraph.h

```
1  #ifndef LGRAPH_H
2  #define LGRAPH_H
3
4  #include <sdsl/bit_vectors.hpp>
5  #include "defs.h"
6  #include <sdsl/wavelet_trees.hpp>
7
8  //! Namespace for the succinct data structure library.
9  using namespace sdsl;
10 using namespace std::chrono;
11 using timer = std::chrono::high_resolution_clock;
12
13 template<class bit_vec_t = bit_vector>
14 class lgraph {
15 public:
16   typedef bit_vec_t bit_vector_type;
17
18 private:
19   bit_vector::rank_1_type b_rank1;
20
21   bit_vector_type b;
```

```
22    bit_vector_type bl;
23
24    std::vector <uint> tmp_l; // temporary label storage
25    vlc_vector <VLC_CODER> L; // final label storage
26    std::vector <uint> tmp_n; // temporary label storage
27    vlc_vector <VLC_CODER> N; // final label storage
28
29  public:
30    const vlc_vector <VLC_CODER> &l;
31    const bit_vector_type &bv; // const reference to the LOUDS sequence
32
33    lgraph() : b(), bl(), bv(b), l(L) {
34      b = bit_vector_type();
35      bl = bit_vector_type();
36    }
37
38    void appendLabel(std::vector <uint> labels = {}) {
39      bit_vec_t tmp_bv = b;
40      tmp_bv.resize(b.size() + labels.size() + 1);
41      tmp_bv[(b.size()) + 1] = 1;
42      b = bit_vector_type(std::move(tmp_bv));
43      for (auto i : labels) {
44        tmp_l.push_back(i);
45      }
46    }
47
48    void appendDistNodeN(std::vector <uint> nodes = {}) {
49      bit_vector_type tmp_bv = bl;
50      tmp_bv.resize(bl.size() + nodes.size() + 1);
51      tmp_bv[(bl.size()) + 1] = 1;
52      bl = bit_vector_type(std::move(tmp_bv));
53      for (auto i : nodes) {
54        tmp_n.push_back(i);
55      }
56    }
57
58    // Initialise support data structure with rank and select support on bitvectors
         ...
59    void finalize() {
60      std::cout << "finializing" << std::endl;
61
62      b_rank1(&b);
63
64      std::cout << tmp_l.size() << " " << tmp_n.size() << " " << b.size() << " " <<
           bl.size() << std::endl;
65
66      L = vlc_vector<VLC_CODER>(std::move(tmp_l));
67      N = vlc_vector<VLC_CODER>(std::move(tmp_n));
68
69      tmp_l.clear();
70      tmp_l.shrink_to_fit();
71      tmp_n.clear();
72      tmp_n.shrink_to_fit();
73    }
74  };
75
76  #endif
```

Code listing A.4: ptree.h

```
#ifndef NTREE_H
#define NTREE_H

#include "signature.h"
#include "defs.h"


typedef struct Portnode_ {
  Portnode_ *next = nullptr;
  Signature *sn;
} Portnode;

typedef struct Flownode_ {
  Portnode *tcp[MAX_PORTS];
  Portnode *udp[MAX_PORTS];
} Flownode;

typedef struct Rootnode_ {
  Flownode *flow_gh[MAXFLOW];
} Rootnode;

#endif
```

Code listing A.5: parser.h

```
#ifndef PARSER_H
#define PARSER_H

#include "defs.h"
#include "signature.h"
#include <vector>
#include <map>

// Some predefined port variables..
inline std::map<std::string, std::vector<uint32_t>> port_vartable = {
    {"$HTTP_PORTS",      {80,  81, 311, 383, 591, 593, 901, 1220, 1414, 1741, 1830,
          2301, 2381, 2809, 3037, 3128, 3702, 4343, 4848, 5250, 6988, 7000, 7001,
         7144, 7145, 7510, 7777, 7779, 8000, 8008, 8014, 8028, 8080, 8085, 8088,
         8090, 8118, 8123, 8180, 8181, 8243, 8280, 8300, 8800, 8888, 8899, 9000,
         9060, 9080, 9090, 9091, 9443, 9999, 11371, 34443, 34444, 41080, 50002,
         55555}},
    {"$FILE_DATA_PORTS", {110, 143}},
    {"$FTP_PORTS",       {{21, 2100, 3535}}}};

int ParseRuleMessage(Signature *sid, char *args);

int ParseRuleContent(Signature *sid, char *args);

int ParseRuleSid(Signature *sid, char *args);

int ParseOptions(Signature *sig, char *rule_opts);

int ParseHeader(Signature *sig, char **toks);

int ParseSigFile(std::vector<Signature *> &sig_list, const char *sig_file);

int ParseRuleFlow(Signature *sig, char *args);

int AppendSignature(Signature *sig, char *line, int signum);
```

```

int GetRuleProtocol(char *proto_str);

int ParsePorts(Signature *sig, char *port_str);

#endif
```

Code listing A.6: parser.cpp

```
#include "parser.h"
#include "signature.h"
#include "defs.h"
#include <iostream>
#include "snort-funcs/misc.h"
#include <cstring>
#include <vector>
#include <map>
#include "suricata-funcs.h"

typedef int (*ParseRuleFunc)(Signature *sid, char *);

typedef struct _RuleFunc {
  char *name;
  ParseRuleFunc parse_func;

} RuleFunc;

static const RuleFunc rule_options[] =
    {
        {RULE_OPT_MSG,     ParseRuleMessage},
        {RULE_OPT_CONTENT, ParseRuleContent},
        {RULE_OPT_SID,     ParseRuleSid},
        {NULL, NULL} /* Marks end of array */
    };

int ParseRuleFlow(Signature *sig, char *args) {
  if (strcasecmp(args, "<-") == 0) {
    sig->flowdir = FLOWDIR_TOCLIENT;
    return 0;
  } else if (strcasecmp(args, "->") == 0) {
    sig->flowdir = FLOWDIR_TOSERVER;

    return 0;
  }
  return -1;
}

int ParseRuleMessage(Signature *sig, char *args) {
  // printf("[PARSEMESSAGE][INFO]: message %s\n", args);
  int ovlen = strlen(args);
  if (ovlen > 1) {
    /* strip leading " */
    args++;
    ovlen--;
    args[ovlen - 1] = '\0';
    ovlen--;
  }

  sig->msg = strdup(args);
  return 0;
```

```
}

int ParseRuleContent(Signature *sig, char *args) {
  int ovlen = strlen(args);
  /* skip leading whitespace */
  while (ovlen > 0) {
    if (!isblank(*args))
      break;
    args++;
    ovlen--;
  }

  /* see if value is negated */
  if (*args == '!') {
    args++;
    ovlen--;
  }
  /* skip more whitespace */
  while (ovlen > 0) {
    if (!isblank(*args))
      break;
    args++;
    ovlen--;
  }

  if (ovlen > 1) {
    /* strip leading " */
    args++;
    ovlen--;
    args[ovlen - 1] = '\0';
    ovlen--;
  }
  if (DetectContentDataParse(args, sig) != 0) {
    return -1;
  } else {
    return 0;
  }
}

int ParseRuleSid(Signature *sig, char *args) {
  sig->sid = atoi(args);
  return 0;
}

int ParseOptions(Signature *sig, char *rule_opts) {
  if (rule_opts == NULL) {
    printf("No rule options.\n");
  } else {
    char **toks;
    int num_toks;
    int i;

    if ((rule_opts[0] != '(') || (rule_opts[strlen(rule_opts) - 1] != ')')) {
      printf("Rule options must be enclosed in '(' and ')'.");
    }

    /* Move past '(' and zero out ')' */
    rule_opts++;
    rule_opts[strlen(rule_opts) - 1] = '\0';
    toks = mSplit(rule_opts, ";", 0, &num_toks, '\\');
```

```
112
113     for (i = 0; i < num_toks; i++) {
114       char **opts;
115       int num_opts;
116       char *option_args = NULL;
117       int j;
118
119       //printf("option: %s\n", toks[i]);
120
121       /* break out the option name from its data */
122       opts = mSplit(toks[i], ":", 2, &num_opts, '\\');
123
124       //printf("   option name: %s\n", opts[0]);
125
126       if (num_opts == 2) {
127         option_args = opts[1];
128         //   printf("   option args: %s\n", option_args);
129       }
130
131       for (j = 0; rule_options[j].name != NULL; j++) {
132         if (strcasecmp(opts[0], rule_options[j].name) == 0) {
133           if (rule_options[j].parse_func(sig, option_args) != 0) {
134             mSplitFree(&opts, num_opts);
135             mSplitFree(&toks, num_toks);
136             return -1;
137           }
138         }
139       }
140       mSplitFree(&opts, num_opts);
141     }
142     mSplitFree(&toks, num_toks);
143   }
144   return 0;
145 }
146
147 int ParseHeader(Signature *sig, char **toks) {
148   /* Set the rule protocol - fatal errors if protocol not found */
149   uint8_t protocol = GetRuleProtocol(toks[1]);
150   switch (protocol) {
151     case IPPROTO_TCP:
152       sig->protocol = IPPROTO_TCP;
153       break;
154     case IPPROTO_UDP:
155       sig->protocol = IPPROTO_UDP;
156       break;
157     case ERROR_RETURN:
158       return -1;
159   }
160
161   if (ParseRuleFlow(sig, toks[4]) != 0) {
162     return -1;
163   }
164
165   if (sig->flowdir == FLOWDIR_TOCLIENT) {
166     if (ParsePorts(sig, toks[3] /* =src port */) != 0) {
167       return -1;
168     }
169   } else {
170     if (ParsePorts(sig, toks[6] /* =dst port */) != 0) {
171       return -1;
```

```
172     }
173   }
174
175   return 0;
176 }
177
178 int ParseSigFile(std::vector<Signature *> &sig_list, const char *sig_file) {
179   int signum = 0;
180   char line[8192];
181   long offset = 0;
182   int lineno = 0;
183   int multiline = 0;
184
185   FILE *fp = fopen(sig_file, "r");
186   if (fp == nullptr) {
187     printf("[PARSE][ERROR]: opening rule file %s", sig_file);
188     return -1;
189   }
190
191   while (fgets(line + offset, (int) sizeof(line) - offset, fp) != nullptr) {
192     lineno++;
193     int len = strlen(line);
194
195     /* ignore comments and empty lines */
196     if (line[0] == '\n' || line[0] == '\r' || line[0] == ' ' || line[0] == '#' ||
          line[0] == '\t')
197       continue;
198
199     /* Check for multiline rules. */
200     while (len > 0 && isspace((unsigned char) line[--len]));
201     if (line[len] == '\\') {
202       multiline++;
203       offset = len;
204       if (offset < sizeof(line) - 1) {
205         /* We have room for more. */
206         continue;
207       }
208       /* No more room in line buffer, continue, rule will fail
209        * to parse. */
210     }
211
212     /* Check if we have a trailing newline, and remove it */
213     len = strlen(line);
214     if (len > 0 && (line[len - 1] == '\n' || line[len - 1] == '\r')) {
215       line[len - 1] = '\0';
216     }
217
218     /* Reset offset. */
219     offset = 0;
220
221     /* Parse the signature */
222     Signature *sig = new Signature();
223     if (AppendSignature(sig, line, signum) == 0 && (sig->content && sig->dstport.
          size() && sig->flowdir)) {
224       sig_list.push_back(sig);
225       signum++;
226     } else {
227       if (sig->content)
228         free(sig->content);
229       if (sig->msg)
```

```
230         free(sig->msg);
231       delete (sig); // use delete, because of "new"
232     }
233
234     multiline = 0;
235   }
236   fclose(fp);
237   return signum;
238 }
239
240 int AppendSignature(Signature *sig, char *line, int signum) {
241   char **toks = NULL;
242   int num_toks = 0;
243   // Signature *sig = new Signature();
244   sig->id = signum;
245
246   toks = mSplit(line, " \t", 8, &num_toks, '\\');
247   char *roptions = toks[7];
248   if (ParseOptions(sig, roptions) != 0) {
249     mSplitFree(&toks, num_toks);
250     return -1;
251   }
252   if (ParseHeader(sig, toks) != 0) {
253     mSplitFree(&toks, num_toks);
254     return -1;
255   }
256   mSplitFree(&toks, num_toks);
257   return 0;
258 }
259
260 int GetRuleProtocol(char *proto_str) {
261   if (strcasecmp(proto_str, RULE_PROTO_OPT_TCP) == 0) {
262     return IPPROTO_TCP;
263   } else if (strcasecmp(proto_str, RULE_PROTO_OPT_UDP) == 0) {
264     return IPPROTO_UDP;
265   } else {
266     return -1;
267   }
268 }
269
270 int ParsePorts(Signature *sig, char *port_str) {
271
272   /* 1st - check if we have an any port */
273   if (strcasecmp(port_str, "any") == 0) {
274     sig->dstport = {ANYPORT};
275     return 0;
276   }
277
278     /* 2nd - check if we have a PortVar */
279   else if (port_str[0] == '$') {
280     auto ports = port_vartable.find(port_str);
281     if (ports != port_vartable.end()) { // found key
282       sig->dstport = ports->second;
283       return 0;
284     }
285     return -1;
286
287   } /* 3rd -  and finally process a raw port list */
288   else {
289     if (port_str[0] == '[') {
```

```
290       int ovlen = strlen(port_str);
291
292       if (ovlen > 1) {
293         /* strip leading " */
294         port_str++;
295         ovlen--;
296         port_str[ovlen - 1] = '\0';
297         ovlen--;
298       }
299     }
300     std::vector<uint32_t> tmpar;
301     int num_toks;
302     char **toks = mSplit(port_str, ",", 0, &num_toks, '\\');
303
304     for (size_t i = 0; i < num_toks; i++) {
305       int tmp_ret = atoi(toks[i]);
306       if (tmp_ret == 0) {
307         mSplitFree(&toks, num_toks);
308         return -1;
309       }
310       tmpar.push_back(tmp_ret);
311     }
312     if (tmpar.size()) {
313       sig->dstport = tmpar;
314     }
315     mSplitFree(&toks, num_toks);
316     return 0;
317   }
318 }
```

Code listing A.7: ltree.h

```
1  /*
2      Inspired by and modified from:
3          https://github.com/simongog/sdsl-lite/blob/master/examples/louds-tree.cpp
4          by Simon Gog
5  */
6  #ifndef LTREE_H
7  #define LTREE_H
8
9  #include <sdsl/bit_vectors.hpp>
10 #include "defs.h"
11 #include <sdsl/wavelet_trees.hpp>
12
13 //! Namespace for the succinct data structure library.
14 using namespace sdsl;
15 using namespace std::chrono;
16 using timer = std::chrono::high_resolution_clock;
17
18 //! A tree class based on the level order unary degree sequence (LOUDS)
       representation.
19 template<class bit_vec_t = bit_vector,
20     class select_1_t = typename bit_vec_t::select_1_type,
21     class select_0_t = typename bit_vec_t::select_0_type,
22     class rank_1_t = typename bit_vec_t::rank_1_type,
23     class rank_0_t = typename bit_vec_t::rank_0_type>
24 class ltree {
25 public:
26   typedef bit_vector::size_type size_type;
27   typedef bit_vec_t bit_vector_type;
```

```
28    typedef select_1_t select_1_type;
29    typedef select_0_t select_0_type;
30    typedef rank_1_t rank_1_type;
31    typedef rank_0_t rank_0_type;
32
33  private:
34    bit_vector_type m_bv;        // bit vector for the LOUDS sequence
35    select_1_type m_bv_select1;  // select support for 1-bits on m_bv
36    select_0_type m_bv_select0;  // select support for 0-bits on m_bv
37    rank_1_type m_bv_rank1;      // rank support for 1-bits on m_bv
38    rank_0_type m_bv_rank0;      // rank support for 0-bits on m_bv
39    std::vector<uint32_t> tmp_s; // temporary label storage
40    wt_int<bit_vector> S;        // final label storage
41    uint select_unique_time = 0;
42
43  public:
44    const wt_int<bit_vector> &s;
45    const bit_vector_type &bv; // const reference to the LOUDS sequence
46    const uint &su_t;
47
48    ltree()
49        : m_bv(), m_bv_select1(), m_bv_select0(), m_bv_rank1(), m_bv_rank0(), bv(m_bv
              ), s(S), su_t(select_unique_time) {
50      bit_vector tmp_bv(2, 0);
51      tmp_bv[0] = 1; // 10 in first two spots...
52      m_bv = bit_vector_type(std::move(tmp_bv));
53    }
54
55    void append(int noc, std::vector<uint32_t> s = {}) {
56      if (s.size() != noc) {
57        printf("[LTREE][ERROR]: No. child does not match labels");
58        return;
59      } else {
60        bit_vector tmp_bv = m_bv;
61        tmp_bv.resize(m_bv.size() + noc + 1);
62        size_type pos = m_bv.size();
63        if (noc) {
64          for (int i = 0; i < noc; i++) {
65            tmp_bv[pos++] = 1;
66            tmp_s.push_back(s[i]);
67          }
68        }
69
70        tmp_bv[pos++] = 0;
71        tmp_bv.resize(pos);
72        m_bv = bit_vector_type(std::move(tmp_bv));
73      }
74    }
75
76    // Initialise support data structure with rank and select support on bitvectors
         ...
77    void finalize() {
78      util::init_support(m_bv_select1, &m_bv);
79      util::init_support(m_bv_select0, &m_bv);
80      util::init_support(m_bv_rank1, &m_bv);
81      util::init_support(m_bv_rank0, &m_bv);
82
83      construct_im(S, tmp_s, 4);
84      tmp_s.clear();
85      tmp_s.shrink_to_fit();
```

```
86    }
87
88    //! Returns the t'th child of v
89    int child(int v, int t) {
90      return nodemap(m_bv_select0(m_bv_rank1(v + t)) + 1);
91    }
92
93    //! Returns unique identifier in [1,n] for v
94    int nodemap(int i) {
95      return m_bv_rank0(i);
96    }
97
98    //! Converts unique identifier to index in bitvector
99    int nodeselect(int v) {
100     return m_bv_select0(v) + 1;
101   }
102
103   //#define DEBUG
104   //! Returns the node identifer for the i'th child labeled l for the node id v
105   int labeledchild(int v, uint32_t l) {
106     int i = nodeselect(v);
107     int s = m_bv_rank1(i) - 1;
108
109     int tmp_s = S.select(S.rank(s, l) + 1, l) - s;
110     return child(i, tmp_s);
111   }
112 };
113
114 #endif
```

Code listing A.8: signature.h

```
1  #ifndef SIGNATURE_H
2  #define SIGNATURE_H
3
4  #include "defs.h"
5  #include <iostream>
6  #include <vector>
7
8  typedef struct Signature_ {
9    uint8_t protocol;
10   std::vector<uint32_t> dstport;
11
12   int id;
13   char *msg;
14   uint16_t clen = 0;
15   uint8_t *content;
16   int sid;
17   int flowdir;
18 } Signature;
19
20 #endif
```

Code listing A.9: defs.h

```
1  #ifndef DEFS_H
2  #define DEFS_H
3
4  #define RULE_OPT_MSG "msg"
```

```
#define RULE_OPT_CONTENT "content"
#define RULE_OPT_SID "sid"
#define RULE_OPT_FLOW "flow"
#define RULE_PROTO_OPT_TCP "tcp"
#define RULE_PROTO_OPT_UDP "udp"
#define TOKS_BUF_SIZE 100
#define FLOWDIR_TOCLIENT 2
#define FLOWDIR_TOSERVER 1
#define MAXFLOW 3
#define MAX_PORTS 65536
#define ERROR_RETURN -1

#define ANYPORT 0

/* Standard well-defined IP protocols.  */
#define IPPROTO_TCP 1
#define IPPROTO_UDP 2
#define IPPROTO_MAX 3

#define VLC_CODER coder::fibonacci
#define TIMEUNIT nanoseconds

#endif
```

Code listing A.10: byte-frequency.py

```
# -*- coding: utf-8 -*-
"""

This program aims to find the ASCII values not in used by the content of a given
    ruleset

"""

import matplotlib.pyplot as plt
import sys

# Customize matplotlib

vals = {}

v_sum = 0

with open(sys.argv[1]) as f:
  for v in [line for line in f]:
    if len(v.strip()):
      if v.strip()[0] != "#":
        for l in v.strip().split(";"):
          if l.split(":")[0].strip() == "content":
            cnt = str(l.split(":")[1]).strip()
            end = len(cnt)
            j = 0
            if cnt[j] == "!":
              j += 1
            cnt.strip()
            if cnt[j] == "\"":
              j += 1
            cnt.strip()
            if cnt[-1] == "\"":
              end -= 1
```

```
            while j < end:
              c = cnt[j]
              if c == "|":
                j_p = cnt.find("|", j + 1)
                if j_p > 0:
                  try:
                    for c in bytearray.fromhex(cnt[j + 1:j_p].replace("|", "")).
                        decode('latin1'):
                      if ord(c) not in vals:
                        vals.update({ord(c): 1})
                        v_sum += 1
                      else:
                        vals[ord(c)] += 1
                        v_sum += 1
                    j = j_p + 1
                  except Exception:
                    break
              else:
                if ord(c) not in vals:
                  vals.update({ord(c): 1})
                  v_sum += 1
                else:
                  vals[ord(c)] += 1
                  v_sum += 1
                j += 1

for x, y in vals.items():
  vals[x] = float(y / v_sum)

vals = dict(sorted(vals.items(), key=lambda item: item[0]))

plt.bar(list(vals.keys()), vals.values(), color='darkorchid')
plt.title(sys.argv[1])
plt.xlabel("Byte")
plt.ylabel("Byte frequency")
plt.savefig("byte-frequency.svg")
```

Efficient representation of data in intrusion detection systems

NTNU

Kunnskap for en bedre verden