Master's thesis

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

Bodil Åberg Mokkelbost

The effectiveness of using the SBOM search algorithm in Intrusion Detection Systems

Master's thesis in Information Security
Supervisor: Slobodan Petrović
December 2021

NTNU
Norwegian University of Science and Technology

Bodil Åberg Mokkelbost

The effectiveness of using the SBOM search algorithm in Intrusion Detection Systems

Master's thesis in Information Security
Supervisor: Slobodan Petrović
December 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

Preface

This thesis is the final part of the Master's degree at the Norwegian University of Science and Technology (NTNU). It is a Master's degree within the field of Information Security and is completed over a period from August 2018 to December 2021.

The main topic of this thesis is the Set Backward Oracle Matching search algorithm and its suitability in an Intrusion Detection System regarding efficiency and security. Security within Intrusion Detection Systems was a field that I found specially interesting during my studies.

Lillehammer, 12th December 2021
Bodil Mokkelbost

Acknowledgment

I would like to thank my supervisor, Slobodan Petrović, for his guidance with this thesis. His suggestions and feedback have contributed for the finished product of this thesis. I will thank my superior at work for giving me some time off to write the thesis, and my colleagues for interesting discussions around the topic of this thesis. I will also thank my friend, Henriette Borgund, for proof reading. Finally I will thank my fiancé for enabling me to complete this masters degree, and my friends and family for motivation and support throughout my studies.

B.M.

Abstract

Malware poses a severe threat to people, companies, and governments. To search for malware in real-time is the main task of an Intrusion Detection System (IDS) to be able to prevent the malware from causing any damage to the system. With an increasing amount of data and more extensive high-speed networks, the IDS' ability to search through data fast becomes more important.

A way to make the IDS faster is to change the search algorithm. Events during the last two decades have also shown that it is possible to launch attacks against the IDS to neutralize it before penetrating the system it protects. Hence, a search algorithm used in an IDS has to be both fast and resilient to possible algorithmic complexity attacks.

This thesis takes a closer look at the very fast Set Backward Oracle Matching (SBOM) search algorithm and its effectiveness in an IDS. The results show that it is up to 6 times faster than the commonly used Aho-Corasick search algorithm. The results also show that it is possible to launch a successful algorithmic complexity attack against an IDS that uses the SBOM search algorithm. Further on, the results show that if implemented with the Aho-Corasick search algorithm, it is possible to get full advantage of the SBOM search algorithm's speed while making it more resilient to algorithmic complexity attacks.

Sammendrag

Skadevare utgjør en alvorlig trussel overfor enkeltpersoner, selskaper og myndigheter. Å kunne søke etter skadevare i sanntid er en av hovedoppgavene til et Intrusion Detection System (IDS) for å forhindre at skadevaren forårsaker skade på systemet. Med en økende mengde data og flere omfattende høyhastighetsnettverk, blir evnen til å hurtig kunne søke gjennom data en viktig faktor.

Ved å endre hvilken søkealgoritme som brukes i en IDS kan man gjøre den raskere til å søke gjennom data. Hendelser de siste to tiårene har vist at det er mulig å angripe en IDS for å sette den ut av spill før videre penetrering av systemet som den beskytter. På bakgrunn av dette må det settes krav til søkealgoritmen som brukes i en IDS. Søkealgoritmen må, i tillegg til å være rask, også være motstandsdyktig overfor algoritmiske angrep, såkalte ”algorithmic complexity attacks”.

Denne masteroppgaven tar en nærmere kikk på den svært raske søkealgoritmen ”Set Backward Oracle Matching algorithm” (SBOM) og hvor effektiv den kan være brukt i en IDS. Resultatene viser at den er opp til 6 ganger så rask som Aho-Corasick søkealgoritmen som ofte blir benyttet i dag. Resultatene viser også at det er mulig å gjennomføre et vellykket algoritmisk angrep mot SBOM algoritmen. Videre viser resultatene at hvis SBOM algoritmen implementeres sammen med Aho-Corasick algoritmen er det mulig å utnytte hastigheten til SBOM søkealgoritmen på en svært god måte og samtidig gjøre den mer motstandsdyktig mot algoritmiske angrep.

Contents

Figures

Tables

Chapter 1

Introduction

To be able to find a pattern in a string is one of the oldest problems in computer science[1]. With the increasing amount of data, an efficient way to search for the pattern in a given string gets more and more important. Several search algorithms have been developed throughout the years to address this problem, each of them addressing different scenarios and, therefore, better in different areas. All algorithms strive to be as fast as possible, but some algorithms are faster when the alphabet is short, others have multiple patterns, and some use approximate searches. Because of that, the choice of the search algorithm is important for achieving satisfactory efficiency of the overall system.

In Intrusion Detection Systems (IDS), a system developed to detect malicious activity in a system or device, pattern recognition is one of the most important and most used ways to detect malicious activity. Efficiency is a high priority in IDS', but equally important is that the algorithm is resistant to algorithmic attacks. An algorithmic attack is an attack that aims at taking down the security device (mostly an IDS) in the network by exploiting vulnerabilities in the search algorithm used in the IDS. Today, the most used algorithm in IDS' is the Aho-Corasick algorithm[2]. The algorithm is a multi-pattern algorithm, but it is not very fast. The reason this algorithm is widely used in IDS is mainly because of the fact that it is impossible to launch an algorithmic attack against it[2]. Over the years, several different algorithms have been proposed as a replacement for the Aho-Corasick algorithm, but the algorithm has not been changed or replaced in IDS.

The Set Backwards Oracle Matching (SBOM) algorithm is a version of the Backward Oracle Matching (BOM) algorithm that is able to recognize sets of strings from a text. The SBOM algorithm has proven to be very fast and performs better when alphabet size, pattern length, and the number of patterns grow. However, it has a worst-case complexity different from the average-case complexity, which means it may be vulnerable to Algorithmic Attacks, though this does not mean that it is feasible to launch such an attack in practice.

1.1 Problem description

Consider a practical case that illustrates the problem:

On a warm summer afternoon, an IT administrator in the Danish company A.P. Møller-Maersk prepares a software update for their network when his computer suddenly turns black. Looking up from his desk, he can see that he is not the only one in the room as one computer after the other turns black, some of them showing a message that their computer was encrypted. To decrypt it, they would have to pay a ransom[3]. A few minutes later, the whole company network was down. Either the computers were infected with the malware, or the computers were taken of the network to prevent infection[3].

The Maersk company had been infected by the ransomware NotPetya, a type of malware, which encrypted all files on the machine[3][4]. This malware also had worm capabilities which means it was capable of spreading to entire networks and encrypting large amounts of data[4][5].

Malware has existed as long as computers[6]. The first malware created were relatively harmless made either as an experiment or as fun for developers and annoyance for the infected users[6][7]. However, during the 1990s and the rise of the Internet, the malware was adopted by criminal groups. As the amount of data in networks increased, Cyberspace also became a domain for intelligence and manipulation. The incidents reported in recent years, [8][9], have shown that malware is not only made for research or annoyance but is today a significant threat to persons and even governments.

The Norwegian research Mørketallsundersøkelsen is a research aiming to shed light on information security, its extent, and the consequence of not taking it seriously[10]. An extract of types of information security incidents are shown in table 1.1. What is even more interesting is that 44% of the companies saying they have experienced one or more information security incidents say that the incident was detected by a coincidence[11]. In 2020, only 11% of these companies reported the incident to the police, which means that these types of attacks practically never get prosecuted[10].

Threat	2016	2018	2020
Attempted databreach/hacking	8%	13%	14%
Virus and/or other malware attacks	20%	21%	11%
Databreach/hacking	2%	5%	4%
Penetration of security systems	2%	2%	1%
DDoS or threats about DDoS	4%	7%	3%

Table 1.1: Extract from Mørketallsundersøkelsen 2020, different information security incidents at Norwegian companies during 2016, 2018 and 2020, adapted from [11].

To be able to detect information security incidents, several measures have to be undertaken. One of these measures is the use of IDS that can detect and alarm

about a possible attack. But even though the IDS have been considerably better during the last decades, malware is still successful in some situations. It is taking advantage of new technologies making the attack surface bigger (see figure 1.1)[6]. The increased amount of malware circulating the Internet combined with new attack vectors puts pressure on the IDS. More extensive high-speed networks require fast IDS to achieve intrusion detection in real-time, and new types of malware require flexible search algorithms that quickly adapt to new search patterns. At the same time, the algorithm has to be robust to deal with large amounts of data over a long time and handle possible algorithmic attacks.

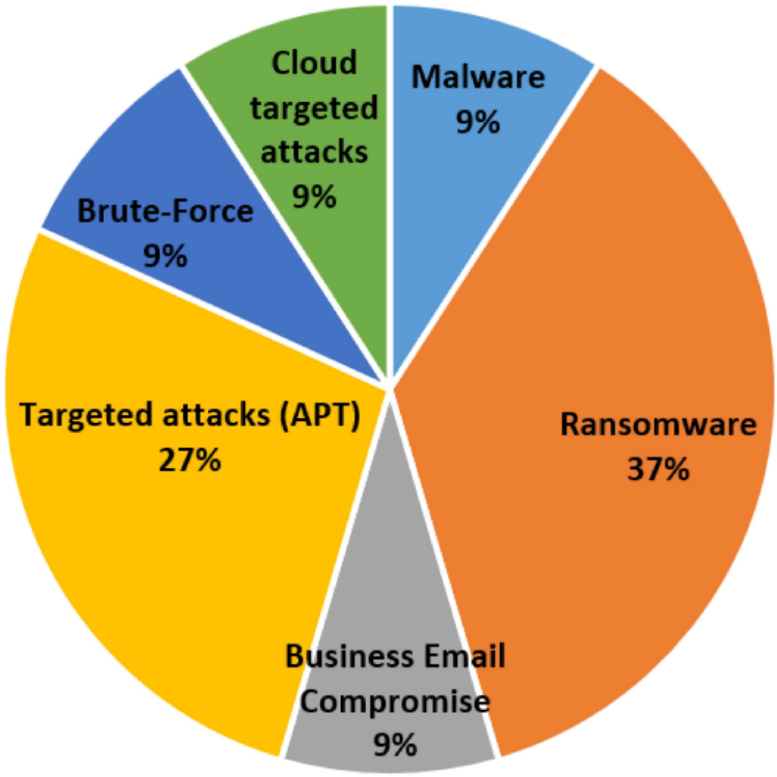


Figure 1.1: The top attack vectors observed in November 2020, adapted from [12].

1.2 Motivation

This thesis aims to see if the SBOM search algorithm can be considered a better choice used in an IDS compared to the Aho-Corasick algorithm based on speed and robustness.

IDS importance increase alongside the increase in the amount of data, size of networks, and amount of targeted and untargeted malware[6]. The IDS therefore

must handle large amounts of data rapidly to be able to achieve real time intrusion detection. This can be done by changing the hardware or software of the IDS. Many algorithms existing today are faster and more efficient than those used in IDS, but many of them are not robust enough and can be used by targeted attacks to make the IDS crash.

1.3 Research questions

In this thesis, we test the following hypothesis:

The SBOM algorithm is a better choice of a search algorithm in an Intrusion Detection System

To test the validity of this hypothesis, three research questions have to be answered:

1. How much better does the SBOM perform compared to the Aho-Corasick algorithm, regarding pattern length and numbers of patterns?
2. How can we use an algorithmic attack to attack the IDS using the SBOM search algorithm?
3. How can we make the SBOM search algorithm more resilient to attacks?

1.4 Contribution

The main contributions of this thesis are as follows:

1. This thesis investigates the effectiveness of algorithmic complexity attacks used on the SBOM algorithm.
2. This thesis develops a combination of the SBOM search algorithm and the Aho-Corasick search algorithm that preserves the speed of the SBOM search algorithm, but in the same time makes it more resilient to algorithmic complexity attacks.
3. This thesis shows how algorithms can be combined in an IDS to better utilize different characteristics of different algorithms.

1.5 Thesis Outline

The rest of the thesis is organized as follows:

1. Chapter 1 gives a brief overview of the topic of the thesis as well as a problem description. The chapter also describes the motivation for the thesis.
2. Chapter 2 presents necessary theory and research on the topic.
3. Chapter 3 presents the research methods used to solve the research questions. The chapter also presents how the experiments are set up and performed so the experiments can be replayed.
4. Chapter 4 looks at ways to modify the SBOM search algorithm making it more suitable for IDS.
5. Chapter 5 presents results from the experiments.
6. Chapter 6 takes a closer look at the results and its validity and possible errors. Combined with chapter 2, the results are discussed to try to answer the research questions.
7. Chapter 7 will conclude the thesis and outlines future work.

Chapter 2

Literature Review

This chapter addresses the relevant theory to give insight into the main hypothesis and the research questions. The chapter firstly addresses malware, before looking into Intrusion Detection Systems (IDS), its history and general structure. Then we look into algorithmic attacks before this thesis' relevant search algorithms are presented. In the end, related work relevant for this thesis is presented.

2.1 Malware

Malware has existed as long as computers[6]. The word *malware* is short for *malicious* and *software* and is software used or created by attackers to gain a purpose on a network or system. In many cases, this can be some sort of financial gain, but it can also be access to the system to either manipulate data, read data or disrupt computer operations[13]. *Malware* is also a general term used for several different types of malicious software, such as *computer viruses, trojans, worms, rootkits, keyloggers, ransomware* and so on[13]. Today, most malware consists of two or more different types, like the NotPetya malware[4].

The first known computer virus was developed already in 1971[7]. The malware could move from system to system (but not clone itself), displaying a message[7]. The first malware found on a personal computer was the program called Brain[13]. The malware spread through floppy disks but as the *Creeper Worm*, it did no harm. The intention of the creators was to show that a computer was not a safe platform[6][13]. At the beginning of the 1990's and the emerging of the Internet, delivery of malware became much easier and in the beginning of the 2000s the use of exploit kits led to a rapid increase in malware delivered over Internet[6]. Then the first financial scams emerged[6]. In the next two decades, it is expected that the number of malware attacks will grow exponentially by doubling or more each year (see Figure 2.1)[6].

A report from Cybersecurity Ventures estimated that ransomware attacks alone cost the world $5 billion USD in 2017[14]. The same report also predicted that this cost would increase to $20 billion in 2021[14], and another report predicts that

cybercrime, in general, will cost $6 trillion USD in 2021, continuing to increase by 15% over the next five years[15].

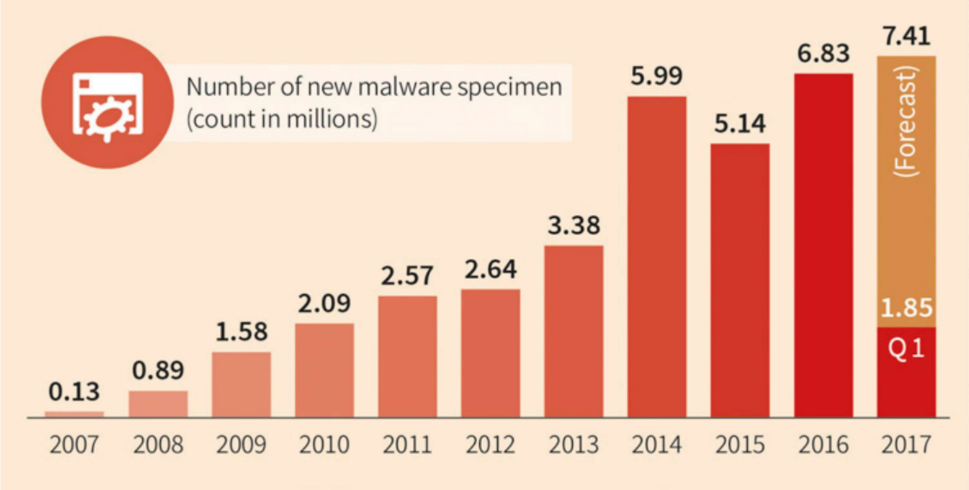


Figure 2.1: The increase in new malware, adapted from [16].

2.2 Intrusion Detection Systems (IDS)

2.2.1 Brief History

At the beginning of the IDS development, the detection included a lot of manual labor by system administrators[17]. With an increase in the amount of data and attacks, ways to make the detection more efficient were needed. In 1980, Anderson published a study on how to improve surveillance at customer sites. This marks the beginning of automated Intrusion Detection[18]. A few years later, the first real-time IDS was developed[18]. During the 1990's the development happened quickly and different types of IDS emerged, as host-based IDS and network-based IDS[18].

2.2.2 Structure

The goal of an IDS is to monitor network assets in order to detect malicious behavior[19]. This behavior can be detected in two different detection modules: Misuse detection or anomaly detection, which is one way to classify the IDS. An anomaly-based IDS learns the system's normal behavior over time and uses this to recognize possible intrusions. Anomaly-based IDS is capable of detecting attacks on the system that have not been seen before, so-called zero-day attacks, but it also has the disadvantage of giving more False Positive Alarms (FPA, generate an alarm for normal traffic) and False Negative Alarms (FNA, not generating

alarm/not recognizing the traffic as malicious). A misuse-based IDS uses signatures in the form of text strings or patterns to detect intrusion. These patterns are characteristics of former known attacks, which means this type of IDS only detects attacks that have happened before[19]. Since the algorithm studied in this thesis looks for patterns in a search text, it is natural to implement it in a Misuse IDS.

In a misuse-based IDS, the most common way of matching patterns is by exact search[2]. With an exact search, each attack is recognized by a signature. A signature is, in this context, a distinctive mark or characteristic being present in a known attack/exploit[20]. These signatures can be strings, fixed offsets, or debugging information[20]. This means that the IDS only detects attacks seen before and that small changes to the attack that results in changes in the signature makes the IDS not recognize the attack[2]. To avoid this, it is possible to perform *approximate search*. In approximate search, the IDS will accept up to a given number of *mistakes* so that minor changes to an attack still will be recognized by the same signature[2]. Approximate search in IDS is still in an experimental state and has not been widely used[2].

An IDS can also be classified by the scope of protection, in other words, what it protects. The IDS can be a Host-based IDS (HIDS), which is similar to antivirus programs. The IDS can also be a Network-based IDS (NIDS). The NIDS is placed strategically in the network to monitor all traffic in/out of the network or between certain points of the network[21]. Both NIDS and HIDS can be either misuse-based or anomaly-based.

An IDS consists of mainly two parts. A preprocessor and a detection algorithm. It takes the incoming traffic and detection module as input (Figure 2.2). The preprocessor collects and formats data so it can be analyzed by the detection algorithm. The detection algorithm uses one or several detection models to determine if the traffic is normal or intrusive[21]. If the detection algorithm recognizes an intrusion, an alert is sent.

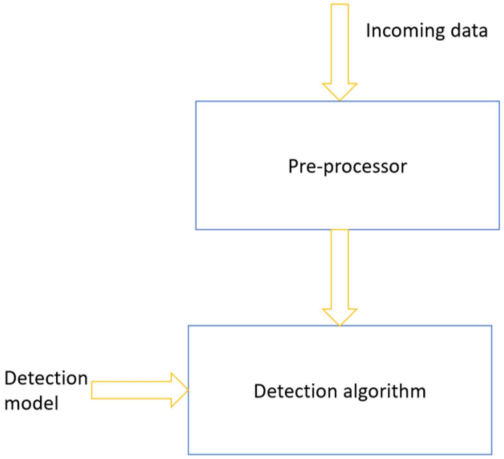


Figure 2.2: Components of an IDS[21]

In a misuse-based IDS, several factors are included to find the optimal search algorithm. Efficiency is important but also handling larger alphabets, and several patterns are key properties. In addition to this, the algorithm has to be resistant to attacks.

2.3 Algorithmic Attacks

Algorithmic attacks are attacks used to neutralize the security device in the network, in most cases the NIDS, [22]. The attack has in recent years become a part of a trend of a two-phased attack where the security device is taken down before the assets it was protecting is attacked[22]. These types of two-phased attacks makes the victim at least less able, and in many cases not able, to detect the actual attack. An example is the attack against SONY in 2011, where account information from up to 101 million users where stolen. In this attack, the IDS was overwhelmed by the amount of data making it drop - or not inspect - all packets. This is called a Distributed Denial of Service (DDoS) attack aimed at the IDS[23].

Another type of algorithmic attack is complexity attack. These attacks exploit the gap between the average-case complexity, which is the amount of resources (memory, cache, etc) the system requires when processing normal packets, and the worst-case complexity, which is the amount of resources the system requires when processing special crafted packets[22]. Up to the early 2000s, it was common to choose an algorithm only based on the average-case complexity, believing the worst-case complexity would not occur in practice[24].

The last way to bypass the IDS is not by attacking the IDS itself but by camouflaging the true positives in an enormous amount of false positives by sending packets that trigger the IDS signatures[25].

2.4 Search algorithms

To solve the string matching problem, finding all occurrences of a given pattern or patterns[1], a search algorithm is used. Several search algorithms have been presented over the last decades, some new, and some variants of already existing ones. Each one focuses on either being faster, simpler, better worst-case complexity, or better average-case complexity. These properties decide in which area the algorithm can be used. As mentioned, the Aho-Corasick algorithm is, because of its properties, the most commonly used algorithm in IDS [26]. With rapid development in technology and continuing increase in data, a faster algorithm to use in IDS is needed.

2.4.1 Terminology

String matching can roughly be divided into three different approaches of how they search through the text[1]. The first approach is to read each character, one by one at each step, check if it matches the pattern and if the whole pattern is found. The other approach is to divide the text into windows of the same length as the pattern p and search for a *suffix* of the pattern backward in the window. The third approach also divides the text into windows of size p, but algorithms using this approach search for a *factor* of the pattern. This approach makes these algorithms the most efficient algorithms in practice[1].

The search algorithms can also be classified according to how they build the trie of the search pattern or patterns. Most algorithms build the trie consisting of nodes. These nodes are linked together with unidirectional links. The first node in the trie is called the zero node or the root node. The rest of the nodes are numbered so the lowest number is closest to the root. If there are several branches in the trie, the second node is in most cases the second child node to the trie and so on (see Figure 2.3). This is called traversal order[1].

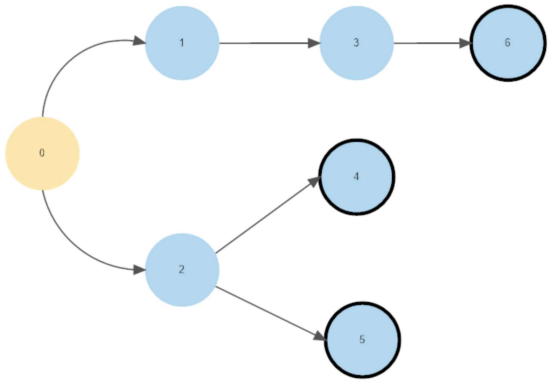


Figure 2.3: A rooted trie

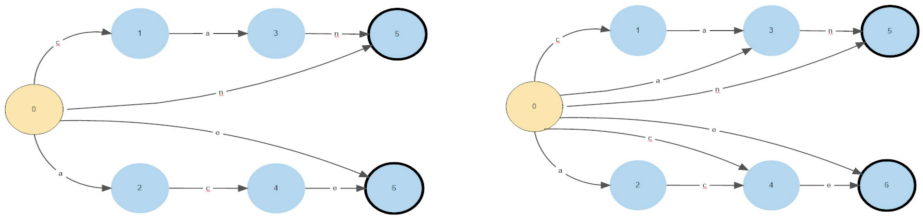
The nodes with no child node are called leaves or terminal nodes. These are marked with thick black lines in Figure 2.3. How the nodes are linked together determines the algorithms *automaton*. The algorithm is a *Deterministic Automaton* if for an input symbol α there exists only one transition to the next state. If there exist several transitions to several different states for the same input symbol α the algorithm is called a *Nondeterministic Automaton*[1]. If the algorithm also is a *Finite State Machine*, which means it has a finite number of states and it can only be in one state at any given time, the algorithm is also called a *Finite Automaton*. These are mostly combined and shortened down to Deterministic Finite Automaton (DFA) or Nondeterministic Finite Automaton (NFA).

Most search algorithms have the same general structure: A *preprocessing phase* where necessary calculations are performed in addition to build the trie or table used for determining shifts between nodes. This phase is followed by a *search phase* or *matching phase* where the search text T is searched through using the trie or table to determine if the pattern p is present or not[27].

In the rest of the thesis, the given terminology is frequently used, together with the definitions given in Table 2.1.

Abbreviation	Explanation
p	The pattern searched for
$P = p_1, p_2 ... p_r$	The set of patterns searched for
r	Number of patterns in P
m	The length of pattern p, or the length of the shortest pattern in the set of patterns P
T	The search text
n	The length of the search text T
$\sum$	The alphabet used in the search
pos	The starting point of the search window in T
j	An incremental value determining what character in the search window or text is read

Table 2.1: Abbreviations



(a) Deterministic Automaton **(b)** Nondeterministic Automaton

Figure 2.4: Rooted trie for Deterministic and Nondeterministic Automaton for the patterns *can, ace*.

2.5 The Knuth-Morris-Pratt (KMP) Search Algorithm

The KMP search algorithm was developed by D.E. Knuth, J.H. Morris and V.R. Pratt already in 1977[28]. In this algorithm, the search for the pattern p in the search text T is done from the beginning by comparing the first character in p with the first character in T. If a match occurs, p_1=T_1, the search continues by comparing p_2=T_2. If there is a mismatch at the first character, p is moved along the text by one character and the comparing continues by checking if p_1=T_2, see Figure 2.5.

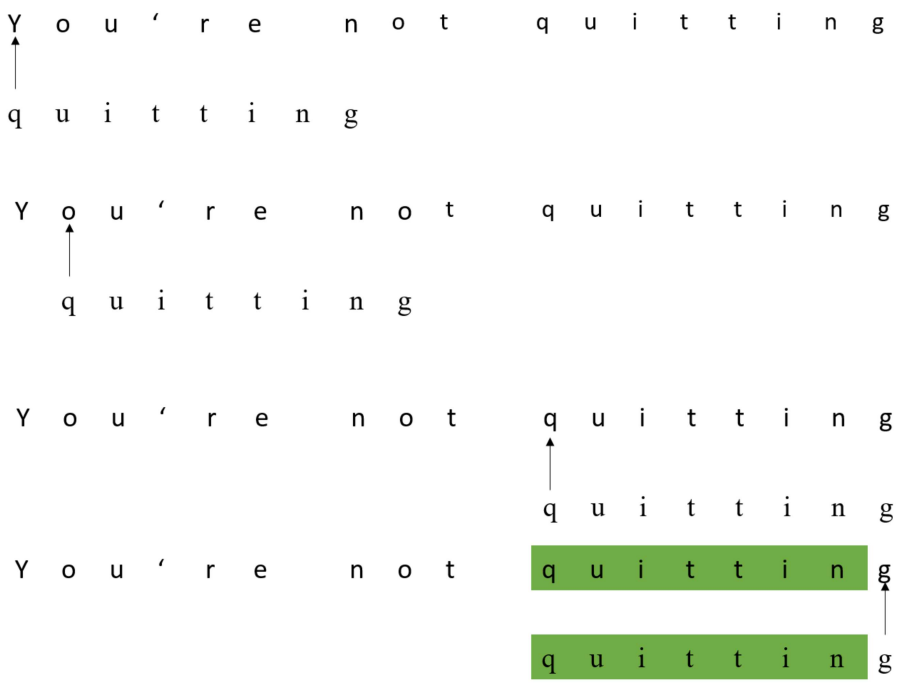


Figure 2.5: Search through the text *You're not quitting*.

How far the pattern is going to shift is based on a table, *next[j]* produced in the preprocessing phase[28]. The 'j' is the current number in p. The pattern is slid *j - next[j]*, see Table 2.2. If the value is zero, the pattern will slide past the current character. To compute the *next[j]*, the program shown in Figure 2.6 is run. In addition, *next[1]* always equals 0.

j	=	1	2	3	4	5	6	7	8
p[j]	=	q	u	i	t	t	i	n	g
f[j]	=	0	1	1	1	1	1	1	1
next[j]	=	0	1	1	1	1	1	1	1

Table 2.2: The pattern shift table

The KMP algorithm has several different extensions making it either faster or

```
t := f[j];
while t > 0 and pattern[j] ≠ pattern[t]
    do t := next[t];
f[j+1] := t+1;
```

(a) KMP processing of *f[j]*

```
j := 1; t := 0; next[1] := 0;
while j < m do
    begin comment t = f[j];
        while t > 0 and pattern[j] ≠ pattern[t]
            do t := next[t];
        t := t+1; j := j+1;
        if pattern[j] = pattern[t]
        then next[j] := next[t]
        else next[j] := t;
    end.
```

(b) KMP processing of *next[j]*

Figure 2.6: The processing of *f[j]* and *next[j]*

```
j := k := 1;
while j ≦ m and k ≦ n do
    begin
        while j > 0 and text[k] ≠ pattern[j]
            do j := next[j];
        k := k+1; j := j+1;
    end;
```

Figure 2.7: Search with the KMP algorithm

detecting several matches. It can also be extended to searching for several patterns, which was done by Aho and Corasick in 1975, creating the Aho-Corasick algorithm[1][28].

2.6 The Aho-Corasick Algorithm

The Aho-Corasick algorithm uses a special automaton which is called the Aho-Corasick automaton[1]. The algorithm uses three functions; a GOTO transition function, a failure transition function (or failure pointer) and an output function[26]. In the following examples, the patterns *snow, snowball,wonder, outside* will be used, along with the text *It's snowing outside, I want to make a snowball*. This gives the following Aho-Corasick Automaton (see Figure 2.8). The yellow

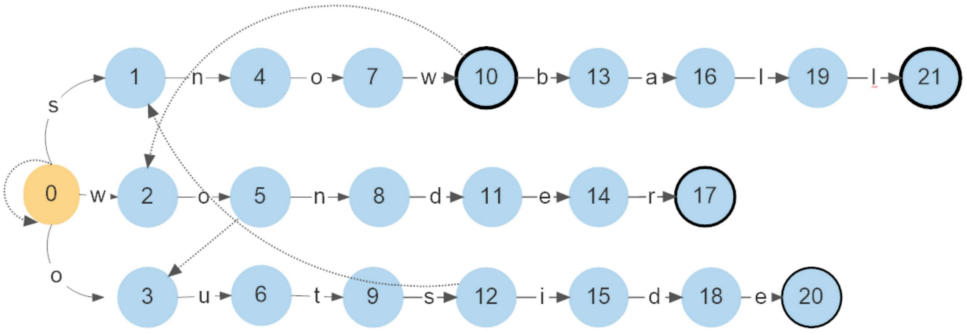


Figure 2.8: Aho-Corasick Automaton for the set of strings *snow,snowball,wondering,outside*

circle is the initial state. The blue states with a thick black circle are terminal states. They will trigger the output function. Each dashed line represents a transition in the failure-function. If there's no dashed line out of a state, the transition goes back to zero and is for simplicity not shown in this figure.

When the search algorithm searches through the text, it will get the result shown in Table 2.3, corresponding to the example given above.

Input text	I	t	'	s		s	n	o	w	i	n	g		o	u	t	s
State	-	-	-	1	-	1	4	7	10	-	-	-	-	3	6	9	12
Output	-	-	-	-	-	-	-	-	10	-	-	-	-	-	-	-	-
Input text	i	d	e	,		I		w	a	n	t		t	o		m	a
State	15	18	20	-	-	-	-	2	-	-	-	-	-	3	-	-	-
Output	-	-	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Input text	k	e		a		s	n	o	w	b	a	l	l				
State	-	-	-	-	-	1	4	7	10	13	16	19	21				
Output	-	-	-	-	-	-	-	-	-	-	-	-	21				

Table 2.3: The search for the patterns in the text (note that the text is devided into three rows)

The strength of the algorithm is that it can detect multiple patterns in one pass [26], which means it reads every character in the search text only once. This

makes it impossible to apply an algorithmic attack[2], but it makes it relatively slow [26].

2.7 The Set Backward Oracle Matching (SBOM) Algorithm

As mentioned previously, one of the factors that are important when considering an algorithm for an IDS, is that it can look for a set of patterns. The Backward Oracle Matching (BOM) algorithm only takes one pattern as input. A version of the BOM algorithm, the Set Backwards Oracle Matching algorithm (SBOM), has been developed[1] to handle multiple search patterns. In this section, the Factor Oracle, which the SBOM search algorithm is built upon, is described in detail. Then the BOM algorithm is presented before it's described how the BOM algorithm is extended into SBOM to be able to search for several patterns.

2.7.1 Factor Oracle

A factor u of a pattern p is a part of the pattern p so that p=xuw, where x, u and w are substrings of p. The factor oracle of a string p is the automaton where all the states are terminal[1]. A string w is recognized by the oracle in the *i-th* state if it labels a path from state 0 to i[1].

The oracle always consists of *m+1* states. The oracle is built by reading the letters of p one by one from left to right, updating the automaton at each step. At each step, it checks if there is a transition out of state l for the given character x. If it exists, no new transition is made. If not, the transition to the current state is made[1]. This way of building the factor oracle by adding external transitions between each position in p is very memory efficient[1], but this also makes it recognize more than the factor of p, as seen in Figure 2.9. The factor automaton, a deterministic automaton able to recognize factors of the pattern[27], only recognizes factors of the pattern. The factor oracle also recognizes *aba*, which is not a factor of the pattern *abbbaab*.

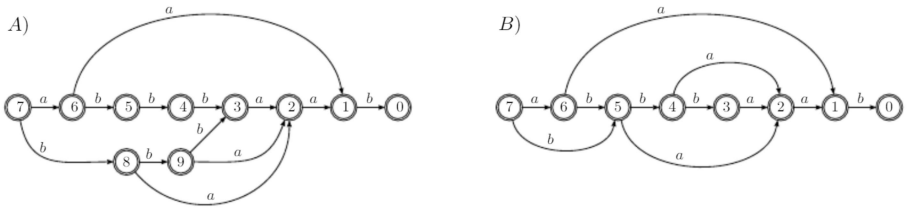


Figure 2.9: The factor automaton (A) and the factor oracle (B), from [27].

The factor oracle can be used in string matching to find the occurrences of p in T, as it is done in the Backwards Oracle Matching algorithm[1].

2.7.2 The Backward Oracle Matching Algorithm

The Backward Oracle Matching algorithm (BOM) is a deterministic acyclic automaton[1]. It takes as input the pattern p of length m and builds the trie, or factor oracle, with $m+1$ states. The trie is built from the reversed pattern (see Figure 2.10). For each transition, the algorithm only cares about what state it is in and what character the next input character is. This makes the algorithm recognize more than the pattern p and factors of p. As an example, in Figure 2.10, if we are in the state 3, the character read before was i. The Factor Oracle has two transition functions out of state 3. If t comes, it goes to the state 4. If u comes, it goes to the state 7. Based on this, the algorithm recognizes, in addition to the pattern *p, qui, quing, quiting* and so on. This is called *weak factor recognition* and is done to gain simplicity and save memory[29]. The fact that it will recognize more than the pattern means that it will also read more characters before reaching an empty state, but efficiency lost by reading more characters is recovered by doing fewer shifts[1].

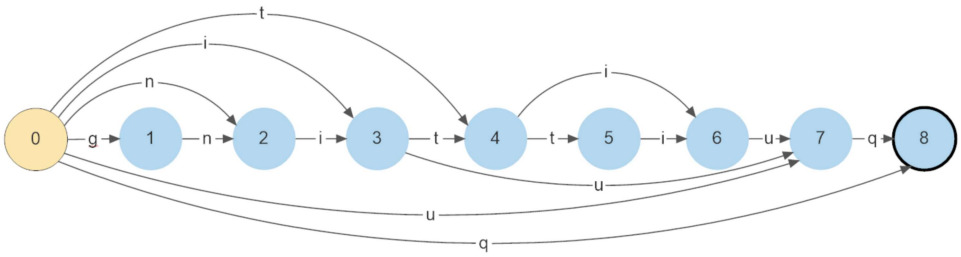


Figure 2.10: Factor Oracle of the reverse pattern *quitting*.

The BOM search algorithm is a so-called skip algorithm. This means that it is capable of skipping larger amount of the text where it knows it will not find an occurrence of the pattern. When the algorithm searches through the text, a window of size m is slid along the text while the search is done backwards in the window. If it comes to a character that has no valid transitions out of the current state, it fails and moves the window to after that character, see Figure 2.11. The basic idea of this is that after reading a text u, and then failing at the character σ, it is sufficient to know that σu is not a factor of p to move the window to after the character σ[1]. As seen in figure 2.11, it skips reading the rest of the window at once a mismatch occures and the pattern cannot be found in the window. This makes the BOM algorithm extremely fast for longer patterns[1].

2.7.3 The Set Backward Oracle Matching Algorithm

The algorithm takes as input a set of strings $P = p_1, p_2 ... p_r$. The trie is built in a very similar way to the Aho-Corasick algorithm. The difference is that all the patterns are shortened to the same length as the shortest p. This ”new” P is called P_{lmin}. The reason for this is to be able to search through the text with a fixed

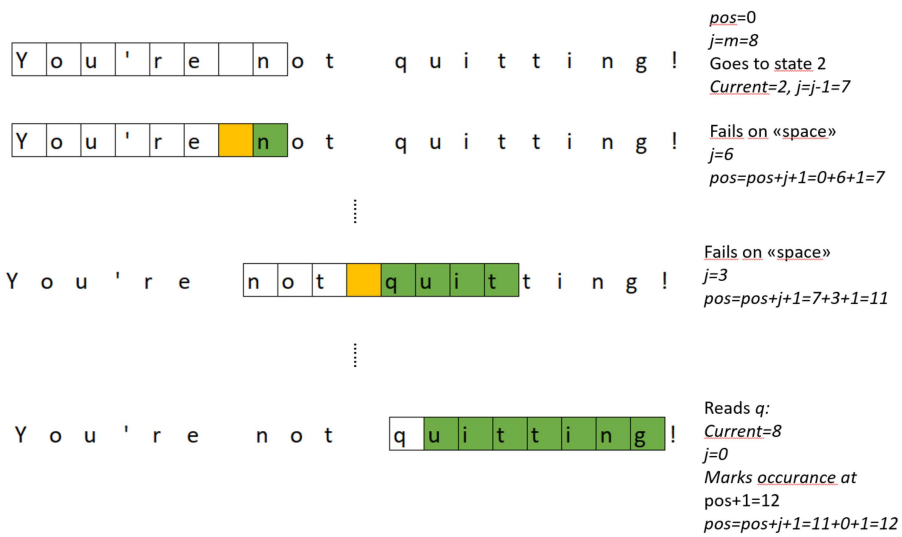


Figure 2.11: Search through the text *"You're not quiting!"*.

window size as with the BOM algorithm[30]. So to continue the example used in the Aho-Corasick algorithm, the shortest p is *snow*. This gives the following $P_{lmin} = snow, wond, outs$.

When searching through the text, the window size is that of the shortest p, which in this case is 4. The window slides through the text, searching from the end of the window towards the beginning. If a match occurs, the text is compared with the strings that may match at this position.

The SBOM algorithm has a worst-case complexity of $O(\ n\ x\ |P|\ )$, but is sub-linear on average[1]. This means that the algorithm in some situations will work slower and because of this it is theoretically possible to attack the algorithm with special crafted packets.

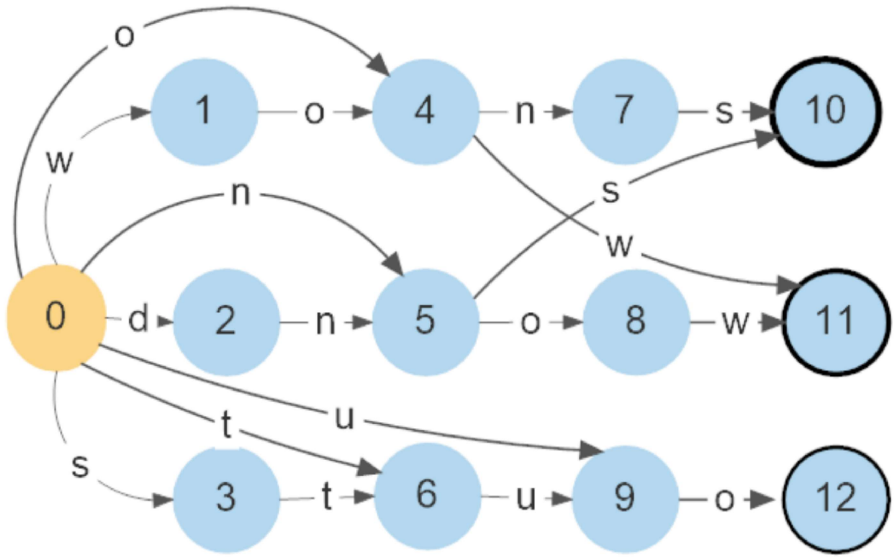


Figure 2.12: The Factor Oracle for the reversed strings $P_{lmin} = snow, wond, outs$.

Figure 2.13: Search through the text *It's snowing outside, I want to make a snowball.*

2.8 Related work

To search for - and find - patterns in a search text is the main task for a Misuse based IDS. Several algorithms with different variations have been proposed[1][31][32][33], yet the field is still a popular subject of study and research. This section looks into what have been proposed in previous research considering the SBOM algorithm, algorithmic attacks and IDS.

2.8.1 SBOM search algorithm

The *factor oracle* was first mentioned in 1999 by Allauzen et. al[30]. Algorithms based on the factor oracle turned out to have very good average time complexity and used far less memory than earlier solutions based on the implementation of the Directed Acyclic Word Graph (DAWG)[30]. In the same paper, they propose the Backwards Oracle Matching (BOM) algorithm and also how it is possible to make this algorithm linear in the worst case by using the Knuth-Morris-Pratt (KMP) search algorithm to make forward reading of some characters in the text[30]. This version of the BOM algorithm was called Turbo-BOM and was in the tests performed in [30] the only algorithm that could be used in real-time[30].

In 2002, the SBOM algorithm was proposed by Navarro and Raffinot[1]. The algorithm proved to be very fast when the patterns were long, and the fastest algorithm to handle many patterns (over 1000)[1].

In 2009, Faro et al. presented two variants of the BOM algorithm, making it more flexible than the original BOM algorithm[27]. They also showed in their tests that one of the suggested variants, the Extended-BOM algorithm outperformed bit-parallel algorithms when the alphabet size was of medium length and the search patterns were short[27].

Jianlong et. al proposed a new version of the SBOM algorithm in [34]. This version looked at what partial string of each pattern gave the minimum time cost when searching for the partial string. This method proved to be 5-20% more efficient than other alternative methods used by other algorithms.

To make the SBOM algorithm more flexible regarding dropping and adding patterns, Zhou suggested an improved SBOM algorithm[35]. This was done by modifying the data structure of the algorithm and showed that the Modified SBOM has greater adaptability and on average is faster than the Aho-Corasick algorithm[35].

In [36], Liu showed that the SBOM algorithm is much faster than the Aho-Corasick algorithm, and also uses less memory. These tests only considered speed in an average time complexity and memory consumption.

In 2006, James Kelly examined several pattern matching algorithms and considered their use in an IDS[37]. He also developed the proposed solution mentioned by Alluzan by using the Aho-Corasick algorithm to perform forward reading of some characters[37]. This version was called Multi-BOM or MBOM and had a sublinear running time on average and linear in worst case since the algorithm

never will read each character more than twice[37]. Kelly also proposed a solution, AUTO, that consisted of several algorithms. This was proposed based on the claim that "no one algorithm can best suit all situations"[37]. In this solution, the algorithm used is decided based on the shortest pattern in the pattern set. Either MBOM or AC-Full (a proposed modification of the Aho-Corasick algorithm) was used in preprocessing and searching[37].

2.8.2 Algorithmic Attacks

In 2009, Ďurian et al. published an article on the Backwards Nondeterministic DAWG Matching (BNDM) search algorithm. In [38] they tuned the algorithm with q-grams making it search through the text even faster than the original BNDM algorithm[38]. This algorithm was later implemented in Suricata. Suricata is an independent open source threat detection engine combining IDS with intrusion prevention and network monitoring[39]. This was done without further testing regarding security[40]. They later found out that it was possible to attack the algorithm using an algorithmic attack, and the algorithm was removed from Suricata[40]. This is one example of an incident where the average case complexity was the only factor when considering what algorithm to use. Related work considering algorithms are mostly concentrated around how fast the algorithm is or how much resources it consumes. Far less work have been conducted regarding if the algorithm can be misused in an Algorithmic Attack.

Corsby and Wallach showed in 2003 how Algorithmic Complexity Attacks can be used as Denial of Service attacks against the security systems in a network[24]. They showed how different common applications have weaknesses where they in worst case were able to crash these applications. They also showed how we could perform algorithmic attacks against hash tables in general and warned about how easy the worst case complexity could be triggered when knowing what algorithm is used[24]. Until then, most algorithms used were chosen based on the average case complexity, assuming worst case complexity would never or very rarely occur in practice.

One early solution to algorithmic attacks was to find a way to detect packets trying to exploit the worst case complexity and drop these packets at an early state. This was done by Khan and Traore[41]. The packets could either be dropped during execution, as *delayed drop*, or before execution, called *early drop*. Different factors could be used to make the decision whether to drop the packet or not, such as the length and size of the packet, a pre-scan of the packet or execution time during the scan of the packet[41].

Afek et. al proposed in 2016 a system that would switch between two different algorithms based on the incoming packet[22]. If the packet was considered normal, it would use the Full Matrix Aho-Corasick algorithm, but if the incoming packet was considered *heavy*, which was a special crafted packet made to trigger the worst case complexity, the system would use the Compressed Aho-Corasick al-

gorithm[22]. This system shifts between processor cores so that heavy and normal packets can be processed simultaneously. This showed to have several advantages, as improving the overall throughput and being able to shift amount of resources allocated to each process.

In 2013, Yu Zhang et. al. published a paper on how to use algorithmic complexity attacks on the SBOM algorithm and WuManber algorithm[42]. Their tests show that these attacks are effective with special crafted packets containing the patterns, but also special crafted packets when only parts of the pattern is known.

2.8.3 IDS

Throughout the years, a considerable amount of research has also been conducted to the IDS, both within signature based IDS, anomaly based IDS and hybrid IDS which is a combination of the two[43]. The main focuses regarding this research has been signature generation[44][45], construction of anomaly detection model[46][47] and how to combine signatures and anomalies to be able to detect more intrusions[48][49].

Chapter 3

Methodology

Research is a systematic process of collecting, analyzing, and interpreting information. It is performed to increase the understanding within an area of interest[50]. The choice of research approach and methods are essential to obtain the most reliable and correct results. This chapter looks into how the research is conducted to answer the research question and consequently either accept or reject the main hypothesis. The choice of research approach, design, and methods are explained, as well as how the data are analyzed. In the end, different problems and limitations to this thesis are discussed.

3.1 Research Approach

The general research approach used in this thesis is a Deductive Research Approach[51][52]. What recognizes the Deductive Research Approach is the quantitative research design, described in the next section, and the ”top-down” reasoning, shown in Figure 3.1.

Figure 3.1: Deductive research approach

The researcher first finds and learns relevant theory. Based on this, the researcher develops a hypothesis and research questions and tests these in several experiments, surveys, etc. Based on these results, the hypothesis is either confirmed or rejected.

3.2 Research Design

Based on the fact that this thesis compares two algorithms against each other, the thesis mostly makes use of quantitative research. The comparison is mostly carried out by comparing numeric data in the form of an experiment, which is a quantitative research design. A quantitative research design is defined as a systematic investigation of a phenomena[51]. Variables are observed through the research and either controlled or not controlled, depending on the research method. The thesis also has features of qualitative research design carried out in Chapter 2 and used to compare the experimental results in Chapter 6.

3.3 Research Methods

With a quantitative research design, it is chosen to use a quantitative hypothesis, or closed-ended questions[51] with additional three research questions as seen in section 1.3. To test this hypothesis, quantitative experiments were performed as a research method. In a quantitative experiment, the theory is tested by examining the relationship among variables[51]. In this thesis, the suitability for the SBOM algorithm in an IDS is tested by implementing the algorithm in code and comparing the results from different datasets against the commonly used Aho-Corasick algorithm. The results are measured in processing time, and the algorithms are tested by manipulating the different independent variables:

- Pattern length
- Length of input text
- Numbers of patterns
- Content of input text (complete pattern match and algorithmic complexity attack)

Several tests are implemented where one variable is manipulated at the time and the others controlled. In that way, we can see how each variable affects the performance of the algorithm itself and also how it affects the performance compared to the other search algorithm. This is called experimental research or a true experiment[53]. The experiment was also performed with a pretest-posttest design[53], which means that tests were performed before an independent variable was manipulated to be able to see what effect the independent variable had on the algorithm. These tests were also used as a pilot study to confirm that the implemented algorithms worked as intended.

3.4 Literature Study

The literature describing the BOM and SBOM algorithm was provided by the thesis' supervisor. Other scientific papers were found in scientific paper collections. In addition, some other sources, such as Information Security reports and lectures, were used.

3.5 Methods of Data Analysis

The results from the experiment had a Ratio Scale of measurement[50]. This means that the scale used in this experiment, the processing time, has an absolute zero point and has equal measurement units. These characteristics make it easy to analyze the data mathematically. The results are processed and visualized in diagrams.

3.6 Problems and Limitations

The author of this thesis had the option of implementing the algorithms in an IDS, but building a whole IDS is beyond the scope of the research. Thus, it was decided to implement only the search algorithms in C++ as a state-of-the-art experiment.

The author had to do a thorough pilot study to verify that the algorithms worked as intended to avoid biases influencing the results. The pilot study was also used to find a systematic way of measuring the processing time to be able to discover deviations or other scenarios that influenced the processing time. In that way, the author would be less doubtful if the results were not as expected.

Chapter 4

Proposed modification of SBOM

This chapter describes the proposed solution to make the SBOM algorithm a preferable choice in an IDS and the work to implement the solution for testing.

The related work that has been implemented on the SBOM algorithm shows that it is in most cases faster than the Aho-Corasick algorithm and uses less memory [34][36]. Many research papers have also been published on how to make the BOM algorithm or SBOM algorithm even faster[27][35]. But, as the example with the BNDM algorithm implemented in Suricata with q-grams[38][40] in Section 2.8, speed is not the only factor to consider when choosing which algorithm to implement in an IDS. Research has also shown that the SBOM algorithm is vulnerable to algorithmic complexity attacks[42].

The SBOM algorithm is vulnerable to algorithmic attacks because of its worst-case complexity. Ideally, the algorithm used in an IDS should have the same average-case complexity and worst-case complexity. This is difficult to achieve, though there are several ways to improve the worst-case complexity of the SBOM algorithm:

1. Use more resources on a general basis. This is not cost-effective and will most likely only push the problem into the future. The attacker has to make more effort, but it is still very feasible to attack the IDS.
2. Use different algorithms. This solution builds on the system proposed by Afek et al. in 2016, where two different algorithms work in parallel[22]. Which algorithm to use on the text is based on an evaluation of the text. In this solution, more resources are required to be able to implement the system, but no additional changes have to be done to the algorithms in the system.
3. Find a combination of algorithms. This is the solution proposed by Kelly in [37]. This solution can be realized in several ways to get the most out of the algorithms used. It is implemented in the software, and therefore it can be easily changed and adapted to each system.

This thesis' proposed solution combines the SBOM algorithm and the Aho-

Corasick algorithm. The proposed solution combines the two algorithms in a way to gain a faster average-case complexity, but still the linear worst-case complexity to the Aho-Corasick algorithm. The goal of the proposed solution is to get as much advantage of the SBOM algorithm speed as possible.

The new algorithm builds the automaton for Aho-Corasick and the factor oracle for the SBOM algorithm. It always starts to search through the text using SBOM but switches over to Aho-Corasick if one or more different conditions are met. These conditions can be set individually for each system to be optimized for each system's needs, but it is necessary to examine different thresholds.

The disadvantage of this solution is that it most likely demands more memory than the solutions available today. The gain is that we can predict the maximum consumption because of the linear worst-case complexity, making it as predictable as the Aho-Corasick algorithm. Moreover, it is almost as fast as the SBOM algorithm in average-case complexity.

4.1 Implementation

The first step towards testing this solution is implementing the SBOM algorithm and the Aho-Corasick algorithm in code. To simplify this process, an already existing implementation of the Aho-Corasick algorithm in C++ is taken from [54]. To be able to run the tests on the algorithm, some modifications were implemented:

1. The algorithm had to take a pattern file as input
2. The algorithm had to take a text file to search through as input
3. The algorithm had to be able to process, as a minimum, all characters in the standard ASCII table

With a functional Aho-Corasick algorithm, the SBOM algorithm has to be implemented in C++ with the same requirements (1-3) as with the Aho-Corasick algorithm. To simplify the implementation, the factor oracle is implemented as a $A \times (m+1)$ table, as recommended in [1], where A is the alphabet size. Each letter is also converted into an ASCII value and then converted down by subtracting 32 so that "space", with ASCII value 32, obtains the new value 0 and the first column in the table. In Table 4.1, the Factor Oracle shown in Figure 2.12 is implemented. For simplicity, only the rows and columns used are shown. The complete table consists of 95 rows and 12 columns (10, 11, and 12 are terminal states). Each column represents a state, and for each possible input, e.g. [0,79], the number of the next state is written, which in this case is state 4. Each terminal state is saved in an array F for each pattern in the set.

When searching through the text and a terminal state is reached, all patterns with the same terminal states are checked with the search text. If a match is found, an action is initiated.

	0	1	2	3	4	5	6	7	8	9
d (100-32=68)	2									
n (110-32=78)	5		5		7					
o (111-32=79)	4	4				8				12
s (115-32=83)	3					10		10		
t (116-32=84)	6			6						
u (117-32=85)	9						9			
w (119-32=87)	1				11				11	

Table 4.1: The factor oracle represented in a table for the reversed strings $P_{lmin} = snow, wond, outs.$. This is the same factor oracle as illustrated in 2.12.

4.2 The SBOM-AC combination

This solution starts searching through each packet with the SBOM algorithm. If a condition is met, the packet is considered to be an attempt of an algorithmic attack, and the IDS continues the search using the Aho-Corasick algorithm.

Two different conditions were considered when constructing this combination:

1. The IP packet has a certain amount of ”almost matches”.
2. The IP packet has a certain amount of terminal states that are not a match.
3. A combination of 1 and 2.

With option 1, the search does not have to exceed the P_{lmin} to begin use the Aho-Corasick algorithm, but it switches algorithm if it reaches a certain point in the window a given set of times. Tests performed on the SBOM algorithm and the Aho-Corasick algorithm shows that the pattern match has to be about 150 characters to make the SBOM algorithm perform worse than the Aho-Corasick algorithm shown in Figure 4.1, but these tests do not account for the P_{lmin} length. With a shorter P_{lmin}, it would in theory be possible to make the SBOM algorithm work slower with a shorter part of a pattern.

Option 2 counts each time the pattern match exceeds the P_{lmin} and the search reaches a terminal state. Then it has to search for the whole pattern and if no match is found a certain amount of times, the IDS continues the search with the Aho-Corasick algorithm.

Depending on the pattern set, and the results from own tests, it might be useful to use both solutions 1 and 2. This combination starts using the Aho-Corasick algorithm if thresholds in option 1 or option 2 are reached.

4.3 Construction of an algorithmic attack

As mentioned in [42], the attacker does not need the whole pattern to perform an algorithmic attack against an IDS using the SBOM algorithm. The advantage of all skip algorithms is that it is not needed to scan the whole search text to

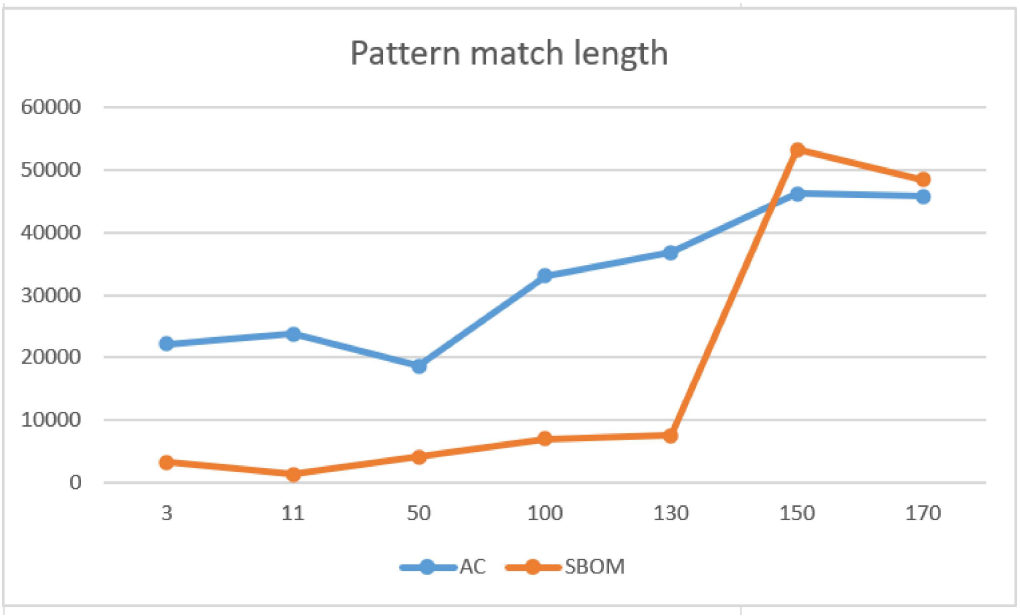


Figure 4.1: Search time for same pattern set when search text contains different lengths of the pattern. Time is measured in μs.

determine if the pattern is present or not, they can skip larger parts of the pattern by determining that the pattern cannot be found in the search window. So by making the algorithm skip as little of the text as possible, it will not be able to use the advantage.

The SBOM algorithm also has a disadvantage that makes it easier to attack. If the algorithm reaches a terminal state after scanning the whole window, it will check the search text for each pattern that has this state as terminal. As an example, we use the pattern set *annual, annually, year*. $P_{lmin} = 4 => annu, year.$ is what will be searched for. If the search text only contains ”annu”, the SBOM algorithm will scan the whole window, and for each time check the text for both *annual* and *annually*:

$$[annu]annuannuannuannuannuannu$$

In this case, it checks if the search text *annuan*=*annual* and if the search text *annuannu*=*annually* before the window is moved and the same check will happen again.

$$annu[annu]annuannuannuannuannu$$

By doing this, each search window is read at least twice and, because of this, it is the most effective way to perform an algorithmic attack.

4.4 Threshold definition

An IP packet can have various lengths. The minimum length is the header size which is 20 Bytes for IPv4 and 40 Bytes for IPv6. In addition, both IPv4 and IPv6 have a payload field that can be up to 65 535 Bytes long[55][56]. To be able to perform an algorithmic complexity attack, each packet inspected should have a time significantly worse than the average time complexity for a data packet of the same size. The time should also be considerably worse than the time searching through the same packet with the Aho-Corasick algorithm. When the worst time complexity is defined for both Aho-Corasick and SBOM, it is possible to tune the SBOM-AC combination to find the thresholds for when to use the Aho-Corasick algorithm and when to use the SBOM algorithm.

The threshold value can vary from options 1 and 2 and from system to system, depending on the pattern set. Because of this, the thresholds found in these tests are not absolute.

Chapter 5

Experimental work

In this chapter, the results from the tests performed on the different algorithms are presented. The tests were performed to help answering the research questions formed in Chapter 1. The chapter also describes how the tests were performed in order to enable reproduction of the tests.

5.1 Testing algorithms

How to test Intrusion Detection Systems (IDS) is a widely discussed topic and there is no universal methodology to test IDS[25]. In this thesis, the goal is to find a way to make the SBOM algorithm more resistant to attacks. This is done by comparing performance measured in time, which is one of the measurable IDS characteristics described in [57]. In addition to time, True Positives (TP), False Positives (FP) and False Negatives (FN) are made account for.

To test these algorithms, several datasets are used. Each dataset is made out of an original dataset containing URLs known to spread malware, phish, spam and defacement. These datasets were created by the Canadian Institute for Cybersecurity[58].

5.1.1 Tests on algorithms

To test the algorithms, several different tests were performed:

1. Run tests on the dataset with no matching patterns.
   These tests are performed to visualize the average case complexity and the difference between the SBOM algorithm and the Aho-Corasick algorithm in performance. In addition, it is interesting to see how the SBOM-AC combination performs compared to the SBOM algorithm.
2. Run tests on special crafted datasets.
   We already know from Section 2.8 that the SBOM algorithm is vulnerable to algorithmic attacks and that we do not need to know the entire pattern to

perform an algorithmic attack. The aim for these tests is to try to find how much of the pattern is needed and if the length of P_{lmin} is significant.

3. Run tests on the SBOM-AC combination with special crafted datasets. The results from 2 are used to decide which conditions to use in the SBOM-AC combination. In these tests thresholds are tuned to make the combination as effective as possible. When the thresholds are decided, the tests in 1 and 2 is repeated to test the SBOM-AC combination on both general performance (average case complexity) and its resilience to algorithmic complexity attacks.

5.1.2 Test Session

The test objective is to define how the SBOM algorithm is vulnerable to algorithmic complexity attacks and if there are ways to mitigate these attacks as a software solution.

The source code for the different algorithms is appended to this thesis (Appendix A). The source code for the Aho-Corasick algorithm is retrieved from [54]. The source code for the SBOM algorithm is written by the author of this thesis, along with the SBOM-AC combination. All source code is written in C++.

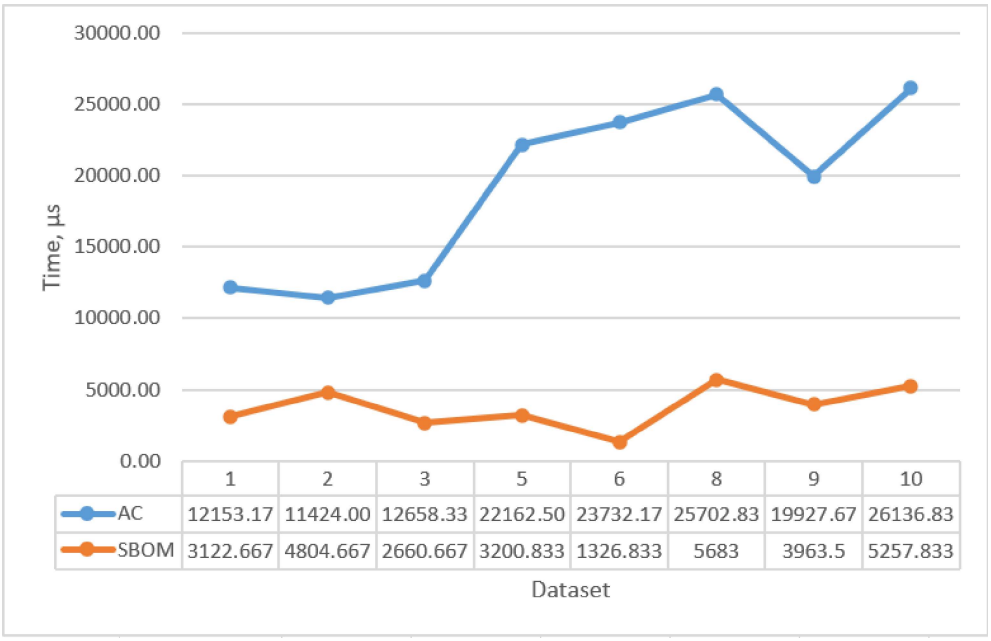
The tests were performed on a Hewlet Packard Z-book G6 i7vPro 9th Generation with 32GB RAM.

All tests were performed six times, where an average value was calculated. An overview of each dataset used and raw results are appended in the appendix (Appendix B and C).

5.2 Experimental Results

5.2.1 Performance

As seen in Figure 5.1, SBOM's performance is much more time efficient than the Aho-Corasick (AC) algorithm on an average basis. Some of these datasets had true positives, which were discovered by both algorithms. None of the algorithms gave false positives or false negatives.



	1	2	3	5	6	8	9	10
AC	12153.17	11424.00	12658.33	22162.50	23732.17	25702.83	19927.67	26136.83
SBOM	3122.667	4804.667	2660.667	3200.833	1326.833	5683	3963.5	5257.833

Figure 5.1: Performance of the two different algorithms - Aho-Corasick (abbreviated to AC) and SBOM - on different datasets, measured in time.

5.2.2 Algorithmic attack

From the results, it is clear that the SBOM algorithm is vulnerable to algorithmic attacks. As long as the pattern match exceeds the P_{lmin}, it does not matter how long the match is. The tests also showed that with shorter P_{lmin}, the attack is more effective. The test was, in addition to the two datasets shown in Figure 5.2, performed on a dataset (dataset 17) with P_{lmin}=3 and the pattern match=4. The SBOM algorithm had an average time of 1.596.608μs, wich is 86 times worse than the AC algorithm, and 65 times worse than the SBOM-AC combination.

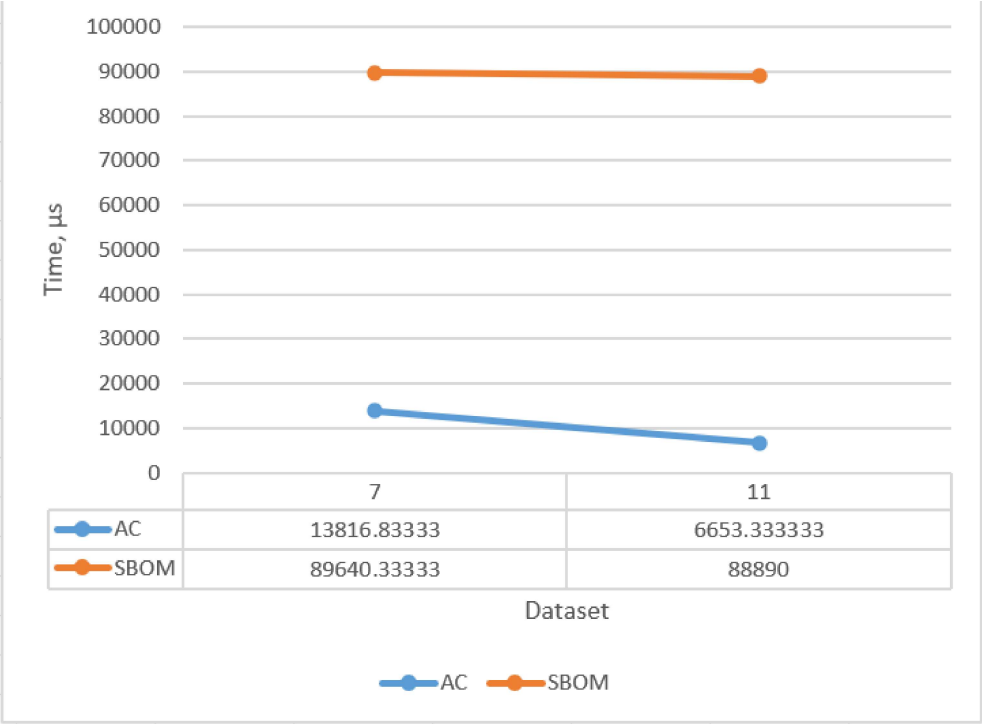


Figure 5.2: Algorithmic attack with P_{lmin} length of 112 in dataset 7 and 40 in dataset 11

Figure 5.3 also shows that it is possible to perform an algorithmic complexity attack in the size of a data packet. An attack is also effective down to a packet size of 10kB if the pattern match exceeds the P_{lmin}.

In Figure 5.4, the author tried to attack the SBOM algorithm with data packets where the pattern match did not exceed P_{lmin}. The data packet only consisted of the letter ”w” and had different size of P_{lmin}.

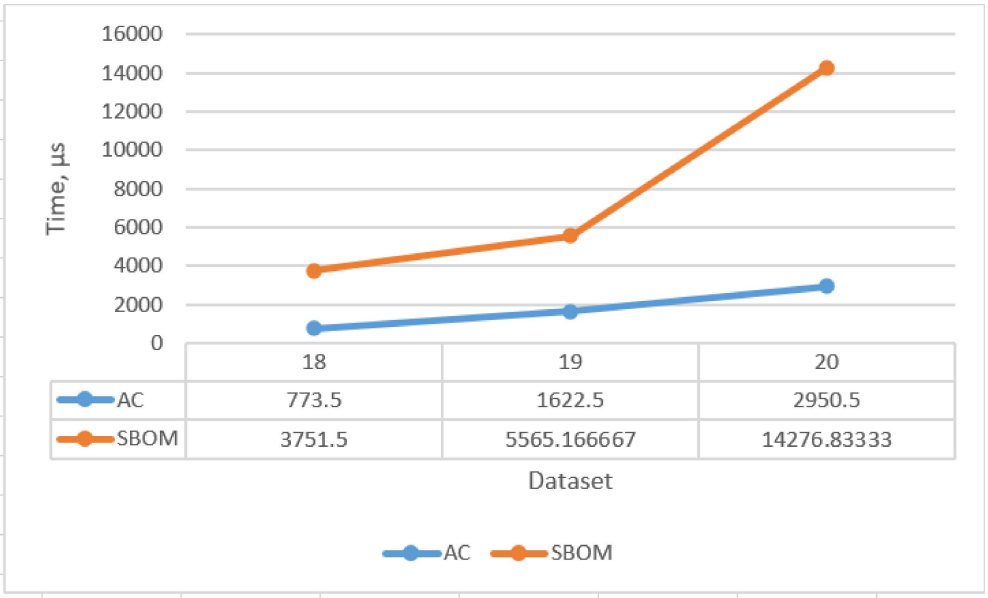


Figure 5.3: Algorithmic attack in the size of a datapacket in increasing size.

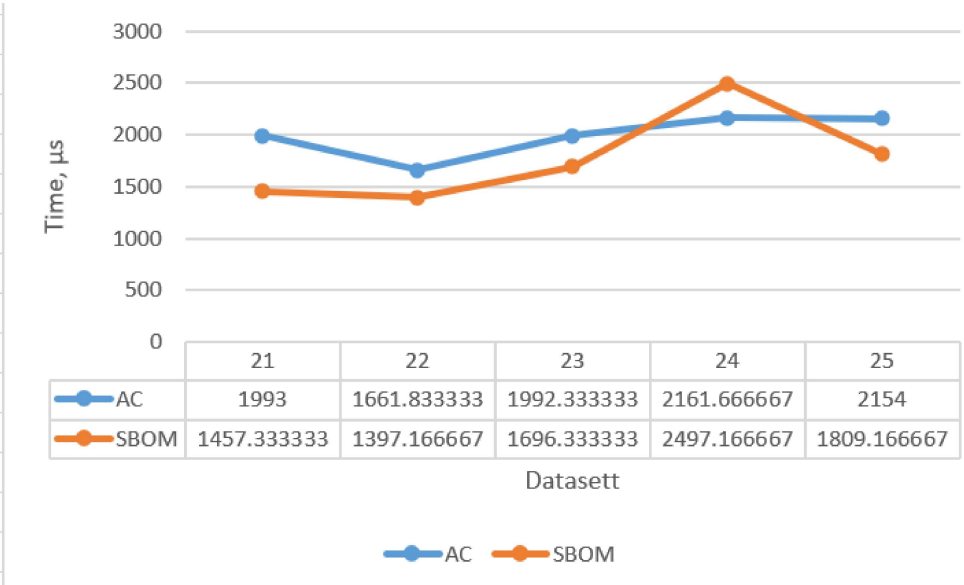


Figure 5.4: Algorithmic attack where pattern match does not exceed P_{lmin}. In these tests, P_{lmin} has been 112, 60, 20, 10, and 4 respectively.

5.2.3 Resilience to algorithmic attacks

In Figure 5.5, the results from dataset 20 were used to tune the threshold of the SBOM-AC combination. The thresholds used is of option 2 from Section 4.2, where the algorithm counts each time the P_{lmin} is exceeded, but no match is found. The thresholds used is 100, 50, 25 and 10, where 25 gave the best results in these test.

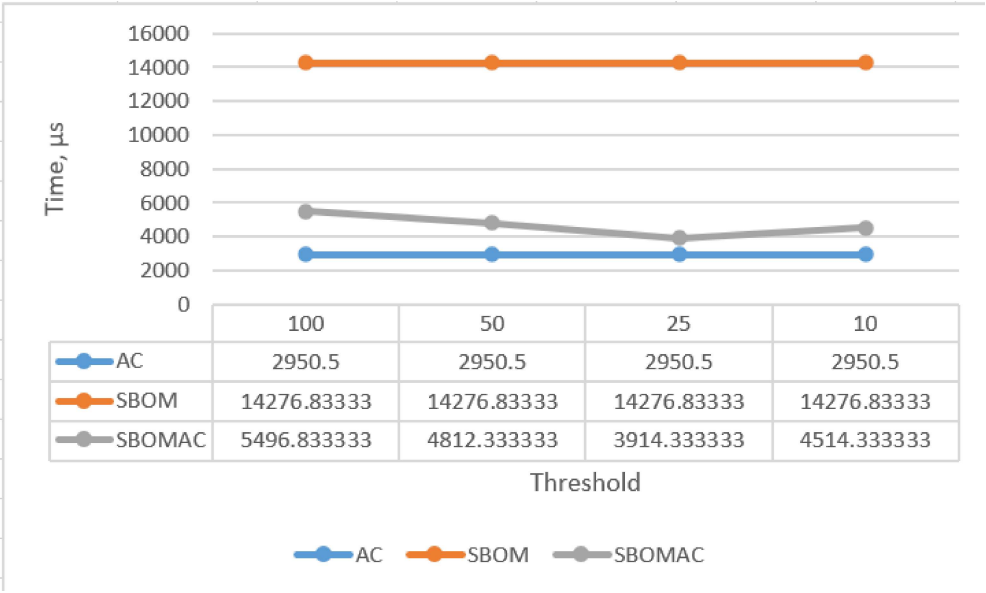


Figure 5.5: The SBOM-AC combination with different thresholds.

From the results from algorithmic complexity attacks where P_{lmin} is not exceeded, the 24th dataset is used. Each time the search window is moved and the number of characters in search window which is read is greater than 4, a variable is incremented. This variable is used as the threshold for when to use SBOM and Aho-Corasick. The thresholds used in these tests are 250, 500, 1000 and 2000. This is an example of the option 1 from Section 4.2.

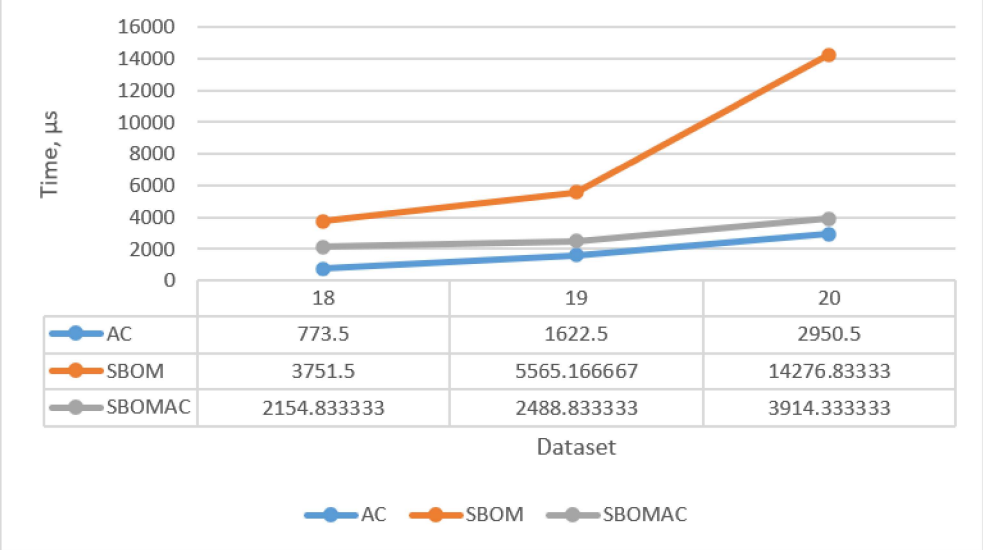


Figure 5.6: The SBOM-AC combination with threshold=25.

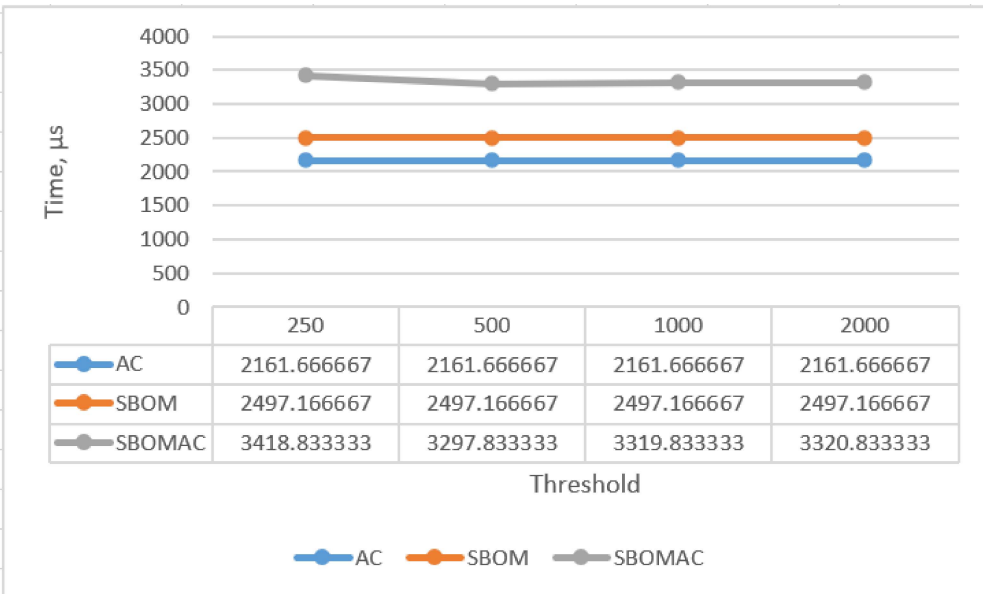


Figure 5.7: The SBOM-AC combination with different thresholds.

	21	22	23	24	25
AC	1993	1661.833333	1992.333333	2161.666667	2154
SBOM	1457.333333	1397.166667	1696.333333	2497.166667	1809.166667
SBOMAC	2261.5	2836.666667	3119.333333	3297.833333	3236.666667

Figure 5.8: The SBOM-AC combination with thresholds=500.

Chapter 6

Discussion

In this chapter, the findings and results from the previous chapter are discussed in the context of the research questions. Lastly, the chapter discusses the relevance of the thesis, the objectives of this thesis, as well as sources of errors and validity.

6.1 SBOM performance

Regarding the first research question, we know from related work that the SBOM algorithm is very fast, especially when the patterns are long or it has to handle many patterns[1], and that it uses less memory than many other algorithms[30].

Results from the performance tests visualized in Figure 5.1 shows that the SBOM algorithm on average performs 6.3 times better than the Aho-Corasick algorithm. It performs better when the P_{lmin} is long (dataset 5, 6 and 8) and is better than Aho-Corasick algorithm at handling larger pattern sets (dataset 3).

6.2 SBOM and algorithmic attacks

Since the SBOM algorithm has a worst-case complexity of $O(n \times |P|)$ it is theoretically possible to attack the algorithm in the use of an algorithmic attack. Figure 5.2 shows that these attacks can be very effective, especially with a shorter P_{lmin} and a pattern match that exceeds the P_{lmin}.

Regarding the second research question "How can we use an algorithmic attack to attack the SBOM algorithm", the author has chosen to focus on algorithmic complexity attacks and distinguish between a DDoS attack aimed at the Intrusion Detection System (IDS) and algorithmic complexity attacks. Because of this, it is not sufficient that it is possible to make the algorithm work slower. It has to be possible to make it work slower in the size of a data packet, and the running time has to be considerably worse than the Aho-Corasick algorithm. One cannot expect to be able to perform an algorithmic complexity attack in one single data packet. Still, if the algorithmic complexity attack is not efficient enough, the attack is more like a DDoS attack.

As shown in Figure 5.3, the SBOM algorithm is slower than the Aho-Corasick algorithm with data packets down to 10kB. This might not be enough to make the algorithm crash, but as the size of the data packet increases, the difference between the running time for the SBOM algorithm and the Aho-Corasick algorithm increase and is up to 4.8 times slower than the Aho-Corasick algorithm.

It is more difficult to create an algorithmic attack if the pattern match does not exceed P_{lmin}, as shown in Figure 5.4. Even though it is possible to make the algorithm work slow, it is not necessarily enough to make the algorithm crash or drop packets. As shown in Figure 5.4, the only time the SBOM algorithm had a worse running time than Aho-Corasick was with dataset 24 and a pattern match that was at 30% of P_{lmin}. Still, the algorithm was only 1.16 times slower than the Aho-Corasick algorithm. The author means that this is not sufficient to perform an algorithmic attack on the IDS.

Even though the author of this thesis has not been able to attack the SBOM algorithm without exceeding P_{lmin}, it is not necessarily impossible, but it can be more difficult than constructing an algorithmic attack where the pattern match exceeds the P_{lmin}. Since the patterns in an IDS can be very short, only a little knowledge of patterns used is sufficient to create this type of attack, and as the results clearly showed, these attacks were increasingly efficient with a decreasing P_{lmin}.

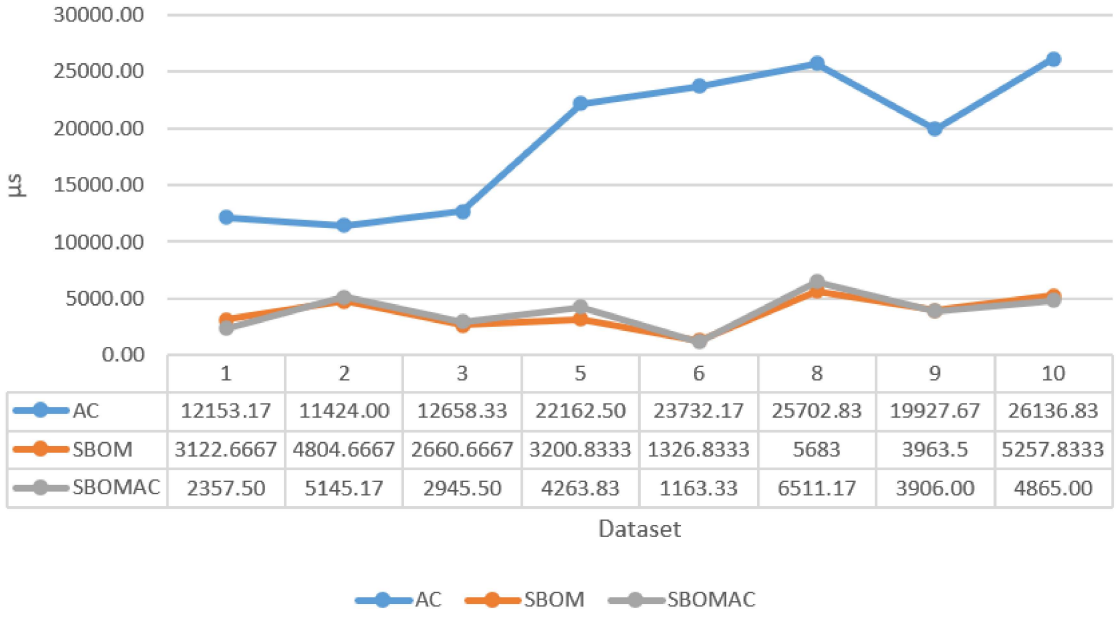
6.3 The SBOM-AC combination

Testing for the thresholds for option 2 in Section 4.2 showed that using the threshold of 25 gave the best results considering the running time of the algorithms. When testing the SBOM-AC combination on the algorithmic complexity attacks, we see that it has a running time considerably better than the SBOM algorithm. On average, it has a running time that is 1.88 times as fast as the Aho-Corasick algorithm, and the running time is faster compared with Aho-Corasick with an increasing data packet size.

Since the author of this thesis considers it possible to attack the SBOM algorithm without exceeding the P_{lmin}, it is still relevant to use option 3 from Section 4.2. The results in Figure 5.8 did not look as good as the results from Figure 5.6. Still, the SBOM-AC has a running time that is on average 1.5 times the running time for Aho-Corasick, which is very good in case of a successful algorithmic complexity attack.

In average-case complexity, the SBOM-AC combination has a running time as fast as the ordinary SBOM algorithm, see Figure 6.1. On average, it is 1.02 times slower than the SBOM algorithm. It is never faster than the Aho-Corasick algorithm in worst-case complexity since it transits from an algorithm working slower due to an algorithmic attack to the Aho-Corasick algorithm. The switch between algorithms also implements several extra operations in the code, which naturally slows the algorithm down. The goal is to get the combination as close to the worst-case complexity of the Aho-Corasick algorithm as possible. In this

case, the gain in an algorithmic attack is already satisfactory, though, with an implementation in an IDS with the tuning of the values based on the pattern set used, we expect a more significant gain. The results achieved in this thesis show that this combination have a linear time complexity in the worst case, which also makes it very difficult to attack through an algorithmic complexity attack.



	1	2	3	5	6	8	9	10
AC	12153.17	11424.00	12658.33	22162.50	23732.17	25702.83	19927.67	26136.83
SBOM	3122.6667	4804.6667	2660.6667	3200.8333	1326.8333	5683	3963.5	5257.8333
SBOMAC	2357.50	5145.17	2945.50	4263.83	1163.33	6511.17	3906.00	4865.00

Figure 6.1: The SBOM-AC combinations performance compared to SBOM and Aho-Corasick

6.3.1 Other possible solutions

As mentioned in Section 2.8, several different solutions have been proposed to avoid possible algorithmic attacks on Intrusion Detection Systems. To compare this solution against these is a possible task for future work, but this section will discuss the different solutions compared to this combination.

The MBOM solution proposed by Alluzan himself and implemented by Kelly in 2006 is also a combination of the SBOM algorithm and the Aho-Corasick algorithm. In this version, the search window is divided into two parts with a critical position. The search starts by scanning from right to left with the oracle. If a mismatch occurs, the critical position is moved to the character to the right of the mismatch and the search continues from critical position with Aho-Corasick. If the critical position is reached, the search continues with Aho-Corasick from the critical position. In this combination, the Aho-Corasick search continues at least to the end of the window, and if it exceeds the window, it continues until there is no prefix match of any pattern or a complete pattern match is found. The window

is shifted to the right, and the search starts again with SBOM[37].

The strength of this combination is that it avoids the extra check the SBOM has to take when a terminal state is reached at the beginning of the window. As the results showed, this is the most effective way to perform an algorithmic complexity attack against the SBOM algorithm. It also has a very good linear worst time complexity. Its weakness is that this solution does not get the best advantage of the SBOM algorithm speed as this thesis proposed solution. For each window shift, it scans at least the amount of characters from the critical position and to the end of the search window twice, and in the worst-case, it scans the entire search text twice, once with SBOM and once with Aho-Corasick. In the solution proposed in this thesis, the average-case complexity is the same as the SBOM, which is very good. The worst-case complexity is not as fast as the MBOM solution, but from the tests performed in this thesis, it is approximately linear. Considering memory consumption, these two solutions should be the same since they combine the same two algorithms, though this has not been tested in this thesis. However, the memory consumption measured in [37] showed that if SBOM is implemented with hash tables, the memory consumption is just slightly higher than the Aho-Corasick algorithm alone[37].

Another possible solution is the solution by Afek et al.[22]. This is a hardware solution, a multi-core platform, where each core runs one or more hardware threads, some handling normal packets and other handling ”heavy” packets simultaneously[22]. If a packet is recognized as ”heavy” and handled by a thread for normal packets, it is switched over to a thread handling heavy packets. The disadvantage of this is somehow the same as with MBOM. It does not get the full advantage of the speed of the average-case complexity algorithm. When no ”heavy” packets are handled, the threads designed to handle these packets process normal packets instead. A greater gain would have been achieved if the threads could dynamically shift between the two algorithms.

6.4 Achievement of the objective

The objective of this thesis was to determine if the SBOM algorithm was suitable for an Intrusion Detection System, as is, or in a combination of search algorithms. Furthermore, the task was to implement the algorithm in code and compare it with the commonly used Aho-Corasick algorithm. The author has also proposed a combination of the SBOM algorithm and the Aho-Corasick algorithm as a more suitable alternative to use in the IDS. As this research can be used to determine what algorithm to use in an IDS, the research questions stated in Chapter 1 are answered and the objective of this thesis has been achieved.

6.5 Sources of errors and validity

The raw results from running the tests on the algorithms showed that the processing time varies from each run. Because of this, it was determined to run each test 6 times and use the average value of these 6 results. If one result had a difference from the other with more than 100%, it was considered a deviation, and the results were discarded. It was then considered if a restart of the system was needed or some processes stopped before the whole test was carried out again.

The tests were performed on a standard office computer. The processing time can, because of this, be affected by other processes running on this computer. The tests have been carried out in the same environment with the laptop on power supply and only necessary processes running.

The tests of the algorithm were not performed on an IDS. The author still considers these results to be comparable since all the tests were performed on the same system and in the same environment.

Chapter 7

Conclusion

Speed has been the main focus for several years regarding creating new algorithms or modifying existing algorithms[31]. Speed is an essential factor in Intrusion Detection Systems (IDS), and is - as the amount of data increases - even more important. When the algorithm is used in an IDS, it is equally important that the algorithm is resilient to algorithmic attacks.

Regarding the first research question, the results from this thesis show that the SBOM algorithm, on average, performs 6 times faster than the Aho-Corasick algorithm considering speed. It also performs slightly better when the pattern is long and with larger datasets. Tests have also shown how it is possible to use an algorithmic complexity attack to attack an IDS running the SBOM algorithm. One of the algorithmic complexity attacks also proved to be very effective, which concludes the second research question.

Because of these results, the conclusion is that the SBOM algorithm is not suitable for use in an IDS. The third research question deals with this problem, and this thesis proposed solution is a combination of SBOM and Aho-Corasick. The experimental results show that the combination does not affect the speed of the algorithm on average as it performs as well as the SBOM algorithm. At the same time, it has a worst time complexity that is linear, which makes it far more resilient to attacks.

The SBOM-AC combination is a software implementation, which means it can be implemented in most existing systems. The implementation can adapt to the system and how it chooses which algorithm to use can be changed and tuned to each system. It is also the only solution - as for the author's knowledge - that preserves the speed of the SBOM algorithm and still has a linear worst-case complexity.

Based on this thesis results, the main hypothesis of this thesis can be partially confirmed. The SBOM algorithm is vulnerable to algorithmic attacks and cannot be used in an IDS as it is. Still, it has shown to be very effective if implemented in a combination with the Aho-Corasick algorithm, which makes it very difficult to attack using an algorithmic complexity attack.

7.1 Future work

To continue the work of this thesis, the SBOM-AC combination should be implemented in an IDS for further testing. Specifically, the tests should focus on what resources the combination consumes, especially regarding memory consumption.

This proposed solution is a proof of concept. Further development and improvement of the code are necessary before a complete implementation. Other ways of choosing the search algorithm can be considered and also combining this solution with other proposed solutions, like the dual-core in [22] and the AUTO proposed in [37].

Bibliography

[1] G. Navarro and M. Raffinot, *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge university press, 2002.

[2] S. Petrovic, *Topic 2.1 misuse detection 1- search*, https:\ntnu.blackboard.com, Online; accessed 01-December-2020, IMT4204 Intrusion Detection in Physical, Virtual Networks, Norwegien University of Science and Technology, 2019.

[3] A. Greenberg, *The Untold Story of NotPetya, the Most Devastating Cyberattack in History*, https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/, [Online; accessed 28-December-2020], 2018.

[4] Microsoft Defender Security Research Team, *New Ransomeware, Old Techniques: Petya adds worm-capabilities*, https://www.microsoft.com/security/blog/2017/06/27/new-ransomware-old-techniques-petya-adds-worm-capabilities/, [Online; accessed 28-December-2020], 2017.

[5] LogRythm Labs, *Petya/NotPetya Poses a Risk to Even Patched Systems*, https://logrhythm.com/blog/detecting-petya-notpetya-ransomware/, [Online; accessed 28-December-2020], 2017.

[6] M. Landesman, *A Brief History of Malware*, https://www.lifewire.com/brief-history-of-malware-153616, [Online; accessed 28-December-2020], 2019.

[7] J. Martindale, *From Pranks to nuclear sabotage, this is the history of malware*, https://www.digitaltrends.com/computing/history-of-malware/, [Online; accessed 28-December-2020], 2018.

[8] D. Kushner, *The Real Story of Stuxnet*, https://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet, [Online; accessed 16-January-2021], 2013.

[9] M. Clayton, *Ukraine election narrowly avoided 'wanton destruction' from hackers*, https://www.csmonitor.com/World/Passcode/2014/0617/Ukraine-election-narrowly-avoided-wanton-destruction-from-hackers, [Online; accessed 16-January-2021], 2014.

[10] Næringslivets Sikkerhetsråd, *MØRKETALLSUNDERSØKELSEN 2016*, https://www.nsr-org.no/uploads/documents/Publikasjoner/morketallsundersokelsen_2016.pdf, [Online; accessed 16-January-2021], 2016.

[11] Næringslivets Sikkerhetsråd, *MØRKETALLSUNDERSØKELSEN 2020*, https://www.nsr-org.no/uploads/documents/Publikasjoner/Morketalls-2020-web.pdf, [Online; accessed 16-January-2021], 2016.

[12] B. Z. Lavi, *November 2020 Malware Trend Reports*, https://www.varonis.com/blog/november-2020-malware-trends-report/, [Online; accessed 28-December-2020], 2020.

[13] N. Milošević, 'History of malware', *arXiv preprint arXiv:1302.5392*, 2013.

[14] D. Braue, *Global Ransomware Damage Costs Predicted To Exceed $265 Billion By 2031*, https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-250-billion-usd-by-2031/, [Online; accessed 04-November-2021], 2021.

[15] S. Morgan, *Cybercrime To Cost The World $10.5 Trillion Annually By 2025*, https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/, [Online; accessed 04-November-2021], 2020.

[16] G DATA Blog, *Malware Trends 2017*, https://www.gdatasoftware.com/blog/2017/04/29666-malware-trends-2017, [Online; accessed 28-December-2020], 2017.

[17] R. Kemmerer and G. Vigna, *Intrusion Detection: A Brief History and Overview*, https://www.computer.org/csdl/magazine/co/2002/04/r4s27/13rRUIJcWgL, [Online; accessed 04-January-2021], 2002.

[18] G. Bruneau, 'The history and evolution of intrusion detection', *SANS Institute*, vol. 1, 2001.

[19] R. Di Pietro and L. V. Mancini, *Intrusion detection systems*. Springer Science & Business Media, 2008, vol. 38.

[20] S. team, *SNORT FAQ*, https://www.snort.org/faq, [Online; accessed 14-September-2021], 2021.

[21] S. Petrovic, *Topic 2.3 misuse detection 3 - zeek (bro), suricata*, https:\ntnu.blackboard.com, Online; accessed 18-Mars-2021, IMT4204 Intrusion Detection in Physical, Virtual Networks, Norwegien University of Science and Technology, 2019.

[22] Y. Afek, A. Bremler-Barr, Y. Harchol, D. Hay and Y. Koral, 'Making dpi engines resilient to algorithmic complexity attacks', *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3262–3275, 2016.

[23] F. Y. Rashid, *ADDoSA*, https://www.eweek.com/cloud/sony-data-breach-was-camouflaged-by-anonymous-ddos-attack/, [Online; accessed 16-March-2021], 2011.

[24] S. A. Crosby and D. S. Wallach, 'Denial of service via algorithmic complexity attacks.', in *USENIX Security Symposium*, 2003, pp. 29–44.

[25] S. Petrovic, *Topic 4 testing ids*, `https:\ntnu.blackboard.com`, Online; accessed 12-July-2021, IMT4204 Intrusion Detection in Physical, Virtual Networks, Norwegien University of Science and Technology, 2019.

[26] X. Wang and D. Pao, 'Memory-based architecture for multicharacter aho–corasick string matching', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 1, pp. 143–154, 2017.

[27] S. Faro and T. Lecroq, 'Efficient variants of the backward-oracle-matching algorithm', *International Journal of Foundations of Computer Science*, vol. 20, no. 06, pp. 967–984, 2009.

[28] D. E. Knuth, J. H. Morris Jr and V. R. Pratt, 'Fast pattern matching in strings', *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[29] A.-N. Du, B.-X. Fang, X.-C. Yun, M.-Z. Hu and X.-R. Zheng, 'Comparison of stringmatching algorithms: An aid to information content security', in *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 03EX693)*, IEEE, vol. 5, 2003, pp. 2996–3001.

[30] C. Allauzen, M. Crochemore and M. Raffinot, 'Factor oracle: A new structure for pattern matching', in *International Conference on Current Trends in Theory and Practice of Computer Science*, Springer, 1999, pp. 295–310.

[31] G. Navarro and M. Raffinot, 'Fast and flexible string matching by combining bit-parallelism and suffix automata', *Journal of Experimental Algorithmics (JEA)*, vol. 5, 4–es, 2000.

[32] C. S. Kouzinopoulos and K. G. Margaritis, 'Exact online two-dimensional pattern matching using multiple pattern matching algorithms', *Journal of Experimental Algorithmics (JEA)*, vol. 18, pp. 2–1, 2013.

[33] T. T. Tran, Y. Liu and B. Schmidt, 'Bit-parallel approximate pattern matching: Kepler gpu versus xeon phi', *Parallel Computing*, vol. 54, pp. 128–138, 2016.

[34] J. Tan, X. Liu, Y. Liu and P. Liu, 'Speeding up pattern matching by optimal partial string extraction', in *2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2011, pp. 1030–1035.

[35] L. H. Zhou, 'A network intrusion detection system with improved sbom algorithm', in *Applied Mechanics and Materials*, Trans Tech Publ, vol. 336, 2013, pp. 2419–2422.

[36] Y. Liu, Q. Liu, P. Liu, J. Tan and L. Guo, 'A factor-searching-based multiple string matching algorithm for intrusion detection', in *2014 IEEE International Conference on Communications (ICC)*, IEEE, 2014, pp. 653–658.

[37] J. Kelly, *An Examination of Pattern Matching Algorithms for Intrusion Detection Systems*, https://curve.carleton.ca/system/files/etd/1fc84226-5a10-41c6-81ce-4e8b8cc94b09/etd_pdf/3eab1f5c799871f4f433bf7d47aba291/kelly-anexaminationofpatternmatchingalgorithmsfor.pdf, [Online; accessed 22-April-2021], 2006.

[38] B. Ďurian, J. Holub, H. Peltola and J. Tarhio, 'Tuning bndm with q-grams', in *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*, SIAM, 2009, pp. 29–37.

[39] T. O. I. S. F. (OISF), *Suricata*, https://suricata.io/, [Online; accessed 16-November-2021], 2021.

[40] S. Petrovic, *Topic 1 ids/ips definition and classification*, https:\ntnu.blackboard.com, Online; accessed 01-December-2020, IMT4204 Intrusion Detection in Physical, Virtual Networks, Norwegien University of Science and Technology, 2019.

[41] S. Khan and I. Traore, 'A prevention model for algorithmic complexity attacks', in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2005, pp. 160–173.

[42] Y. Zhang, P. Liu, Y. Liu, A. Li, C. Du and D. Fan, 'Attacking pattern matching algorithms based on the gap between average-case and worst-case complexity', *Journal on Advances in Computer Network*, vol. 1, no. 3, pp. 228–233, 2013.

[43] R. Singh, H. Kumar, R. K. Singla and R. R. Ketti, 'Internet attacks and intrusion detection system: A review of the literature', *Online Information Review*, 2017.

[44] S. Lee, S. Kim, S. Lee, J. Choi, H. Yoon, D. Lee and J.-R. Lee, 'Largen: Automatic signature generation for malwares using latent dirichlet allocation', *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 771–783, 2016.

[45] A. Shabtai, E. Menahem and Y. Elovici, 'F-sign: Automatic, function-based signature generation for malware', *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 41, no. 4, pp. 494–508, 2010.

[46] C. Jia and F. Yang, 'An intrusion detection method based on hierarchical hidden markov models', *Wuhan University Journal of Natural Sciences*, vol. 12, no. 1, pp. 135–138, 2007.

[47] G. Thatte, U. Mitra and J. Heidemann, 'Parametric methods for anomaly detection in aggregate traffic', *IEEE/ACM Transactions On Networking*, vol. 19, no. 2, pp. 512–525, 2010.

[48] Y. Li and L. Guo, 'An active learning based tcm-knn algorithm for supervised network intrusion detection', *Computers & security*, vol. 26, no. 7-8, pp. 459–467, 2007.

[49] H. T. Elshoush and I. M. Osman, 'Alert correlation in collaborative intelligent intrusion detection systems—a survey', *Applied Soft Computing*, vol. 11, no. 7, pp. 4349–4365, 2011.

[50] P. D. Leedy and J. E. Ormrod, *Practical research*. Pearson Custom Saddle River, NJ, 2005, vol. 108.

[51] J. Cresswell, *Research design*. Sage publications Thousand Oaks, 2014.

[52] J. H. Alnes, *Metode*, `https://snl.no/hypotetisk-deduktiv_metode`, [Online; accessed 28-January-2021], 2017.

[53] O. Blakstad, *Experimental Research*, `https://explorable.com/experimental-research`, [Online; accessed 29-October-2021], 2008.

[54] S. Priom, *AhoCorasick.cpp*, `https://gist.github.com/ashpriom/b8231c806edeef50afe1`, [Online; accessed 6-September-2021], 2014.

[55] Wikipeida, *IPv4*, `https://en.wikipedia.org/wiki/IPv4`, [Online; accessed 14-September-2021], 2021.

[56] Wikipedia, *IPv6 Packet*, `https://en.wikipedia.org/wiki/IPv6_packet`, [Online; accessed 14-September-2021], 2021.

[57] P. Mell, V. Hu, R. Lippmann, J. Haines and M. Zissman, *An overview of issues in testing intrusion detection systems*, 2003.

[58] University of New Brunswick, *Datasets*, `https://www.unb.ca/cic/datasets/index.html`, [Online; accessed 27-January-2021], 2016.

Appendix A

Source code of the implemented algorithms

A.1 The Aho-Corasick Algorithm

ahocorasick.cpp

```
#include <iostream>
#include <sstream>
#include <string>
#include <chrono>
#include <fstream>

#include "aho_corasick.hpp"

using namespace std;
using namespace std::chrono;

int main(){
    string p; //name of pattern file
    string t; //name of search text file
    string text; //stores search text
    vector<string> keywords;
    cout << "Enter pattern file:\n";
    cin >> p;
    ifstream infile (p);
    std::string line;
    while (getline(infile, line)){ //reads line by line and pushes the pattern on to
    //the keywords vector
        keywords.push_back(line);
    }
```

```
    cout << "Enter search file \n";
    cin >> t;
    ifstream file{t}; //opens file for reading
    copy( istream_iterator<char>{ file >> noskipws }, {}, back_inserter( text ));
    //saves text into 'text'
    file.close();

    vector<char> T;
    int z=0;
    for (int x=0;x<text.length();x++){//pushes text into vector and skips linebreaks
        if (text.at(x)!='\n'){
            T.push_back(text.at(x));
            z++;
        }


    }

    buildMatchingMachine(keywords, ' ', '~');
    auto start = high_resolution_clock::now();
    int currentState = 0;
    int o=0;
    for (int i = 0; i < T.size(); ++i) {
       currentState = findNextState(currentState, T.at(i), ' ');
       if (out[currentState] == 0) continue;
       // Nothing new, let's move on to the next character.
       for (int j = 0; j < keywords.size(); ++j) {
           if (out[currentState] & (1 << j)) { // Matched keywords[j]
                o++;
                //cout << "Match nr. " << o << " with keyword nr. "
                //<< j+1 << " appears from "
                //    << i - keywords[j].size() + 1 << " to " << i << endl;
           }
       }
   }
    cout << o << '\n';
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    cout << "Execution time:" << duration.count() << "micros\n";

   return 0;

}
```

aho_corasick.hpp

```
 #ifndef AHO_CORASICK_HPP
#define AHO_CORASICK_HPP

using namespace std;
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
#include <sstream>
#include <fstream>
#include <cassert>
#include <climits>
#include <cstdlib>
#include <cstring>
#include <string>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <deque>
#include <stack>
#include <list>
#include <map>
#include <set>

const int MAXS = 6 * 50 + 10; // Max number of states in the matching machine.
                              // Should be equal to the sum of the length of
                              //all keywords.

const int MAXC = 94; // Number of characters in the alphabet.

int out[1000]; // Output for each state, as a bitwise mask.
               // Bit i in this mask is on if the keyword with index i appears when the
               // machine enters this state.

// Used internally in the algorithm.
int f[1000]; // Failure function
int g[1000][MAXC]; // Goto function, or -1 if fail.

// Builds the string matching machine.
//
```

```
// words - Vector of keywords. The index of each keyword is important:
//         "out[state] & (1 << i)" is > 0 if we just found word[i] in the text.
// lowestChar - The lowest char in the alphabet. Defaults to 'a'.
// highestChar - The highest char in the alphabet. Defaults to 'z'.
//               "highestChar - lowestChar" must be <= MAXC, otherwise we will
//               access the g matrix outside its bounds and things will go wrong.
//
// Returns the number of states that the new machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(const vector<string> &words,
char lowestChar = ' ', char highestChar = '~') {
    memset(out, 0, sizeof out);
    memset(f, -1, sizeof f);
    memset(g, -1, sizeof g);

    int states = 1; // Initially, we just have the 0 state

    for (int i = 0; i < words.size(); ++i) {
        const string &keyword = words[i];
        int currentState = 0;
        for (int j = 0; j < keyword.size(); ++j) {
            int c = keyword[j] - lowestChar;
            if (g[currentState][c] == -1) { // Allocate a new node
                g[currentState][c] = states++;
            }
            currentState = g[currentState][c];
        }
        out[currentState] |= (1 << i);
        // There's a match of keywords[i] at node currentState.
    }

    // State 0 should have an outgoing edge for all characters.
    for (int c = 0; c < MAXC; ++c) {
        if (g[0][c] == -1) {
            g[0][c] = 0;
        }
    }

    // Now, let's build the failure function
    queue<int> q;
    for (int c = 0; c <= highestChar - lowestChar; ++c) {
    // Iterate over every possible input
        // All nodes s of depth 1 have f[s] = 0
        if (g[0][c] != -1 and g[0][c] != 0) {
```

```
            f[g[0][c]] = 0;
            q.push(g[0][c]);
        }
    }
    while (q.size()) {
        int state = q.front();
        q.pop();
        for (int c = 0; c <= highestChar - lowestChar; ++c) {
            if (g[state][c] != -1) {
                int failure = f[state];
                while (g[failure][c] == -1) {
                    failure = f[failure];
                }
                failure = g[failure][c];
                f[g[state][c]] = failure;
                out[g[state][c]] |= out[failure]; // Merge out values
                q.push(g[state][c]);
            }
        }
    }

    return states;
}

// Finds the next state the machine will transition to.
//
// currentState - The current state of the machine. Must be between
//                0 and the number of states - 1, inclusive.
// nextInput - The next character that enters into the machine. Should be
//             between lowestChar and highestChar, inclusive.
// lowestChar - Should be the same lowestChar that was passed to "buildMatchingMachine".

// Returns the next state the machine will transition to. This is an integer between
// 0 and the number of states - 1, inclusive.
int findNextState(int currentState, char nextInput, char lowestChar = ' ') {
    int answer = currentState;
    int c = nextInput - lowestChar;
    while (g[answer][c] == -1) answer = f[answer];
    return g[answer][c];
}

#endif //AHO_CORASICK_HPP
```

A.2 The SBOM Algorithm

sbom.cpp

```
#include<string>
#include<iostream>
#include<algorithm>
#include<iterator>
#include<vector>
#include <sstream>
#include<fstream>
#include<cstdlib>
#include<chrono>
#include<cstring>



using namespace std;
using namespace std::chrono;

int main(){
    string pfile; //name of pattern file
    string tfile; //name of search text file
    string text; //stores search text
    int p=0; //number of patterns
    int pmin; //length of shortest pattern
    string pr;
    vector<string> keywords;//patterns read from file
    //vector<string> rkeywords;//patterns read from file reversed

    cout << "Enter pattern file:\n";
    cin >> pfile;
    ifstream infile (pfile);
    string line;

    while (getline(infile, line)){
    //reads line by line and pushes the pattern on to the keywords vector
        keywords.push_back(line);

        if (p==0){//saves the length of shortest pattern in pmin
            pmin=line.length();
        }
        else{
            if (pmin>line.length()){
```

```
                pmin=line.length();
            }
        }
        //reverse_copy(line.begin(), line.end(), rline.begin());
        //rkeywords.push_back(rline); //reversed patterns
        p++;
    }

////////////////////////////PREPROCESSING////////////////////////////////

    int P[100][2000]={};//matrix with ascii value to each pattern
    int Q[100][2000]={0};//outgoing states
    int S[143][p*pmin]={0};//table of states
    for (int x=0; x<143; x++){//sets hole table to empty value -1
        for (int y=0; y< p*pmin; y++){
            S[x][y]=-1;
        }
    }
    vector<string> rkeywords;
    string pattern;
    int F[p]={};//Saves terminal states for each pattern


    for (int x=0; x<p; x++){
        pattern= keywords.at(x);
        string pshort;

        for (int y=0; y<pmin; y++){

              pshort.push_back(pattern.at(y));


        }
        string pr=pshort;

        reverse_copy(pshort.begin(), pshort.end(), pr.begin());

        rkeywords.push_back(pr); //reversed patterns

        for (int y=0; y<=pmin; y++){
            if (y==0){
                P[x][y]=-1;
            }
            else{
```

```
            unsigned char a = pr.at(y-1);
            P[x][y]=(int) a-32; //ascii value of each pattern saved in matrix
        }

    }
}

int current=0;
int s=0;//state number to put current state in

for (int y=1; y<=pmin; y++){

    for (int x=0; x<p; x++){
        if (P[x][y-1]==-1){//has no parent
            s=0;


        }
        else {//has parent
            s=Q[x][y-1];

        }
        if (s!=0){
            if (S[P[x][y]][0]==-1){
               S[P[x][y]][0]=current+1;
            }
        }
        if (S[P[x][y]][s]==-1){//if no transition from the state to next state
            S[P[x][y]][s]=current+1;
            Q[x][y]=current+1;

            for (int z=y; z>0; z--){
            //checks for other possible transitions to current
                for (int v=p-1; v>=0; v--){
                    if (P[v][z]==P[x][y-1] && S[P[x][y]][Q[v][z]]==-1
                    && Q[v][z]!=current+1){
                        S[P[x][y]][Q[v][z]]=current+1;

                    }
                }
            }
            current++;
        }
        else{
```

```
                Q[x][y]=S[P[x][y]][s];
                for (int z=current; z>0; z--){
                //checks for other possible transitions to current
                    for (int v=p; v>0; v--){
                        if (P[v][z]==P[x][y-1] && S[P[x][y]][Q[v][z]]==-1){
                            S[P[x][y]][Q[v][z]]=current+1;

                        }
                    }
                }
            }

        if (y==pmin){
            F[x]=S[P[x][y]][s];//saves the terminal state for each pattern
        }

        }
    }
//////////////////////////////END PREPROCESSING //////////////////////

    cout << "Enter search file \n";
    cin >> tfile;
    ifstream file{tfile}; //opens file for reading
    copy( istream_iterator<char>{ file >> noskipws }, {}, back_inserter( text ) );
    //saves text
    file.close();


    vector<int> TXT;
    vector<char> T;
    vector<int> C;
    for (int x=0;x<text.length();x++){//transforms text to ascii
        if (text.at(x)!='\n'){
            T.push_back(text.at(x));
            TXT.push_back(text.at(x)-32);

        }

    }
//////////////////////////SEARCHING//////////////////////////////////////
auto start = high_resolution_clock::now();

int pos=0; //Search window start pos
int o=0;//occurances
```

```
int a=0;//almost match
int nm=0; //no match after checking the plmin match with full p
int c=0;


while (pos<=T.size()-pmin)
{
    current=0;
    int j=pmin;
    while (j>0 && current!=(-1))
    {
        char bla=T.at(pos+j-1);
        int ntest=TXT.at(pos+j-1);
        current=S[TXT.at(pos+j-1)][current];
        j--;
        c++;

    }
    if (current!=(-1)) {
        int m=0; //to make it not incremate nm if it is a match with another password
        for (int x=0;x<p; x++){
            if (F[x]==current){
            //Verify all the patterns in F(Current) one by one against the text
                int pl=keywords.at(x).length();
                string pm;
                if (pos+pl<=T.size()){
                    for (int y=pos; y<pos+pl; y++){
                    pm.push_back(T.at(y));
                }

                }
                int res=pm.compare(keywords.at(x));
                if (res == 0){
                    o++;
                    m++;
                    //cout << "Match nr." << o << "at pos " << pos << "
                    //with pattern nr." << x+1 << '\n';
                }
                else if (m==0){
                    nm++;
                }
            }
        }
```

```
    }
    else if (j<=pmin-1){
        a++;

    }


    pos=pos+j+1;

}
cout << a << ' ' << nm << c << '\n';
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
cout << "Execution time:" << duration.count() << "micros\n";

return 0;

}
```

A.3 The SBOM-AC combination

SBOMAC.cpp

```
#include<string>
#include<iostream>
#include<algorithm>
#include<iterator>
#include<vector>
#include <sstream>
#include<fstream>
#include<cstdlib>
#include<chrono>
#include<cstring>

#include "aho_corasick.hpp"


using namespace std;
using namespace std::chrono;

int main(){
    string pfile; //name of pattern file
    string tfile; //name of search text file
    string text; //stores search text
    int p=0; //number of patterns
    int pmin; //length of shortest pattern
    string pr;
    vector<string> keywords;//patterns read from file
    //vector<string> rkeywords;//patterns read from file reversed

    cout << "Enter pattern file:\n";
    cin >> pfile;
    ifstream infile (pfile);
    string line;

    while (getline(infile, line)){
    //reads line by line and pushes the pattern on to the keywords vector
        keywords.push_back(line);

        if (p==0){//saves the length of shortest pattern in pmin
            pmin=line.length();
        }
        else{
            if (pmin>line.length()){
```

```
                pmin=line.length();
            }
        }

        p++;
    }

////////////////////////////////PREPROCESSING//////////////////////////////
    buildMatchingMachine(keywords, ' ', '~');
    //builds the matching machine for AhoCorasick

    int P[100][2000]={};//matrix with ascii value to each pattern
    int Q[100][2000]={0};//outgoing states
    int S[143][p*pmin]={0};//table of states
    for (int x=0; x<143; x++){//sets hole table to empty value -1
        for (int y=0; y< p*pmin; y++){
            S[x][y]=-1;
        }
    }
    vector<string> rkeywords;
    string pattern;
    int F[p]={};//Saves terminal states for each pattern


    for (int x=0; x<p; x++){
        pattern= keywords.at(x);
        string pshort;

        for (int y=0; y<pmin; y++){

              pshort.push_back(pattern.at(y));


        }
        string pr=pshort;

        reverse_copy(pshort.begin(), pshort.end(), pr.begin());

        rkeywords.push_back(pr); //reversed patterns

        for (int y=0; y<=pmin; y++){
            if (y==0){
                P[x][y]=-1;
            }
```

```
            else{
                unsigned char a = pr.at(y-1);
                P[x][y]=(int) a-32; //ascii value of each pattern saved in matrix
            }

        }
    }

    int current=0;
    int s=0;//state number to put current state in

    for (int y=1; y<=pmin; y++){

        for (int x=0; x<p; x++){
            if (P[x][y-1]==-1){//has no parent
                s=0;


            }
            else {//has parent
                s=Q[x][y-1];

            }
            if (s!=0){
                if (S[P[x][y]][0]==-1){
                   S[P[x][y]][0]=current+1;
                }
            }
            if (S[P[x][y]][s]==-1){//if no transition from the state to next state
                S[P[x][y]][s]=current+1;
                Q[x][y]=current+1;

                for (int z=y; z>0; z--){
                //checks for other possible transitions to current
                    for (int v=p-1; v>=0; v--){
                        if (P[v][z]==P[x][y-1] && S[P[x][y]][Q[v][z]]==-1
                        && Q[v][z]!=current+1){
                            S[P[x][y]][Q[v][z]]=current+1;

                        }
                    }
                }
                current++;
            }
```

```
            else{
                Q[x][y]=S[P[x][y]][s];
                for (int z=current; z>0; z--){
                //checks for other possible transitions to current
                    for (int v=p; v>0; v--){
                        if (P[v][z]==P[x][y-1] && S[P[x][y]][Q[v][z]]==-1){
                            S[P[x][y]][Q[v][z]]=current+1;

                        }
                    }
                }
            }

        if (y==pmin){
            F[x]=S[P[x][y]][s];//saves the terminal state for each pattern
        }

        }
    }
//////////////////////////////END PREPROCESSING//////////////////////

    cout << "Enter search file \n";
    cin >> tfile;
    ifstream file{tfile}; //opens file for reading
    copy( istream_iterator<char>{ file >> noskipws }, {}, back_inserter( text ) );
    //saves text
    file.close();


    vector<int> TXT;
    vector<char> T;
    for (int x=0;x<text.length();x++){//transforms text to ascii
        if (text.at(x)!='\n'){
            T.push_back(text.at(x));
            TXT.push_back(text.at(x)-32);

        }

    }
//////////////////////////////SEARCHING////////////////////////////////////
auto start = high_resolution_clock::now();

int pos=0; //Search window start pos
int o=0;//occurances
```

```
int a=0;//almost match
int nm=0; //no match after checking the plmin match with full p


while (pos<=T.size()-pmin)
{
    current=0;
    int j=pmin;
    while (j>0 && current!=(-1))
    {
        current=S[TXT.at(pos+j-1)][current];
        j--;

    }
    if (current!=(-1)) {
        int m=0; //to make it not incremate nm if it is a match with another password
        for (int x=0;x<p; x++){
            if (F[x]==current){
            //Verify all the patterns in F(Current) one by one against the text
                int pl=keywords.at(x).length();
                string pm;

                for (int y=pos; y<pos+pl; y++){
                    pm.push_back(T.at(y));
                }
                int res=pm.compare(keywords.at(x));

                if (res == 0){
                    o++;
                    m++;
                    //cout << "Match nr." << o << "at pos " << pos << "
                    //with pattern nr." << x+1 << '\n';
                }
                else if (m==0){
                    nm++;
                }
            }
        }



    }
    else if (j<=(pmin-3)){
        a++;
```

```
    }


    pos=pos+j+1;

    if (nm>=25 || a>=500){
        cout << "Switching to AC\n";
        int currentState=0;
        for (int i = pos; i < T.size(); ++i) {
            currentState = findNextState(currentState, T.at(i), ' ');
            if (out[currentState] == 0) continue;
            // Nothing new, let's move on to the next character.
            for (int k = 0; k < keywords.size(); ++k) {
                if (out[currentState] & (1 << k)) { // Matched keywords[k]
                    o++;
                    //cout << "Match nr. " << o << " with keyword nr. " << j+1 <<
                    //" appears from "
                    //    << i - keywords[k].size() + 1 << " to " << i << endl;
                }
            }
        }
        pos=T.size();
    }


}
cout << a << ' ' << nm << '\n';
auto stop = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(stop - start);
cout << "Execution time:" << duration.count() << "micros\n";

return 0;

}
aho_corasick.hpp
#ifndef AHO_CORASICK_HPP
#define AHO_CORASICK_HPP

using namespace std;
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
```

```
#include <sstream>
#include <fstream>
#include <cassert>
#include <climits>
#include <cstdlib>
#include <cstring>
#include <string>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <deque>
#include <stack>
#include <list>
#include <map>
#include <set>

const int MAXS = 6 * 50 + 10; // Max number of states in the matching machine.
                              // Should be equal to the sum of the length
                              // of all keywords.

const int MAXC = 94; // Number of characters in the alphabet.

int out[1000]; // Output for each state, as a bitwise mask.
               // Bit i in this mask is on if the keyword with index i appears when the
               // machine enters this state.

// Used internally in the algorithm.
int f[1000]; // Failure function
int g[1000][MAXC]; // Goto function, or -1 if fail.

// Builds the string matching machine.
//
// words - Vector of keywords. The index of each keyword is important:
//         "out[state] & (1 << i)" is > 0 if we just found word[i] in the text.
// lowestChar - The lowest char in the alphabet. Defaults to 'a'.
// highestChar - The highest char in the alphabet. Defaults to 'z'.
//               "highestChar - lowestChar" must be <= MAXC, otherwise we will
//               access the g matrix outside its bounds and things will go wrong.
//
// Returns the number of states that the new machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(const vector<string> &words,
    char lowestChar = ' ', char highestChar = '~') {
```

```
    memset(out, 0, sizeof out);
    memset(f, -1, sizeof f);
    memset(g, -1, sizeof g);

    int states = 1; // Initially, we just have the 0 state

    for (int i = 0; i < words.size(); ++i) {
        const string &keyword = words[i];
        int currentState = 0;
        for (int j = 0; j < keyword.size(); ++j) {
            int c = keyword[j] - lowestChar;
            if (g[currentState][c] == -1) { // Allocate a new node
                g[currentState][c] = states++;
            }
            currentState = g[currentState][c];
        }
        out[currentState] |= (1 << i);
        // There's a match of keywords[i] at node currentState.
    }

    // State 0 should have an outgoing edge for all characters.
    for (int c = 0; c < MAXC; ++c) {
        if (g[0][c] == -1) {
            g[0][c] = 0;
        }
    }

    // Now, let's build the failure function
    queue<int> q;
    for (int c = 0; c <= highestChar - lowestChar; ++c) {
    // Iterate over every possible input
        // All nodes s of depth 1 have f[s] = 0
        if (g[0][c] != -1 and g[0][c] != 0) {
            f[g[0][c]] = 0;
            q.push(g[0][c]);
        }
    }
    while (q.size()) {
        int state = q.front();
        q.pop();
        for (int c = 0; c <= highestChar - lowestChar; ++c) {
            if (g[state][c] != -1) {
                int failure = f[state];
                while (g[failure][c] == -1) {
```

```
                    failure = f[failure];
                }
                failure = g[failure][c];
                f[g[state][c]] = failure;
                out[g[state][c]] |= out[failure]; // Merge out values
                q.push(g[state][c]);
            }
        }
    }

    return states;
}

// Finds the next state the machine will transition to.
//
// currentState - The current state of the machine. Must be between
//                0 and the number of states - 1, inclusive.
// nextInput - The next character that enters into the machine. Should be
//             between lowestChar and highestChar, inclusive.
// lowestChar - Should be the same lowestChar that was passed to "buildMatchingMachine".

// Returns the next state the machine will transition to. This is an integer between
// 0 and the number of states - 1, inclusive.
int findNextState(int currentState, char nextInput, char lowestChar = ' ') {
    int answer = currentState;
    int c = nextInput - lowestChar;
    while (g[answer][c] == -1) answer = f[answer];
    return g[answer][c];
}

#endif //AHO_CORASICK_HPP
```

Appendix B

Dataset used for testing

Nr	Size search text	Size pattern file	P_{lmin}	Number of positives	Number of hits
0	98KB	723B	112	2	8(7+1)
1	969KB	723B	112	2	8(7+1)
2	969KB	998B	32	0	0
3	969KB	909B	188	0	0
4	1641KB	723B	112	1	10000
5	1880KB	723B	112	0	0
6	1662KB	723B	112	0	0
7	1112KB	723B	112	0	0
8	1932KB	723B	112	0	0
9	1444KB	723B	112	0	0
10	1715KB	723B	112	0	0
11	528KB	650B	40	0	0
12	991KB	585B	260	0	0
13	528KB	585B	260	0	0
14	1512KB	585B	260	0	0
15	1809KB	585B	260	0	0
16	1539KB	585B	260	0	0
17	630KB	590B	3	0	0
18	10KB	585B	112	0	0
19	19KB	585B	112	0	0
20	37KB	585B	112	0	0
21	55.5KB	727B	112	0	0
22	55.5KB	674B	60	0	0
23	55.5KB	634B	20	0	0
24	55.5KB	625B	10	0	0
25	55.5KB	619B	4	0	0

Appendix C

Raw Results

C.1 Aho-Corasick

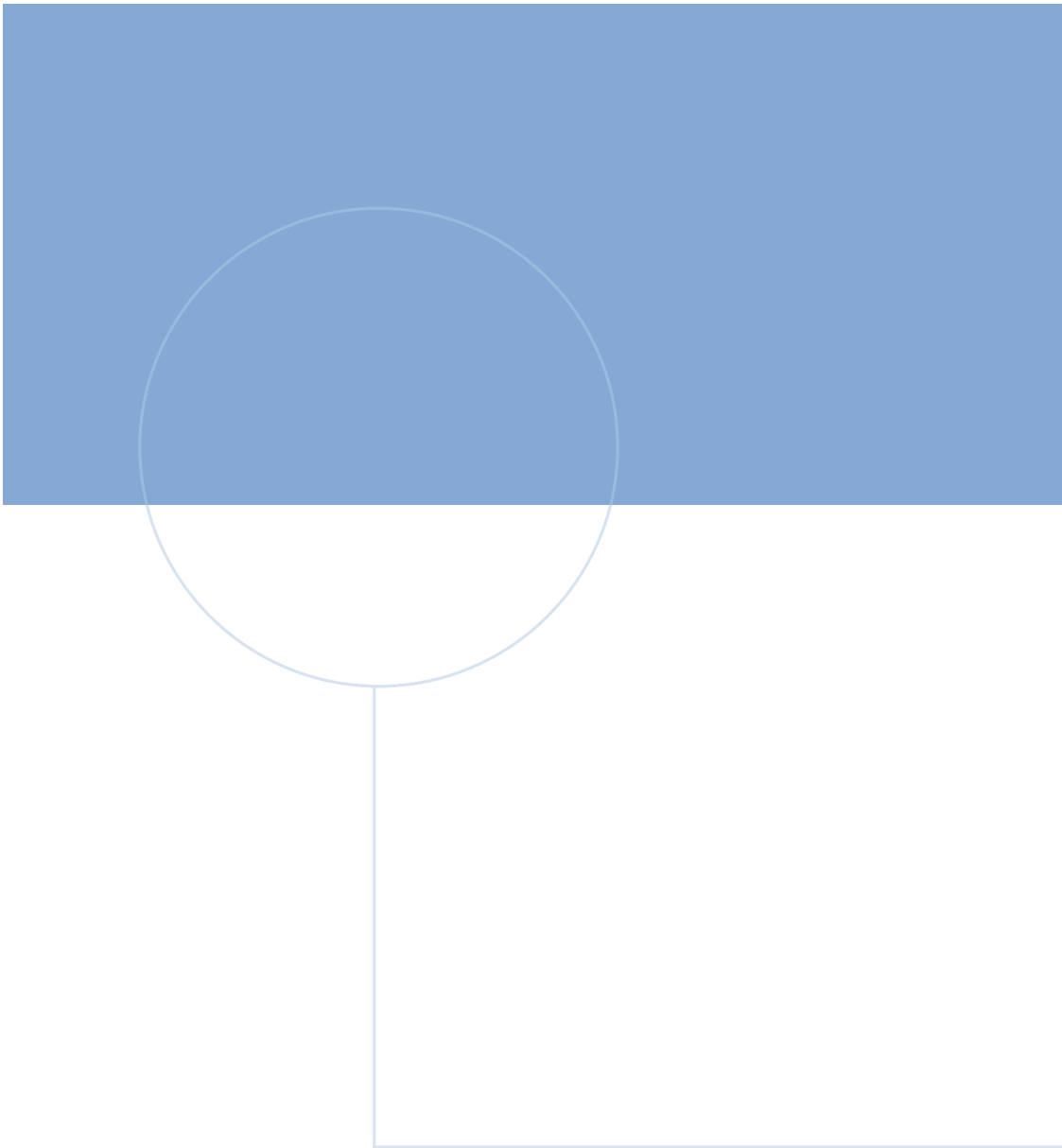
Dataset	Run						Average
	1	2	3	4	5	6	
1	12002	11994	12000	11923	12996	12004	12153.16667
2	10999	10973	21515	2006	11052	11999	11424
3	11999	12001	11940	12058	10999	16953	12658.33333
4	19999	19964	19983	20990	19973	19938	20141.16667
5	21999	22037	21938	21999	21999	23003	22162.5
6	21938	21619	33115	21819	21939	21963	23732.16667
7	13952	13996	13963	13993	12996	14001	13816.83333
8	26231	28249	23861	23260	27074	25542	25702.83333
9	17679	18586	23162	20705	20408	19026	19927.66667
10	29122	23739	23853	24920	23158	32029	26136.83333
11	5999	7149	5690	6962	7035	7085	6653.333333
17	15472	18697	19756	22525	17424	16964	18473
18	346	604	839	979	1344	529	773.5
19	1219	1319	1289	1232	1373	3303	1622.5
20	3957	3180	2178	2950	2450	2988	2950.5
21	1991	1992	1992	1996	1994	1993	1993
22	1992	1995	997	998	1995	1994	1661.833333
23	1980	996	2994	1995	1996	1993	1992.333333
24	2993	995	3000	1994	2992	996	2161.666667
25	1990	1960	2994	1992	1993	1995	2154

C.2 SBOM

Dataset	Run						Average
	1	2	3	4	5	6	
1	1939	1973	8938	1961	1963	1962	3122.667
2	4519	3712	5610	3810	4762	6415	4804.667
3	1891	3900	2009	1829	2228	4107	2660.667
4	85001	85998	84983	90001	86000	84984	86161.167
5	3987	3030	3170	3010	3068	2940	3200.833
6	1998	3000	0	0	2963	0	1326.833
7	88939	90000	89971	89952	88977	90003	89640.33
8	3006	4507	5735	6752	6893	7205	5683
9	3239	3615	2914	4731	3672	5610	3963.5
10	7240	3792	4233	5794	4494	5994	5257.833
11	92260	86001	86213	93807	87998	87061	88890
17	1422508	1435800	1616981	1865282	1692767	1546309	1596607.83
18	3896	3377	3754	3349	4068	4065	3751.5
19	4567	5491	6622	6623	5320	4768	5565.167
20	11822	13698	13852	11415	22033	12841	14276.833
21	2574	1002	2174	994	1002	998	1457.333
22	1991	1437	996	966	1995	998	1397.167
23	999	1993	999	1996	2863	1328	1696.333
24	1994	2991	2087	2991	1961	2959	2497.167
25	1991	1997	997	1994	1882	1994	1809.167

C.3 SBOM-AC Combination

Dataset	Run						Average	Comment
	1	2	3	4	5	6		
1	2030	2000	1967	2962	3013	2173	2357.5	
2	7186	5082	10539	2198	2798	3068	5145.167	
3	2277	4878	2946	2434	3247	1891	2945.5	
4	102978	90711	90159	88885	92006	86680	91903.167	
5	4118	4216	5457	2124	3742	5926	4263.833	
6	1072	0	1945	0	1963	2000	1163.333	
7	20162	16130	15964	15959	14985	20257	17242.833	
8	4854	3641	6466	9540	4010	10556	6511.167	
9	2480	4684	5786	5479	1816	3191	3906	
10	5189	5257	5323	4004	5439	3978	4865	
11	8143	4181	9981	8279	7999	8414	7832.833	
17	18143	28651	22512	32836	27652	17079	24478.833	
18	2962	1994	1993	1993	1994	1993	2154.833	nm=25
19	2993	1999	1994	2995	1994	2958	2488.833	nm=25
20	5358	3306	5114	2561	4260	8275	4812.333	nm=50
20	2713	4585	4110	2482	5066	4530	3914.333	nm=25
20	4993	5221	6258	7076	4552	4881	5496.833	nm=100
20	4633	4387	5785	4618	3866	3797	4514.333	nm=10
21	3986	1995	1997	2594	1001	1996	2261.5	a=500
22	2992	2989	2065	2992	2993	2989	2836.667	a=500
23	2839	4988	2991	2989	2995	3993	3465.833	a=250
23	3159	4350	3011	2993	2683	2520	3119.333	a=500
23	2994	2958	3991	2727	2992	2960	3103.667	a=1000
23	1994	3000	3988	2000	2958	2991	2821.833	a=2000
24	3957	2998	3649	2960	2991	3958	3418.833	a=250
24	2994	2989	2991	3514	3342	3957	3297.833	a=500
24	3989	2991	2994	2993	3956	2996	3319.833	a=1000
24	3991	2996	2958	2994	3992	2994	3320.833	a=2000
25	2993	3956	2992	2991	2957	3531	3236.667	a= 500



NTNU

Norwegian University of
Science and Technology