PATTERN MATCHING ALGORITHMS
FOR INTRUSION DETECTION SYSTEMS

A Thesis submitted in partial fulfillment of the requirements
For the degree of Master of Science
In Computer Science

By
Stanimir Bogomilov Belchev

July 2012
The thesis of Stanimir Bogomilov Belchev is approved:

Prof. John Noga  
Date

Prof. Robert McIlhenny  
Date

Prof. Jeff Wiegley, Chair  
Date

California State University, Northridge
DEDICATION

I dedicate this paper to my parents, family, and friends for their continuous support throughout the years. I am grateful they continuously prompted me to go back to school and continue to educate myself – this has helped me a lot in life.

I also want to thank my girlfriend (and future wife) Nadia for always staying by my side and supporting me every step of the way. Her passions for science has had a great impact on me and prompted me to explore subjects that I wouldn’t think would be interesting to me anyways.

Thank you!!!
ACKNOWLEDGEMENTS

I would like to thank the Computer Science department at California State University Northridge for their help and support. Especially I would like to thank my committee members – Prof. Jeff Wiegley (chair), Prof. Robert McIlhenny, and Prof. John Noga. I admire their passion to computing and their dedication to teaching their students.

My stay at California State University Northridge was an invaluable experience that gave me inspiration, knowledge, and love towards computing. Every class was fun and exciting, even when I had to do my homework late at night. The staff and faculty and the Computer Science department was professional and very knowledgeable and they certainly helped me grow as an individual.

Thank you all!
# TABLE OF CONTENTS

Signature Page ........................................................................................................... ii

Dedication .................................................................................................................. iii

Acknowledgements ................................................................................................... iv

List of Tables ............................................................................................................ vi

List of Figures .......................................................................................................... vii

List of Algorithms ................................................................................................. viii

Glossary of Acronyms ............................................................................................ ix

Abstract ................................................................................................................... xi

Chapter 1: Introduction ............................................................................................ 1

Chapter 2: Overview of Intrusion Detection Systems .............................................. 4
  2.1 Firewall versus IDS ......................................................................................... 5
  2.2 Passive versus Reactive IDSs ......................................................................... 6
  2.3 Network-based versus Host-based IDSs ......................................................... 7
  2.4 Distributed IDSs ............................................................................................ 10
  2.5 Signature-based versus Anomaly-based Detection ......................................... 12
  2.6 Wireless IDSs ................................................................................................. 14
  2.7 IDSs Software Implementations ..................................................................... 16

Chapter 3: Snort Intrusion Detection System ............................................................. 18
  3.1 Overview of Snort IDS ................................................................................... 19
  3.2 Snort Architecture ......................................................................................... 22
3.3 Snort Rules

Chapter 4: Traits of Pattern Matching Algorithms

4.1 Overview of Pattern Matching
4.2 Signature Set Size
4.3 Alphabet Size
4.4 Keywords Length
4.5 Computational Complexity

Chapter 5: Single-Keyword Pattern Matching Algorithms

5.1 Brute Force (BF) Algorithm
5.2 Knuth-Morris-Prath (KMP) Algorithm
5.3 Boyer-Moore (BM) Algorithm
5.4 Karp and Rabin Algorithm

Chapter 6: Multiple-Keyword Pattern Matching Algorithms

6.1 Aho-Corasick (AC) Algorithm
6.2 Wu-Manber (WB) Algorithm
6.3 Commentz-Walter (CW) Algorithm

Chapter 7: Conclusion

Bibliography
LIST OF TABLES

5.2 Pre-computational table of the next function in the KMP algorithm.............................36
5.3 Pre-computation of the SHIFT table in the Boyer-Moore algorithm ...............................42
6.2 Calculation of the min function in the Wu-Manber algorithm ........................................57
6.2 SHIFT and HASH tables in the Wu-Manber algorithm ....................................................60
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3</td>
<td>A typical deployment of a Network Intrusion Detection System</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Distributed Intrusion Setection System</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Architecture of Snort IDS</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Sample Snort rule detecting attempts from a telnet session</td>
<td>21</td>
</tr>
<tr>
<td>4.4</td>
<td>Distribution of the string length in the Snort default database</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Normal and worst-case operation of a hash table.</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>Example of Moyer Moore Pattern Matching Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>Keyword tree for P = {he, she, his, hers}</td>
<td>52</td>
</tr>
<tr>
<td>6.2</td>
<td>Example of Aho-Corasick automation</td>
<td>54</td>
</tr>
<tr>
<td>6.3</td>
<td>Relation between search time and the length of patterns in Wu-Manber algorithm</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Commentz-Walter keyword trie including both shift functions</td>
<td>66</td>
</tr>
</tbody>
</table>
LIST OF ALGORITHMS

5.1 Java implementation of Brute Force algorithm ..............................................................34
5.2 Knuth-Morris-Pratt Next Function ..................................................................................35
5.2 Knuth-Morris-Pratt Matching Algorithm .........................................................................39
5.3 Boyer-Moore bad character pre-processing ......................................................................44
5.3 Pseudo Code of Boyer-Moore Algorithm ........................................................................45
5.4 Pseudo code for Karp-Rabin string matching algorithm ..................................................49
6.1 Aho Corasick Goto Function .........................................................................................54
6.1 Aho-Corasick Multiple Pattern Matching Algorithm .......................................................55
6.2 Pseudo-code of Wu-Manber algorithm ..........................................................................63
6.3 Commentz-Walter multiple pattern-matching algorithm pseudo-code. .........................63
GLOSSARY OF ACRONYMS

BASE – Basic Analysis and Security Engine

CIDE - Classless Inter-Domain Routing

DFA – Deterministic Finite Automata

DIDS – Distributed Intrusion Detection System

DoS – Denial of Service

DDoS – Distributed Denial of Service

HIDS – Host Intrusion Prevention System

ID – Intrusion Detection

IDS – Intrusion Detection System

IP – Internet Protocol

IPS – Intrusion Prevention System

LAN – Local Area Network

NAT – Network Address Translation

NIDS – Network Intrusion Prevention System

WIDS – Wireless Intrusion Detection System
ABSTRACT

PATTERN MATCHING ALGORITHMS FOR INTRUSION DETECTION SYSTEMS

By

Stanimir B. Belchev

Master of Science in Computer Science

Intrusion Detection Systems (IDSs) have become an indispensable tool for recognizing and responding to suspicious activities targeted at computing resources. They are responsible for analyzing the network traffic and identifying potentially malicious patterns of information. The essence of each Intrusion Detection System are the underlying algorithms that perform the pattern matching process. These algorithms must have a low algorithmic complexity and must be able to match the input to multiple patterns simultaneously.

In this paper we present a variety of string matching algorithms that are used in contemporary Intrusion Detection Systems. We explain the fundamental differences in each approach and compare the algorithms based on criteria such as their application domain, flexibility with various length keyword sets and run time complexity. Since there is no perfect algorithm that solves the general pattern matching problem we present different solutions that can be applied in different scenarios. We also introduce a widely used open-source IDS – Snort – and present its architecture and operations.
Chapter 1

Introduction

With the constant expansion of the Internet in the past few decades the need for secure networks has become greater than ever. Network security today nowadays face a tremendous challenge to keep their networks protected against the overwhelming amount of attacks that are launched on a daily basis. It is estimated that the current cost of the hacking industry is from $10 - $100 billion a year and the street value of a working, weaponized exploit could be as much as $100,000 [2]. At the same time, the cyber security industry is projected to be around $25 billion annually [17]. These staggering statistics point out that the demand for implementing and deploying state-of-the-art defense mechanisms against intruders has reached its optimal state.

Verizon’s 2011 Data Breach Investigation Report confirms the fact that security breaches have certain geographic tendencies. Most of the attacks originate from East Europe or Asia and usually from countries that are not economically advanced [1]. There are many factors that attribute to this propensity. High unemployment rates, minimal risk when committing a cybercrime, and lack of severe consequences in most cases are among the leading influences. The main problem is that individuals can be financially elevated much quicker if they successfully break into an important computer network rather than working for a company that develops security systems. Unfortunately this issue is a global problem and a solution is not likely to surface in the near future.

Without a doubt Intrusion Detection Systems (IDSs) play an important role in every contemporary security system. Even more significant are the underlying algorithms that these systems use in order to detect anomalies in the standard behavior of the internal
environment of an organization. While the study of pattern matching (searching for a specific set of keywords in an input text) has been studied very extensively in the past few decades the network security industry is still evolving, adapting, and responding to the persistent pressure exerted by cyber criminals. Therefore in this thesis we will examine a variety of pattern matching algorithms and explain the application domain for each one.

In Chapter 2 we present the background of Intrusion Detection systems and explain the different types. We also look at comparable devices that interact with IDSs in order to understand the similarities, differences and the necessity of deploying each one in a practical environment. In Chapter 3 we explore one of the most popular and widely-used open-source IDSs – Snort. We examine its architecture, rules, and signatures used to perform the pattern matching processes. Next, Chapter 4 explains the major traits that exist in the pattern matching mechanism. We see how signature and alphabet sizes can affect the performance of each algorithm and based on these properties we can determine which algorithm is a good candidate for implementation in IDS. In Chapter 5 we examine a diversity of single-keyword pattern matching algorithms which were the foundation of most network security algorithms. While simpler in their operation, single-keyword string-matching algorithms give us a fundamental understanding of how we can recognize an attack among the rest of the valid traffic passing through the network. However, in a real environment these types of approaches are not practical as their performance is much slower and moreover we almost always have the necessity to search for a set of patterns than just a single pattern. Then we continue to examine the more complex multiple-keyword pattern matching algorithms and compare their use and application to their predecessors in Chapter 6. Traditional approaches such as Aho-
Corasick and Wu-Manber present key ideas how we can speed up the pattern recognition process while looking into multiple strings in parallel. In this paper we focus on the software implementations of these algorithms and not on the hardware platforms they are running on. Finally, in Chapter 7 we present the conclusion of our research as well as future areas of study.
Chapter 2

Overview of Intrusion Detection Systems

In order to ensure that optimal measures are taken against cyber criminals, professionals in the cyber security industry have suggested that a comprehensive set of tools must be used to form a solid shield around a company’s virtual intellectual property. A vital part of these tools are Intrusion Detection Systems (IDSs).

The underlying operation of IDSs hasn’t changed much since their first inception in the late 80’s and early 90’s. In general, an IDS is deployed on the inside of the network, past all firewalls and routers. Depending on the scale of the organization’s infrastructure and the need for optimal security (banks, government associations, etc.) multiple systems might be installed in separate nodes of the network. The primary function of an IDS is to monitor incoming and outgoing network traffic and apply various algorithms in order to detect patterns of information, called signatures, that correspond to a specific intrusion. Once an attack is detected the system triggers an alarm and logs the event in a database so that it can be properly examined by a security professional. However, before we attempt to write an algorithm that recognizes these patterns we must establish a definition for an intrusion which is not a trivial task. Modern computer systems and networks are very complex with thousands of processes being executed every second on different levels. Therefore we must study their architecture very well in order to identify when a system or network is behaving abnormally. There are many different types of intrusions and examples of those are buffer overflow, targeted towards a web server or DDOS attack, exploiting a software bug or system misconfiguration, cracking system passwords, sniffing unprotected traffic, scanning for open ports, or
taking advantage of vulnerabilities in the design of a specific networking protocol. We can generalize that an intrusion is an attempt to compromise the integrity or availability of a specific resource on the network [3].

All Intrusion Detection Systems are designed to perform several key functions after being deployed. These features include (but are not limited to):

- To monitor network traffic, users, target machines
- To monitor system files, event logs, and system resources
- To analyze the system’s susceptibility to attacks and to identify potential risks
- To detect abnormal system behavior and to adapt to changes in the environment
- To detection violations of policies and unauthorized access
- To run continuously without human interaction
- Must not cause a network or system crash should the IDS becomes unavailable
- They need to provide the minimum amount of false-positive alarms

While all the above characteristics will comprise the perfect IDS we all know that such a system doesn’t exist and is impossible to implement. Next, in this text, we will examine the different types of Intrusion Detection System and identify the features each one possesses.

2.1 Firewall vs. IDS

It didn’t take long for security administrator to realize that a properly configured firewall is not enough to provide absolute proper protection for their networks. This is because we configure firewalls to block or allow certain traffic based on criteria such as packet
header, source/destination IP addresses, specific protocols, ports or interfaces but they fail to detect attacks on specific applications or services. Moreover firewalls are unable to distinguish various patterns of events that are coming in through the wire while in many cases these signatures hold vital information about a possible security risk. This is where Intrusion Detection Systems become useful. Their purpose is to complement firewalls by analyzing the traffic passed through them and in some cases use the information provided from the firewall to help determine the validity of the traffic. For example firewalls perform Network Address Translation (NAT) between public and private IP addresses and this data could be useful to the IDS because they can recognize the actual IP address of the host performing the attack.

Since firewalls and IDSs can quickly add up to the overall cost of an organization many professionals need to choose between both approaches. If this is the case it is always better to purchase a robust firewall because by nature they are active devices where IDSs are passive (we will explain the difference later in this text). Therefore having a firewall to work in conjunction with an intrusion detection system provides a better solution for securing a computer network.

2.2 Passive vs. Reactive IDSs

All Intrusion Detection Systems are passive devices. This is, they merely monitor the network packets and should a suspicious event be detected they trigger an alarm and log the occurrence in a database [8]. The alarm could be an e-mail sent to the administrator, a message that pops up on the screen, sound an alarm, a mobile text message, or a combination of all. In order to further investigate the nature of each event and determine its legitimacy an analyst must manually examine all logs and take appropriate corrective
actions such as reconfiguring the firewall or installing necessary system patches. We classify IDSs as passive because they take no action in preventing the attack. On the other hand we have reactive IDSs that we call Intrusion Preventions Systems (IPSs). They have all the characteristics IDSs do with one major difference – they are designed not only to recognize malicious network patterns but also to stop them. When an IPS senses an intrusion it can apply various protection mechanisms to neutralize the attack – reconfigure other security appliances (such as routers and firewalls), denying access to the source IP address, denying access to the target machine/service/application, or terminating an active user session.

Another significant difference between IDSs and IPSs is their location. IPSs watch the live network traffic (which is useful in early catching of viruses, worms, and Trojans) while IDSs only look at a copy of that traffic. This poses some additional problems. If an IPS detects valid traffic and it categorizes it as “bad” it will deny the source machine access to the network resources – users can be deprived of access to assets without a valid reason. In general, if an IPS is used there will be some latency in the network especially for large organizations. Since a decision must be made before a packet is allowed this adds some extra time for the traffic to reach its destination. Moreover, if the IPS stops working all incoming and outgoing traffic will halt which means that additional fail-save mechanism need to be implemented in order to ensure the traffic is not interrupted. Depending on the organization’s needs and goals administrators can deploy an IDS, an IPS, or both while considering all advantages and disadvantages of each approach.
2.3 Network-based vs. Host-based IDSs

In order to further classify IDSs we can look into the types of attacks each system recognizes and the number of devices they protect. In general, Network Intrusion Detection Systems (NIDSs), are responsible for protecting multiple hosts (or subnets of hosts) while Host Intrusion Detection Systems (HIDSs) are designed to specifically monitor single machines. Figure 1 shows a typical deployment of a NIDSs within a small business. A few different components comprise a NIDS:

- **IDS Sensor** – a hardware device responsible for monitoring the network packets as they arrive from the router and the DMZ switch.

- **Management Server** – after the packets are detected a copy is sent off to the Management Server which is accountable for performing preliminary analysis of the packets and categorizing them based on the types of attacks detected. This helps security officers to more easily examine the logs. The Management Server then logs the alerts in a Database Server which is some instances could be the same machine.

- **IDS Console** – this is a terminal which administrators use to log into the system and examine the alerts in greater detail. The IDS console could be any type of desktop or laptop and sometimes it might not be located inside the organization’s Intranet – this machine could be an outside host that connects to the Management Server via a secure VPN connection.
NIDSs are the more popular type of systems deployed within organizations nowadays. They are installed in multiple nodes of the internal network and are responsible for monitoring network traffic coming in or leaving various hosts. They investigate application, transport, and network layers traffic and determine if distrustful activities have occurred [4].

HIDSs, on the other hand, are installed onto single machines and designed to solely protect the host they reside on. But most importantly, HIDSs do not focus on scanning the network packets but instead they monitor the system behavior by constantly monitoring file modifications, changes in system resources (disc space, CPU, memory, cache, bandwidth), suspicious performance of various applications, changes in the event
logs, user activity, and audit trails of the machine. Since many attacks these days are focused solely on web servers or various web applications, Host Intrusion Detection Systems can be deployed and configured to specifically monitor the activities of a network service or application. If an atypical event is detected they generate an alert and log the event for further investigation.

It is apparent that both NIDSs and HIDSs are used for different purposes and understanding the basic functionality of each one can be very useful when deploying these systems within large enterprises. Due to the fact that both systems complement each other, vendors have begun designing hybrid solutions that include features of both systems. This makes it much easier to ensure optimal protection is achieved with less effort.

2.4 Distributed IDS

There is one more loosely-defined type of IDS – Distributed Intrusion Detection System (DIDS). Its primary use is in large organizations or institutions where protection of intellectual property is of critical importance (government facilities, data centers, banks, healthcare providers). In such networks we usually deploy several NIDSs or HIDS in the most significant nodes. Each NIDS usually monitors only a few devices (such as web server and FTP servers) instead of being responsible for the whole network. This way the traffic monitoring is segregated and the load on each IDS is decreased which leads to more accurate detection. All NIDSs report to a NIDS Management Station where alerts are logged and analyzed. In return, the Management Station can make configuration changes to each of the systems based on information about new potential threats. DIDSs
are very helpful against organized attacks because even if intruders manage to overcome one system the rest may still be functioning. Another useful application is protection against worms and viruses. If a virus is leaked into one subnet the NIDS Management System can take preventive measures to isolate the rest of the subnets ensuring they are not compromised.

Figure 2. Distributed Intrusion Detection System
2.5 Signature-based vs. Anomaly-based Detection

After we have examined the different types of Intrusion Detection Systems we must explain an important aspect of all types of IDSs – the detection mechanism. It is fundamentally important to acknowledge the differences of the two main types of detection – signature-based and anomaly-based. Signature-based detection is the more common approach used in modern IDSs. Its principle is based on signatures – patterns (or blocks) of information that represent a specific security risk [5]. These signatures are pre-defined by the vendors offering IDS solutions. Example signature definition may include the following:

- Numerous attempts of using network credentials to gain access to a target device, especially if the usernames are “root”, “admin”, or “administrator”. This could be a trait of a brute force attack.
- An incoming email with subject line “Monthly Invoice” along with an attachment named “MonthlyInvoice.exe”. This could represent a virus or a malware.
- Packets with forged IP headers might represent a spoofing activity
- Executing excessive amounts of identical commands in a short period of time could be a trait of a Denial of Service (DoS) attack.
- A submission of a web form that contains the string “or 1=1 --” could be a signature that represents the infamous SQL injection attack.

From the above examples we can deduce that signature-based detection works best when we clearly pre-define a set of potential harmful activities that may occur as a pattern in the network traffic. This method is very simple as it merely involves comparisons between strings in the network queue and signatures defined in the rule set of an IDS. As
we can easily see this approach is very good in recognizing known attacks but very ineffective is discovering new threads. The dependability of IDSs based on signature-based detection relies heavily on how well and how extensive these signatures are defined. With new attacks constantly emerging on the horizon it is critical that IDSs vendors quickly identify these threads, determine their essential signatures, and implementing them within their systems. This, of course, is a time-consuming process which adds even more overhead on companies providing Intrusion Detection Systems.

Anomaly-based detection, on the contrary, is a much more sophisticated approach for detecting intruders. The underlying concept is very close to the human behavior. First we define a “normal” state of a target device or a network node and then we monitor to see if any deviation from this “normal” behavior will occur. This process is also referred to as profiling because we must pre-define various profiles for users, hosts, services, applications or network segments. Usually it takes several weeks to complete a full and accurate profile before the system is deployed. This is necessary because we must collect as much information as possible in order to define what is normal or natural behavior. For example we can monitor the amount of network traffic originating of a particular machine or the number of daily emails sent by the same. Over time we would have a good understanding of the average amount for each activity which will be the base for future comparison. Should a host become infected with a worm, for example, which triggers sending large amounts of emails to various recipients the system will easily detect this behavior as abnormal and generate an alert.

There are two types of profiles used in anomaly-based IDSs – static and dynamic. Static profiles are built once before the system is deployed and are not changed during its
operation. This method will become more inaccurate over time since it is normal for a user (or a host) to take upon new responsibilities which are perfectly valid. Static profiles are prone to generate a large amount of false positives which is a big disadvantage for security administrators. Dynamic profiles are designed to adapt as the behavior of the target device changes slightly over time. They always stay current and produce a much smaller amount of false-positive. However, there is a significant problem with dynamic profiling. Attackers can use this approach to their advantage and act upon a target device in small fractions over time. Since this change in behavior is so gradual the IDS might consider this to be a normal deviation of the initial profile and instead of triggering an alert to begin adjusting its profiles accordingly. Over time intruders can “train” the system that accepting malicious traffic is actually “normal” and only a manual intervention of the security officer can correct that problem. With the complexity of modern networks increasing rapidly it becomes more challenging to establish correct profiles for all devices on the network.

The greatest advantage of the anomaly-based approach is that it can recognize novel attacks without the need of explicitly describing what constitutes an attack. While security threads might be different in their implementation, the results they cause are, in general, very similar. Therefore anomaly-based systems will easily recognize new, emerging attacks.

2.6 Wireless IDSs

It is evident that wireless networks have their own security problems and safeguarding them from the perspective of deploying an Intrusion Detection Systems requires a different approach. This is because in order to break into a Local Area Network (LAN) an
intruder must usually overcome one or more security devices while if someone wishes to
attack a Wireless Local Area Network (WLAN) they simply need to be in close
proximity to that network [4]. The majority of the wireless attacks involved intercepting
messages and extracting their information or inserting fabricated messages into the
network stream which can harm the endpoint devices (man-in-the-middle attack). The
types of systems that are intended to protect wireless networks are called Wireless
Intrusion Detection Systems (WIDSs).

The architecture of WIDSs and NIDSs/HIDSs is very similar in general but the
important difference between them originates from the way they perform their scanning
patterns. While NIDSs detect and examine every packet that comes in or leaves the
network Wireless Intrusion Detection Systems can only look at segments of the traffic.
This is because wireless network sensors are capable only of monitoring one channel at a
time for each of the two possible frequency bands (2.4 GHz and 5.0 GHz). The obvious
problem here is that parts of the wireless traffic are never examined thus vulnerabilities
might not be detected. In order to remedy this problem, an approach called *channel
scanning* is used which triggers the sensors to constantly switch between channels a few
times per second so that the majority of the packets are captured[7].

Since WIDSs work solely on the WLAN protocol level they provide more
accurate results than network and host-based ones and provide fewer false-positives. At
the same time operating on only one protocol is a disadvantage since these systems are
limited in the types of attacks they can detect. The list below consists of the most
common types of events that a WIDS can recognize:
• Unauthorized access to WLANs – when intruders are trying to gain access of the wireless network without possessing the proper credentials

• Weak, misconfigured, or completely unprotected WLANs – WIDSs can detect if network administrators have left “holes” in the wireless system while configuring it.

• Unusual network patterns – very much like HIDSs, WIDSs can sense a change of the usual traffic to and from a particular host which can trigger an alert.

• Detection of network scanners – wireless network scanners are very popular tools for detecting open or poorly protected wireless networks. WIDSs can detect that so that administrators can correct the problem.

• Denial of Services Attacks – this type of attack exists in all types of networks and is characterized by sending large amounts of information to a particular host trying to “overwhelm” that device so it stops operating

• Man-in-the-middle attacks – as we already mention earlier this is introducing false messages into the wireless network which can lead to compromised the end node devices.

While WIDSs are still maturing and constantly improving their operations implementing them certainly adds additional protection to each wireless network.

2.7 IDSs Software Implementations

There are various developments of Intrusion Detection Systems on the current market which are mostly divided into commercial and open-source products. Among commercial products some of the most popular brands are Symantec Critical System Protection – a
hybrid network IDS (www.symantec.com), TripWire – a HIDS which does an excellent job in monitoring systems files’ integrity (www.tripwire.com), and BlackICE – a NIDS with a built-in firewall (www.networkice.com). While these products are a good choice for a system that will protect an organization against hackers, they could be very expensive, costing in the tens of thousands of dollars. For companies who are on limited budgets a good starting option would be considering open-source Intrusion Detection Systems. As the open-source community rapidly expands and develops more software then we can easily find mature and reputable systems that have been around for a long time and have been adopted by many enterprises. A few of the leaders in open-source IDSs development are Snort (www.snort.org), Bro (www.bro-ids.org), BASE (base.secureideas.net), and OSSEC (www.ossec.net).

In this thesis we will use Snort IDS to examine its operations in greater details as well as implement and test various algorithms for finding malicious patterns.

Finally, we must note that there are various hardware implementations of Intrusion Detection Systems but the focus of this text is solely on the software solutions.
Chapter 3

Snort Intrusion Detection System

3.1 Overview of Snort IDS

Snort is an open-source IDS/IPS developed by Sourcefire (www.sourcefire.com). It’s founder is Martin Roesch who developed the first release of Snort in 1998. Since then the system underwent many changes and evolved from a basic packet analyzed into a full-featured IDS adapted by thousands of organizations. Snort is a hybrid IDS that performs signature and protocol-based packet inspection on the network level combined with anomaly-based detection on the system level. It interacts with many other third-party open-source projects that provide various additional functionalities such as grammar correction tools, output systems, rules management tools, etc. The system is available for Microsoft Windows and Linux distributions. There are many Graphical User Interfaces (GUIs) available for Snort such as Basic Analysis and Security Engine (BASE – http://base.secureideas.net) and Snorby (www.snorby.org).

Snort IDS runs into four main modes and depending on the exact needs can be used in various scenarios:

- **Packet Sniffer Mode** – this is the most basic use of the application. The system simply captures network packets and display it in the output window.

- **Packet Logger Mode** – this mode will not only read the network traffic but also record it into a database for later inspection.

- **IDS Mode** – a full-featured Intrusion Detection System that performs packet analysis based on the available rules.
• IPS Mode – when Snort is configured in this mode it will allow for packets to be dropped if a positive match against a known signature is found.

3.2 Snort Architecture

The architecture of Snort IDS is very similar to other Intrusion Detection Systems. First Snort needs to capture the network packets. Since it doesn’t have a native interface for this purpose it relies on the Libpcap application which is a Unix-based Application Programming Interface (API) for low-level packet sniffing. Microsoft Windows systems can use the WinPcap library instead. It is important to mention that Snort specifically captures raw packets on the data-link layer level, before important header information will be stripped off by the operating system. The header information is useful in many cases when determining various types of attacks, such as spoofing attacks.

Figure 3. Architecture of Snort IDS

Next, the Packet Decoder begins extracting information about each packet starting from the Data Link Layer all the way up to the Application Layer (for more information about the seven Open System Interconnection (OSI) Layers see http://visitdocwiki.cisco.com/wiki/Internetworking_Basics). The decoded data is being
stored in an object and passed onto the preprocessors. Preprocessors play a very important role in every IDS. As we already discussed one of the main disadvantages on Intrusion Detection Systems is they are not that effective against attacks not based on signatures. Therefore Snort uses a pool of processes (preprocessors) that examine the packets before they reach the Detection Engine and checks if any suspicious behavior is detected that cannot be recognized by signature matching. Preprocessors also play a role in normalizing the network traffic after it has been decoded so that it can be analyzed by the Detection Engine more easily. This is important because there are many known attacks that are specifically designed to target the scanning engine of an IDS by injecting malicious patterns, also known as mimicry attacks [18]. When traffic is normalized, most of these attacks become futile. Another advantage of preprocessors is that users can write their own and implement them within their systems.

The most significant component of Snort, and any other IDS, is the Detection Engine. It has two primary responsibilities – to parse the application rules and to perform signature detection. Before Snort initializes it must read each rule line by line and load them into the memory. New rules cannot be introduced while the engine is running. (In the next sub-chapter we examine and explain Snort rules in more detail). Based on the currently present rules the Detection Engine begins to scan each network packet and determine if there is a signature match. Should a match occur it is then being logged in the Alert Database and possible outputted on the screen. For high-volume networks the Alert Database can grow large very quickly thus proper purging and archiving should be standard operating procedure.
3.3 Snort Rules

Snort rules are statements that indicate what type of information we are looking for in a string. They could contain one or more signatures (patterns to be matched) that describe a specific network or system behavior. Each rule is divided in two parts: rule header and rule option. The rule header contains information such as the specific action to be performed (alert, log or drop), the applicable protocol (TCP, UDP, ICMP), source/destination IP addresses, CIDR block and port. The signature, alert messages and description about which portion of the packet should be inspected comprise the rule option. One can change the order in which the rules are loaded or disable unnecessary rules. Currently there are over 11,000 rules that are available on www.snort.org [37].

Snort users are also free to write their own rules or to modify the existing ones. In Figure 4 we present a sample Snort rule that will detect a user’s attempts to issue the “su” command in order to gain root access through a telnet session.

```
alert tcp $TELNET_SERVERS 23 -> $EXTERNAL_IP any (msg:"TELNET Attempted SU from wrong group"; content: "su root"; flow: from_server,established; nocase; sid:715; rev:6;)
```

**Figure 4.** Sample Snort rule detecting attempts from a telnet session

Below we will identify the different components of this rule. The part up to the left parenthesis is the rule header and the part inside the parenthesis is the rule option.

- The command “alert” is the action to be taken when the system detects a packet matching the rule description. In this case the action is simply to generate an alert.
- “tcp” means that this rule will only examine TCP traffic
- The variable “$TELNET_SERVERS” contains a list with IP addresses of all available telnet servers in the network. This variable is defined in the configuration file `snort.conf`.

- The number “23” correlates to the port number the rule will be applying to. Port 23 is the telnet port for unencrypted text communications.

- The variable “EXTERNAL_IP” corresponds to all the external IP address outside the private network. This means an alert will be generated if the user issuing the “su” command is outside the internal network. Internal requests will be allowed.

- The keyword “any” defines that any port should be match for any of the IPs listed in variable “EXTERNAL_IP”.

- Next we have the message to be displayed as part of the alert when a positive match is found.

- The “content:” option identifies what keyword(s) must be present in the packet in order to issue an alert.

- The “flow” argument allows us to apply direction to the network traffic. In our case we’re applying the rule on traffic flowing from the server.

- The option “nocase” ignores the case type in which the statement was written.

- The “sid” keyword is used to uniquely identify each Snort rule. It is useful when using outside plugins that need to interact directly with a rule.

- The “rev” keyword identifies the revision of this rule. It is important because this is how signatures are update with the most recent information.

From the above example we can conclude that the granular construction of Snort rules allows us to be flexible in our rules implementation so that we can be looking for very
specific types of information. This helps the speed of the search algorithm as well as produce less false-positive alarms.
Chapter 4

Traits of Pattern Matching Algorithms

There are many pattern matching algorithms that are subject to research by the Computer Science community. Naturally, not all algorithms will be applicable to Intrusion Detection Systems – some might be too slow, other might produce more false-positives, a third set may take lots of system resources. Therefore we must take into account several different traits that are important when developing or implementing algorithms that will be used as security mechanisms in computer networks.

4.1 Overview of Pattern Matching

Pattern matching is considered simultaneously a science and art. It is a complex field that encompasses many aspects of Computer Science and Computer Engineering. The actual definition of pattern matching, however is quite simple. Given a set of keywords and an input string we need to find all instances of any keyword in the input string [13]. We also must clarify that the keyword set must possess two properties – must be non-empty and finite. The first requirement that we impose on the set of keywords, to be non-empty, is trivial but necessary since pattern matching by definition cannot be performed against empty sets. Moreover, that set needs to be finite so that we can eventually produce meaningful results and determine if a match has occurred. When applying this concept to Intrusion Detection Systems our set of keywords will be the set of signatures loaded in the detection engine and the input string will be the packets stream that enters the IDS. Even though the data flow to be processed is uninterrupted and could be considered
infinite, the pattern matching process performs data segmentation (in the IDS pre-processor) so that we feed finite sets of packets into the engine.

Besides Intrusion Detection Systems, pattern matching has many applications such as parsers, spam filters, digital libraries, word processors, search engines, and many more. A well-known pattern matching application is grep – a Unix-based plain-text searching utility. It is used to match any text or regular expression to an input. The syntax is very simple as we present two examples below. In the first one we search for any occurrence of the string “csun” in the input file “sample.txt”. In the second example we output all lines that begin with “cs” followed by any character, followed by “n”.

grep csun sample.txt

grep ^cs.n sample.txt

There are various implementations of grep which use different algorithms but the two most used are the Knuth-Morris-Pratt algorithm and the Aho-Corasick algorithm, both of which will be discussed in detail in the next chapters.

4.2 Signature Set Size

The size of the set of keywords divides pattern matching algorithms into two categories – single-keyword algorithms and multiple-keyword algorithms [13]. Initially, all pattern matching procedures started as simple approaches where the keyword size was only one. While they might have been perfectly suitable for certain applications (word processors) they were notorious for not being fast and efficient enough for other applications (search engines, IDSs), which required more sophisticated tactics. Despite the fact that single-keyword algorithms have hardly any presence in modern network security systems they
are fundamentally important to establish a good foundation of the intricacies of pattern matching.

Multiple-keyword algorithms have been a subject of extensive research in the past decades and many implementations, improvements, and variations have been proposed. The ability to search for multiple signatures simultaneously is critical for Intrusion Detection Systems because, as we previously stated, the increase of attacks against networks require generating a large set of signatures which need to be quickly matched in the constant stream of information. Also multiple-keyword algorithms have theoretically and practically lower complexities than their single-keyword predecessors which makes them good candidates for implementation into a NIDS. Especially when dealing with large sets of keywords. Currently Snort has thousands of signatures describing various malicious patterns. This is a relatively large set considering the performance requirements for an algorithm to be able to recognize a pattern almost instantaneously in this pool of signatures. Taking this in consideration, not all pattern matching algorithms are designed to perform well with such large keyword sets. This is why engineers have specifically adapted methods that are very efficient with large keyword sets but not necessarily with small ones. This poses a problem because in certain situations, where variables change dynamically, we are unable to use only one approach. Therefore we must carefully evaluate the environment in which algorithms will be used so that they yield optimal results.

4.3 Alphabet Size
The alphabet size also plays an important role when selecting a pattern matching algorithm to be implemented into an NIDS. Since the alphabet in Intrusion Detection
Systems are bound by the length of the bytes from the network flow, it is easy to determine that we will be dealing with a 256-character alphabet (a byte contains eight bits each of which can have two possible values – 0 or 1, hence $2^8=256$). These 256 values represent every possible keystroke on a regular keyboard including some values reserved for the system itself. This alphabet is indeed quite large and we can compared it to other alphabets to see the difference. The English alphabet, for example, has 52 possible values (uppercase and lowercase letters) and the DNA sequencing alphabet has only four possible values – A, C, T, and G. This means the approach to solving the pattern matching problem in terms of efficiency and accuracy is much more difficult than if we are creating algorithms applicable in other topics. Raffinot [16] has done an extensive research on this topic and has proposed which algorithms perform best when dealing with small and large alphabets respectively. For example, the Boyer-Moore algorithm’s bad character shifting is not very practical when applied to small alphabets, but is very useful when dealing with larger ones such as the ASCII alphabet (128 characters). We will cover some of these algorithms in greater depth in the next chapters.

### 4.4 Keywords Length

The length of the keywords (the descriptive patterns by which we identify intrusions) used to perform the pattern matching process also play an important rules in the algorithmic performance. It is evident that the greater the keyword length – the slower the algorithm will be and the more system resources will be needed to execute the comparison. Some keywords can be very small - an order of only a few bytes because they primarily deal with IP addresses. Other patterns can be very lengthy – over 50 bytes. This is because as attacks become more organized and complex, a greater rule description
is required to match all the possible attributes of the attack. Since we cannot risk compromising the security of a network, we have to be as descriptive as we can when defining a rule in any Pattern Matching System (see Chapter 3.3 for more information about rules). Figure 5 represents a distribution scale for the number of rules in Snort versus the length of each string in a Snort rule in bytes. We can observe that the mean string length is around 15 bytes which is still a relatively long value for some algorithms to cope with.

![Figure 5. Distribution of the string length in the Snort default database](image)

However there is another interesting fact that we can extrapolate from the figure above. Most of the attack definitions (keywords) are in the range of 4-16 bytes which means that these are known signatures that have been present for a long time. This translates into the fact that the majority of network attacks have relatively short patterns which is beneficial because it takes less effort (processing time) to detect them. As for any other criteria, there are string-matching algorithms that perform well when the shortest keyword is indeed short, while other are only beneficial when dealing with only long patterns. For that purpose we can use a simple technique to generally improve the runtime of the algorithm. If we know we are implementing a method that is very quick
with long keywords then we can combine all short-keyword rules into fewer, long-keyword rules without compromising the detection ability of the engine [17]. The same is true about the opposite scenario – we can break up long descriptive rules into smaller ones and feed them into detection engines that work primarily with short strings.

4.5 Computational Complexity

When we evaluate an algorithm based on the criteria we previously presented (keyword size, alphabet size, signature set size) we can measure the performance of this algorithm in a theoretical and practical environment. In Computer Science we refer to computational complexity as the worst-case time that an algorithm can take in order to solve a given problem [19]. Average-case and best-case times are also studied as they provide valuable information of the algorithmic performance in specific scenarios but we are usually most interested in the worst-case scenario since it clearly defines the limitations of the algorithm in practice.

In order to understand why studying the algorithmic complexity is important we have to examine the types of attacks that are launched against Intrusion Detection System. Probably the most popular type of attack is the Denial of Service (DoS) attack which can transpose into a Distributed Denial of Service (DDoS) attack. Its purpose is to target the detection engine of the IDS by flooding it with an extreme amount of packets in order to either significantly slow down its performance to the point that many packets will be missed or to completely disable the scanning process. But there is a specific subset of DOS attacks that do not use the traditional method of deluging the network device. Instead, they use a very small amount of information that is specifically designed to invoke the worst-case behavior of the algorithm in a recursive manner. These attacks
are classified as algorithmic complexity attacks [21]. They can be especially dangerous and hard to detect because the incoming packets do not have to contain malicious signatures.

Figure 6 depicts a hash table operation when incoming packets are distributed and stored into the network system. Instead of the input (network packets for example) is distributed evenly among the hash buckets, all the packets end up in the same bucket, resulting in a lengthy linked list. If the linked list becomes too large and the system resources cannot continue to store the information, then we can expect some unexpected behavior of the system such as dropping packets. Smith and Estan presented the backtracking attack, which is an algorithmic complexity attack that can slow down the rate of signature matching to 1.5 million times than the average rate [20]. In practice this would mean that the IDS will be considered non-operational, especially for high-traffic networks.

![Figure 6. Normal (left) and worst-case (right) operation of a hash table.](image)

In order for someone to launch an algorithmic complexity attack against an IDS they must have prior knowledge about the type of algorithm deployed in the system. Since there are many pattern matching algorithms not all will be susceptible to the same input. Commercial IDS vendors constantly modify and improve their algorithms and keep them
secret so that they do not end up in the possession of a hacker. At the same time there are many open-source IDSs that use known string matching algorithms which are widely available to the general public. This makes it easier for attackers to acquire this information and use it against network systems.

In this chapter we presented various factors that play important roles in deciding which algorithm is best to be used in a particular scenario. The bottom line is there is no single, best algorithm and we need to aim towards carefully evaluating the network and system environments so that an optimal choice is made. In the next two chapters we are going to present some of the most popular types of algorithms used in Intrusion Detection Systems and examine their strengths, weaknesses, and application.
Chapter 5

Single-Keyword Pattern Matching Algorithms

We use single keyword pattern matching algorithm to find all occurrences of a specific keyword in a given input. Due to the fact that the size of the keyword set is one, we can deduct that the application of these types of algorithms will be very limited in modern Network Intrusion Detection Systems since it is infeasible to construct only one signature that encompasses all known malicious patterns. Therefore the use of single string algorithms is limited to instances such as data processors, text editors, search engines, images analyzers, and parsers. Nevertheless, understanding their foundation and construction is essential in order to be able to design robust multiple keyword algorithms. In this chapter we will examine the most popular single phrase pattern matching algorithms.

5.1 Brute Force (BF) Algorithm

Perhaps the most basic method of approaching the problem of pattern matching is the Brute Force (BF) algorithm. This technique is very simple and easy to follow. Let’s assume we have text (input) $T$ with length $n$ and a pattern (keyword) $P$ with size $m$. The algorithm begins by comparing the pattern to the text, scanning left to right, one character at a time, until there are no more matching characters. If a mismatch occurs, the algorithms shifts the pattern one character to the right. Here is the generic pseudo-code of the algorithm:
**Step 1** Align pattern at beginning of text

**Step 2** Moving from left to right, compare each character of pattern to the corresponding character in text until:
- all characters are found to match (successful search); or
- a mismatch is detected

**Step 3** While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

In the example below the pattern “text” is to be found in the text T. The algorithm will search to see if the first character of the pattern - “t” – occurs anywhere in the text. We underlined these instances for better visualization. If there is a match the algorithm begins comparing the second character of the pattern – “e” to the next position of the text. If this time we encounter a mismatch we label this instance as a *false start*. The process continues until we reach the end of the string.

T: **In this text we present the Brute Force algorithm**

P: **text**

The Brute Force algorithm is very easy to implement. First we have a *for loop* with a variable \( i \) that keeps track of the difference between the lengths of the text and the string. This is because even if a character match is found there might not be enough characters at the end of the string to account for a complete pattern match therefore we can abandon the search. The *while loop* starts matching symbols from the patterns to the text, one at a time. If there is a character match it bumps the index \( j \) to the next symbol until no more matches are found.
public static int brute(String text, String pattern){
    int n = text.length(); // n is length of text
    int m = pattern.length(); // m is length of pattern
    int j; // pointer to keep track of the position
    for(int i=0; i <= (n-m); i++) {
        j = 0;
        while ((j < m) &&
            (text.charAt(i+j) == pattern.charAt(j)) )
            j++;
        if (j == m) // match at i
            return i;
    }
    return -1; // no match
}

Algorithm 1. Java implementation of Brute Force algorithm

It is not difficult to see that this approach is very trivial and naïve. The worst-case performance of Brute Force is O(mn) since the for loop is called at most n-m+1 times and the while loop is executed m times for every iteration of the outer loop. In the average searches, however, the computational time will be close to O(n+m), especially when the alphabet is rather large. This is because in large-alphabet languages we are much more likely to encounter repetitive strings. If the alphabet is of a smaller size (the binary one for example), the performance significantly decreases as the below example show.

Example of a worst case:
- T: "aaaaaaaaaaaaaaaaaaaaaaaaab"
- P: "aaaaab"

Example of an average case:
- T: "this is an average string matching example"
- P: "store"
Considering the worst-case running time of Brute Force we can conclude that is not a good choice for an algorithm that could be implemented in an IDS but it can successfully be used in other applications.

5.2 Knuth-Morris-Pratt (KMP) Algorithm

This algorithm was introduced by Don Knuth, Jim Morris, and Vaughan Pratt in 1977. It is quite similar to the Brute Force approach regarding scanning the text left to right, however we are now using information from the previously compared characters in order to determine the maximum possible shift of the pattern to the right. The idea is to avoid comparisons with elements from the text \( T \) that have previously been compared with some elements of the pattern \( P \). In order to achieve this task, KMP preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself. The pre-calculation is done in time \( O(m) \) and is called the \textit{next function} \( F[j] \) [13]. This function is an array that represents the size of the largest prefix of \( P[0...j] \) which is also a suffix of \( P[1...j] \). The KMP algorithm states that the most we can shift the pattern in order to avoid redundant comparisons is namely the length of the \textit{next function}.

1. Algorithm nextFunction(P)
2. \( F[0] = 0 \)
3. \( i = 1 \)
4. \( j = 0 \)
5. while \( i < m \) //we have a continued substring in \( p \) from the start of \( p \)
6. \( \text{if } P[i] == P[j] \)
7. \( F[i] = j + 1 \)
8. \( i++ \)
9. \( j++ \)
10. \( \text{else if } j > 0 \) then //use nextFunction to shift \( P \)
11. \( j = F[j - 1] \)
12. \( \text{else} \)
13. \( F[i] = 0 \) //no match, we weren’t in a substring
14. \( i++ \)

\textbf{Algorithm 2.} Knuth-Morris-Pratt \textit{Next Function}
Algorithm 2 represents the pseudo-code of the *next function*. It uses a simple *while* loop to find repeated substrings of the pattern itself which are prefixes of the pattern. This way the function generates knowledge about how the pattern matches against shifts to itself. It then logs the results in a table which will be used during the second stage of the algorithm – the string matching. The first table values of $F[j]$ are always fixed – namely “-1” and “0” so the construction of the table actually begins at position $j[2]$. This is so if we have a mismatch in the very first character we simply slide the pattern one position to the right. Table 1 shows the final computation of the *next function* based on the pattern “abcacadabca”. The variable $i$ represents the indexes of $P$ and $j$ will be the indexes of the text $T$. Remembering the objective of $F[j]$ – to find the largest prefix of the pattern $P[0\ldots j]$ that is also a suffix of $P[1\ldots j]$ – it is easy to see that this string is namely “abca”.

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[j]$</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>$F[j]$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

*Table 1.* Pre-computational table of the *next function* in the KMP algorithm

Next, we will take the example below and trace how the algorithm works. We can follow the pseudo-code in Algorithm 3 for reference.

- $j = 012345678901234567890123$
- $T = abca abcacad abcacadabca$
- $P = abcacadabca$
- $i = 01234567890$

We start matching the characters left to right, like we did in the BF algorithm. The first mismatch occurs at $T[4] \neq P[4]$. Here instead of shifting the pattern by one, we use the pre-computed value of $j$ in Table 1 to find out what is the maximum amount of shifting we can perform. We calculate the value of $j$ by using the formula $j = j + i - F[j]$. In this case we have $j = 0 + 4 - 1 = 3$. The initial position of $j$ is 0 and the position where the
mismatch occurred is $i = 4$. The index $F[i]$ is taken from the pre-computed table by the *next function*. From this formula we see that the new value of $j$ is $j=3$ which means we slide the pattern across to $j=3$ (new starting point) and we reset $i=0$ (the point of failure).

- $j = 012345678901234567890123$
- $T = abca abcacad abcacadabca$
- $P = abcacadabca$
- $i = 01234567890$

In the next iteration we repeat the procedure and see that there is a mismatch at $T[4] \neq P[1]$. Here the shifting equation becomes $j = j + i - F[j] = 3 + 1 - 0 = 4$. Therefore we only shift the pattern one point at $j = 4$.

- $j = 012345678901234567890123$
- $T = abca abcacad abcacadabca$
- $P = abcacadabca$
- $i = 01234567890$

The next iteration is trivial and exactly the same as the previous step therefore we can omit the calculation where $T[4] \neq P[0]$ and only show the result.

- $j = 012345678901234567890123$
- $T = abca abcacad abcacadabca$
- $P = abcacadabca$
- $i = 01234567890$

The next matching fails at $T[12] \neq P[7]$ (see above) since the empty character doesn’t equal to “a” and we calculate the new shifting value of $j$. We can notice that in this iteration we are making a significant jump since $j= 2 + 7 - 1 = 8$. 
The next step is also trivial, when we find a mismatch between “space” and “a”.

Finally, at \(j=13\), we encounter a complete match of the string with the pattern. The algorithm successfully matches all the positive comparison until the end of the string and reports that the string is found. If we count the number of comparisons we can see that it took only 28 in the KMP algorithm compared to 154 when using Brute Force. This is a significant improvement in the computational time. The main algorithm’s pseudo-code can be seen in Algorithm 3. It consists of one \textit{while} loop and uses the pre-computed indices by the \textit{next function}. If a substring mismatch occurs the algorithm uses the values of \(j\) to find out how much to shift the pattern to the right.

The complexity of KMP is \(O(m+n)\) since the construction of the table \(T\) takes \(O(m)\) time and the string matching process takes \(O(n)\) time. As the alphabet size increases the performance of KMP decreases. This is because with large alphabets we will encounter more mismatches.
1. Algorithm KMPMatch (T, P)
2. \( F = \text{nextFunction}(P) \)
3. \( i = 0 \)
4. \( j = 0 \)
5. while \( i < n \)
6. \( \text{if } T[i] = P[j] \)
7. \( \quad \text{if } j = m - 1 \)
8. \( \quad \text{return } i - j \) //match
9. \( \quad \text{else} \)
10. \( \quad i++ \)
11. \( \quad j++ \)
12. \( \text{else} \)
13. \( \quad \text{if } j > 0 \)
14. \( \quad j = F[j - 1] \)
15. \( \quad \text{else} \)
16. \( \quad i++ \)
17. \( \text{return } -1 \) // no match

**Algorithm 3.** Knuth-Morris-Pratt Matching Algorithm

### 5.3 Boyer-Moore (BM) Algorithm

Developed in 1977 by Bob Boyer and J Strother Moore, the Boyer-Moore algorithm shares some similarities with the KMP algorithm but at the same time introduces three novel ideas in order to approach the pattern matching task more efficiently. These ideas are:

- Matching the pattern backwards (right to left scan)
- The *bad character shift* rule
- The *good suffix shift* rule

**Right to Left Scan**

Understanding the right to left scan is similar to the traditional left to right scan used by the previous algorithms we already introduced. Instead of matching characters at \( T[0] == P[0] \) we begin by matching the rightmost symbol of the pattern to the corresponding
symbol in the text (this is also known as the *looking glass* technique). At this point it is unclear why this approach is better because without introducing the other two methodologies there is no real advantage and the algorithm will still run in time $O(mn)$.

The following example illustrates the right to left scan approach. The character comparison begins at position 5. Since there is a mismatch at $T[4] \neq P[4]$ we start shifting the pattern to the right and again we start comparing the last character of the pattern with its corresponding index in the text. In our case we will be comparing $T[5] == P[4]$.

- $j = 01234567890$
- $T = abbadabacba$
- $P = babac$ (before shift)
- $P = babac$ (after shift)

The actual shift amount will be determined by the *bad character shift* rule and the *good suffix shift* rule therefore we must examine these concepts next.

**Bad Character Shift Rule**

The idea here is simple to understand. Let’s say that the rightmost character in $P$ is $x$ and the corresponding character in $P$ is $y$. Initially the text and pattern are aligned to the left and we can see that a mismatch occurs in position 5 since $x \neq y$. In this case it will be helpful if we know in advance the rightmost position of $x$ in $P$ because then we can safely shift the pattern as many positions as it takes to align $x$ in $T$ with the rightmost occurrence of $x$ in $P$. Hence we know that when we align the pattern and the text this way we will immediately get a match at least for the first symbol. In our case after the shift is complete we have “b”=“b”.

- $j = 01234567890$
- $T = abbababacba$
- $P = babac$ (before shift)
P = babac (after shift)

Furthermore we can examine a case when upon a mismatch at \( x \neq y \), if \( x \) does not appear anywhere in the pattern then we can safely shift past the point of mismatch without having to compare any of the intermediate information. In cases such as this one the matching runs at a sub-linear time due to the large pattern jumps.

- \( j = 01234567890 \)
- \( T = abbadabacba \)
- \( P = \text{babac} \) (before shift)
- \( P = \text{babac} \) (after shift)

We can formally define the *bad character shift* procedure as follows. When a mismatch occurs at \( T[i] = j \) two scenarios may follow:

- If \( P \) contains \( j \), then shift \( P \) to align the last occurrence of \( j \) in \( P \) with \( T[i] \)
- Else, shift \( P \) to align \( P[0] \) with \( T[i + 1] \).

In order to achieve this goal we must pre-process the pattern \( P \) against the alphabet \( \Sigma \) in order to build a special mapping called the *last occurrence* function \( L[22] \). We define \( L(c) \) as:

- The largest index \( i \) such that \( P[i] = c \) or
- -1 if the index does not exist

The below example illustrates the pre-computation of this table. It is essentially an array of indices of the characters in the alphabet and their rightmost occurrences in the pattern.

\[ \Sigma = \{a, b, c, d\} \]
\[ P = \text{abacaba} \]
We must note that when implementing this concept in Intrusion Detection Systems we will always be dealing with an alphabet size of 256 (the extended ASCII table) therefore the size of our table will also be 256. The computation of this table occurs in time $O(m+s)$ where $m$ is the length of the patterns and $s$ is the size of the applicable alphabet.

**Good Suffix Shift Rule**

Even though the *bad character shift* rule is highly efficient it does not work all the time since there will certainly be cases where upon a mismatch the character where the mismatch occurred will appear somewhere else in the text. This is especially true with small alphabets such as the DNA sequencing alphabet. In order to deal with this problem we introduce the *good suffix shift* approach. The idea here is that we want to use knowledge of the previously matched characters in the pattern’s suffix. In other words, if we already matched $t$ characters in the text $T$ we need to find the smallest shift in the pattern that will align a substring of $P$ of the same $s$ characters. If we label the already matched sub-string in $T$ with $t$ and the mismatched character in $T$ with $x$ then in case of a mismatch we shift the pattern to the right until the first occurrence of $t$ in $P$ such that the next character $y$ in $P$ holds the property $y \neq x$. Otherwise, we shift the pattern right to the largest prefix of $P$ that aligns with a suffix of $t$. In order to keep track of these shifting trends we must create another table that will keep track of how many characters we have already matched.
Let’s take the example below. We begin matching the string right to left and we find two
matches in the first two characters. But the next character is a mismatch at T[7] ≠ P[7].
Since we have already matched a suffix of the pattern which is namely “ab” we can shift
the pattern to the right until that same substring “ab” aligns with its corresponding match
in the text. We must note that there are two more occurrences of “ab” in the pattern and
we are only allowed to shift the pattern to the rightmost one.

- $j = 01234567890123456789012345$
- $T = \text{bbacdcba}abcddcdaddaabcdba$
- $P = \underline{cabbabdbab}$ (before shift)
- $P = \underline{cababdabab}$ (after shift)

The second case that might appear when the **good suffix shift** rule (see Algorithm 4) is
applied is when there is no full sub-string alignment within the pattern. If we look at the
below representation we can see that we have already matched three symbols – “abc”.
Next we see a negative match at T[7] ≠ P[7]. We examine the pattern to see if the suffix
“abc” appears anywhere and it clearly does not. Therefore we start chopping off the
pattern in order to find the next largest substring that can be matched in the pattern. In our
case this sub-string is “bc” therefore we align the pattern accordingly so that the largest
suffix of P aligns with its corresponding sequence in T.

- $j = 012345678901234567890123456$
- $T = \text{bbacdcba}abcdbdabcaabcdba$
- $P = \underline{bcbbabdbabc}$ (before shift)
- $P = \underline{bcbbabdbabc}$ (after shift)
1. void occFunction() {
2.     char a;
3.     int j;
4.     for (a = 0; a < alphabetsize; a++)
5.         occ[a] = -1;
6.     for (j = 0; j < m; j++){
7.         a = p[j];
8.         occ[a] = j;
9.     }
10. }

Algorithm 4. Boyer-Moore bad character pre-processing

To summarize, the Boyer-Moore algorithm works by calculating two preprocessing tables to determine the next appropriate alignment after each mismatch. The first table calculates how many positions ahead of the current position to start the next search (bad character shift rule). The second table makes a similar computation indicating how many characters were matched successfully before a mismatch (good suffix shift rule). Based on these two ideas we can perform a quick string matching following the example in Figure 7.

```
  a  p  a  t  t  e  r  n  m  a  t  c  h  i  n  g  a  l  g  o  r  i  t  h  m
```

Figure 7. Example of Moyer Moore Pattern Matching Algorithm

Initially, we have a mismatch during the first comparison. Since “t” exists in P we use the bad character shift rule to move the pattern two positions in order to re-align the mismatched character with its rightmost occurrence in P. Next, we compare “e” and “m”
in our second comparison. Clearly they’re not equal and since “e” doesn’t appear anywhere within P we can shift the pattern past “e”. It is trivial to follow the rest of the algorithm and to observe how to string is matched completely. Algorithm 5 shows the pseudo-code of the BM algorithm.

| 1. Algorithm BoyerMooreMatch(T, P, S) |
| 2. L = occFunction () |
| 3. i = m - 1 |
| 4. j = m - 1 |
| 5. while i > n - 1 |
| 6. if T[i] = P[j] |
| 7. if  j = 0 |
| 8. return i //match at i |
| 9. else |
| 10. i = i - 1 |
| 11. j = j - 1 |
| 12. else //character-jump |
| 13. l = L[T[i]] |
| 14. i = i + m – min(j, 1 + l) |
| 15. j = m - 1 |
| 16. return -1 //no match |

**Algorithm 5.** Pseudo Code of Boyer-Moore Algorithm

The best-case for the Boyer-Moore algorithm is attained if at each attempt the first compared text symbol does not occur in the pattern. Then the algorithm requires only $O(n/m)$ comparisons due to the large shifts in the text input. The worst case is still $O(nm)$ which happens when dealing with small alphabets [24]. This implies quadratic complexity which is far than ideal. The pre-computation of both tables happens in $O(m + |\Sigma|)$. These results make the Boyer-Moore algorithm faster than the KMP algorithm and the advantage increases with increasing the size of the alphabet. As far as space requirements we need two tables pre-computed which reveals complexity of $O(m + |\Sigma|)$. 
5.4 Karp-Rabin (KR) Algorithm

The Karp-Rabin Algorithm was created by Michael Rabin and Richard Karp in 1987. They used a completely different approach than the single-keyword methods we already discussed in this chapter. The main idea is that instead of using comparisons it involves mathematical computations which more specifically extends to the notion of hashing. The application of hashing (converting each string into a numeric value) has always been a useful approach when it comes down to string matching because we can use it in order to test if two strings are the same. If both words have different hash values then we can be certain they are different [27]. But if their hash values are the same we cannot conclude they are the same string and will have to perform further comparisons (usually via Brute Force). The reason is because it is practically impossible to assign a unique value to every single substring derived from a 256-character alphabet. In order to keep the hash values to a manageable set we must allow for two different strings to map to the same hash bucket in the associative array which mean we assume that hash collisions will rarely occur. We formally define these collisions with \( h(x_1) = h(x_2) \), where \( x_1 \neq x_2 \). We can significantly decrease the number of hash values used by applying the modulo operation (mod).

At first look it does not seem logical that hashing is applied in pattern matching because the hash function is applied to each substring of the text which implies an algorithmic complexity of \( O(nm) \). Although there is no way to improve the hashing speed for a single character we can improve this process for groups of characters that share similar properties. For this purpose we can represent each of the extended ASCII characters with their decimal values – “a” = 97, “b” = 98, “c” = 99 and so on. The Karp-Rabin algorithm states that for a pattern \( P \) with length \( m \) we need to calculate the hash
function of every possible $m$-character substring in the text $T$, and compare if it is equal to the hash value of the pattern itself [26]. We can represent the hash function by $h(k) = k \pmod{q}$ where $q$ is a large prime integer. Each hash value is then calculated based on the value of the substring in the previous position hence the computation is done in time $O(1)$ for each substring. We can see exactly how this is done by reviewing the general pseudo-code of the Karp-Rabin approach in Algorithm 6.

Now we will examine how the algorithm works with the following example:

- $T = \text{aabbcaba}$
- $P = \text{cab}$

First we calculate the hash values of the pattern $P$ and the first $m$-substring of $T$. These values are called the initial fingerprints [28].

- $h(\text{cab}) = (99 + 97 + 98) \pmod{3} = 294 \pmod{3} = 0$
- $h(\text{aab}) = (97 + 97 + 98) \pmod{3} = 292 \pmod{3} = 1$

Since the first two hash values are different we continue by sliding the “window” to the right and calculating the hash value of the next $m$-substring. But here the trick is to be able to update the fingerprints at a constant time instead of calculating a brand new hash value. We can use the following approach. We already know that $h(\text{aab}) = 1$. From here we can simply add to this number the value of the next character that appears in the text (which is “b” in our case) and subtract the value of the string leaving our sliding window (in this case – “a”). So we get $1 + 98 - 97 = 2$. Now we perform the modulo operation to get $2 \pmod{3} = 2$. This is indeed the same value as if we were to calculate it using the long method.
- $T = \text{aabb}\text{caba}$
- $P = \text{cab}$
- $h(\text{abb}) = 2$

Since there is a mismatch we move onto the next substring by using the same procedure.

- $T = \text{aabb}\text{caba}$
- $P = \text{cab}$
- $h(\text{bbc}) = 1$

Again there is a mismatch so we move on.

- $T = \text{aabb}\text{caba}$
- $P = \text{cab}$
- $h(\text{bca}) = 0$

Here we see that $h(\text{bac}) = h(\text{cab}) = 0$ so we need to verify if the strings match by doing extra verification. No match is detected so we continue to the right. Finally we observe a complete match since both the hash values and the character comparison yield positive results.

- $T = \text{aabb}\text{caba}$
- $P = \text{cab}$
- $h(\text{cab}) = 0$

Given that we are using a sufficiently large prime number for the hash function we can ensure that the number of hash collisions will be insignificant and not worth accounting for. This means the algorithm’s complexity is $O(n)$ for the average case because there is only one comparison needed per m-substring. Of course we can always construct a text and pattern to invoke the worst-case scenario which is $O(nm)$ but this is only possible if the prime integer is small.
1. Karp-Rabin-Matcher(T,P,d,q)
2. n = length(T)
3. m = length(P)
4. h = d^{m-1} mod q
5. p = 0
6. t_0 = 0
7. for i = 1 to m  //preprocessing
8. p = (d*p + P[i]) mod q  //checksum of P
9. t_0 = (d*t_0 + T[i]) mod q  //checksum of T[1…m]
10. for s = 0 to n-m  //matching
11. if p = t_s
12. if P[1..m] = T[s+1..s+m]  //Checksums match.
13. Now test for false positive.
14. print “Pattern occurs with shift” s
15. if s < n-m
16. t_{s+1} = (d^*(t_s-T[s+1]*h) + T[s+m+1]) mod q
17. // Update checksum for T[s+1..s+m] using
    checksum T[s..s+m-1]  

Algorithm 6. Pseudo code for Karp-Rabin string matching algorithm
Chapter 6

Multiple-Keyword Pattern Matching Algorithms

It is apparent that when choosing a pattern-matching algorithm which will be implemented in a Network Intrusion Detection System we much choose one that supports matching of multiple string simultaneously. This is necessary because there are many different signatures containing attack definitions that need to be checked against. Over the past few decades there have been many solutions presented, tested, and implemented. In this chapter we will take a look at some of the fundamental algorithms that are currently used in the contemporary intrusion detection.

6.1 Aho-Corasick (AC) Algorithm

First introduced in 1975 by Alfred Aho and Margaret Corasick, this is one of the most widely used algorithms for multiple string matching. One of its applications is the engine behind the popular Unix utility grep (see Chapter 4.1). AC is closely related to the Knuth-Morris-Pratt algorithm as it generalizes its functionality to handle sets of strings instead of a single and is based on the technique of prefix searching. Aho and Corasick used the principle of constructing a deterministic finite automata (DFA) from the keywords set and using the it to process the text string in a single pass [32]. We define our new search pattern $P = \{P_1, P_2, ..., P_q\}$ to be a finite set of strings (keywords in terms of NIDS), also referred to as a dictionary. The operation of the pattern matching machine is controlled by three functions:
A Goto Function

The goto function is basically a trie (a prefix tree used to hold an associative array of strings) of the patterns we are searching for. The function takes as parameters a state $s$ and an input $a$ and maps them into some state or returns $\text{fail}$ if unsuccessful. The formal definition is as follows: $g(s, a) = s' | \text{fail}$.

A Failure Function

This function tells the algorithm what to do when there is no suitable match. It maps a state into a new state and is invoked when the goto function fails. We define the failure function as $f(s) = s'$. The value of the function at state $s$ is the state that contains the longest suffix of $P_q$ that also exists as a prefix of another string in the DFA. If we recall, this is the same principle we described that was used in the KMP algorithm.

An Output Function

This output function defines a set of keywords for every state of the DFA. We formalize that with $\text{output} (s) = \text{keywords}$.

Before we continue we must explain what is a deterministic finite automata. It is a machine that takes strings as input and based on a few predefined rules it tells whether the string is accepted or not. Moreover, given a state and an input symbol, there is only one transition state that the machine can jump to. A DFA is represented by a state (transition diagram) and possesses the following five properties:
• Q is a finite set of states
• \( \Sigma \) is finite set of symbols (the alphabet)
• \( \delta : Q \times \Sigma \rightarrow Q \) is the transition function
• \( q_0 \in Q \) is the start state
• \( F \subseteq Q \) is the set of final (accepting) states

In order to visualize the construction and operation of Aho-Corasick we will use an example with pattern \( P = \{he, she, his, hers\} \). Based on this dictionary keyword tree DFA will look as follows.

![Keyword tree for \( P = \{he, she, his, hers\} \)](image)

**Figure 8.** Keyword tree for \( P = \{he, she, his, hers\} \)

In order to build the keyword tree, the algorithm uses the *goto* function and begins at the root node – \( q_0 \) (see Figure 8). It begins inserting elements from the dictionary in order, one by one, and creates a new node for every new symbol. After the keyword “he” is inserted and a the first path is created the algorithm marks \( q_1 \) as an accepting state. Since the next word – “she” begins with a different letter, he algorithm branches out from the
root to establish a new path and insert all characters. Next, the keyword “his” shares a common first letter with the first word therefore the algorithm only has to insert the latter two characters by branching out of state $q_1$ and marking the third accepting state. By the same logic, since “hers” contains a substring that is already part of a branch of the trie, the algorithm continues inserting the remaining suffix past the acceptance state for “he”. It is easy to see that this tree is built in time $O(|P_1| + \ldots + |P_q|) = O(n)$. Algorithm 7 shows the simple pseudo code that is used to build this goto function.

Next we show how the keyword tree is extended into an AC automaton which allows for patterns to be matched without the need for the algorithm to backtrack (skip comparisons that have been previously made) upon a failed link. In order to build the automaton we take advantage of the failure function. It creates a map of all states and their corresponding destinations when a mismatch is encountered. At the root level we have a loop back to itself if the pattern does not match one of the first character of the keyword tree. The acceptance states indicate which keywords the algorithm will accept during the traversal.

The actual searching mechanism (see Algorithm 8) takes the first keyword $P_i$ and follows the path down the tree that is labeled by the character comprising that keyword. If the path leads to an acceptance state – the keyword is in the dictionary. If the path end before a final state, the string is not in the dictionary and we use the failure function to hop onto another node and keep the search.
The complexity of the Aho Corasick algorithm is $O(n+m+k)$ which is a linear time compared to the single-keyword matching approaches. It takes $O(n)$ time and space to build to keyword tree, $O(m)$ time to perform the comparisons (failure links) and $k$ is the actual number of patterns found in the text $T$ (the output links). We can expect the worst-case time performance to be $O(2n)$. The performance of AC does not depend on the keyword set which makes it a good candidate for NIDSs where the set of patterns is relatively large.

Figure 9. Example of Aho-Corasick Automaton

Algorithm 7. Aho Corasick Goto Function
Algorithm 8. Aho-Corasick Multiple Pattern Matching Algorithm

6.2 Wu-Manber (WM) Algorithm

The Wu-Manber (WM) multiple-string pattern matching algorithm was developed by Sun Wu and Udi Manber at the University of Arizona in 1991. The algorithm was actually constructed while Wu and Manber were working on developing the UNIX tool agrep which is a cousin of the already popular tool grep (see Chapter 4.1). The main difference between both tools is that agrep searches a file for strings or regular expressions with approximate matching capabilities [33]. The basic idea of this algorithm is inherited by the Boyer-Moore single-keyword approach and more specifically using the bad character shift rule to skip over irrelevant parts of the pattern in order to avoid redundant comparisons. But instead of searching for a single pattern, we simultaneously search the input text for a set of patterns $P = \{p_1, p_2, p_3, \ldots, p^m\}$. Each string is considered separately as a sequence of substrings $p^j = a_1a_2\ldots a_m$. At the preprocessing stage the Wu Manber algorithm constructs three tables – a SHIFT table, a HASH table and a PREFIX table,
which we will examine shortly in this chapter. The patterns pre-processing of WM is based on the following steps:

- Calculate \( \text{min} \)
- Initialize the SHIFT table
- Complete the SHIFT table
- Complete the HASH table

**Calculate \( \text{min} \)**

In order to determine what is the maximum safe shift size Wu and Manber decided to first calculate the minimum length of a pattern from the set of patterns, which we label \( \text{min} \). In order to ensure the algorithm does not miss any matches we cannot jump forward into the text by a value greater than \( \text{min} \). This creates a small problem because if we have at least one string of size 2 it wouldn’t be possible for us to shift by more than two characters at a time which degrades the efficiency of the algorithm. Fortunately, when implementing the WM strategy in Intrusion Detection Systems, this problem is not that significant since, as we previously explained in Chapter 4.1, the majority of the search patterns are between 4 and 16 bytes which allows for reasonable size jumps.

The calculation of \( \text{min} \) is as simple as counting the minimum number of adjacent bytes, excluding the wildcards, that appear in any signature. Let’s consider the following example. \( P = \{ \text{announce, annual, annually} \} \). We can see the distribution of \( \text{min} \) across all patterns in Table 6 below. Since “annual” is our shortest keyword we can conclude that \( \text{min} = 6 \) for this example.
<table>
<thead>
<tr>
<th>announce</th>
<th>annual</th>
<th>annually</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

**Table 3.** Calculation of the *min* function in the Wu-Manber algorithm

**Initialize the SHIFT table**

In their algorithm, Wu and Manber use a novel approach and examine the text in blocks (of size $B$) instead of as single characters which allows us to make more intelligent decisions about how to skip over a safe amount of symbols. Therefore we have $T = \{t_1, t_2, t_3, \ldots, t_n\}$. If in our keyword set from an alphabet of size $c$, we have $k$ patterns then the total size of all patterns will be $M = k \times \text{min}$. They determined that the most practically suitable block size is either $B = 2$ or $B = 3$ which is derived by the formula $B = \log_c2M$ [34]. In our example we will use $B = 2$. The SHIFT table works similarly to the SHIFT table in the Boyer-Moore algorithm – it determines the number of bytes that can be safely skipped. The only difference is that it determines the shift value based on the last $B$ characters (the suffix) instead of the prefix of the word. This means we are comparing the patterns and the strings backwards which has been shown to produce better results in some algorithms.

During the pre-processing stage, the initialization of the SHIFT table is essentially setting all values to $\text{min} – B + 1$ [34]. We explain why this is when examining the construction of the SHIFT table next.
Complete the SHIFT table

There are two types of SHIFT tables that can be used in WB. A complete SHIFT table contains an entry for every possible substring of length B which implies a table size of $|\Sigma|^B$. In order to reduce the table size a compressed table can be used where we are mapping multiple substrings to the same hash value. However, in this case we need to reduce the shift values of each set of strings that map to the same integer to the minimum of their values. In other words we are trading off a reduced table size for a decreased amount of jumps. Since the preprocessing stage (for a fixed set of keywords) is done only once there is not a significant difference which type of table is used however modern Wu-Manber modifications tend to favor the compressed approach. When demonstrating how WM works further in this chapter we will also use the compressed table approach.

In order to complete the SHIFT table we need to extract all distinct 2-byte sequences (since we decided to work with a block of $B=2$). They are namely an, nn, no, ou, un, nc, ce, nu, ua, al, ll, ly. Let $q_{xy}$ be the rightmost ending position of $xy$ in any of the pattern substrings. For example “an” appears as a substring in all three keywords but “ly” as a substring only in “annually”. Then we have $q_{an} = 2$ and $q_{ly} = 8$. There are two scenarios we can observe from here:

1. If a block $B_i$ from the text $T$ does not appear as a substring in any of the keywords this implies we can shift the characters to the right by a maximum shift of $\text{SHIFT}[xy] = \text{min} – B + 1$. This simply implies we are shifting the matching window to the right by the size of the entire block $B$.

2. If a block $B_i$ is found in any of the keywords we find its rightmost occurrence ($q_{xy}$) in a string $p_i$ we execute $\text{SHIFT}[xy] = \text{min} – q_{xy}$. 
Finally, we set all the values of the SHIFT table to the minimum of the two calculations above and our table is complete. Next, we show how the HASH and PREFIX tables are constructed, however they are only used when the SHIFT value of a sub-string is equal to zero. We use them to decide which patterns are likely candidates for a match and also to verify that a match exists.

**Complete the HASH table**

The purpose of this table is to store the hash value of each possible block $B_i$. It contains an index that maps every block $B_i$ that is used in the SHIFT table to an integer. It also contains the list of all patterns that have the same suffix. When searching a text, we calculate the hash value of the block inside the current match window and lookup the hash table to get the list of all patterns that contain the same block character as their suffix.

Every time we perform a shift operation we know we are not at the end of a substring that can potentially be a match. But if our shift equals to zero we are definitely at the right edge of a possible match. We formally represent this as:

\[
\text{SHIFT}[xy] = \min - q_{xy} = 0.
\]

If this condition is true then we set HASH[xy] to all signatures that have a pattern substring ending at “xy”. An example is shown below based on the previously established values $q_{an} = 2$ and $q_{ly} = 8$:

- HASH[an] $\rightarrow$ announce; annual; annually
- HASH[ly] $\rightarrow$ annually
The complete construction of the SHIFT and HASH tables is shown in Table 4.

<table>
<thead>
<tr>
<th>String</th>
<th>an</th>
<th>nu, nn</th>
<th>no, ou</th>
<th>un, nc</th>
<th>ua, al</th>
<th>ce</th>
<th>ll</th>
<th>ly</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHIFT</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>HASH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4. SHIFT and HASH tables in the Wu-Manber algorithm

<table>
<thead>
<tr>
<th>String</th>
<th>ce</th>
<th>ly</th>
<th>al</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASH</td>
<td>{1,3}</td>
<td>{1,3}</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Table 5. HASH tables in the Wu-Manber algorithm

**PREFIX Table**

Since a particular HASH bucket can contain many entries that map to it, it becomes more efficient to build and keep track of a third table, PREFIX table, than contains all the $B'$ characters of each pattern. Like with the SHIFT table, a good value is $B' = 2$ [34]. When we observe $\text{SHIFT}[xy] = 0$ it means we could have a potential match therefore instead of comparing every single value stored in the HASH table we can narrow the results by performing a lookup in the PREFIX table.

**The Scanning Stage**

After the three tables have been constructed we are ready to explain how the scanning mechanism works. It consists of four steps:
1. Compute a hash value $h$ based on the current block of $B$ characters from the input text $T$. We start at position $y[\text{min} - B + 1] \ldots y[i])$.

2. If $\text{SHIFT}[h] > 0$ (we are not within a matching substring), we shift the text to the right by the corresponding hashed value and return to Step 1. If $\text{SHIFT}[h] = 0$, we continue to Step 3.

3. Compute the hash value of the text prefix (starting $\text{min}$ symbols to the left of the current position). We name this variable $\text{txt\_prefix}$.

4. Check for each position whether $\text{PREFIX}[P] = \text{txt\_prefix}$. If they are equal, we check the actual pattern against the input text to determine if a match exists.

Finally we show a complete trace of the Wu-Manber algorithm using an input $T = \text{"IBM-annual-conference-announce"}$ and $P = \{\text{announce, annual, annually}\}$. Since $\text{\text{min}} = 6$ we start the comparisons at position 6 working backwards.

- **IBM-annual-conference-announce**
  $\text{SHIFT}[\text{an}] = 4$

- **IBM -annual-conference-announce**
  $\text{SHIFT}[\text{al}] = 0$. $\text{List} = \text{HASH}[\text{al}] = \{2\}$. Compare $p_2$ against the corresponding text substring and denote a match is found. Shift search positions by 1.

- **IBM -annual-conference-announce**
  $\text{SHIFT}[\text{e}] = 5$

- **IBM -annual-conference-announce**
  $\text{SHIFT}[\text{e-}] = 5$
- IBM -annual-conference-announce
  \( \text{SHIFT[ou]} = 3 \)

- IBM -annual-conference-announce
  \( \text{SHIFT[ce]} = 0 \).
  \( \text{List} = \text{HASH [ce]} = \{1, 3\} \).
  Compare \( p^1 \) against the corresponding text substring and denote a match is found.

Many variations of the Wu-Manber algorithm have been implemented in Snort and other Intrusion Detection Systems and are currently used today because WM is an efficient approach for simultaneous pattern searching. The run time of this algorithm is estimated based on the assumption that both input text and search patterns are random strings with uniform distribution. In practice, of course, this is not the case, which means performance may vary depending on the application of the algorithm. The SHIFT table is constructed in \( O(M) \) time since each \( B \)-size substring of any pattern is examined once and it takes constant time on the average to examine it. The scanning phase has two scenarios. The first one is when \( \text{SHIFT[xy]} \neq 0 \). In this case we simply shift the scanning window without performing any more operations. In the second case, where \( \text{SHIFT[xy]} = 0 \), we have to consult the HASH and PREFIX tables to verify if we have a match. Therefore we have to determine how many shifts we expect to have on the average. Wu and Manber found that number to be \( \geq \frac{m}{2} \) which implies that considering it takes \( O(B) \) to build each has function the total amount of work required for perform all non-zero hash mappings is \( O(BN/min) \) [34]. The worst-case scenario of the algorithm is still \( O(MN) \).

From Figure 10 we can see the relationship between the length of a pattern and the run time of the algorithm. For patterns longer than 2 bytes the algorithm performs significantly faster which confirms the algorithm is suitable for intrusion detection in a practical environment.
Figure 10. Relation between search time and the length of patterns in Wu-Manber algorithm

The pseudo code of Wu-Manber can be found in Algorithm 8. We can see the order of operations that comprise the entire scope of the algorithm.

```
1. WuManber(P = {p1, p2, ..., pr}, T = t1t2...tn)
2. //Preprocessing
3. Compute a suitable value of B (e.g. B = log2M)
4. Construct tables SHIFT and HASH;
5. // Searching
6. pos = min;
7. while pos ≤ n do
8.     i = h1(tpos−B+1...tpos);
9.     if SHIFT[i] = 0
10.     then
11.         list = HASH[h2(tpos−B+1...tpos)];
12.         Verify all patterns in list against the text;
13.         pos++;
14.     else
15.         pos = pos + SHIFT[i];
16. fi
17. od
```

Algorithm 8. Pseudo-code of Wu-Manber algorithm
6.3 Commentz-Walter (CW) Algorithm

Developed in 1979 by Beate Commentz-Walter, this algorithm is a hybrid between the multiple-keyword matching approach of Aho-Corasick and the single-string method of Boyer Moore. The technique, adapted from the AC algorithm, works by building a keyword trie (see Chapter 6.1) which in conjunction with the shifting procedure from the BM algorithm creates a fast system for searching simultaneous patterns in a text. As in many other algorithms (some of which we already examined in this paper) we can divide CW into two phases – keyword pre-processing and text scanning. The pre-processing part consists precisely of building a keyword trie but this time the keywords are populated in reverse. In order to explain the operation of Commentz-Walter we must introduce several terminologies. Each node of the trie contains a character of a keyword and is labeled by \(v\), except the root node which is labeled with \(\varepsilon\) denoting the empty symbol. Like in the Wu-Manber algorithm we calculate the length of the shortest strings and refer to it as \(w_{\text{min}}\).

We also introduce two sets that categorize some of the nodes – \(d\) and \(w\). They purpose is as follows:

- \(d(v)\) denotes the depth of each node \(v\) in the trie. In other words, the number of intermediate nodes, including \(v\), that are required to reach the root.
- \(w(v)\) denotes a word that is created by merging consecutive characters from the root to the node \(v\).

Formally we can represent the node depth by the following:

\[
\begin{align*}
  d(v) &= 0 & \text{if } v = \varepsilon, \text{ or} \\
  d(v) &= d(v') + 1 & \text{if } v \text{ is a child of } v'
\end{align*}
\]
After the keyword trie is established the pre-processing stage calls for the creation of four functions that are responsible for keeping track of the shifting during the scanning phase. The functions are *out*, *shift1*, *shift2*, and *char* [38].

The simplest function of all is *out*. It essentially takes a node $v$ as an input and determines whether the path from the root to the node $v$ represents a keyword that is part of the set of keywords $K$. If so, the *out* function returns that keyword; otherwise it does not return anything.

Next we need to establish two more sets, *set1*(v) and *set2*(v), that are used by *shift1* and *shift2* respectively. The first set is a collection of all nodes in the trie that have a higher depth than $v$ and that also end with the same suffix – $w(v)$ [38]. In other words, for every node $v$, the algorithm finds all other paths from the root of the trie to $v$ that share a common suffix. Then a logical connection is created between the two end nodes. In Figure 11 this connection is denoted by a dashed arrows. Then we have *set2*(v), a subset of *set1*(v), which contains all nodes that are starting points of paths from $v$ to the root that match any of the keywords in K. That is to say we are looking for all starting nodes that have an *out* function different than 0. All connections to nodes in *set2*(v) are represented by sold arrows in Figure 11. We formally represent these two sets as:

- $\text{set1}(v) = \{v' : w(v) \text{ is a proper suffix of } w(v')\}$
- $\text{set2}(v) = \{v' : v' \in \text{set1}(v) \text{ and } \text{out}(v) \neq 0\}$

Now we are ready to define both shift functions *shift1* and *shift2* [38].

- $\text{shift1}(v) = 1$ if $v = r$, otherwise:
- $\text{shift1}(v) = \min (w_{\text{min}}, \{d(v') - d(v)\})$, where $v' \in \text{set1}(v)$
- \( \text{shift2}(v) = \text{wmin} \) if \( v = r \), otherwise:
- \( \text{shift2}(v) = \min (\text{shift2}(\text{parent node of } v), \{d(v') - d(v)\}) \), where \( v' \in \text{set2}(v) \)

**Figure 11.** Commentz-Walter keyword trie including both shift functions

The population of the \( \text{shift1} \) values throughout the nodes of the tree begins with the root where the value is always one. For all other nodes the algorithm takes the minimum value
of \( w_{\text{min}} \) and the difference between the depth of the child node and the parent node. All nodes that do not have pointers to nodes in either set will be assigned a value of 3 which is the value if \( w_{\text{min}} \). By the same structure we construct the values for the \( \text{shift2} \) function based on the formula outlined above.

The last part of the pre-processing stage is to define a function \( \text{char} \) that is mapped to the values of the accepted alphabet [39]. It is used in the verification phase of the algorithm.

\[
\text{char}(a) = \min( w_{\text{min}} + 1, d(v))
\]

The scanning stage of Commentz-Walter is essentially traversing the subject string from left to right but performing the matching from right to left. As soon as a match fails the starting point for matching is shifted to the right by the values of both shift functions and the matching starts again. Algorithm 9 presents the pseudo-code of the original algorithm although many other variations have been presented.

The worst case performance of Commentz-Walter is poor – \( O(mn) \). This is because some substrings of the input text can be scanned multiple times, i.e. we have the problem of backtracking. Some of the later variations of CW have addressed this problem and tried to overcome it. The pre-computation time is linear – \( O(M) \) – and it depends on the total length of all keywords used in the process. The actual expected running time of CW is linear although the algorithm yields different results in different environments. The application of Commentz-Walter in a practical environment has declined over the years and while it is still applicable in some instances, faster algorithms, such as Wu-Manber, have been preferred when considering algorithms for intrusion detection.
Algorithm 9. Commentz-Walter multiple pattern-matching algorithm pseudo-code
Chapter 7

Conclusion

With the Internet constantly expanding and traffic through network security devices increasing tremendously the need of developing fast algorithms for search malicious patterns in the flow of information is vital. In this paper we have made an introduction to Intrusion Detection Systems, as a whole, and examined their types, operation, characteristics and their importance for the contemporary networks security industry. In particular, we have explored the operations of one of the most widely used open-source systems – Snort – and examined how to construct rules that are used for recognizing specific patterns of information. We have also presented different traits when it comes to building and utilizing a pattern matching algorithm. There are many algorithms that have been developed over the past decades but not everyone is suitable to be integrated in an Intrusion Detection System. Factors such as handling large patterns, large signature sets, and variable alphabets are important to take into consideration therefore we presented how each criteria would affect the performance of an algorithm.

Finally, we examined several pattern matching algorithms that have been widely adopted for use in various environments and specifically in the field of Intrusion Detection Systems. Boyer-Moore and Knuth-Morris-Pratt are classical examples of fast single-keyword algorithms. Even though they are not directly applicable in NIDSs they lay the foundation for extending these approaches into simultaneously matching multiple keywords and are important for use to understand the principles of fast strings matching. The most significant multiple-keyword matching algorithms – Aho Corasick and Wu-Manber – have shown different but efficient approaches for solving the problem of
finding information in parallel. By using finite automata or various shift tables these algorithms have been widely accepted in Snort and many other NIDSs. While it is difficult to calculate what is their practical performance a good estimate is their achieve slightly sub linear search times.

The field of pattern matching is currently on the rise and many people are getting involved in refining and speeding up existing algorithms or developing completely new approaches. For example a recent study has suggested that using the Genetic Algorithm (GA) can be successfully used in IDSs. The algorithms works by taking the patterns matching problem and converting it into a biological model by using chromosome-like data structures [40]. It can detect anomalies in the behavior of a computing device based on past and present data. These type of innovative approaches need to be researched more extensively because they can show us new methods that can be applied in Computer Science and specifically in the field of Intrusion Detection.
Bibliography


