An Examination of Pattern Matching Algorithms for Intrusion Detection Systems

By

James Kelly

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfilment of

the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario, Canada

August 2006

©Copyright

James Kelly, 2006

The undersigned hereby recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis,

An Examination of Pattern Matching Algorithms for Intrusion Detection Systems

submitted by

James Kelly

Dr. Frank Dehne

(Director, School of Computer Science)

Dr. Paul Van Oorschot

(Thesis Supervisor)

Carleton University

August 2006

Abstract

Multiple-pattern matching algorithms are the heart of many network intrusion detection systems' signature matching engines. They allow these engines to quickly search for many patterns simultaneously in input passing through such systems, but often consume most of the processing time. Thus, they should be as fast as possible to ensure system scalability into networks of ever-increasing speeds. Concurrently they must enforce security so that they are not susceptible to algorithmic complexity attacks.

We provide a comprehensive overview of significant pattern matching algorithms and discuss their suitability for these kinds of systems. Using the Snort network intrusion detection system as a platform, we implement and compare several apposite algorithms. Multiple Backward Oracle Matching has not been used in intrusion detection to our knowledge, and we introduce it in options we add to Snort: MBOM and AUTO. Our AUTO option is a new approach to pattern matching in Snort using multiple algorithms.

Acknowledgements

I want to thank especially the delicate and encouraging research guidance and sagacious feedback given by Paul C. van Oorschot over the course of the graduate studies leading to this work. As well, I wish to thank Anil Somayaji, Tim Furlong, Glenn Wurster, James Muir, and the rest of the Carleton Computer Security Lab members for their support and feedback on various parts of the material in this thesis. I am grateful to have been among you during my time at Carleton University.

I also want to thank the commitee members and chairman—Paul C. van Oorschot, Pat Morin, Ali Miri, and Michiel Smid—who reviewed this thesis.

Many thanks go to Ontario's Ministry of Training, Colleges and Universities, Nortel Networks, and Carleton University and for their generous financial support through various scholarships.

Last but not least, I would like to thank my family members. Thank you for giving me the support and love to complete my work.

Contents

Abstract **ii**

Acknowledgements **iii**

Glossary of Acronyms **x**

1 Introduction and Overview **1**

2 Background on Intrusion Detection Systems **4**

2.1 Passive versus Reactive IDSs 5

2.2 Misuse-based versus Anomaly-based IDSs 7

2.3 Host-based versus Network-based IDSs 9

2.3.1 Traits of Signature-based and Network-based IDSs 10

2.3.2 Deploying a Signature-based NIDS 13

2.3.3 Architecture of a Signature-based NIDS 14

2.4 Chapter Summary 20

3 Single-Keyword Pattern Matching Algorithms **21**

3.1 Brute Force Algorithm 23

3.2 Karp-Rabin Algorithm 24

3.3 Knuth-Morris-Pratt Algorithm 27

3.4 Boyer-Moore Algorithm 29

3.5 Backward Oracle Matching Algorithm 39

3.6 Chapter Summary 46

4 Multiple-Keyword Pattern Matching Algorithms 47

4.1 Aho-Corasick Algorithm 49

4.2 Commentz-Walter Algorithm 51

4.3 Wu-Manber Algorithm 60

4.4 Fan-Su Algorithm 64

4.5 Set Backward Oracle Matching Algorithm 72

4.6 Chapter Summary 78

5 Pattern Matching for NIDS Signatures 82

5.1 Algorithm Requirements 82

5.1.1 Searching for Multiple Patterns Simultaneously 83

5.1.2 Searching for Large Sets of Patterns 83

5.1.3 Searching With a Large Alphabet Size 84

5.1.4 Searching With a Wide Range of Keyword Lengths 85

5.1.5 An Algorithm Designed for the Average and Worst Cases . . . 86

5.1.6 Extended Searching Characteristics 89

5.2 Candidate Algorithms to Fulfill Requirements 89

5.3 Chapter Summary 93

6 Software Solutions That Have Been Proposed 94

6.1 The First Multiple-Keyword Pattern Matching Solutions for Snort . . 95

6.2 Current Solutions in Snort 2.6 97

6.3 Piranha 100

6.4 Deterministic Memory Efficient String Matching 104
6.5 Chapter Summary 108

7 Implementing and Comparing Pattern Matching Algorithms for Snort 109

7.1 Algorithms Applicable For Snort 110
7.2 Adding the New Algorithms 112
7.3 Evaluating Our Algorithms In Snort 120
7.4 Chapter Summary 124

8 Further Discussion and Concluding Remarks 125

8.1 Pattern Matching Algorithms in Other Security Applications 125
8.1.1 Antivirus Software 125
8.1.2 Spam Detection Software 128
8.2 Pattern Matching in Hardware 129
8.3 Future Work in the MBOM Snort Options 130
8.4 Concluding Remarks 131

A Modifications to Snort 133

A.1 Adding the New Search Method Options 133
A.2 The MBOM Option 145
A.3 The MBOM2 Option 164
A.4 Changes to the AC-Std Option 185

List of Figures

2.1 An overview of Snort's architecture 16

2.2 A sample Snort rule 17

3.1 Boyer-Moore algorithm's good suffix shift examples [16] 34

3.2 Three data structures that hold all factors of a word 40

3.3 Iterations in the construction of a factor oracle for the word `abbbaab` 43

3.4 The matching phase of the Backward Oracle Matching algorithm . . . 45

3.5 Example of a suffix oracle for the string `abbbaab` 45

4.1 An example of a trie versus an Aho-Corasick automaton. 52

4.2 An example of a basic Commentz-Walter style trie for x ={"hers", "his", "she", "he" } 53

4.3 Commentz-Walter style trie with sets and shift functions for keyword set {"cacbaa", "acb", "aba", "acbab", "ccbab",} 55

4.4 Commentz-Walter algorithm B1 58

4.5 Commentz-Walter style trie with new sets for algorithm B1 59

List of Tables

5.1 Table of algorithms' fulfillment for NIDS desirable features 90

7.1 Average results for Snort search methods on machine 1 121

7.2 Average results for Snort search methods on machine 2 122

List of Algorithms

3.1 Brute Force Single-Keyword Matching Algorithm 24
3.2 Karp-Rabin Single-Keyword Matching Algorithm 28
3.3 Knuth-Morris-Pratt Single-Keyword Matching Algorithm 30
3.4 Boyer-Moore Bad Character Shift Pre-computation Algorithm 33
3.5 Boyer-Moore Good Suffix Shift Pre-computation Algorithm 35
3.6 Boyer-Moore Single-Keyword Matching Algorithm 37
3.7 Boyer-Moore-Horspool Single-Keyword Matching Algorithm 38
3.8 Factor Oracle Construction Algorithm 42
3.9 Backward Oracle Matching Single-Keyword Matching Algorithm . . . 44
4.1 Aho-Corasick Multiple-Keyword Matching Algorithm 50
4.2 Commentz-Walter Multiple-Keyword Matching Algorithm B 56
4.3 Wu and Manber Multiple-Keyword Matching Algorithm 65
4.4 Fan and Su Multiple-Keyword Matching Algorithm 67
4.5 Factor Oracle Construction Algorithm From a Keyword Set 75
4.6 Set BOM Multiple-Keyword Matching Algorithm 77
7.1 Multiple BOM Multiple-Keyword Matching Algorithm 117

Glossary of Acronyms

AC - Aho-Corasick. A multiple-pattern matching algorithm. (Page 49).

AC-Banded - An AC algorithm option in Snort. (Page 97).

AC-Bitmap - An AC algorithm option for Snort proposed by Tuck et al. [86]. (Page 104).

AC-Full - An AC algorithm option in Snort. (Page 97).

AC-Path - An AC algorithm option for Snort proposed by Tuck et al. [86]. (Page 104).

AC-Std - An AC algorithm option in Snort. (Page 97).

BM - Boyer-Moore. A single-pattern matching algorithm. (Page 29).

BMH - Boyer-Moore-Horspool. A single-pattern matching algorithm. (Page 36).

BOM - Backward Oracle Matching. A single-pattern matching algorithm. (Page 39).

BSOM - Backward Suffix Oracle Matching. A single-pattern matching algorithm. (Page 46).

CW - Commentz-Walter. A set of multiple-pattern matching algorithms. (Page 51).

DAWG - Directed acyclic word graph. (Page 39).

DFSA - Deterministic finite state automaton. (Page 64).

HIDS - Host-based intrusion detection system. (Page 9).

IDS - Intrusion detection system. (Page 3).

IPS - Intrusion prevention system. (Page 6).

KMP - Knuth-Morris-Pratt. A single-pattern matching algorithm. (Page 27).

MBOM - Multiple Backward Oracle Matching. A multiple-pattern matching algorithm. (Page 2).

MWM - Modified Wu-Manber. A multiple-pattern matching algorithm option in Snort. (Page 97).

NIDS - Network-based intrusion detection system. (Page 9).

SBMH - Setwise Boyer-Moore-Horspool. A multiple-pattern matching algorithm. (Page 95).

SBOM - Set Backward Oracle Matching. A multiple-pattern matching algorithm. (Page 72).

SBSOM - Set Backward Suffix Oracle Matching. A multiple-pattern matching algorithm. (Page 76).

WM - Wu-Manber. A multiple-pattern matching algorithm. (Page 60).

Chapter 1

Introduction and Overview

Since their gain in popularity, intrusion detection systems have begun to be used frequently as one component of an effective layered security model for an organization. Various alterations of what started as monitoring systems [7, 32, 43] spurred interest, and intrusion detection quickly became known as an important computer security tool for individual computers as well as in computer networks. Today they are used in many places both inside and outside security perimeters and in many different ways. Always quintessential is that the information collected through detection can be made into powerful intelligence if put to use to strengthen computer security in the areas of intrusion prevention, preemption, deterrence, deflection, and countermeasures.

Understandably, a protected system or network is only as secure as its defences are strong. In the intrusion detection systems that we focus on in this thesis, we show how pattern matching is a critical ability, and that it must be a strength of the system. Unfortunately, in the past it has been identified as a visible and exploitable weakness, and as such, has been the topic of much specialized research for some years now. Although intrusion detection systems have various uses as is explained further in Chapter 2, many types of these systems rely heavily on pattern matching

within certain core components. Moreover, pattern matching is widely used in many computer security applications.

The main contributions of this thesis are to examine the older and well-studied problem of pattern matching and its solutions fully, and present the applicability of the solutions to pattern matching within the context of network intrusion detection systems. We show that many popular current and past solutions may not be suitable at all within this context; furthermore, some solution domains have not even yet been examined. In our assessment of pattern matching algorithms we discovered one particular solution domain of pattern matching using factor oracles [3, 4, 5] that, to our knowledge, until now, has not yet been seen in practice within network intrusion detection systems. However, it may well have an appropriate place. Markedly, we clarify, modify, and use the Multiple Backward Oracle Matching (MBOM) algorithm [6]. In this thesis we give both a detailed theoretical explanation and pseudocode of this solution's algorithm and the code that we used to implement the algorithm. Neither of these are available in the original MBOM publication by Allauzen and Raffinot [6] which focuses on Set Backward Oracle Matching (see Section 4.5) and only introduces MBOM in passing as a minor contribution. Although it was not their primary focus, MBOM should be of high interest to the network intrusion detection community because of its performance characteristics as a multiple-pattern matching algorithm. We discuss this further in Chapter 7.

In addition to these contributions, we use the popular and free network intrusion detection system, Snort [72, 77], to compare what we justify to be the best selection of applicable pattern matching algorithms in practice. Specifically, we implement three new options for Snort using the existing MBOM algorithm. One option in particular, our AUTO option (see Chapter 7), takes a new approach to pattern matching in Snort because it uses more than one algorithm—unlike any of the other pattern

matching approaches available in the existing Snort options. It decides on a suitable algorithm depending on the pattern group (see Chapter 2) characteristics before the searches take place. Our AUTO option proves to be a serious contender and worth consideration given that it performs better or about equally as well as Snort's current fastest approach.

In the balance of this thesis Chapter 2 provides background material on intrusion detection systems, as well as a detailed introduction of the signature-based network intrusion detection systems, such as Snort, that we focus on in this thesis. The next four chapters (Chapters 3–6) go over a large body of pattern matching research and pattern matching within network intrusion detection systems. In order to gain a proper understanding of multiple-pattern matching algorithms for use in a signature- and network-based IDS, we first explore types of simpler and classical single-keyword pattern matching in Chapter 3. These often helped inspire the multiple-keyword pattern matching (multiple-pattern matching) algorithms we go through in Chapter 4. Subsequently, we evaluate how the presented multiple-pattern matching algorithms fare in the domain of network intrusion detection systems in Chapter 5. We then proceed to investigate several contemporary software-based multiple-pattern matching algorithms that are appropriate and suggested for use in practice in the target type of intrusion detection system (Snort and those that are similar) in Chapter 6. Chapter 7 begins by presenting our rational behind selecting what we identify as a subset of algorithms applicable for use within Snort. The MBOM algorithm that we implement in Snort—which is not detailed in other works—is also documented before we go on to compare the selected algorithms in practice through our experimentation and results. Chapter 8 presents future work and other uses of our contributions presented herein, and discusses future direction for pattern matching in intrusion detection like creating hardware-based network intrusion detection systems.

Chapter 2

Background on Intrusion Detection Systems

In this chapter we give the background for intrusion detection systems (IDSs). This content along with the state of the art in pattern matching algorithms used within IDSs (in Chapters 5) are important to fully grasp in order to understand the contributions of this thesis. Intrusion detection covers a broad range of digital security because IDSs have a wide range of uses. In general, these systems automate the process of extracting intelligence about past or present actions that attempt to compromise the confidentiality, integrity, or availability of a resource.

The definition of an intrusion in this context is not fixed, but rather is a concept that changes depending on the administration or objective of the system. More specifically, the intelligence and information provided by an IDS is contingent upon how the system is being used, and is as important as the chosen IDS itself. Indeed, there are many ways to use IDSs. If and when an IDS discovers an intrusion, regardless of how it has been defined, it is common for a system to make a record or report of the intrusion, typically by way of logging or generating an alert that is sent off to an

appropriate party [13]. More and more, these systems are built to act not only as a judge of intrusions, but also to react to them as we show in the next section.

In the subsequent sections of this chapter we review a classification of intrusion detection systems that is familiar to the intrusion detection community. Considering this classification helps to narrow the focus of the context in which an action or inaction constitutes an intrusion. An understanding of this classification will clarify the scope of the contributions of this thesis. We examine more closely the domain of IDSs where our contributions are positioned. Section 2.1 discusses passive versus reactive IDSs. Section 2.2 discusses misuse-based versus anomaly-based IDSs. Section 2.3 discusses network-based versus host-based IDSs. It is in this section that we dig deeper showing that the implementation and comparison work of this thesis (see Chapter 7) is contained in the intersection between network-based and misuse-based IDSs. Section 2.3.1 focuses on this specific intersection under examination, and when it is appropriate to use systems in this class. Section 2.3.2 deals with best practices of how network-based IDSs like these are used within networks. Finally, Section 2.3.3 gives an overview of the architecture of an example IDS in this class which illustrates and motivates the importance of the algorithms used within such systems like those discussed in the following chapters.

2.1 Passive versus Reactive IDSs

Traditionally, intrusion detection systems were *passive* monitoring systems [7]. As the name indicates, the nature of detection does not involve any form of response to the intrusion. The model upon which classic IDSs are supposed to have been built is, therefore, that of a passive system. A passive system is one in which sensors detect intrusions and report them to the system's reporting engine which, depending on its

capability and configuration, could format and log the intrusion to a database, a file, or even a computer console. A passive system may also signal an alarm of some kind, but its distinguishing characteristic is that it does not deal directly with the intrusion to stop it or prevent future intrusions of the same sort in any way. To the contrary, in a *reactive* IDS a response to the suspicious activity is performed by, for example, logging off a user or by reprogramming a firewall to block network traffic from the suspected malicious source [80].

If the system takes action to directly affect the current or future intrusions it may be designated as an intrusion prevention system (IPS) [43]. For example, in the context of a network-based intrusion prevention system, the system may be able to directly terminate or rate-limit connections. This differs very little from a reactive intrusion detection system in only that it performs the action itself. A reactive intrusion detection system, in contrast to the above example, might have signaled a firewall or another network appliance to terminate the malicious connection under suspicion. Firewalls and even application layer firewalls differ in the sense that they do not usually have the capability to search for anomalies or specific content patterns (or keywords) called signatures (discussed further in Section 2.2) to the same degree as intrusion detection and prevention systems do [37, 86]. Despite the classification that we present here, we point out that it is of course not impossible to make a firewall with all the same capabilities that an IDS possesses. It is often simply a matter of which names and terms are chosen to best market the system. Furthermore, although here the example of an intrusion prevention system is network-based, intrusion prevention systems could be host-based acting to deny potentially malicious activity [26, 74, 75].

2.2 Misuse-based versus Anomaly-based IDSs

Misuse detection is very comparable to classical or first-generation virus scanning. It involves dealing with the input to the system and searching it for what the IDS rules refer to as patterns of misuse, however those rules define them. Accordingly, we distinguish that in the context of IDSs, the term "misuse" does not have to refer to an attack by an insider or authorized user [13]. A pattern in this context may be very simple, like looking for a specific string of bytes at a given position or any position. It may also be rather complex like matching a regular expression, for example, involving the presence of one string and the subsequent omission of another string within a certain range of bytes. It may not even involve what is typically thought of as a pattern, but instead search for a predefined harmful state that constitutes an abuse. These misuse patterns are very often of the same nature as patterns in strings or regular expressions, and in the area of IDSs are referred to as signatures. Consequently, misuse-based intrusion detection systems are also known as signature-based intrusion detection systems (and sometimes knowledge- or rule-based IDSs) [13, 21].

The nature of the patterns present in an intrusion detection system depends on the power of the system itself and its intended use. Naturally a system that can search for matches of complex patterns must have a more complex language to allow the system's users to describe the patterns. Conjointly and in general, systems that match complicated patterns would also be expected to take longer to process the input than a system that can only match simpler patterns. These patterns or signatures are predefined and preloaded into a system before it starts processing input. When the system starts processing input and, in effect, searching for possible signature matches, the relevance of the signature complexity and the number of signatures may greatly

affect the speed of processing which, depending on the circumstances, may be very important.

Because the misuse-based IDSs are only as good as their signatures, the effectiveness of the system is clearly evident offline by simply looking at the completeness of the rule set or signatures it will search for. Often this list of signatures is referred to as a database of signatures. With this style of IDS the primary resource that is updated frequently is the database of signatures. Although often the terms "signature" and "rule" are used closely and in connection, veritably they are different; it is custom for a rule to hold a signature along with supplementary information such as the alert to report if the signature is encountered in a search.

While true that these kinds of systems only detect known attack classes, it does not mean they are not valuable for detecting new attacks. It is plausible to misconstrue that known attack-class signatures identify old attacks, and yet, this is not necessarily the case. Known attack classes are simply ones for which a vulnerability exists. It may or may not have been already exploited in a specific attack. Also, depending on the level of sophistication of the system's signatures it may be able to stop whole classes of possible attacks with a certain set of signatures.

By stark contrast to misuse detection, *anomaly-based* (also called behaviour-based) *intrusion detection* systems do not rely on definitions of what is suspected as malicious, incorrect, or abnormal. Rather they are programmed to identify what is normal; hence, they should also identify what falls outside this range. Typically the system's heuristics of what is normal are learned through self-learning and keeping statistics [21], but rules may also be input from a user. Anomaly IDSs can, thus, be characterized as identifying unknown actions, and consequently, the output from such a system may be harder to interpret [21]. Unfortunately, it is also possible that the conditioning process of learning normal behaviour is corrupted if the initial

conditioning happens during an attack or another anomaly.

Sometimes *specification-based detection* as another classification is considered, which is slightly different than misuse and anomaly detection. Instead of detecting bad or anomalous states it aims to detect states that are known not to be good; that is, it detects actions that violate a specification of valid actions (often on a per-program basis) [13]. It can be thought of as anomaly detection with all good behaviour pre-programmed (specified) rather than having the IDS learn the typical and normal behaviour.

2.3 Host-based versus Network-based IDSs

The difference between host- and network-based intrusion detection is markedly the location of the system and most importantly its input. Systems that can handle both types of input are called hybrids or distributed intrusion detection systems (DIDSs) [13].

A *host-based intrusion detection* system (HIDS) consists of an application, generally software, on a machine that is designed to inspect input actions that are internal to the machine like system calls, application and audit logs, file-system modifications, and other host activities and states. A commonality often seen in HIDSs is the use of an object or checksum database that catalogs the last or known good states of the objects being monitored. Attackers that know of a HIDS on their target system may try to circumvent the HIDS's detection by covering up traces of their attacks through modifying entries in this database so as to not set off alarms during the next HIDS scan. For this reason a HIDS database needs to be strongly, often cryptographically, protected.

A *network-based intrusion detection* system (NIDS) may take the form of an inde-

pendent network appliance or device tapped into the network with associated processing capabilities. It monitors network activity, and therefore, its input is solely in the form of the traffic on the network. Since frequently attacks on networks or machines within them originate outside of the network in question, NIDSs have a wide range of possible attacks to detect from the outside (ingress). These typically include, but are not limited to, denial of service (DoS) attacks, port-scans, spreading viruses, and attempts to break into or exploit vulnerabilities in computer systems by malicious individuals, worms, or other malware self-spreading on the network. However, NIDSs can also help to warn about or guard against sensitive data and attacks within the network or leaving the relevant network (egress).

The main niche of examination in this thesis is the intersection between network-based intrusion detection systems and signature-based intrusion detection systems. In the next sections we explore this particular sort of system in the details relevant to the remainder of this thesis.

2.3.1 Traits of Signature-based and Network-based IDSs

When considering intrusion detection systems both host- and network-based systems have their own traits, advantages, and strengths, but there is no need to choose between them. Both may be easily used together—independently in the case of separate systems, or dependently in the case of hybrid systems. While both kinds of systems have distinctive and common traits, understanding those of the network-based systems of interest in this thesis provides insight as to how they are useful and when. Network-based intrusion detection systems have at least seven possible characteristics that make them appropriate for detecting intrusions [54]. Actual advantages and disadvantages with respect to these traits are different on a system-by-system basis.

1. **Low cost of ownership for wide coverage**, meaning that a single system can detect intrusions directed at a whole network of hosts. This entails multiple costs, as opposed to just monetary costs.

2. **Packet analysis** of network traffic containing known network protocol fields and payload data in the packets as well.

3. **Evidence gathering** that is permanent, meaning that the existence of certain traffic in the network cannot be denied, hidden, or undone. It may be captured and kept as evidence.

4. **Real-time detection** is most often used (and sometimes response); although, capturing traffic and later passing it through a NIDS is often possible.

5. **Suspicious activity detection**, meaning that even attacks that do not succeed can be detected and researched depending on how and where the system is setup.

6. **Verification** of the type of traffic expected in accordance with the existing security policy or as a complement to other monitors.

7. **Operating system independence**, meaning that the system works regardless of what operating systems are running on the hosts in the relevant network. Nevertheless, depending on the actual NIDS's use, it may still be beneficial to be aware of the operating systems running on the resources under protection.

Security administrators frequently opt to include a signature-based NIDS as part of their layered defence solution. This is largely due to a specific instance of this kind of system called Snort [72, 77]. Snort is a free, GPL-licensed, open-source NIDS distributed by Sourcefire [78]. A key component in the success of Snort is its freely available database of signatures which is regularly updated—albeit, there is now a

paid update service available as well, providing more frequent updates. This significantly lessens the burden for administrators by having a prepared and widely reviewed list of vulnerabilities that affect networks world-wide. Naturally, depending on the use of the intrusion detection system, the list could and probably should be pruned and tailored to the needs for the circumstances where the system is to be deployed.

With respect to signature- versus anomaly-based detection methods, selecting an IDS is no longer an either/or proposition because many systems use both techniques. However, due to the wide-spread use of Snort and other factors, signature-based intrusion detection systems may be more prevalent than anomaly-based intrusion detection systems or hybrids within the context of networks [41]. Though this could be the case for various reasons, one strong contributing reason is certainly because signature-based approaches are usually able to process input (network traffic) at much faster speeds and empirically with far less resources (processing and memory) [84]. This assumes that statistical modeling and the self-learning processes typically used in anomaly detection are more complicated and slower than the tasks performed in misuse detection; this is something that is always changing. Moreover, there is some degree of inscrutability with respect to the actions of an anomaly-based IDS whenever self-learning is involved. Thus, the problem of false positives and understanding the true positives is another significant reason.

Systems that are possible to extend into hardware implementations—where processing and memory restrictions are of even more concern—are also likely to grow faster than solutions that are not hardware-friendly. Of all these things, the speed factor is especially important in a network setting where real-time processing may be desired without interruption by overloading the system with too much or the wrong kind of input. With increasing network speeds this has become even more significant; currently, even the signature-based NIDSs are faltering to keep up to modern-day net-

work speeds at or over the gigabit (one billion bits) per second rates. An important challenge addressed in this thesis is looking at ways to speed up signature matching engines and algorithms used for NIDS as to, in effect, speed up the whole system's processing of input. Because of the factors described in Section 2.3.2 that NIDSs often deal with, it is understandably all the more crucial why NIDS performance is so heavily stressed and sought to be improved.

2.3.2 Deploying a Signature-based NIDS

Network-based intrusion detection systems face many of the security troubles that most software faces, but moreover, they must also deal with network communications issues. Nowadays, deploying an IDS in a local network means it is dealing with switched media not shared media—which was commonplace only a few years ago—unless dealing with a wireless network. This means that instead of network hubs and bridges, layer-2 switches like those present in modern-day Ethernet local area networks (LANs) are in place [54]. Consequently, all of the connections on the network are no longer able to be intercepted by anyone other than the parties in communication which is a good thing for network speed, protocols, and privacy. In other words, Ethernet broadcast domains have been isolated to, in effect, create point-to-point networks.

In this style of full-duplex (as opposed to half-duplex) network, a NIDS is typically setup to have access to all traffic inside, outside, or inside and outside the network by configuring port mirroring from a switch or by use of a network tap [54]. Regardless of where the NIDS is to be setup on the network, it is always required to configure its network interface on which it detects in *promiscuous mode*—unless configured in inline network-node IPS style, that is, it is able to block and affect network traffic. A network interface in promiscuous mode intercepts all traffic on its receiving channels

not just the traffic destined for its own media access control (MAC) or layer-2 address. Such a network interface in relation to a NIDS is referred to as a sensor interface. It is commonplace to not allow any traffic destined to or coming from the IDS sensor's interface as it could pose a security risk. In this way the sensor interface only listens or intercepts traffic destined to others. In addition, in many cases the transmit channel for this network interface can be disabled altogether. It is well regarded that communication with the IDS (or the IDS's sensors nodes) is best done over a separate interface [54].

When using a switch with port mirroring to the NIDS, the bandwidth and speed needed to receive the accumulation of traffic on the network link is required to be multiple times greater and faster because it is receiving traffic from multiple links combined into one link. For this reason sometimes multiple NIDSs are used for the sake of load balancing. Lastly, when deploying multiple NIDSs within one network it is crucial for certain kinds of NIDSs to see both directions of a connection to be able to detect SYN floods and other types of attacks [54]. To achieve this a flow switch [54] can ensure both directions of the same connection are sent to a single sensor interface.

2.3.3 Architecture of a Signature-based NIDS

To examine the architecture of a NIDS we will present an overview of Snort as an example and sample architecture. Understanding the Snort model is valuable here because it defines the typical framework in which this type of system's pattern matching algorithms are found. Furthermore, it is general enough that it often serves as a rough baseline for what many network-based intrusion detection systems look like if they do any signature matching at all.

For example, Bro [66], another NIDS, does signature matching, but also builds

in additional functionality to understand the context of the signature. Bro uses a different and what some may perceive as a more complex policy language to describe signatures and unusual activities. Bro is generally capable of the same functionality as Snort with respect to the detection engine. In fact, a Snort signature database can easily be converted to the Bro policy language using a script that is available with Bro [67]. Moreover, Bro naturally does things Snort cannot, unless Snort is configured with custom Snort preprocessors (see Preprocessors below). Bro can analyze network traffic at a much higher level of abstraction with powerful facilities for storing information about past activity and incorporating it into analyses of the current activity [67]. Unfortunately, this makes Bro more complex than Snort, and thus, it is less suited for extremely high-speed large networks.

A diagram summarizing the Snort architecture and the data flow of packets through the IDS is given in Figure 2.1.

Getting Data off the wire

The Snort data flow proceeds first to acquire traffic from the network link via libpcap [55]. Libpcap is a Unix-based implementation of the PCAP application programming interface for capturing data-link layer frames on a network. For our purposes we use Snort on the Linux operating system. Because PCAP implementations exist for most other operating system platforms as well, Snort is considered a platform independent application (it could be run on Microsoft Windows using WinPcap). Once the frames are captured they are passed through a series of Snort decoder routines that fill out all the structures associated with the layer 2 through 4 protocols. These structures, which we will refer to as packets, are then sent through the registered set of preprocessors.

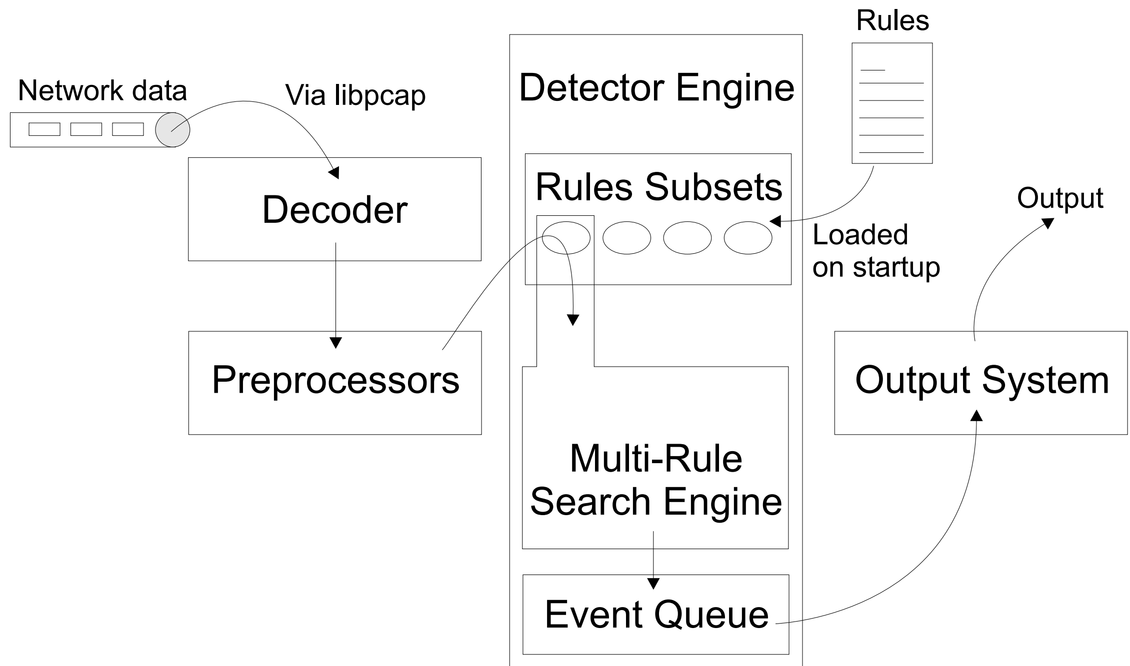


Figure 2.1: An overview of Snort's architecture

Preprocessors

Snort's preprocessors perform two fundamental functions. They either manipulate packets so the detection engine—the next stop in the data flow—can properly analyze them, or they examine traffic for suspicious use that cannot be discovered by signature detection alone [53]. Snort has a variety of configurable preprocessors (plug-ins), most of which have been added to combat new methods of NIDS evasion [53]. Examples of simple uses for preprocessors would be reassembling fragmented packets or decoding some disguised HTTP header fields. However, since preprocessors are customizable they can be made arbitrarily complex to detect intrusions that Snort's signature matching alone cannot find. After packets are circulated through all the preprocessors, they are sent on to the heart of Snort: the detection engine.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 22
(msg:"EXPLOIT ssh CRC32 overflow /bin/sh"; flow:to\_server,established;
content:"/bin/sh"; reference:bugtraq,2347; reference:cve,CVE-2001-0144;
classtype:shellcode-detect; sid:1324; rev:3;)
```

Figure 2.2: A sample Snort rule

Detection Engine

The detection engine has two main roles. First, upon starting or refreshing the Snort process and before any packets pass through the data flow, part of Snort's detection engine parses the Snort rules file line by line. This is the file containing the signature database. The rules used to be prioritized in this file so that at runtime Snort would build the rules structure in a certain order and then process rules in that same prioritized order, but since version 2.0, Snort has a rule optimizer (see discussion below).

Snort Rules

Snort's rules are each made up of two parts called the header and the option. The header contains prerequisite information that must match the protocol fields defined within the packet data structure. This is information containing fields such as the protocol type, IP address, and port number that a packet must have before Snort examines the option part of the rule. The option section contains the signature itself. This consists of the specific pattern to search for as well as some auxiliary information about the signature; for example, a vulnerability it detects and a message to log. We include a sample rule in Figure 2.2 with the option section delimited by the parentheses.

In this example the header is concerned with the source and destination IP addresses as well as ports. The source information is on the left side of the arrow and

the destination information is on the right side of it. The "any" keyword in the header signifies that any port number will match in this case for the source port of a packet. The "$" symbols indicate variables which when loaded by Snort get substituted for their configured value.

As mentioned above, Snort has a rule optimization process with a rule classifier and manager that utilize a set-based methodology for managing rules and applying them to packets when it comes time to examine them. Rule subsets (groups) are first formed by the classifier based on unique rule and packet parameters using a classification scheme based on set criteria [76]. Based on the parameter variations, the rules are divided into each possible subset that could possibly match packets passing through the system. When a packet passes through, it is the rule manager component of Snort that ensures the packet is associated to the correct subset. Naturally if a packet does not even match a single subset, then it simply passes through the packet queue without any processing. However, often this is not the case and a subset is selected. Once a rule subset is selected for a packet the next component of the detection engine, the multi-rule search engine, begins processing it.

Multi-rule Search Engine

The multi-rule search engine is broken into three distinct searches based on unique Snort rule properties [76]:

1. **Protocol field search** – The protocol field search allows a rule to specify a value in a particular protocol field to search for.

2. **Generic content search** – The generic content search allows a rule to specify a byte string to match against anywhere in the packet including the payload data.

3. **Packet anomaly search** – The packet anomaly search allows a rule to specify characteristics of a packet or packet header that is cause for alarm.

The packet anomaly search is a specific type of signature detection in Snort that would qualify as an atypically complex signature. An example of a packet anomaly rule is one that looks for ICMP packets of over 800 bytes in length [76]. Note that under the classification presented in this chapter, the packet anomaly search is still considered misuse detection given that is it searching for a known misuse.

The multi-rule search engine uses a configurable high-performance multiple-pattern matching algorithm to find all occurrences of generic content patterns which is the lengthiest part of the whole packet processing [36, 86]. Some possible algorithms for use here will be examined closely in the following chapters (Chapters 4 and 5). When a match is found in any of the three search types, a Snort rule is fully validated, and an event is generated and added to the event queue. It is the event selector component that processes the event queue.

Event Queue

The event queue allows Snort to keep track of multiple rule matches for every packet that passes through the IDS. The event selector prioritizes these events with the longer matches having higher priority. Currently, the event selector always selects the event in the queue with the highest priority to be sent to Snort's output system [76].

Output System

The output system is determined by which of Snort's output plug-ins are configured for use. Output plug-ins can range from simple comma-delimited output to complex relational database output. Also, a unified binary output format has been specifically

designed for Snort to outsource the writing to databases, which has commonly been a performance bottleneck in the past. This specific component that handles the output from Snort is called Barnyard [77].

2.4 Chapter Summary

In this chapter we presented the required background on intrusion detection systems to fully grasp the concepts and contributions of this thesis. First, we introduced a classification of intrusion detection systems that organized them as passive or reactive; signature- (misuse-), specification-, or anomaly-based; and lastly host- or network-based. Intrusion detection systems classically fall to one side of these in all three of these classifications; however, we also noted that as systems become increasingly complex they tend to be partial or full hybrids of many contrasting types. With respect to the network-based intrusion detection systems we introduced a popular exemplification, Snort, which also falls into the class of signature-based NIDSs. We further examined its architecture and attributes in preparation for the following chapters which introduce the state of the art in pattern matching algorithms relevant to NIDSs such as Snort.

Chapter 3

Single-Keyword Pattern Matching Algorithms

Pattern matching algorithms solve the general keyword pattern matching problem. That is, given a fixed and finite non-empty set of keywords and an input string, they find all occurrences of any of the keywords in the input string [88]. In this problem the input string is finite as well, but often a set of (multiple) input strings is used as input when searching for the keywords. In our case in particular, the input strings will be packets in the detection engine of the NIDS, and therefore, there will be many of them to process rapidly. This implies an alphabet size of 256 (quite large), and the size of the set of keywords on the other hand will be extremely small in comparison to the number of input strings. Furthermore, the set of keywords is known before the algorithm begins processing the input. Should this not be the case, if efficient modifications to the keyword set are needed, the searching process is known as dynamic string matching [63].

Herein, we refer to computation performed on the set of keywords before processing the input as offline computation, pre-computation, or preprocessing. Because the

time involved in pattern matching (processing the input in search of matches) will far outweigh the pre-computation time, the performance of the pre-computation is not emphasized. This is typical when analyzing a pattern matching algorithm, and partly because the length of the input to be processed may not be available at pre-computation time and takes priority over the length of the algorithm parameters to be preprocessed (i.e. the keyword set). Thus, our assumptions about the keyword set and input string set are common assumptions.

The general keyword pattern matching problem has a specific instance that has been shown empirically as easier to solve than the general problem. Before examining algorithms of interest to solve the general problem in Chapter 4, we present some classic algorithms solving this special case as necessary background and a stepping stone to understanding the solutions to the general problem. This special situation is the case when the size of the keyword set is one. This is also known as a *singleton keyword set* [88].

For the discussion of the single-keyword pattern matching algorithms we use the convention that the keyword x has length m and the input string y has length n. The lengths represent the number of characters and in our case a character is any possible 8-bit configuration, thus, taking one byte. We do not consider the case of multiple input strings as the algorithms are simply repeated for all input strings. Of course customarily, the pre-computation only needs to be done once before processing the first of the input strings. Note that in our pseudocode for the pattern matching algorithms presented in this thesis, we use the convention of outputting all the indexes of the character (byte) in y that matches the leftmost character in the keyword x (or in a keyword from the keyword set in the multiple-pattern matching algorithms presented in following chapters).

The two key criteria we examine with the description of each algorithm is the

running-time performance of the algorithm and the memory space required. The running-time performance, also referred to as time complexity, is measured in the number of machine steps, and in this case we are primarily concerned with character or byte comparisons. Of course, fewer steps correspond to a faster and more favorable algorithm. The time complexity will be considered for the average and worst case of the algorithm's execution. By this we mean that the performance may depend on the keyword and the input as well as the algorithm itself. Another way to think of this is that the time complexity changes depending on the algorithm parameters. The average- and worst-case time complexities are the cases when the parameters cause the algorithm's performance to respectively either behave as expected on average or degrade to its worst performance—whereby the time and memory complexity may increase. The second criterion, the amount of memory consumed while the algorithm runs, is considered only in addition to the necessary space to store the keyword and input. Of course the keyword—and in the next chapter, the keyword set—must always be stored; thus, we take this for granted and ignore it in our analysis. Often the same assumption is made with the input. Although, the input may actually be coming in on-the-fly, and most algorithms need only to keep a certain amount—sometimes called a window—of the input. Generally, usage of less memory is favorable.

3.1 Brute Force Algorithm

The brute force algorithm's methodology is very simple to understand. The brute force algorithm consists in checking, at all positions in the input between positions 0 and $n - m$ (left to right), whether an occurrence of the pattern starts there or not. After each attempt, the algorithm shifts the pattern by exactly one position to the right [16]. Algorithm 3.1 gives the pseudocode for this simple approach to

single-keyword pattern matching.

This algorithm is considered naive based on the fact that it does no pre-computation on the keyword. However, that is advantageous if memory space is a concern, since it keeps only a small constant (not a function of n or m) amount of information about its current position. The worst-case time complexity of this algorithm is $O(m(n-m+1))$ (equals $O(nm)$) [25], and the expected number of comparisons is at most $2(n-m+1)$ (equals $O(n)$) for randomly chosen strings [25]. That is, on average we perform a constant number of comparisons at each position.

Algorithm 3.1 Brute Force Single-Keyword Matching Algorithm

```
1:  procedure BRUTE_FORCE(x, m, y, n)
        ▷ Input:
        ▷ x ← array of m bytes representing the keyword
        ▷ m ← integer representing the keyword length
        ▷ y ← array of n bytes representing the text input
        ▷ n ← integer representing the text length

2:      for j = 0 → n − m do                          ▷ For every possible character in y
3:          i ← 0

4:          while i < m and x[i] = y[i + j] do
5:              i ← i + 1               ▷ i = count of matching characters at and after y[j]
6:          end while

7:          if i ≥ m then
8:              output j
9:          end if
10:     end for
11: end procedure
```

3.2 Karp-Rabin Algorithm

Rabin and Karp [49] proposed a pattern matching algorithm that also generalizes to two-dimensional pattern matching [25]. This algorithm is based on the work of

Rabin, and specifically, uses the Rabin fingerprinting technique [68] which can be most simply thought of as a number-theoretic notion for the purposes of the discussion here. Rabin's fingerprinting technique is similar to a hash function in its use in this algorithm. It has special mathematical properties which we discuss further that account for it sometimes being referred to as a rolling hash.

A Rabin fingerprint is a short tag like a hash value for a larger input [15]. These fingerprints share the same property with output values from hash functions like MD5 [71] in that if two fingerprints are different, then the corresponding inputs that were used to create them are different. Furthermore, there is only a small chance of two different objects having the same fingerprint [15]. We call this property a *small probability of collision*.

Consider the fingerprint of the keyword and the set of all $n - m + 1$ fingerprints made of the $n - m + 1$ substrings of length m found in the input string of length n. We can compare these fingerprints instead of actually comparing the keyword against the portion of the input. This saves comparisons at each position (compare to the inner loop in Algorithm 3.1) because instead of comparing things of length m, we compare only the fingerprints. Moreover, typically if we used a normal hash function to calculate the fingerprints it would take $O(m)$ time to generate a fingerprint (linear in the length of the input to be hashed); however, using Rabin's fingerprinting method we can calculate a fingerprint on an input of length m in constant time and using few simple machine operations. This is not the case in general, but in the case at hand we can incrementally update the fingerprint result as the window of length m slides over the input. During this time we maintain only two fingerprints: the current one for the window and the one for the keyword which together take $O(\text{size of fingerprint})$ memory space.

Consider two byte strings for the input and the keyword. The first fingerprint

over the first m bytes $(0...m-1)$ of the n byte input is calculated in time $O(m)$, but when we shift the window to calculate the fingerprint of the input for the next m bytes $(1...m)$ the fingerprint can be updated in constant time. This is true for all subsequent positions as we shift the window over the input string. The key to the speed of this algorithm is that the full fingerprint calculation is done only twice: first for the keyword itself and once for the first m bytes of the input. The first of these operations can be considered pre-computation since it is only done once. The second of these operations will have to be done once per input string to be searched, but we assume this is still only once in our pseudocode (only one input string y is accepted). Hence, we say that the pre-computation phase of the Karp-Rabin algorithm takes time $O(m)$. As the window slides through the input updating the fingerprint result, the result and the keyword's fingerprint can be compared to check for a match. This sliding and comparison process will happen exactly $n-m+1$ times and each step can be done in constant time; therefore, the matching phase of this algorithm runs in expected time $O(n-m+1)$.

Unfortunately, if the fingerprints match in the comparison it does not guarantee a pattern match because there is a negligibly small probability that two Rabin fingerprints are the same when the sources from which they were created are different (for a discussion on this probability see Broder [15]). In order to guarantee a match the actual keyword and the window portion of the input must be directly compared. When the fingerprints match but the keyword is not matched it is referred to as a *spurious hit* [25]. In theory if we had a spurious hit at every position of the input checked we would have a worst-case time complexity of $O(m(n-m+1))$ which is no better than the brute force algorithm. In practice, spurious hits can be made very infrequent by fine-tuning the parameters to the Rabin fingerprinting algorithm. Therefore, if we do not concern ourselves with this unlikely worst case the Karp-Rabin algorithm is a very

good and simple algorithm to implement once the Rabin fingerprinting algorithm is handled. Lastly, it is also common to see implementations that use a simpler function than the original Rabin fingerprint function. This works and keeps the same time complexity so long as the new function maintains the constant time update property so that a window slide to the next position and the corresponding fingerprint update is done in constant time. The update to the fingerprint at each position taking place in constant time is theoretical, but in practice this operation may be simple or complex depending on the chosen function; thus, this is an important consideration when changing the function. Algorithm 3.2 demonstrates the Karp-Rabin method using a simpler function than the Rabin fingerprinting technique. This simpler function from Corman et al. [25] and uses a prime q. Arithmetic computations are performed modulo q. For efficiency in this algorithm with this simpler function, the prime value q should be chosen such that $256q$ fits just inside one computer word, which allows all the necessary computations to be performed with single-precision arthmetic [25]. The constant 256 is chosen to match the alphabet size for our purposes.

3.3 Knuth-Morris-Pratt Algorithm

Knuth, Morris and Pratt [51] proposed a keyword pattern matching algorithm (herein the KMP algorithm) that runs left to right over its text input in linear time and improves on an algorithm proposed earlier by only Morris and Pratt [59]. This algorithm's pre-computation creates an array with information about how the keyword matches against shifts of itself [25]. For example knowing we have matched exactly r characters somewhere in the input allows us to determine that certain shifts are invalid; thus, avoiding the shifts that the naive brute force algorithm executes only to then fail at a subsequent match attempt [25].

Algorithm 3.2 Karp-Rabin Single-Keyword Matching Algorithm

```
 1: procedure KARP_RABIN(x, m, y, n, q, a)
        ▷ Input:
        ▷ x ← array of m bytes representing the keyword
        ▷ m ← integer representing the keyword length
        ▷ y ← array of n bytes representing the text input
        ▷ n ← integer representing the text length
        ▷ q ← integer value of prime number to use for simple fingerprinting
        ▷ a ← integer representing the alphabet length (256 for our purposes)

 2:     h ← a^(m−1) mod q                         ▷ Used in fingerprint calculation
 3:     p ← 0                                     ▷ Keyword fingerprint
 4:     t_0 ← 0                   ▷ Fingerprint of sliding window over text input

 5:     for i = 1 → m do                          ▷ Pre-computation
 6:         p ← (a ∗ p + x[i]) mod q              ▷ Calculate keyword fingerprint
 7:         t_0 ← (a ∗ t_0 + y[i]) mod q          ▷ Calculate initial fingerprint
 8:     end for

 9:     for s = 0 → n − m do                      ▷ Matching
10:         if p = t_s then                       ▷ Fingerprints match
11:             if x[1...m] = y[s + 1...s + m] then   ▷ Confirm match
12:                 output s
13:             end if
14:         end if

15:         if s < n − m then
16:             t_{s+1} ← (a(t_s − y[s + 1] ∗ h) + y[s + m + 1]) mod q   ▷ Update t
17:         end if
18:     end for
19: end procedure
```

As a concrete example let us say the keyword pattern $x = \texttt{ababaca}$ and we have just matched 5 characters in y starting at position i only to find a mismatch for the character `c` at position $i + 5$. The naive shift of the brute force algorithm would shift the pattern by one position, but of course it would fail because the character b at $y[i + 1]$ would not even match the first character of the keyword x. The KMP algorithm knows what the appropriate shift should be, and may shift over multiple positions without missing any potential matches. The trick to doing this correctly is in the pre-computation step where the prefix function for the keyword is built into an array. If we define x_r as the first r characters of x that have been matched at any point, the prefix function array table at position r ($table[r]$) contains the length of the longest prefix of x that is a proper suffix of x_r. Using this information the shift to the next position is always possible to calculate as (the current position) + (the number of matched characters before the mismatch (r)) - ($table[r]$) [25].

Algorithm 3.3 below shows the KMP algorithm and the pre-computation step of how the prefix function array ($table$) is created in time and with memory space $O(m)$. In the worst case the matching phase of the KMP algorithm executes $2n - 1$ character comparisons [16].

3.4 Boyer-Moore Algorithm

The review of the Boyer-Moore [14] algorithm is an important contribution herein not only because it is generally a popular choice, but it also inspired aspects of the Commentz-Walter [23] algorithm and others that we review in the next Chapter. The suffix-based idea used by the Boyer-Moore algorithm is considered as the most well-known and efficient, yet still basic, keyword pattern matching algorithm in general. More specifically, it is very efficient given a large alphabet which we have to cope

Algorithm 3.3 Knuth-Morris-Pratt Single-Keyword Matching Algorithm

```
1:  procedure KMP(x, m, y, n)
        ▷ Input:
        ▷ x ← array of m bytes representing the keyword
        ▷ m ← integer representing the keyword length
        ▷ y ← array of n bytes representing the text input
        ▷ n ← integer representing the text length

2:      table ←COMPUTE_PREFIX_KMP(x, m)                ▷ Pre-Computation
3:      q ← 0
4:      for i = 1 → n do                                ▷ Matching
5:          while q > 0 and x[q + 1] ≠ y[i] do
6:              q ← table[q]
7:          end while
8:          if x[q + 1] = y[i] then
9:              q ← q + 1
10:         end if
11:         if q = m then
12:             output i − m
13:             q ← table[q]
14:         end if
15:     end for
16: end procedure

17: procedure COMPUTE_PREFIX_KMP(x, m)
        ▷ Input:
        ▷ x ← array of m bytes representing the keyword
        ▷ m ← integer representing the keyword length

18:     table ← newArray[m + 1]                         ▷ Prefix Shift Table
19:     table[1] ← 0
20:     k ← 0
21:     for q = 2 → m do
22:         while k > 0 and x[k + 1] ≠ x[q] do
23:             k ← table[k]
24:         end while
25:         if x[k + 1] = x[q] then
26:             k ← k + 1
27:         end if
28:         table[q] ← k
29:     end for
30:     return table
31: end procedure
```

with herein since we assume any 8-bit pattern is a valid character. We do present other more complex and less well-known algorithms that outperform this algorithm subsequently.

Although the idea of a large alphabet is subjective, in string matching 256 is fairly large and what is of importance here is that as the size of the alphabet (number of characters in the alphabet) grows so does the potential for larger shifts. This algorithm searches from left to right over the input, but performs character comparisons within its sliding window of size m in reverse order (right to left). It uses two pre-computed functions on the keyword which help it run in time as fast as $O(n/m)$ (the best case) by skipping over parts of the input that are not necessary to check during matching.

The first and simpler pre-computation step done by the Boyer-Moore algorithm is the creation of the *bad character shift table* (see Algorithm 3.4). This table is essentially an array indexed by all characters in the alphabet storing integers that represent how far the algorithm may shift upon a mismatch. At all characters indices not in the keyword a value equal to the keyword's length is stored since (keeping in mind the reverse order comparison) if a character in the input is encountered that does not appear in the keyword we can shift entirely past it as in the example below (steps 1 through 3). All table positions of characters present in the keyword store the distance from the right most character in the keyword. Therefore, if the character `a` started the keyword and did not appear anywhere else in the keyword then the bad character shift table's (call it *bctable*) value at position `a` *bctable*[`a`] would be $m - 1$ (as seen in Step 0 below). When the character does exist in the keyword this bad character shift allows the algorithm to immediately realign the keyword's right most appearance of the character that was mismatched to the character that caused the mismatch in the input (see step 4 in the example below). Because this kind

of realignment could potentially result in a negative shift in some situations there is another more complex pre-computed function which is also looked up at match time and the maximum value returned from both functions is used. We look at this second pre-computed function next, which in the example below would lastly align the keyword under the letters `algorithm` in the input whereby a match is found (not shown in the example).

Example:

Input: `example with the boyer-moore algorithm`

Keyword: `algorithm`

Step 0. The shift values for the characters present in the keyword (all other characters have value 9):

```
algorithm
876543210
```

Step 1. First the `w` in the input is mismatched with the `m` in the keyword.

```
        |
example with the boyer-moore algorithm
algorithm
        |
```

Step 2. Now the pattern can be shifted by 9 which corresponds to the value 9 in the table at index `w`. Next the `b` in the input is mismatched with the `m` in the keyword.

```
                 |
example with the boyer-moore algorithm
         algorithm
                 |
```

Step 3. The pattern can be shifted by 9 which corresponds to the value 9 in the table at index `b`. Next the `r` in the input is mismatched with the `m` in the keyword.

```
                          |
example with the boyer-moore algorithm
                  algorithm
                          |
```

Step 4. The pattern can be shifted by 4 which corresponds to the value 4 in the table at index `r`.

```
                              |
example with the boyer-moore algorithm
                      algorithm
                              |
```

Algorithm 3.4 Boyer-Moore Bad Character Shift Pre-computation Algorithm

1: **procedure** COMPUTE_BAD_CHARACTER_SHIFTS_BM(x, m)
    ▷ Input:
    ▷ $x \leftarrow$ array of m bytes representing the keyword
    ▷ $m \leftarrow$ integer representing the keyword length

2:     $alphabet_size \leftarrow 256$
3:     $table \leftarrow \textbf{new}Array[alphabet_size]$ ▷ Bad Character Shift Table

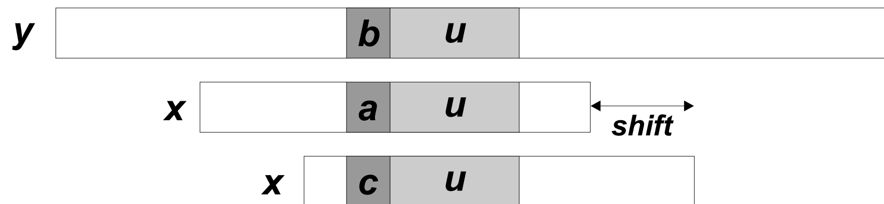
4:     **for** $i = 0 \rightarrow alphabet_size - 1$ **do**
5:         $table[i] \leftarrow m$
6:     **end for**

7:     **for** $i = 0 \rightarrow m - 1$ **do**
8:         $table[x[i]] \leftarrow m - i - 1$
9:     **end for**

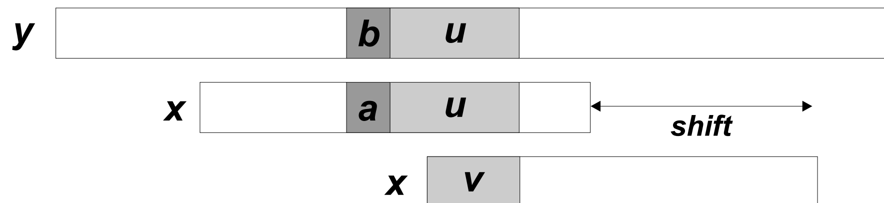
10:     **return** *table*
11: **end procedure**

The second pre-computation step done by the Boyer-Moore algorithm is the creation of the *good suffix shift table* (see Algorithm 3.5). This table is stored in an array of size $m+1$. Similar to the KMP algorithm this part of the Boyer-Moore algorithm's pre-computation creates an array with information about how the keyword matches against shifts of itself. The good suffix shift table contains values that provide one of two forms of shifting [16].

As an example let us say a mismatch occurs between the character $x[i] = \texttt{a}$ of the keyword and the character $y[j+i] = \texttt{b}$ of the input during an attempt at position j (of the keyword sliding window), and let $u = x[i+1...m-1] = y[j+i+1...j+m-1]$ which is the part already matched. The good suffix shift consists in aligning the segment u with its rightmost occurrence in x that is preceded by a character different from $x[i]$ which was the mismatched character (see Figure 3.1 part i) [16]. If there exists no such segment then the second form of the shift consists in aligning the longest suffix v of u with a matching prefix of x (see Figure 3.1 part ii) [16].



i) The good suffix shift, *u* re-occurs preceded by a character *c* different from *a*.



ii) The good suffix shift, only a suffix of *u* re-occurs in *x*

Figure 3.1: Boyer-Moore algorithm's good suffix shift examples [16]

Algorithm 3.5 Boyer-Moore Good Suffix Shift Pre-computation Algorithm

```
1:  procedure COMPUTE_GOOD_SUFFIX_SHIFTS_BM(x, m)
        ▷ Input:
        ▷ x ← array of m bytes representing the keyword
        ▷ m ← integer representing the keyword length

2:      table ← new Array[m]                                ▷ Good Suffix Shift Table
3:      suf_table ← new Array[m]              ▷ Temporary Suffix Information Array
4:      suf_table[m − 1] ← m
5:      g ← m − 1
6:      j ← 0
7:      for i = m − 2 → 0 do                                      ▷ Building suf_table
8:          if i > g and suf_table[i + m − 1 − j] < i − g then
9:              suf_table[i] ← suf_table[i + m − 1 − j]
10:         else
11:             if i < g then
12:                 g ← i
13:             end if
14:             j ← i
15:             while g ≥ 0 and x[g] = x[g + m − 1 − j] do
16:                 g ← g − 1
17:             end while
18:             suf_table[i] ← j − g
19:         end if
20:     end for
21:     for i = 0 → m do                                            ▷ Building table
22:         table[i] = m
23:     end for
24:     j ← 0
25:     for i = m − 1 → −1 do
26:         if i = −1 or suf_table[i] = i + 1 then
27:             while j < m − 1 − i do
28:                 if table[j] = m then
29:                     table[j] ← m − 1 − i
30:                 end if
31:                 j ← j + 1
32:             end while
33:         end if
34:     end for
35:     for i = 0 → m − 2 do
36:         table[m − 1 − suf_table[i]] ← m − 1 − i
37:     end for
38:     return table
39: end procedure
```

Finally, the Boyer-Moore matching algorithm (see Algorithm 3.6) itself performs very well on average thanks to spending $O(m+$ size_of_alphabet$)$ in pre-computation. With a large size alphabet this algorithm is frequently expected to make large shifts of length m bypassing large sections of the input. In the best case a running time of $O(n/m)$ can be achieved as mentioned above, but in the worst case performance is still quadratic or $O(nm)$ [63] with a periodic pattern (one having repeated cycles). With a non-periodic pattern as the worst case, Cole proved that there are at most $3n$ character comparisons [22]. Finally, memory requirements of the entire algorithm are $O(m+$ size_of_alphabet$)$ bytes because of the two arrays computed by the two pre-computation functions.

Although the Boyer-Moore algorithm is very fast in practice, there have been numerous improvements made to it. The first notable improvement was made by Horspool [46]. Horspool's variant simplifies the Boyer-Moore algorithm greatly while also making it faster in general. Horspool noted the bad character shift was usually the longest shift and considered a modification that allows us to omit the second more complex shift table. Horspool's algorithm works as follows [63].

Upon each move of the sliding window to position i we compare the rightmost character in x ($x[m-1]$) against $y[i+m-1]$ (call it a). Assuming they match, we check right to left against the keyword until we find the whole keyword or we find a mismatch at some text character. Subsequently, regardless of a match or mismatch, we shift the window according to the next occurrence of a in x. Pseudocode for what is commonly called the Boyer-Moore-Horspool (BMH) algorithm is given in Algorithm 3.7.

Lastly, many more new suffix-based string matching algorithms have been proposed that follow-up on the ideas of Boyer and Moore. Notably, Sunday's slight variant [81] of Horspool's algorithm performs shifts of even longer distances safely by

Algorithm 3.6 Boyer-Moore Single-Keyword Matching Algorithm

```
 1: procedure BM(x, m, y, n)
        ▷ Input:
        ▷ x ← array of m bytes representing the keyword
        ▷ m ← integer representing the keyword length
        ▷ y ← array of n bytes representing the text input
        ▷ n ← integer representing the text length

        ▷ good_table ← array of m elements – see Algorithm 3.5
        ▷ bad_table ← array of 256 elements – see Algorithm 3.4

        ▷ Pre-Computation
 2:     good_table ←COMPUTE_GOOD_SUFFIX_SHIFTS_BM(x, m)
 3:     bad_table ←COMPUTE_BAD_CHARACTER_SHIFTS_BM(x, m)
 4:     j ← 0

 5:     while j ≤ n − m do                                        ▷ Matching
 6:         i ← m − 1
 7:         while i ≥ 0 and x[i] = y[i + j] do
 8:             i ← i − 1
 9:         end while
10:         if i < 0 then
11:             output j
12:             j ← j + good_table[0]
13:         else
14:             j ← j + MAX(good_table[i], bad_table[y[i + j]] − m + 1 + i)
15:         end if
16:     end while
17: end procedure
```

Algorithm 3.7 Boyer-Moore-Horspool Single-Keyword Matching Algorithm

```
 1: procedure BMH(x, m, y, n)
        ▷ Input:
        ▷ x ← array of m bytes representing the keyword
        ▷ m ← integer representing the keyword length
        ▷ y ← array of n bytes representing the text input
        ▷ n ← integer representing the text length

 2:     alphabet_size ← 256                                   ▷ Pre-computation
 3:     table ← new Array[alphabet_size]                      ▷ Horspool Shift Table

 4:     for i = 0 → alphabet_size − 1 do
 5:         table[i] ← m
 6:     end for
 7:     for j = 0 → m − 1 do
 8:         table[x[j]] ← m − j − 1
 9:     end for

10:     i ← 0                                                 ▷ Matching
11:     while i ≤ n − m do
12:         j ← m − 1
13:         while j > 0 and x[j] = y[i + j] do
14:             j ← j − 1
15:         end while
16:         if j = 0 then
17:             output i
18:         end if
19:         i ← i + table[y[i + m − 1]]                       ▷ Shift
20:     end while
21: end procedure
```

only changing the shift to $i \leftarrow i + table[y[i+m]]$. In theory this results in a faster algorithm; however, in practice the Horspool variant beats the Sunday variant due to a lower number of memory references [63]. Furthermore, variants such as that of Galil [40] exist that bound the worst-case running time of these algorithms to $O(n)$ while suffering only slightly on the average.

3.5 Backward Oracle Matching Algorithm

Before we introduce the Backward Oracle Matching (BOM) algorithm [3, 4, 5] we introduce a few concepts as prerequisites to understanding the algorithm itself. Firstly, we introduce the idea of factor-based string matching which leads to average optimal running times, assuming that the characters of the text are independent and occur with equal probability [63]. A *factor* of a word is any substring of the word. This term is used to mean substring herein when discussing factor oracles. If we have a structure to tell us all factors of a keyword then we can use it as follows to match the keyword in searches.

Suppose that we read backward a factor u of the keyword, and that we mismatch on the next character a in the input text. That is, by using our structure as described above, we know au is definitely not a factor of the keyword. We may shift our window safely past the occurrence of a in the text [63]. Two such data structures can be created that recognize the set of factors for a word. Namely, they are directed acyclic word graphs (DAWGs or suffix automata) and factor oracles (factor automata). DAWGs, albeit more complex and timely to construct, were used first for string matching in the Backward DAWG Matching (BDM) algorithm [29], and subsequently in the Backward Non-deterministic DAWG Matching (BNDM) algorithm [62] which is a bit-parallel and improved version of BDM. Factor oracles were proposed

in string matching with the realization that to shift the window in the factor search as described, it suffices to know that au is not a factor of the keyword. Previous approaches focused on identifying u as a factor of the keyword. We continue the explanation of the BOM algorithm after first explaining factor oracles.

A *factor oracle* is an automaton data structure proposed by Allauzen et al. [3] to catalog all of the factors of a word. Other data structures exist that do the same thing differently such as a suffix trie or a suffix automaton (DAWG) [30]. Figure 3.2 shows an example comparing the three such types of data structures.

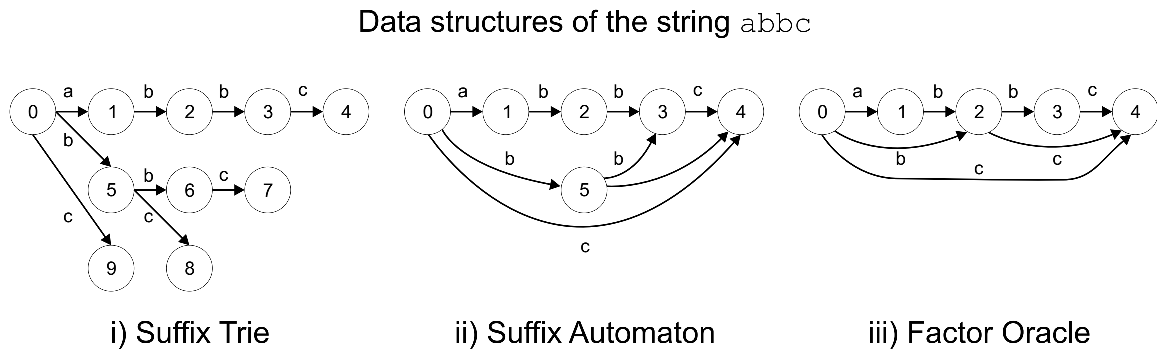


Figure 3.2: Three data structures that hold all factors of a word

The factor oracle is a special type of deterministic automaton that specifies every state as an accepting state. This means, starting from the initial state and at every state, it accepts only a factor (substring) of the word it is built upon. That is, it contains transitions to recognize factors of the keyword it is built upon. During the input of a word into the factor oracle automaton, if at any given state a transition of a certain character is not defined, then the word being input cannot exist as a factor of the keyword. Note that a factor oracle may also sometimes accept factors that are not present in the keyword upon which it is built [3]. The factor oracle is the data structure used in the BOM algorithm and is built using the keyword x. It holds that the factor oracle will always have $m + 1$ states and have between m and $2m - 1$

transitions [3]. For example, for the strings `aaaa` (best case) and `aaab` (worst case) the oracle would have m and $2m - 1$ transitions respectively.

There are many ways to construct a factor oracle for a keyword x; however, we will present the algorithm from Allauzen et al. [4] in Algorithm 3.8. Other algorithms that are not as useful in practice, but that help to understand factor oracles better are given in [19]. Algorithm 3.8 also uses an array S for a supply function over the nodes. The supply function specifies the end of the first $repet(i)$ where $repet(i)$ is the longest suffix of the first i characters of x ($x[0]...x[i-1]$) that appears at least twice. The construction of the factor oracle is the only pre-computation necessary for the BOM algorithm, and it has a linear (in m) running time and uses linear (in m) memory which is of course kept throughout the matching phase as well. Figure 3.3 shows an example of the steps of building a factor oracle one character at a time as performed by Algorithm 3.8.

To apply the factor oracle in a pattern matching algorithm is straightforward. The algorithm consists of the same rules as the earlier proposed Backward DAWG Matching algorithm [29, 30] except with the factor oracle in place of the DAWG or suffix automaton. The BOM algorithm is very fast in practice for long patterns—so larger shifts are possible—and even works well for small alphabets. On the average (over all possible inputs, albeit worse for worst-case inputs) its running time is thought to be optimal [4]. The matching phase of the BOM algorithm uses the oracle of the reversed keyword x. The search for a match proceeds right to left within a window that is shifted left to right (as in the Boyer-Moore algorithm [14]). The search from right to left stops when the characters scanned from y are no longer recognized by the oracle as a factor, meaning it is certainly not a factor of the reversed pattern [3]. If the matching stopped because of a mismatch at some character $y[i]$ then the window can safely shift to the left entirely past $y[i]$ and start scanning again. Algorithm 3.9

Algorithm 3.8 Factor Oracle Construction Algorithm

```
 1: procedure CONSTRUCT_FACTOR_ORACLE(x, m)
        ▷ Input:
        ▷ x ← array of m bytes representing the keyword
        ▷ m ← integer representing the keyword length

 2:     Create oracle                                   ▷ Factor oracle automaton
 3:     Create state_0 of oracle
 4:     S ← new Array[m]                                ▷ Initialize elements to 0
 5:     S[0] ← −1                 ▷ Supply function of initial state is undefined
 6:     k ← S[0]

 7:     for i = 1 → m do
 8:         Create state_i of oracle
 9:         Add new transition from state_{i−1} to state_i with label x[i − 1]
10:         k ← S[i − 1]
11:         while k > −1 and there is no transition from state_k with label x[i − 1]
                do
12:             Add new transition from state_k to state_i with label x[i − 1]
13:             k ← S[k]
14:         end while
15:         if k = −1 then
16:             S[i] ← 0
17:         else
18:             S[i] ← state index of wherever state_k leads to with transition x[i − 1]
19:         end if
20:     end for
21:     return oracle
22: end procedure
```

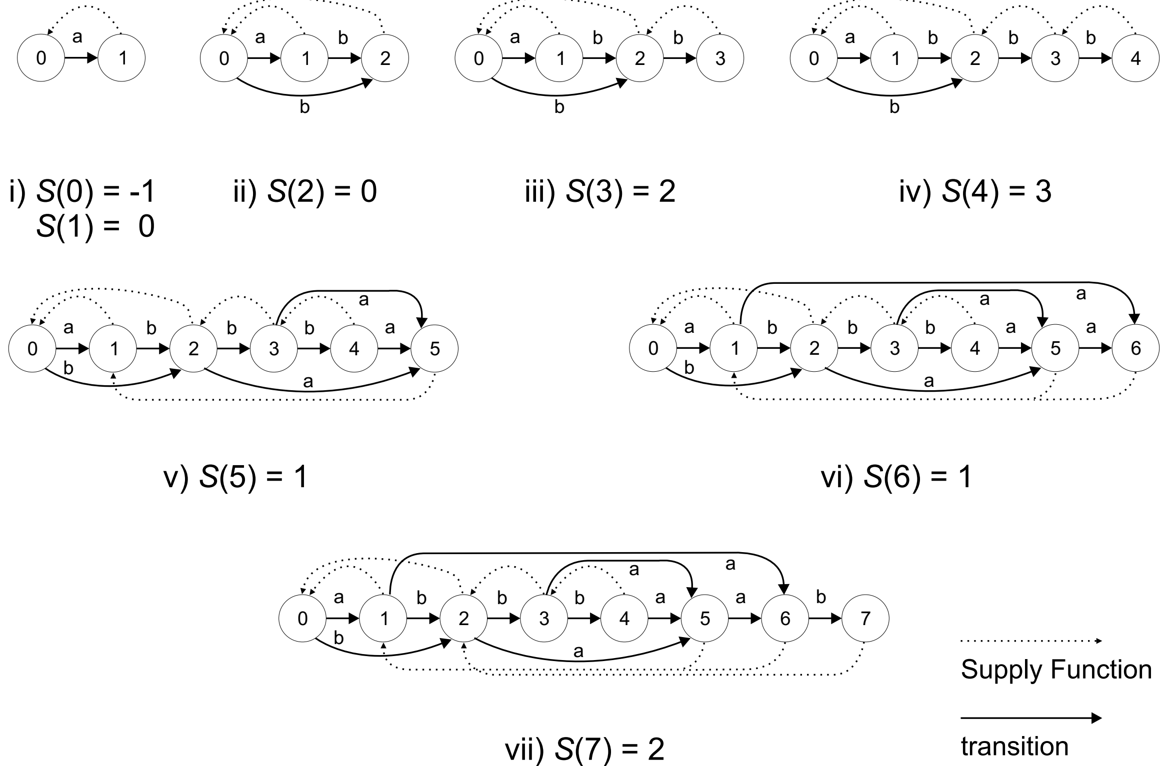


Figure 3.3: Iterations in the construction of a factor oracle for the word `abbbaab`

gives the pseudocode for the Backward Oracle Matching algorithm, and Figure 3.4 depicts the matching phase as described.

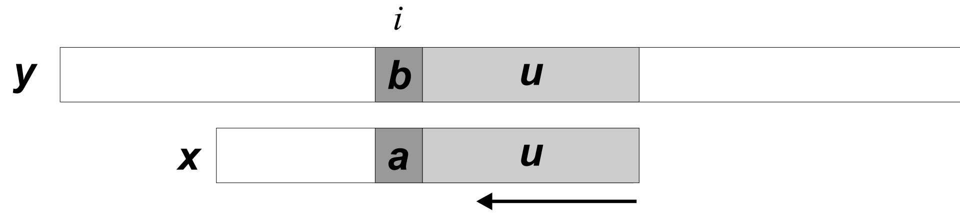
As mentioned on the average the BOM algorithm is optimal and thus sublinear in n [18] (see Chapter 4 for the meaning of sublinear herein). However, in the worst case the BOM algorithm has quadratic behaviour or more specifically has an $O(nm)$ running time. Allauzen et al. [4] show how to avoid this worst-case running time by their implementation of TurboBOM, an algorithm that scans using a combination of both the KMP algorithm [51] (detailed in Section 3.3) and the BOM algorithm. Namely, in a given window of length m the KMP algorithm scans forward first up to a critical point (usually less than half the window length), and then the BOM algorithm takes over, which of course provides for larger shifts. In addition to factor oracles, Allauzen et al. present the idea of suffix oracles and the Backward Suffix

Algorithm 3.9 Backward Oracle Matching Single-Keyword Matching Algorithm

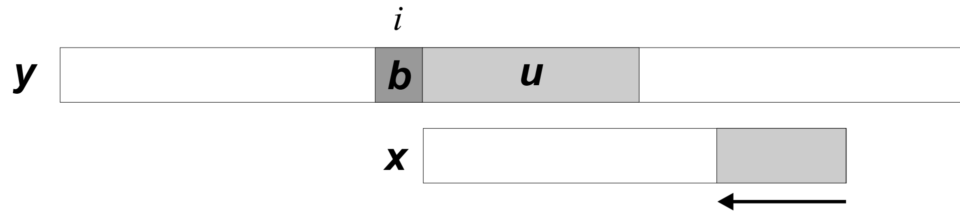
```
1: procedure BOM(x, m, y, n)
     ▷ Input:
     ▷ x ← array of m bytes representing the keyword
     ▷ m ← integer representing the keyword length
     ▷ y ← array of n bytes representing the text input
     ▷ n ← integer representing the text length

2:   x^R ← reverse(x)                                        ▷ reverse keyword
3:   oracle ← CONSTRUCT_FACTOR_ORACLE(x^R, m)                ▷ Pre-Computation

4:   for i = 0 → n − m do                                    ▷ Matching
5:     curState ← state_0 of oracle
6:     j ← m
7:     while curState exists do
8:       curState ← take transition labeled y[i + j] from curState
9:       j ← j − 1
10:    end while
11:    if j = 0 then
12:      output i − m + 1
13:      j ← 1
14:    end if
15:    i ← i + j
16:  end for
17: end procedure
```



i) Mismatch occurs at position i while searching



ii) New search starts with window shifted past i

Figure 3.4: The matching phase of the Backward Oracle Matching algorithm

Oracle Matching (BSOM) algorithm [4] which is similar and not shown here. In suffix oracles the only terminal states are those that recognize a suffix of the word upon which they are built. That is, a state s of a suffix oracle is terminal if and only if there is a path labeled by a suffix of the word from the initial state leading to s [4]. Figure 3.5 shows an example of a suffix oracle. Lastly, Allauzen and Raffinot [6] extend the BOM and BSOM algorithms to multiple-keyword pattern matching algorithms which we examine later (see section 4.5).

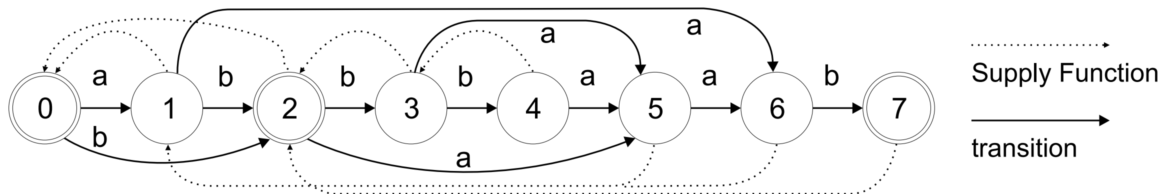


Figure 3.5: Example of a suffix oracle for the string `abbbaab`

3.6 Chapter Summary

In this chapter we presented a small sample of the state of the art in single-keyword pattern matching algorithms. In particular, we examined single-keyword pattern matching algorithms which inspired many extensions and techniques that are used in searching for multiple patterns in multiple-keyword pattern matching. In the next chapter we present several multiple-keyword pattern matching approaches from a generic standpoint. As in this chapter, for all the algorithms we give pseudocode that should easily be adaptable into a programming language.

Chapter 4

Multiple-Keyword Pattern Matching Algorithms

The pattern matching algorithms pertaining to the general keyword pattern matching problem are the ones of particular interest in this thesis. In this chapter we examine some of the proposed algorithms to solve this problem. Cleophas et al. [18] have presented a comprehensive taxonomy and toolkit of pattern matching algorithms that updates and collects the past works of Watson and Zwaan [89, 90]. That taxonomy will give the reader a much more complete idea of what is available in terms pattern matching algorithms, and how the different basic types and variations of algorithms have arisen. This section will only go over explanations and pseudocode of certain representative algorithms to match multiple keywords. Of course the set of algorithms that solve this problem is growing all the time and slight variations of the algorithms do exist; however, we try to present the algorithms herein as generally as possible and without any attunement to a programming language.

We use the same naming conventions and inspect the same criteria as the previous chapter. The difference in this section is that we have multiple keywords and multiple

keyword lengths. We choose to represent these as the array $x[0...p\text{-}1]$ of size p storing the keywords (which are byte arrays themselves) and the array $m[0...p\text{-}1]$ of size p storing the integer values representing the lengths of the keywords. Thus the byte array representing some keyword in x and position i has length $m[i]$ ($x[i][0...m[i]\text{-}1]$). The sum of the lengths of all keywords will be M. The input text byte array is still named y and has length n. Noted again is that in our pseudocode for the pattern matching algorithms presented in this thesis, we use the convention of outputting all the indexes of the character (byte) in y that matches the leftmost character in a keyword from the keyword set.

In the last chapter the Boyer-Moore [14] and BOM [3, 4, 5] algorithms introduce the notion of *sublinear running times* which we continue to use as a term for certain algorithms in this chapter. We use the term "sublinear" as has been seen in analyzing average-case behaviour in a good deal of modern string matching literature [5, 14, 18, 63, 90]. That is, the number of symbol comparisons is sublinear (less than linear) in the length of the input string. Note that this usage of the "sublinear" term is not strictly in agreement with most authoritative sources on asymptotic notation (i.e. Big-Oh, etc.) where a sublinear running time typically indicates $o(n)$. Typically a string matching algorithm herein said to have a sublinear average running time will on average run in cn steps, where c is $1/d$ and $d > 1$. The constant d is usually the length of the keyword (m in single-keyword algorithms) or the length of the shortest keyword (in multiple-keyword algorithms), but formally $o(n)$ describes a function f where if $f(n)$ is $o(g(n))$, then $0 \leq f(n) < c * g(n)$ for all constants $c > 0$. Because this must hold true for all values of c, the algorithms herein cannot be said to be $o(n)$ because for some c the definition is violated.

4.1 Aho-Corasick Algorithm

There are many approaches to recognizing patterns that involve using finite automata (also referred to as finite state machines). The Aho-Corasick (hereafter AC) algorithm [2] is one such classic algorithm. This algorithm also shares characteristics with the Knuth-Morris-Pratt [51] algorithm described in Section 3.3 [23, 25]. The idea is that a finite automaton is constructed using the set of keywords during the pre-computation phase of the algorithm and the matching involves the automaton scanning the input text string reading every character in y exactly once and taking constant time for each read of a character.

It is essential to first understand finite automata theory (which we do not cover here) to understand the AC algorithm's description. The notation we use for the AC automaton is a 7-tuple $(Q,\ q_0,\ A,\ \Sigma,\ g,\ f,\ o)$, where:

- Q is a finite set of states,
- $q_0 \in Q$ is the start (initial) state,
- $A \subseteq Q$ and is the set of accepting states,
- Σ is the input alphabet accepted,
- g is a function from Q x Σ into Q, called the good (or goto) transition function,
- f is a function from Q into Q, called the fail (or failure) transition function, and
- o is a function from Q into Q, called the output function.

If the automaton is in a state q and reads input character (byte) a, it moves (transitions) to state $g(q,\ a)$ if defined otherwise it moves to state $f(q)$. Also if the

automaton is in a state q, and q belongs to the set A then q is said to be an *accepting state*. Function o, the output function, returns whether or not any state $q \in A$. Aho and Corasick's original algorithm uses a function called *output* to test this and furthermore returns the keyword matched at the accepting state. The AC algorithm's automaton is such that a transition into an accepting state indicates a match of one or more keywords. The pseudocode for the matching phase of the algorithm is given below (see Algorithm 4.1). Further pseudocode of the construction of g, f and *output* are given in the original paper [2].

Algorithm 4.1 Aho-Corasick Multiple-Keyword Matching Algorithm

```
1: procedure AC(y, n, q0)
       ▷ Input:
       ▷ y ← array of n bytes representing the text input
       ▷ n ← integer representing the text length
       ▷ q0 ← initial state

2:     state ← q0
3:     for i = 1 → n do                                   ▷ Matching
4:         while g(state, y[i]) = fail do     ▷ while g(state, y[i]) is undefined
5:             state ← f(state)                 ▷ use the failure function
6:         end while
7:         state ← g(state, y[i])
8:         if o(state) ≠ ∅ then
9:             output i             ▷ This an accepting state, i.e. state ∈ A
10:        end if
11:    end for
12: end procedure
```

The AC algorithm uses a refinement of a trie (keyword tree) to store the set of keywords in a string matching special automaton. Figure 4.1 below shows the difference between a normal trie and an AC automaton. In the automaton $g(q,\ a)$, represented by a solid edge, shows the state entered from current state q by matching character (byte) a; thus, edge $(q,\ g(q,\ a))$ is labeled by a (if there are multiple matching characters at a state there are multiple edges). Also $f(q)$, represented by a

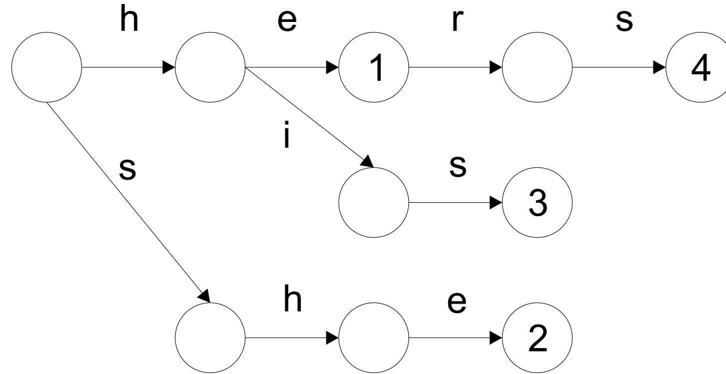
dashed edge, shows the state entered when the input character a does not match (no $g(q, a)$ is defined). If the dashed edge would simply point to the initial state it is not shown.

Building the AC automaton takes running time linear in the sum of the lengths of all keywords. This involves constructing the keyword tree for the set of keywords and then converting the tree to an automaton by defining the functions g and f and labeling states in A with the keyword(s) matched. The space or memory requirements of the AC algorithm can be taken directly from the automaton built during the precomputation because it is the only structure used in matching. Unfortunately the space can be quite large depending on the alphabet and keyword set. In the worst case it would be $O(M * |\Sigma|)$ where $|\Sigma|$ is the size of the alphabet Σ.

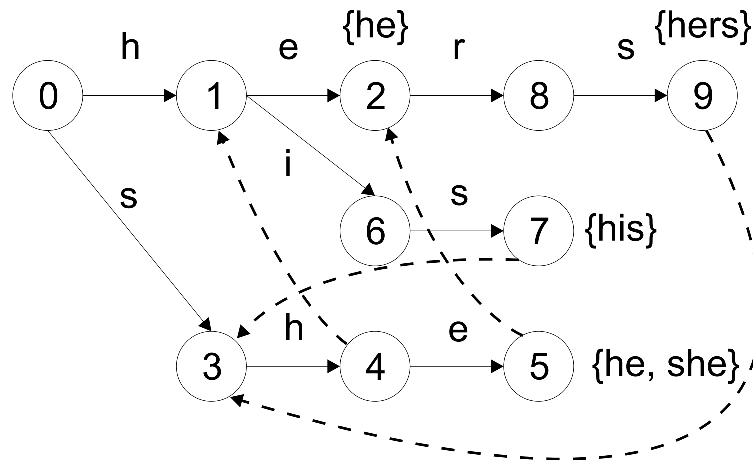
Once the automaton is built the matching is straightforward and involves simply stepping through the input characters one at a time and changing the state of the automaton—which happens in constant time. At every step we check if there's a match by seeing if the current state is an accepting state. Using this simple functionality the AC matcher always operates in $O(n)$ running time.

4.2 Commentz-Walter Algorithm

Commentz-Walter [23] achieved creating a mesh of both the Aho-Corasick [2] multiple-keyword pattern matching algorithm which has a running time linear in n and the Boyer-Moore [14] single-keyword pattern matching algorithm which runs in time sub-linear in n on average. The resulting multiple-keyword pattern matching algorithm matches multiple patterns simultaneously using a trie-like structure similar to Aho and Corasick, and using skips or shifts (i.e. filtering) similar to Boyer and Moore. Commentz-Walter also noted that the quadratic ($O(nm)$) worst-case running time



i) A keyword tree for x = {"hers", "his", "she", "he"}



ii) An Aho-Corasick finite automaton for x = {"hers", "his", "she", "he"}
Dashed: fail transitions; those not shown lead to $q0$

Figure 4.1: An example of a trie versus an Aho-Corasick automaton.

behaviour of the Boyer-Moore algorithm could be improved upon to be linear in n as shown in the work of Galil [40]. As such, Commentz-Walter derived two different algorithms called B and B1 which have quadratic ($O(n * \max\{m[0], m[1], ..., m[p-1]\})$) and linear ($O(n)$) worst-case running times respectively. Algorithm B is the main work of the simpler of Commentz-Walter's literature [23] and has a simpler pre-computation phase than B1. Furthermore, B1 takes more memory during the pre-computation and search than B by remembering the input text bytes that were already scanned [23].

Details of B1 are available in Commentz-Walter's lengthier report [24]. Both algorithms maintain a pre-computation phase that is linear in the total length of all keywords or $O(M)$, and both achieve slightly sublinear (in n) matching-phase running times on average [23] which could be as good as $O(n \,/\, \min\{m[0], m[1], ..., m[p-1]\})$ in the best case. Algorithm B will be described first starting with the functions created during the pre-computation phase.

The pre-computation phase of algorithm B starts by creating a basic trie data structure using the reversed keywords. Each trie node v, except the root node, is labeled with a character (byte) from a keyword. For a brief background on the trie data structure see [23]. A simple diagram of a trie data structure is given in Figure 4.2 below as it differs slightly from the concept presented by the Aho-Corasick trie and classic trie used in Figure 4.1. The next step of the pre-computation is the creation of four functions called *out*, $shift1$, $shift2$, and *char*.

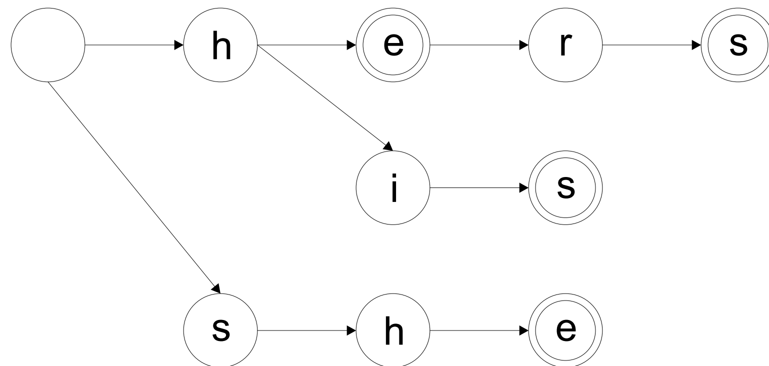


Figure 4.2: An example of a basic Commentz-Walter style trie for x ={“hers”, “his”, “she”, “he” }

The *out* function receives a trie node v and returns whether or not the path to the root from node v represents a keyword. If so, *out* returns the keyword. Otherwise it returns nothing (the empty set is denoted $\emptyset$), and the path from v to the root is simply a proper suffix of one or more keywords in the keyword array x.

For the construction of functions $shift1$ and $shift2$ we first define two sets, d, and w over the nodes in the trie, excluding the root node [23].

- $d(v)$ represents the depth of node v in the trie. That is, the number of nodes—including v—between v and the root node.
- $w(v)$ represents the word made from the concatenation of the node labels from the root to node v.
- $set1(v) = \{\ v';\ w(v)$ is a proper suffix of $w(v')\ \}$.
- $set2(v) = \{\ v';\ v'$ is an element of $set1(v)$ and $out(v') \neq \emptyset\ \}$.

Essentially $set1(v)$ indicates the set of all nodes at a deeper level in the trie than node v whose paths down from the root also end in $w(v)$. $set2(v)$ is the set of all nodes in $set1(v)$ that also mark the beginning of a path to the root such that the path represents a keyword from the keyword array x.

Now, where $wmin$ is the length of the shortest keyword, for each node, functions $shift1$ and $shift2$ are defined by [23]:

- If v is the root node then $shift1(v) = 1$ and $shift2(v) = wmin$, otherwise
- $shift1(v) = \min\{\ wmin,\ \{\ k;\ k = d(v') - d(v)\ \}\ \}$, where v' is an element of $set1(v)$, and
- $shift2(v) = \min\{\ shift2($ parent node of v $),\ \{\ k;\ k = d(v') - d(v)\ \}\ \}$, where v' is an element of $set2(v)$.

Finally, we define function $char$ over the values of the accepted alphabet [23]:

- $char(a) = \min\{\ wmin + 1,\ d(v)\ \}$, where v is a node with label a.

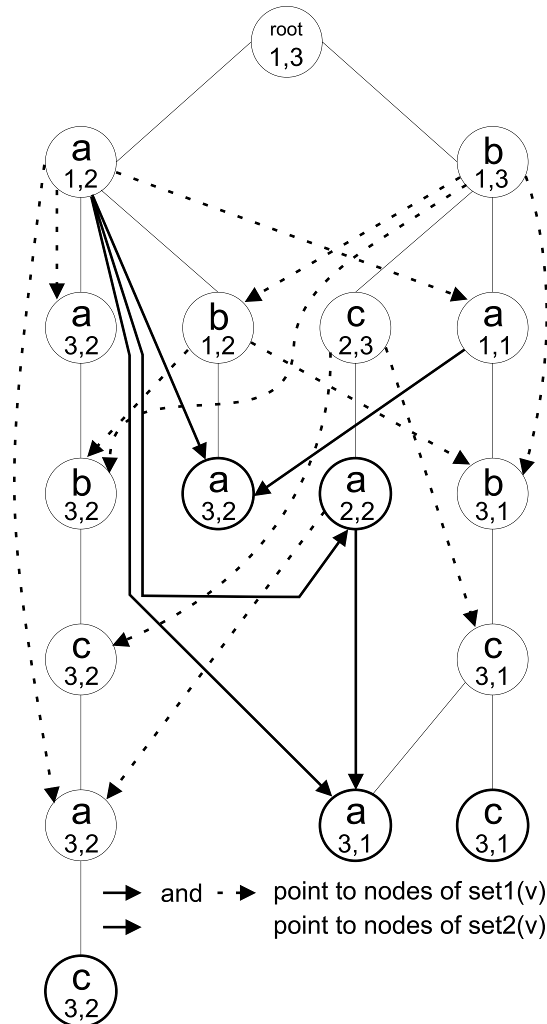


Figure 4.3: Commentz-Walter style trie with sets and shift functions for keyword set {"cacbaa", "acb", "aba", "acbab", "ccbab",}

Figure 4.3 is an example showing the relationships of the two sets as well as the values of the two shift functions on the nodes of a trie made by this algorithm.

The matching phase of the Commentz-Walter algorithm B is based on a combination of ideas from the Aho-Corasick trie and the Boyer-Moore right-to-left matching. The algorithm can be thought of as aligning the trie root under $y[wmin + 1]$ in the input text and scanning from right to left, and shifting the whole trie right upon detecting a mismatch. The details of the matching phase of Commentz-Walter's

algorithm B are given in pseudocode format in Algorithm 4.2 below.

Algorithm 4.2 Commentz-Walter Multiple-Keyword Matching Algorithm B

1: **procedure** CW($y, n, m, p, root$)
  ▷ Input:
  ▷ $y \leftarrow$ array of n bytes representing the text input
  ▷ $n \leftarrow$ integer representing the text length
  ▷ $m \leftarrow$ array of keyword lengths
  ▷ $p \leftarrow$ number of keywords
  ▷ $root \leftarrow$ root node of the trie

2:   $v \leftarrow root$ ▷ The current node
3:   $i \leftarrow \min\{m[0], m[1], ..., m[p-1]\}$ ▷ i points to the current position in y
4:   $j \leftarrow 0$ ▷ j indicates depth of the current node v

5:   **while** $i \leq n$ **do** ▷ Matching

6:     **while** v has child v' labeled $y[i-j]$ **do**
7:       $v \leftarrow v'$
8:       $j \leftarrow j + 1$
9:       **if** $out(v) \neq \emptyset$ **then**
10:         **output** $i - j$ ▷ Path from v to root matches $y[i-j]$ to $y[i]$
11:       **end if**
12:     **end while**
  ▷ Shifting
13:     $i \leftarrow i+ \min \{ shift2(v), \max \{ shift1(v), char(y[i-j]) - j - 1 \} \}$
14:     $j \leftarrow 0$

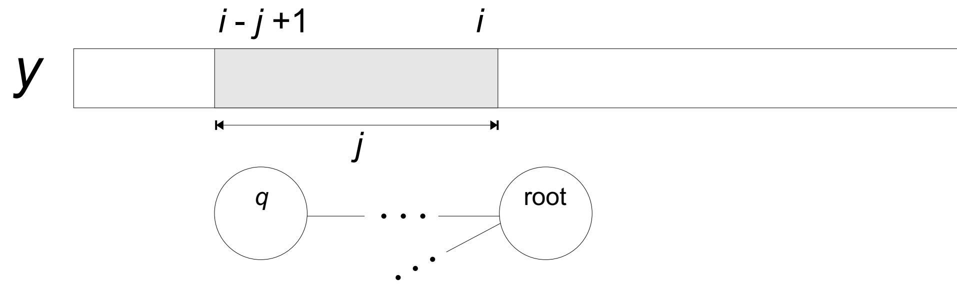
15:   **end while**
16: **end procedure**

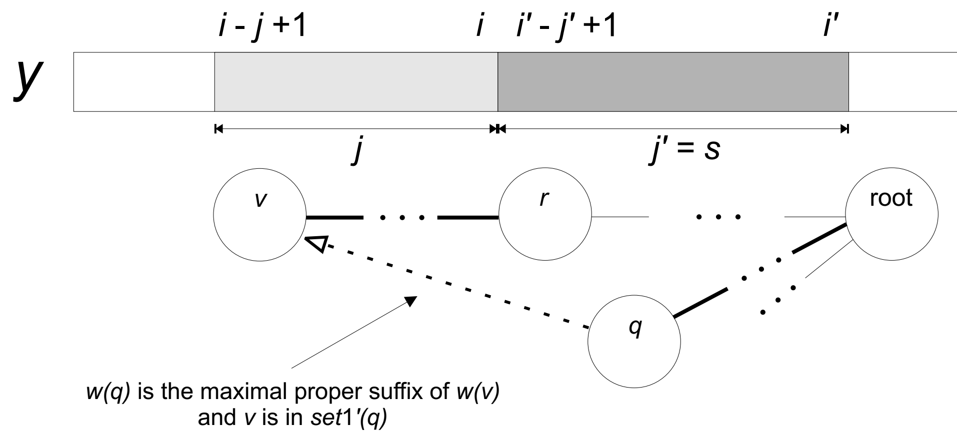
In Commentz-Walter's algorithm B some substrings of the input text y are scanned over and over in the worst case which leads to the quadratic behaviour of the running time during the matching phase [24]. In algorithm B1 [24] we have the exact same trie as for algorithm B; however, in order to reduce the worst case matching phase running time to linear in n, we use a stack that remembers the characters of the input that have just been scanned. The size of the stack could, in theory, grow as large as n, the length of y, but fortunately only the last $wmax$ (where $wmax$ is the length of

the longest keyword in x) entries of the stack are needed. This means the memory or space requirement during matching is still proportional to the keyword set or M in particular.

Keeping in mind that we always scan right to left and we shift left to right, we realize that characters or substrings of y are getting scanned more than once in algorithm B. Algorithm B1 eliminates this. Namely, if we are currently scanning y at some position i, then the root of the trie could be thought of as positioned on its side under $y[i+1]$. Let's assume that in the inner while loop of Algorithm 4.2 $j > 0$ at some point and a mismatch occurs (see Figure 4.4 part i). The loop then ends and we know that j characters were matched. Let's call the current node in the trie q. At this point a shift takes place which can be thought of as moving the trie to the right. Let's call the distance of the shift s. Now the current position becomes $i' = i + s$ and the root of the trie lies under $y[i' + 1]$. Let j' denote the variable j after the shift.

Let's assume that subsequent to the shift of distance s we match s characters, thus at some point $j' = s$ and we denote the current node r. Now $i' - j' = i$ and the next j characters to the left in y have already been compared against the trie before the shift took place (see Figure 4.4 part ii). What algorithm B1 effectively does to bypass re-comparing against those j characters is that it remembers them using the stack. Let's assume that the algorithm kept going and re-compared against the following j characters. At that point the current node v in the trie would not be node q; however, it is important to understand that v would be in $set1(q)$. That is, $w(q)$ is a proper suffix of $w(v)$. Node v may be in the $set1$ collection of many other nodes in the trie as well, but let's assume that the difference in depth between q and v is the minimal difference of depth as compared to all the other nodes that may have v in their $set1$ collections. Commentz-Walter [24] designates this important subset of $set1(q)$ as $set1'(q)$, where for any member node t of $set1'(q)$, $w(q)$ is the





ii) After the shift, *q* is retrieved from the stack

Figure 4.4: Commentz-Walter algorithm B1

maximal proper suffix of $w(t)$ compared to other nodes that contain t in their $set1$ set. Thus, in this instance $v \in set1'(q)$. Note that a comparable subset of $set2$ exists for all nodes called $set2'$ where the nodes in $set2'$ are always nodes t where $out(t) \neq \emptyset$. Commentz-Walter also notes that if v and q were in an Aho-Corasick style trie, that $f(v) = q$, where f is the Aho-Corasick failure function. Figure 4.5 is an example showing the relationships of the two new sets (compare with Figure 4.3).

The way that Algorithm B1 overcomes re-comparing against the j characters as described above is that i (or a pointer to $y[i]$) and a pointer to node q would be in one item on the stack. Algorithm B1 would have put this information on the stack prior to

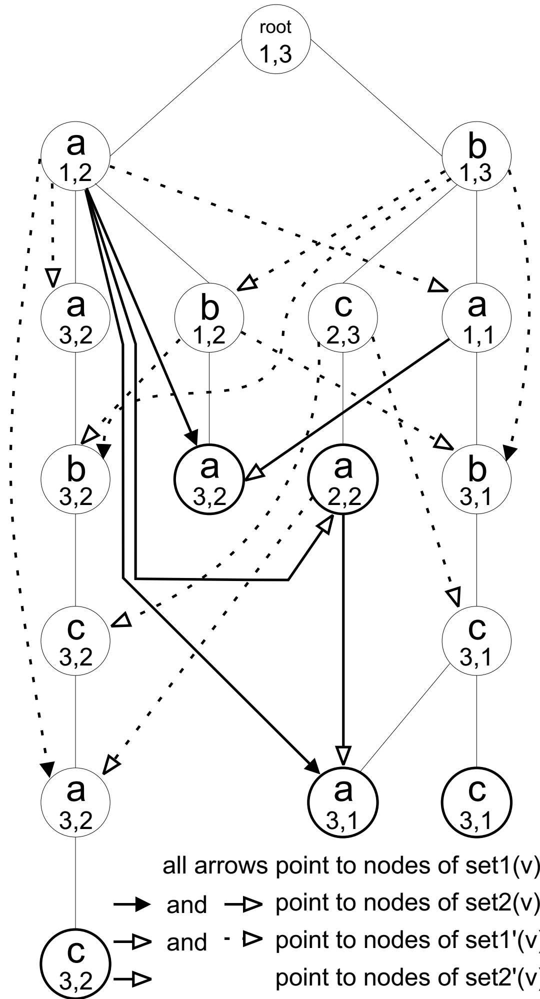


Figure 4.5: Commentz-Walter style trie with new sets for algorithm B1

the original shift of course. Note that this information from the stack is not enough. Additionally, a $where1$ function is calculated in the pre-computation that allows algorithm B1 to skip over the j characters. This function would take the current node in the trie (in this case node r) and the node pointed to by the item on the stack and indicate (return) that the algorithm may skip over j characters and move the current node to node v which is a descendant of node r. Commentz-Walter [24] provide the details of how this function is calculated. Furthermore, the scenario is still more

complicated than described because there could be multiple keywords that match on the path skipped from r to v, and alternatively the $where1$ function could return $\emptyset$ indicating that no node like v exists as a descendant from r. These cases are all handled by algorithm B1, although it requires a rather complicated pre-computation phase and a slightly more complicated matching phase. Commentz-Walter also shows how to improve the pre-computation of algorithm B1 in an algorithm called B2 [24]; confessedly, the average matching phase performance suffers slightly.

In conclusion, for our purposes in regards to the variegated set of Commentz-Walter algorithms, since the running-time of the pre-computation is trivial, the B1 algorithm is the best choice all around, though avowably very complex in order to achieve this running time.

4.3 Wu-Manber Algorithm

Wu and Manber created the UNIX tool agrep [91] to search for many patterns in files. In doing so they created a new algorithm to search for multiple patterns simultaneously [92]. They claim and demonstrate that their tool is substantially faster than both the UNIX tools egrep and fgrep. fgrep initially used the Aho-Corasick [2] algorithm, and subsequently it was upgraded to use a modified Commentz-Walter [23] algorithm [92, 45]. agrep is shown to match tens of thousands of patterns faster than both other tools [92].

Like the Commentz-Walter algorithm, this algorithm borrows the idea of skipping over parts of the text input using a skip table just as the Boyer-Moore [14] algorithm does to achieve a sublinear running time while matching patterns. The design of the algorithm concentrates on typical natural language searches rather than on worst-case behaviour [92].

This algorithm uses three tables built during the pre-computation phase: a SHIFT table, a HASH table, and a PREFIX table. The SHIFT table is similar to the Boyer-Moore bad character skip table, and the other two tables are only used when the SHIFT table indicates not to shift—with a shift value of zero—because there's a potential match at the current position under examination in the input. As with the Boyer-Moore shifting, the size of the shift is limited to the length of the pattern and in this case, the length of the minimum length pattern (call it *minlen*). Therefore, short patterns in the keyword set inherently make this algorithm less efficient [92].

A unique trait of this algorithm is that it looks at blocks of text instead of single characters at a time. Denoting again by M the sum of the keyword lengths and c the size of the alphabet, then the block length B is optimized for running time and space when $B = \log_c 2M$ (in practice $B = 2$ or 3 is recommended by Wu and Manber) [92]. The SHIFT table values determine the shift based on the last B bytes rather than just one. A hash function is used to map blocks to an integer used as an index into the SHIFT table. The total possible space required for this is $O(c^B)$. For our alphabet we have a SHIFT table of 256^B or $65,536$ elements when $B = 2$.

Let $A = a_1, a_2, ..., a_B$ be the B bytes in the text input y that we are currently scanning. There are two cases [92]:

1. A does not appear as a substring in any keyword (pattern) of x. In this case, we can clearly shift *minlen* - B + 1 bytes in the text input. SHIFT[$hash(A)$] = *minlen* - B + 1.

2. A appears in some keywords of x. In this case, we find the rightmost occurrence of A in any of the keywords; let's assume that A ends at position q of $x[k]$ and that A does not end at any position greater than q in any other keyword of x. Then SHIFT[$hash(A)$] = *minlen* - q.

So essentially to construct the SHIFT table all values in SHIFT are initially set to *minlen* - $B + 1$. Subsequently going through all keywords in x we map each possible substring of size B (a_{j-B+1} ... a_j) into SHIFT, and set the corresponding value to the minimum of its current value and *minlen* - j [92]. The running time to compute the SHIFT table is $O(BM)$ or if we count setting all elements of SHIFT, again, there are c^B many. Wu and Manber also discuss the possibility of compressing the SHIFT table when larger values of B are used by mapping different strings into the same entry—using the hash function—as long as the value in the table is the minimum of all of the exact shift values for the different strings. This would cause the algorithm to sometimes shift less than what it could do; however, this would not cause matches to be missed. Compressing the SHIFT table is a classic space-time tradeoff.

The index into the SHIFT table is also the index into the HASH table. When SHIFT[i] $= 0$ we look at HASH[i] which contains a pointer *pt* to a list of pointers to the keywords sorted by the hash values of the last B bytes. To find the end of the list of pointers we keep incrementing the pointer until it is equal to the value in HASH[$i + 1$] (because the whole list is sorted according to the hash values). As a result of this all keywords with the same suffix of length B bytes will map to the same entry in the HASH table. The number of elements in the HASH table needs only to be as large as the number of times SHIFT[i] $= 0$ over all values i. In the worst case it could be large, but in practice this does and should not happen given the control over the keyword set.

There could potentially be many entries in the list of pointers to keywords at a particular HASH entry. This is where the PREFIX table is useful instead of going through all of the keywords at the entry. The PREFIX table contains the mapping of all keywords' first B' characters ($B' = 2$ is the recommended value to use). Thus, the PREFIX table needs $O(c^{B'})$ space ($c^{B'}$ many elements). So the HASH table not only

contains, for each suffix, the list of all patterns with this suffix, but also hash values of their prefixes. Although Wu and Manber describe taking the hash of the prefix, it is actually the B' characters that are *minlen* bytes to the left of the end of the keywords. It would only truly be the prefix for keywords of size *minlen*. When the algorithm is in the matching phase the corresponding hash of the prefix in the text input is calculated (by reading *minlen* bytes to the left), and it can be used to filter keywords whose suffix is the same but whose prefix is different in the list of pointers. Wu and Manber claim this to be effective because of an assumption that very few keywords share the same prefix and suffix [92].

Although the pre-computation phase is slightly complex to build these three tables it is shown to be done quickly in practice in Wu and Manber's original implementation which is available in the C programming language. The space efficiency is directly represented in the tables and depends largely on the choice of B and the keyword set. The larger values of B mean that more entries will be present in the SHIFT table; however, it is optionally compressible as mentioned above. In addition to the three tables described, a PAT_POINT table stores the keywords, as does our array x with p elements described at the beginning of the chapter. PAT_POINT is indexed by the same pointers retrieved from the HASH table that serve as indexes into the PREFIX table.

The main matching phase of the algorithm is referred to by Wu and Manber as the scanning stage. It proceeds in four steps as follows:

1. Compute a hash value h based on the current B characters of the text (e.g. $y[i - B + 1]$ ... $y[i]$).

2. If SHIFT$[h] > 0$, shift the text by this value and return to step 1; otherwise, go to step 3.

3. Compute the hash value of the prefix of the text (by reading *minlen* bytes to the left); call it $text_prefix$.

4. Check for each *pt*, where HASH[h] $\leq pt <$ HASH[$h+1$], whether PREFIX[pt] $= text_prefix$. When they are equal, check the actual keyword (given by PAT_POINT[pt] in the list of pointers) against the text directly and output matches.

The pseudocode below (see Algorithm 4.3) is given for the matching phase of the Wu-Manber algorithm; although, the algorithm is highly dependant on parts of the C programming language. Some of the details of the hash functions below are hidden, but can easily be found in the mgrep.c file of the agrep package available from ftp://ftp.cs.arizona.edu/agrep.

The analysis of the expected running-time complexity of the main matching phase is shown by Wu and Manber to be slightly less than linear in n, the length of the input text. This analysis assumes both an input text and keywords that are random byte strings with uniform distribution [92].

4.4 Fan-Su Algorithm

The Fan-Su [35] algorithm is a combination of the Aho-Corasick [2] and Boyer-Moore [14] algorithms. Similar to the AC algorithm, this algorithm matches using a deterministic finite state machine or automaton (DFSM or DFSA) with differences inspired by the Boyer-Moore algorithm's ability to skip over portions of the input where a match is impossible. Fan and Su describe three main differences between a normal DFSA matching approach (the AC algorithm) and their proposed algorithm as follows [35]:

Algorithm 4.3 Wu and Manber Multiple-Keyword Matching Algorithm

```
1: procedure WM(y, n, B, B', SHIFT, HASH, PREFIX, PAT_POINT)
       ▷ Input:
       ▷ y ← array of n bytes representing the text input
       ▷ B ← integer representing the suffix block length
       ▷ B' ← integer representing the prefix block length
       ▷ SHIFT ← SHIFT table (see description above)
       ▷ HASH ← HASH table (see description above)
       ▷ PREFIX ← PREFIX table (see description above)
       ▷ PAT_POINT ← table of pointers to keywords (like our x it has m keywords)

2:     m ← min {length of all keywords}                              ▷ minlen
3:     i ← m − 1
4:     while i ≤ n do                                                ▷ Matching
5:         h ← hash(y[i − B + 1], ..., y[i]) ▷ hash over B bytes back from index i in y
6:         shift ← SHIFT[h]
7:         if shift = 0 then                                         ▷ Suffix block matches
8:             text_prefix ← hash(y[i − m + 1], ..., y[i − m + 1 + B'])
9:             p ← HASH[h]                                           ▷ a C style pointer
10:            p_end ← HASH[h + 1]                                   ▷ a C style pointer
11:            while p < p_end do
12:                if text_prefix = PREFIX[p] then                   ▷ Prefix matches
13:                    px ← PAT_POINT[p]                 ▷ Pointer to the current keyword
14:                    len ← length of px                ▷ Length of current keyword
15:                    j ← 0                             ▷ Count of matched characters
16:                    while j < len and y[i − len + 1 + j] = px[j] do
17:                        j ← j + 1
18:                    end while
19:                    if j ≥ len then
20:                        output i − len + 1
21:                    end if
22:                end if
23:                p ← p + 1
24:            end while
25:            i ← i + 1                                     ▷ Shift only by one place
26:        else
27:            i ← i + shift                                 ▷ Skip part of the text
28:        end if
29:    end while
30: end procedure
```

1. They call their state transition function the *goto* function which essentially determines the structure of the DFSA. The difference they implement is that in the pre-computation this function and, thus, the DFSA are created from the set of reversed keywords. In the matching phase of the algorithm all keywords are matched from their rightmost characters.

2. Just as the Boyer-Moore and Wu-Manber [92] algorithms do, this algorithm shifts the input past impossible matches while limiting the size of the shift to the length of the minimum length keyword (call it *minlen*). During the matching phase the first character retrieved is the $minlen^{th}$ character. The algorithm searches backward from there until a transition to the start state is indicated by the *goto* function.

3. When the *goto* function indicates a transition from the start state to itself, the $skip_1$ table (see discussion below) is consulted to determine which character in the input we will compare to next. Essentially the shift that can be performed is similar to the shift indicated by the Boyer-Moore bad character skip table. Otherwise, when the *goto* function indicates a transition from a state other than the start state to the start state, the $skip_2$ table (see discussion below) is consulted to determine the appropriate shift.

It should be clear how to construct the *goto* function given the description of the AC algorithm above. Furthermore, Fan and Su also use a function called *output* that receives a state as a parameter and returns the set of matching keywords associated with the state. This is a function similar to the A set of accepting states and the Aho-Corasick *output* function.

The pseudocode for the matching algorithm is given in Algorithm 4.4. The only remaining part of the algorithm is the pre-computation involving the generation of

Algorithm 4.4 Fan and Su Multiple-Keyword Matching Algorithm

1: **procedure** FS(y, n, q_0)
  ▷ Input:
  ▷ $y \leftarrow$ array of n bytes representing the text input
  ▷ $n \leftarrow$ integer representing the text length
  ▷ $q_0 \leftarrow$ initial state

2: $i \leftarrow \min$ {length of all keywords} ▷ *minlen*
3: $state \leftarrow q_0$
4: **while** $i \leq n$ **do** ▷ Matching
5: **if** $goto(state, y[i]) = q_0$ **then**
6: **if** $state = q_0$ **then** ▷ From the start state to itself
7: $i \leftarrow i + skip_1(y[i])$
8: **else** ▷ From another state to the start state
9: $i \leftarrow i + skip_2(state, y[i])$
10: $state \leftarrow q_0$
11: **end if**
12: **else**
13: $state \leftarrow goto(state, y[i])$ ▷ Transition to another state
14: **if** $output(state) \neq \emptyset$ **then**
15: **output** i
16: **end if**
17: **end if**
18: **end while**
19: **end procedure**

the skip tables. The rather lengthy pseudocode for their generation is included in Fan and Su's original paper [35]. Our discussion here only details the steps needed for their generation. The $skip_1$ table is straightforward to compute and involves two steps as follows:

1. For each character c in the alphabet set $skip_1[c] = minlen$.

2. For the jth character of the ith keyword $pattern_i$, set $skip_1[pattern_i[j]]$ to the lesser of its current value and the length of $pattern_i$ - j.

The $skip_1$ table resembles the Boyer-Moore bad character shift table. Similarly the $skip_2$ table resembles the Boyer-Moore good suffix shift table, and as such its generation is much more convoluted.

When the $skip_2$ table is being consulted it is because the next state is the start state, but the current state is not the start state. This indicates that there was a partial match for some number of characters. Let us call the partial match u and the number of characters matched (the length of u) $depth(u)$. Fan and Su describe the purpose of the $skip_2$ table to be to shift through the input such that one of two possible cases is satisfied as follows [35]:

1. The input will align after the shift with the next rightmost reoccurrence of u in a certain keyword.

2. The input will align after the shift such that the longest prefix of a certain keyword will align with a suffix of u in the input.

Constructing $skip_2$ can be done with the help of the DFSA's structure which in this algorithm's case is the *goto* function and the failure function or the equivalent of f in the Aho-Corasick algorithm. The $skip_2$ table can be generated using the following three steps [35]:

1. For every combination of pairs of states t and characters c set the value of $skip_2(t, c) = depth(t) + minlen$, where $depth(t)$ is the depth of the state in the trie and, hence, the length of the string t.

2. For every state t, state s, and character c such that $f(t) = s$ (f is the failure function) set $skip_2(t, c)$ to the lesser of its current value and $depth(t)$ - $depth(s)$ + $depth(s)$ (i.e. just $depth(t)$).

3. Each character c and final or accepting state t indicates a match of the keyword v. To find the longest prefix of v such that it is a suffix of u for each state s in the *goto* graph where state s indicates a match of the keyword u, then if $f(t) = r$ (f is the failure function) set $skip_2(s, c)$ to the lesser of its current value and $depth(t)$ + $depth(s)$ - $depth(r)$.

With both $skip_1$ and $skip_2$ tables built, rather than looking up the shift value in those tables during the matching phase, Fan and Su decided to encode the skip tables shift values into the state transition table in a fashion as follows so that no additional access to the skip tables is necessary upon a mismatch:

1. The state transition table is indexed by current state s and input character c.

2. Each entry in the state transition table consists of a tuple containing an indicator field and a data field which is interpreted differently depending on the indicator which can be a 1 or a 0.

3. If the indicator is set to 1 the data field is occupied by the state number of the next state that can be reached on input character c.

4. If the indicator is set to 0 then:

(a) If s is the start state (usually state 0) then the data field is occupied by $skip_1(c)$.

(b) If s is the not start state (usually state 0) then the data field is occupied by $skip_2(s, c)$.

Interpreting the state transition table during matching is now the only real difference in Fan and Su's DFSA algorithm. To summarize, when an entry in the table is read using current state s and character c from the input, the next state is indicated by the second item in the tuple and the shift is -1 or one character leftward in the input string if the first item in the tuple is a 1 (i.e. a match of the character has occurred, we move leftward as in the Boyer-Moore algorithm, and we transition to the next state effectively with the *goto* function or the g function of the Aho-Corasick algorithm); otherwise, the next state is the start state and the shift in the input string is rightward and indicated by the second item in the tuple. Note that upon a mismatched character that is not in any of the keywords, the first item in the tuple will always turn out to be a 0.

The complexity analysis of the algorithm's pre-computation shows that more work is done than simply constructing the *goto* and failure (f) functions as is done in the AC algorithm, but these processes are at least as involved, meaning that the time required to construct those functions is linearly bounded by the total length of all keywords [35]. Constructing the $skip_1$ table takes time linear in the total length of all keywords plus (+) the size of the alphabet [35]. Usually the sum of the keywords' lengths will be the dominant factor here. Constructing the $skip_2$ table takes the most time and is the dominant part of the pre-computation taking time linearly bounded by the number of keywords, times ($*$) the total length of all keywords, and times ($*$) the size of the alphabet [35]. Unfortunately this is rather lengthy, but Fan and

Su describe the process to be done rapidly because they did not measure the pre-computation time for more than ten keywords. Their assumption states that real applications need only to match a few keywords at the same time—which differs from that of Wu and Manber, who set out to handle keyword sets of sizes in the thousands.

The final memory requirements of the algorithm are represented mainly by the tuple encoded state transition table which is linearly bounded by the number of states times ($*$) the size of the alphabet. The number of states will always be less than the total length of all keywords, but it is essentially linearly bounded by it. Thus, the memory or space requirements are finally concluded to be linearly bounded by the total length of all keywords times ($*$) the size of the alphabet which can be rather large as is the case in the Aho-Corasick algorithm.

The running-time analysis of the matching phase of the algorithm is similar to that of the Boyer-Moore algorithm. In the best case a shift of *minlen* is performed every time giving a running time bounded by the length of the input text n divided by *minlen* or $O(n \ / \ minlen)$. Knuth showed that the worst case for the Boyer-Moore single-keyword pattern matching algorithm was essentially quadratic or proportional to the length of the keyword times the length of the input text [51]. Similarly, in the worst case, the Fan-Su algorithm shows quadratic behaviour [35], and the worst-case running time would be proportional to the input text length n times ($*$) the total length of all patterns M. Like many others, Fan and Su were mostly concerned with the average case because the worst case is assumed to be rare in practice. To analyze the average-case running time a probabilistic model is used [35]—as for the Boyer-Moore algorithm. The average-case running time is bounded by the length of the input text n times ($*$) the ratio R between the cost of discovering the mismatch and the distance to shift the rightward upon finding the mismatch. The details of the probabilistic model to calculate the expected value of this ratio R are given in the

algorithm's original paper, and examining it here is beyond the scope of this thesis. Fan and Su examine three scenarios for the expected value of R as follows [35]:

1. For a random alphabetic input text R is expected to be less than 0.2.
2. For an English input text R is expected to be less than 0.4.
3. For a random binary input text R is expected to be greater than 1.0.

Interpreting this means that for an English text approximately only 40% of the characters in the input text need to be examined during the matching phase. Fan and Su also show the results of their experimentation which essentially determine that R and thus the running time grows as the number of keywords increases, but diminishes slightly as the keywords' lengths increase. Lastly, because R is large for random binary input texts, matching binary patterns using this algorithm is not recommended [35].

4.5 Set Backward Oracle Matching Algorithm

The Set Backward Oracle Matching (SBOM) algorithm [6, 63] is the extension of the Backward Oracle Matching (BOM) algorithm [3, 63] to multiple keywords (see Section 3.5 for for a review of factors and oracles), with the factor oracle constructed for a set of words instead of just one word. The running time and space of the pre-computation phase of the algorithm, which creates the factor oracle, is linear in M, the sum of the keyword lengths. Unlike many pattern matching algorithms—especially for multiple keywords—this algorithm claims to work well for all kinds and sizes of alphabets and for various ranges of p, the number of keywords [6]. Experimental comparisons [6] show the new tool Ogrep [6] containing the SBOM algorithm consistently and significantly outperforming GNU's fgrep and Wu and Manber's agrep [91]. Experimental

maps displayed in graphs [63] show more specifically that SBOM does better than the Wu-Manber algorithm for small alphabet sizes in general. SBOM is also better than Wu-Manber for all alphabet sizes with keyword set sizes of 1000 and greater, but only for keyword sets with the shortest pattern of length 20 to 25 or greater.

To understand SBOM fully we first look at constructing a factor oracle for a set of words. The first main step to constructing a factor oracle from a set of words is to create a trie (keyword tree) out of the set of words. The form the trie takes for this algorithm is that of the Aho-Corasick trie where the links are transitions for the automata and the links are labeled—not the nodes. There are at most $M+1$ nodes, but typically fewer because common prefixes are usually found in some keywords, thus, reducing the size of the trie. Besides the transitions that are created from the trie itself we create additional transitions called *external transitions*. There can be at most M many of such external transitions.

The construction of the factor oracle uses a function called the supply function S—as in the BOM algorithm—which for some state q we identify another state k as its supply state [6]. We denote this $S(q) = k$. If a state q does not have a supply state then $S(q) = \emptyset$. Furthermore, to calculate the supply state for a state q we must have first computed the supply states for all states which correspond to nodes at all levels between q and the initial state or root node, and thus, we calculate the supply function for states (nodes) in a breadth-first traversal on the oracle under construction (trie). Where I is the initial state (root node), we start by defining $S(I) \leftarrow \emptyset$. Next we proceed in a breadth-first traversal over the states of the oracle following these steps [63]:

1. For current state q, initialize the variable $k \leftarrow S(parent(q))$, where $parent(q)$ represents the parent state (node) of q in the oracle (trie).

2. Initialize the variable $\sigma \leftarrow$ (label of the transition from $parent(q)$ to q).

3. If $k = \emptyset$, then $S(q) \leftarrow I$.

4. If $k \neq \emptyset$ and there is no transition from state k labeled by σ, then build a transition from state k to state q labeled by σ. Set $k \leftarrow S(k)$ and return to step 3 above.

5. If $k \neq \emptyset$ and there is a transition from state k labeled by σ leading directly to a state s, then $S(q) \leftarrow s$ and we are done processing state q.

The worst-case time complexity of constructing a factor oracle on a set of keywords is $O(M)$ [63]. We give its pseudocode in Algorithm 4.5. Navarro and Raffinot [63] give the details of the CONSTRUCT_TRIE subroutine which are omitted here. The implied difference (see Algorithm 4.5) in the our CONSTRUCT_TRIE subroutine is that in the construction of the trie all states corresponding to an entire keyword in x are marked as terminal. This includes at least all leaves in the trie.

This is not strictly all the pre-computation necessary for the SBOM algorithm. The SBOM algorithm uses CONSTRUCT_FACTOR_ORACLE_MULTI as a subroutine to construct the factor oracle in time $O(p * lmin)$, where $lmin$ is the length of the shortest keyword and p is the number of keywords in x. The subroutine is not exactly used as described, but the approach is fairly close. The SBOM algorithm creates a factor oracle from the reverse prefixes of length $lmin$ of the keywords in x. Furthermore, every terminal state q (leaf in the trie) also holds a set $F(q)$ of the indexes to which whole keywords in x they correspond. In our pre-computation call to CONSTRUCT_FACTOR_ORACLE_MULTI in Algorithm 4.6 we note that we assume these changes in Algorithm 4.5.

The matching phase of the SBOM algorithm, therefore, uses a sliding window of length $lmin$ to scan the text input y. In this window we read right to left the longest

Algorithm 4.5 Factor Oracle Construction Algorithm From a Keyword Set

```
1: procedure CONSTRUCT_FACTOR_ORACLE_MULTI(x, m, p)
       ▷ Input:
       ▷ x ← array of p keywords
       ▷ m ← array of p integers that are the keyword lengths
       ▷ p ← integer representing the number of elements in x and m

2:     oracle ← CONSTRUCT_TRIE(x, m, p)                    ▷ Details Omitted
           ▷ Marks all states corresponding to an entire keyword in x as terminal (all
           leaves)

3:     I ← oracle root
4:     S[I] ← ∅                             ▷ Array S stores supply function values

5:     for all states q ∈ oracle (q ≠ I) in a breadth first transversal do
6:         k ← S[parent(q)]                   ▷ parent(q) is q's parent in the trie
7:         σ ← label on transition from state k to q
8:         while k ≠ ∅ and (there is no transition from k to q labeled σ) do
9:             Create transition from k to q with label σ
10:            k ← S[k]
11:        end while
12:        if k = ∅ then
13:            S[q] ← I
14:        else
15:            S[q] ← state index of wherever k leads to with transition σ
16:        end if
17:    end for
18:    return oracle
19: end procedure
```

suffix that labels a path from the initial state [63]. One of two cases may occur as follows [63].

1. We mismatch at some point on a character a in the input y, and we safely shift the window to align the left side just past a in y.

2. We reach the beginning of the window successfully and the factor oracle is in some state q. Because rarely q may be associated with a factor that is not present in any keyword, we double check that we have actually read the reversed prefix of the keywords in $F(q)$. If we have actually read the reversed prefix of the keywords in $F(q)$, we verify each possible occurrence of a keyword match by comparing each keyword in $F(q)$ against the input y.

The worst-case time complexity of the SBOM algorithm is $O(n * M)$ which is quite bad, but on average it performs sublinearly in n [63]. We give its pseudocode in Algorithm 4.6.

Allauzen and Raffinot [6] extend Set Backward Oracle Matching to also use suffix oracles whereby longer shifts are achieved; however, the construction of suffix oracles is more complicated and not discussed here. Their approach to string matching using suffix oracles is called Set Backward Suffix Oracle Matching (SBSOM). Of more interest for our applications is bounding the worst-case time on both these algorithms to be linear in n. This is achieved [6] by combining the Aho-Corasick algorithm [2] (see Section 4.1) with either SBOM or SBSOM. The respective resulting algorithms are called MultiBOM and MultiBSOM. Just as SBOM and SBSOM are the multiple-keyword counterparts to the single-keyword algorithms BOM and BSOM (see Section 3.5), MultiBOM and MultiBSOM are multiple-keyword counterparts to the single-keyword linear-running-time TurboBOM and TurboBSOM algorithms (see Section 3.5). To actually achieve the linear worst-case running time, we begin each

Algorithm 4.6 Set BOM Multiple-Keyword Matching Algorithm

```
1: procedure SBOM(x, m, p, y, n)
       ▷ Input:
       ▷ x ← array of p keywords
       ▷ m ← array of p integers that are the keyword lengths
       ▷ p ← integer representing number of elements in x and m
       ▷ y ← array of n bytes representing the text input
       ▷ n ← integer representing the length of y

2:     oracle ← CONSTRUCT_FACTOR_ORACLE_MULTI(x, m, p)  ▷ Pre-computation
3:     lmin ← min{m[0], m[1], ..., m[p − 1]}            ▷ Min. keyword length
4:     I ← oracle root

5:     i ← 0                                              ▷ Matching
6:     while i ≤ n − lmin do
7:         q ← I
8:         j ← lmin − 1
9:         while j ≥ 0 and q ≠ ∅ do
10:            q ← state reached from q by transition labelled y[i + j]
11:            j ← j − 1
12:        end while
13:        if q ≠ ∅ and j = −1 then
14:            if y[i]...y[i + lmin − 1] is a prefix in a keyword of F(q) then
15:                Check for matches in keywords of F(q) against y and output
                       matches
16:                j ← 0
17:            end if
18:        end if
19:        i ← i + j + 1                                  ▷ Shift
20:    end while
21: end procedure
```

search through the sliding window by searching forward (left to right) using the AC algorithm. The backward search, using the oracle to find a mismatch, proceeds up to some point where the AC search stopped. Details on this algorithm are not complete in references used herein because they focus on SBOM primarily and not on the bounding the worst case; however, we provide further details about MultiBOM in Section 7.2 where we rename MultiBOM to MBOM after some modifications and elucidation. Because MultiBOM and MultiBSOM (and our MBOM) use the AC algorithm in this forward searching fashion the characters (bytes) of y are read at most once by each sub-algorithm (the AC pass and the backward pass with the oracle) for a total of at most $2n$ reads [6]. For the best performance the transitions for the factor oracle and AC automaton (trie) should be available in constant time; although, this typically involves using more memory.

4.6 Chapter Summary

Above we have covered a select few of the algorithms that offer solutions to the general keyword matching problem discussed at the beginning of this chapter. We have presented the algorithms in a fashion intended to be easy to understand along with background (see Chapter 3), diagrams, and pseudocode. This serves as both a state-of-the-art detailed survey, and provides necessary context for what follows in the remainder of this thesis.

The Aho-Corasick [2] algorithm is a classic solution which has been used in many applications and as a core element of other pattern matching algorithms. It is useful because of its worst-case running time which is linear in n, just as the Knuth-Morris-Pratt algorithm [51] is for matching a single keyword in a text. The Commentz-Walter [23] algorithm seems to be the first multiple-pattern matching algorithm that

successfully achieved a sublinear average running time thanks to the application of the filtering idea from Boyer and Moore [14] where parts of the text are skipped. As there exists a Horspool variant [46] which simplifies the Boyer-Moore algorithm, we note that [63] presents a set-based Horspool algorithm simpler than Commentz-Walter algorithm; although, it is only efficient for keyword sets with few keywords and with large alphabets.

More algorithms do exist and they are worth briefly mentioning here. The idea of bit-parallelism, semi-numerical techniques (as used in Karp-Rabin [49]), and low level bit-wise operations to manipulate the text and patterns are sometimes used in multiple-pattern matching algorithms for their low level operations speed and suitability to the problem of approximate pattern matching.

Kim and Kim [50] present such a solution with good experimental results that outperform Wu and Manber's agrep [91]. Unfortunately, as with many algorithms it is not suited to a wide range of applications. Their is based on a compact encoding scheme which is best used with small alphabets. In practice, any bit configuration of a byte gives an alphabet of size 256 which is not encodable in a compact manner (it requires of course 8 bits per byte or symbol in the alphabet). Although not discussed by Kim and Kim [50], this is significant as their experimental results are achieved using small alphabets like that of an English text and a DNA sequence which both contain alphabets encodable into less than 8 bits (e.g. DNA sequences use only 4 symbols or two bits per symbol).

Prior to Kim and Kim's algorithm, Baeza-Yates and Gonnet created the popular Shift-OR algorithm [12] primarily to match single keywords which they extended to handle don't care symbols, classes of characters, and multiple patterns as well. This algorithm works well for short keywords especially, and although it does not skip any part of the text, it may outperform sublinear algorithms with small pattern

sets and patterns of short lengths due in part to its suitability for fast and simple implementation with rapidly executable bit operations.

Navarro and Raffinot have also studied and implemented bit-parallel approaches to suffix automata [61, 62]. The tool *nrgrep* [61] is based on the bit-parallel simulation of a non-deterministic suffix automaton. Bit-parallelism in string matching algorithms is often associated with complex pattern matching (such as regular expressions) and approximate string matching as is the case with nrgrep [61]. Unfortunately, bit-parallel approaches with automata are harder to extend to the multiple-pattern matching problem.

Baeza-Yates and Navarro more recently have concluded in their extensive study of string matching algorithms [11] that with respect to the multiple-pattern matching problem, in general, bit-parallelism solutions like the Multiple Shift-And algorithm [12] are not as useful or extendable as the ideas and approaches of automata and filtering (skipping over text that is not present in any keyword). One would expect this to be an authoritative conclusion given that Baeza-Yates and Navarro were both pioneers and experts in the area of using bit-parallelism in string matching [12, 62]. Navarro and Raffinot [63] reiterate this conclusion for any size keyword set that is not small.

More recent developments in string matching have started to use extensions of automata such as directed acyclic word graphs (DAWGs or suffix automata) [29, 30], factor oracles [3, 5], and suffix oracles [4, 5]. Of the multiple-keyword matching variety are the DAWG-Match algorithm [27], the Multiple Backward DAWG Matching (MultiBDM) algorithm [69], the Multiple Backward Non-deterministic DAWG Matching (MultiBNDM) algorithm [62]—which also uses bit parallelism and is the most recent of them—and the Set Backward DAWG Matching (SBDM) algorithm [30] which is average-case optimal while maintaining a worst-case linear running time

[11]. In practice, the Set Backward Oracle Matching (SBOM) and Set Backward Suffix Oracle Matching (SBSOM) algorithms [6] are simpler and faster [11, 63] while still extendible to have optimal linear running times in the worst case [6] as shown above. Moreover, they use substantially less memory [63].

Lastly, more discussion on single- and multiple-keyword string matching solutions can be found in other resources [9, 10, 11, 28, 63, 30]. In the next chapter (Chapter 5) we will see which specific characteristics are desirable and acceptable for the pattern matching algorithm of a NIDS.

Chapter 5

Pattern Matching for NIDS Signatures

Matching patterns in a NIDS is a problem more specialized than the general keyword matching problem described in Chapter 4. In this chapter we will look at this problem and what specific requirements make it so particular. Furthermore, we will re-examine the candidate algorithms presented in Chapter 4 to indicate which specific needs they fulfill, and in turn, prescribe algorithms that could be helpful for those wishing to create the best possible pattern matching solution in a NIDS. In the context of signature matching in a NIDS the signature database corresponds to the keyword set and the network packets which the system scans correspond to the text input for a pattern matching algorithm.

5.1 Algorithm Requirements

Regarding modern developments and proposals on NIDSs, numerous studies reveal several important idiosyncrasies that NIDSs must address in order to accomplish their

signature matching goals. To better understand NIDSs and to make educated choices for the signature matching engine of NIDSs and similar applications, we examine these requirements in turn.

5.1.1 Searching for Multiple Patterns Simultaneously

Perhaps the most basic requirement for general purpose signature matching is an ability to match multiple patterns quickly to the point where it effectively happens simultaneously. This is due to the fact that signature matching in a NIDS typically requires matching a very large number of signatures. Thus, there is indeed a keyword set of size greater than one, suggesting a multiple-keyword pattern matching algorithm. Single-keyword pattern matching algorithms are in general better-known and understood to most; however, in this domain they are of no direct use, as serially searching the input text for keywords one at a time is far less efficient.

5.1.2 Searching for Large Sets of Patterns

When presented with the task of intrusion detection we note that the number of known intrusions is growing and is almost surely to continue to do so. This growth was observed in the past in the rapid expansion of the size of the signature database for the Snort NIDS. Tuck et al. [86] display a graph of the growth of the Snort rule database, indicating the database nearly tripled between the years of 2001 and 2004. In 2006 the Snort signature database comes with approximately 6000 rules. Custom rules may be also defined further expanding the size of the database. Also a single rule may contain multiple signatures (patterns to search for). However, often many signatures are disabled in tailoring to the individual deployment needs, resulting in the opposite effect.

Furthermore, depending on the NIDS there may be multiple keyword sets used, depending on the particular input. This is also a phenomenon present in the Snort groups of rules (see Chapter 2). It is certainly probable that some groups are quite large and some quite small. This means that it is possible that one algorithm may not work for all sizes of groups. It is normal that as the keyword set (group) size increases we see a decrease in the performance of pattern matching algorithms, and different algorithms scale differently. There are, in general, more algorithms that work well on small keyword sets than large keyword sets, but algorithms suited for matching using a large keyword set might not be as well suited when dealing with a small keyword set. Thus, a NIDS may use more than one algorithm if applicable to its implementation. Lastly, we note that small keyword set sizes suggest that the algorithm may run with fewer cache misses in practice.

Nonetheless, as just one example, Snort demonstrates how a NIDS can contain a large database of signatures. This of course translates to the requirement for the support of a large keyword set.

5.1.3 Searching With a Large Alphabet Size

Because network packets have no restrictions on what kind of data they carry, NIDSs' input and signatures have no restrictions on the alphabet. In short, any byte of input can contain any of the 256 possible values, and hence we are dealing with an alphabet of size 256.

With respect to most string matching literature this is a large alphabet. Typical alphabet sizes considered in string matching literature are: 4, for DNA/RNA sequences; 52, for the English dictionary; or 128 for ASCII. NIDSs, however, may be used to search for binary patterns in network packets resulting in requiring them to work on a larger alphabet of size 256. Many string matching algorithms designed to

work well with small alphabets are, thus, not at all suitable for NIDSs. While Navarro and Raffinot [63] demonstrate the effect of changing the alphabet size for different multiple-pattern matching algorithms, their experimental results, shown as various maps, do not include alphabets larger than 64. Nonetheless, we are able to get an idea of some algorithms unlikely to work well for large alphabets. Alas, overall, more algorithms are suited to small alphabets than to an alphabet of size 256. One possible way of coping with this inherent large alphabet size is to consider only 4 bits at a time instead of 8[1]. To our knowledge this is an unexplored method of shrinking the alphabet down to 16 symbols for NIDSs. Doing so may improve the performance of certain pattern matching algorithms, but we leave this research area for future work.

5.1.4 Searching With a Wide Range of Keyword Lengths

The lengths of the individual keywords within a keyword set can have great consequences on the performance and memory requirements of an algorithm used for matching. A requirement of a NIDS matching signatures is that the algorithm used must be capable of handling patterns of various lengths. Typically this is not a problem for multiple-keyword string matching algorithms until the length of a single keyword in the set gets very small (1 – 3 characters). This, of course, is a serious concern for the algorithms that perform best when the length of the shortest keyword in the set is not very small. Filtering-inspired algorithms that skip over parts of the input search data can only ever shift their windows by this minimum length. Of course this becomes most problematic if there is a keyword of length one in the set. Almost always these filtering inspired algorithms work best with a minimum keyword length that is quite large as to increase the window's shift distance because the longer shift distances is how they achieve their sublinear running times on the

[1] This idea was suggested by P. Morin.

average. Unfortunately, filtering style algorithms are only as good as their weakest link; they cannot shift further than the length of the shortest keyword in the set.

For the Snort NIDS, Tuck et al. [86] display a graph of the distribution of the lengths of unique keywords found in an older Snort database. It indicated that indeed there are some keywords with lengths one and two, although most keywords have lengths between 4 and 15. This indicates that to date, known attack patterns are generally short. Because these short keywords are known a priori it would certainly be best to try to take them out of the signature database or else extend them to be longer with content before or after them if possible when the length of the shortest keyword has any impact on the algorithm being used.

5.1.5 An Algorithm Designed for the Average and Worst Cases

Because NIDSs detect and sometimes stop attacks, attackers would naturally like to break them in some way so their real attacks can go unnoticed. For this reason NIDS are typically and intentionally targeted by attackers for denial of service attacks to try to cause the system running the NIDS's sensor to overload and begin dropping packets. One such recently discovered manner to take down a system is through an algorithmic complexity attack [31].

An algorithmic complexity attack is an attack whereby denial of service is possible without the traditional overwhelming flood of inputs typically considered necessary. Instead of the overpowering flood of general input that causes the capitulation of the system to the point that it is useless, an algorithmic attacker crafts the input such that the worst-case system behaviour is unremittingly invoked again and again. Using this technique can often have devastating effects on systems that are unprepared for

this even without vast amounts of input. Depending on the algorithms present in a given system and how much they are relied upon, it may be that hardly any input of this specially crafted nature is necessary in order to assemble a successful attack. Awareness and prevention is the best defense against these attacks.

For algorithmic complexity attacks to be possible and practical for attackers, they must have two things: knowledge of the underlying algorithms in the system (and possibly the architecture as well), and the ability to manipulate at least some influential parts of the input. Almost all systems take input from a source that can be manipulated, some more than others. Knowledge of the algorithms and data structures in a system may be something harder to achieve or guess at unless the code for the system is readily available such as it is for open-source software projects.

In systems connected to a network, attackers can often easily mount an algorithmic complexity attack because they can send any amount of any kind of traffic they wish. It is important to understand that although this traffic may attempt to invoke the worst-case behaviour for a system, the traffic does not need to look malicious in any way. In fact, there is usually no way of telling if it is legitimate or not. NIDSs are particularly susceptible to this and their sensors are purposely made to face attack traffic in hopes of detecting attacks. This is often traffic from sources like the Internet that cannot be controlled by the owners of the system. This means that potential attackers most often have the ability to manipulate their input to the NIDS to a large extent. Furthermore, and to make matters worse, a few popular NIDSs are indeed open-source software, and for the rest we generally don't recommend security through obscurity for kinds of systems like a NIDS where it is likely going to be targeted for attacks. That is to say, relying on the fact that an attacker can't guess, reverse engineer or otherwise obtain knowledge of the system's details is generally a bad assumption to make for a NIDS.

Snort and Bro are two open-source NIDSs that have already been seen as susceptible to algorithmic complexity attacks. The renewed interest in these attacks is largely due to the research of Crosby and Wallach [31], in which they demonstrate bringing down the Bro NIDS. Specifically, they attack Bro with a specially crafted stream of network packets at a rate of 16 Kbps and cause the system running Bro to begin dropping approximately 70% of all traffic going to it. Obviously for a NIDS, whose goal it is to detect attacks, dropping traffic is unacceptable. This attack demonstrates the severe threat of the algorithmic complexity attacks with very little traffic at all given that the line rates targeted for most NIDSs are now between a thousand and one hundred thousand times greater than that which was needed to topple the Bro NIDS. Tuck et al. [86] also describe in less detail an attack against Snort's Wu-Manber algorithm variation for its default pattern matching algorithm. Recently, more algorithmic complexity attacks have been investigated in non-IDS applications [39].

In summary, not just the pattern matching algorithm, but all algorithms, data structures, and architecture should be well engineered in a NIDS to prevent attackers from invoking ongoing worst-case behaviours. The best way to do this is designing with the average- and worst-case running time and memory complexities in mind for all of the mechanisms employed in the NIDS. We note, however, that of all algorithms it is usually the pattern matching algorithm of the signature matching engine that is crucial to NIDS performance. In systems such as Snort, approximately 70% to 80% of the NIDS's processing time is consumed by the signature engine's pattern matching algorithm [86].

5.1.6 Extended Searching Characteristics

The problem of extended string matching as described by Navarro and Raffinot [63] involves searching for keywords with special characteristics beyond what the exact (general) keyword matching problem considers. This is a more sophisticated kind of searching between the problems of simple string matching (considered herein until now) and matching regular expressions which are the most complex pattern types of all to match. The extensions typically seen in general are: case insensitivity, don't care symbols, wildcards, classes of characters, bounded length gaps, optional characters, and repeatable characters.

In NIDSs for signature matching, case insensitivity is often a requirement, but is almost always an easy feature to add to an algorithm that doesn't provide for case insensitivity already. Other extensions are not common enough to call a requirement for pattern matching algorithms in a NIDS; however, certain specialized systems may benefit from having certain kinds of pattern extensions mentioned here.

5.2 Candidate Algorithms to Fulfill Requirements

In this section we look at the multiple-keyword pattern matching algorithms presented in Chapter 4 and the requirements as described in Section 5.1. Based on the requirements set forth for a NIDS, we compare and contrast the algorithms in Table 5.1 and provide explanations of the data in the table below.

The Aho-Corasick (AC) algorithm [2] works well in general except that we rate it poorly on its ability to handle large alphabets and large keyword sets. This is due to the fact that the memory it requires to perform well in these cases is exceedingly large. Typically, the fastest access for state transitions comes from the implementation of

	AC	CW-B	CW-B1	WM	FS	SBOM	MultiBOM
Linear worst-case running time	✓		✓				✓
Sublinear running time on average		✓	✓	✓	✓	✓	✓
Supports many keyword lengths	✓						
Supports a large alphabet				✓		✓	✓
Supports a large keyword set				✓		✓	✓
Simple to understand & implement	✓				✓	✓	

Table 5.1: Table of algorithms' fulfillment for NIDS desirable features

a table that can be on the order of (size of the alphabet * total number of states). Because the AC state machine is essentially built from a trie the number of states is often large as well, up to $O(M)$ is the worst case. Despite this the AC algorithm is widely used for its simplicity in many contexts and many applications don't require a large alphabet. Moreover, the AC algorithm is currently being used in NIDSs such as Snort; albeit, with a lot of enhancements such as table compression that reduce the memory usage of the algorithm. This brings us to the last and important fact regarding the running time complexity of the AC algorithm. Its linear bound on the worst-case running time makes it a suitable algorithm for NIDSs, and when a NIDS is under attack this is undoubtedly important. However, there have been numerous attempts to speed up Snort's performance using other algorithms like the Wu-Manber algorithm [92] where a better average-case solution was achieved. There is high demand for this because of the push to bring NIDSs like Snort to handle gigabit network speeds and beyond. The trouble with such an idea is that tweaking Snort or any NIDS to be able to handle those speeds without being able to handle worst-case scenarios under those speeds may not benefit the NIDS users and puts them at serious risk. For the time being it seems that the AC algorithm's steadfast running time is dependable for NIDSs which is crucial when under attack. Another recent development in multiple-pattern matching is using the AC algorithm as a component within another primary algorithm to bound the whole worst-case running time. This

is something we feel hasn't yet gotten any attention in the NIDS pattern matching research.

The Commentz-Walter (CW-B and CW-B1) algorithms [23] are the same in all respects except in its worst-case running time. Typically research surrounding the Commentz-Walter algorithms refers to only one algorithm which is assumed implicitly to be the simpler algorithm B. In fact, most research does not even reference Commentz-Walter's work in the original technical report which is where we have found the only explanation of the B1 algorithm even though it is described to be the better of the two algorithms in all respects with the exception of its additional difficulty to understand and complexity to implement. The B1 algorithm may in fact be very useful to NIDS research given its worst-case running time complexity is linear in the input length.

The weak points of both Commentz-Walter algorithms are the algorithms' heavy memory usage. In particular algorithm B1 trades off better performance while consuming even more memory. We rate the Commentz-Walter algorithms poorly with respect to handling a wide range of keyword lengths due to its reduced performance as the size of the minimum length keyword decreases. We note that this is the case with all algorithms that use a suffix-based searching approach that proceeds backward within a window. However, this same approach buys these algorithms a sublinear running time in the average case. Finally, both algorithms do take up more memory than the AC algorithm; large keyword sets tend to use a lot of memory which may be a concern in some cases. In addition, although we do not focus on pre-computation complexities, we note that CW-B1 has substantial time and space overhead during pre-computation.

The Wu-Manber (WM) algorithm [92] as mentioned above has been used in Snort in the past for its fast sublinear average-case speed, however it performs badly under

an algorithmic complexity attack and has a quadratic (in n) worst-case running time. Furthermore, its performance becomes poor with a large variance in the keyword sizes, especially as keywords become very small since it uses a suffix-based approach where shifts are limited to the size of the shortest keyword. In most other respects WM is excellent, handling a large alphabet size, though with quite a bit of memory. Although, Wu and Manber describe how to easily compress the space usage without sacrificing too much performance. Also, WM was designed to be able to handle massive keyword sets, and it does so very well. Lastly, we rank WM as marginal to poor with respect to its ability to be implemented from scratch. Generally, most research implementations take code from the source of Wu and Manber's freely available agrep [91] tool. Still, the code is quite convoluted and the many tables and hash functions used, along with its dependence on C programming structures like pointers, make the algorithm harder to understand overall.

The Fan-Su (FS) algorithm [35], as detailed in Section 4.4, performs badly on random binary data which implies a large alphabet, and it has a quadratic running time complexity in the worst case. This makes it a poor choice for NIDSs. Nevertheless, it is fast on the average and its implementation is a fairly easy extension of the AC algorithm. Because FS also uses a suffix-based approach, its ability to deal with small patterns decreases performance. As a final point, FS does not deal well with large keyword sets for matching, taking up more memory than the Aho-Corasick algorithm, as well as causing pre-computation to be quite lengthy [35].

The Set Backward Oracle Matching (SBOM) algorithm [5, 63] fulfills many of the requirements and some partially which we will explain. Firstly, on average the SBOM algorithm is often the fastest in practice (along with the WM algorithm) for typical string matching applications. It is markedly the simplest of all algorithms to implement; although, it is not as widely known as other algorithms because it

is more recent. Also the oracle it constructs will use less memory than the Aho-Corasick trie because it is built on reversed keywords truncated to the minimum of all keyword lengths. This results in the SBOM algorithm being comparatively excellent for large keyword sets, and progressively better for large alphabets as the minimum keyword length increases and as the keyword set size increases. Unfortunately, one weak point is that this algorithm is factor-based, and performs searches backward within a window like the suffix-based algorithms meaning that it does not handle keyword sets well if they contain very small keywords. Another weak point of the SBOM algorithm is its worst-case running time is very bad. However, the enhanced version, MultiBOM [5] which uses AC as a sub-component, successfully introduces a linear bound on the worst-case performance. As mentioned above, MultiBOM has not yet received attention in NIDS research. Although it is still slower than SBOM in practice, MultiBOM's average-case performance is sublinear and much faster than AC alone. One stumbling block of MultiBOM, shared by SBOM, is its ability to handle small keyword lengths.

5.3 Chapter Summary

In this chapter we presented the state of the art in pattern matching algorithms for use within a signature matching engine of a NIDS. We reviewed the topic of pattern matching within the context of NIDSs by identifying and discussing the requirements that such a signature matching engine has and how certain algorithms fulfill those requirements. Finally, we compared and contrasted the multiple-pattern matching algorithms from Chapter 4. Although there is no all-around perfect choice, in our discussion we explained how suitable each algorithm is in a NIDSs such as Snort.

Chapter 6

Software Solutions That Have Been Proposed

In this chapter we look at various solutions proposed in the recent research of others. Of particular interest are the solutions pertaining to the Snort NIDS and other NIDSs where signature matching comprises an important part of the processing time. Almost all modern research ascertains the importance of the multiple-keyword pattern matching algorithms used in a NIDS's signature matching engine. Although NIDS engineers may have elucubrated excellent signature matching engines over years of development, in testing, NIDSs such as Snort still reveal that pattern matching is often the most expensive and lengthy process they perform.

In this thesis we are only considering solutions that have been proposed that are software-based; however, hardware-based solutions are also currently being proposed and progressively becoming more realizable with the decrease in costs of hardware like memory. Nevertheless, at the present time we believe software solutions to be of broader interest and greater practicability. We begin our investigation of proposed solutions by examining research that led to the use of multiple-keyword pattern match-

ing algorithms in Snort, and then follow with a moderately detailed examination of the pattern matching options available in the current version of Snort. Subsequently, we introduce a couple of proposals that offer new benefits to Snort.

6.1 The First Multiple-Keyword Pattern Matching Solutions for Snort

Before version 2.0, Snort simply used the Boyer-Moore [14] single-keyword pattern matching algorithm to search for signature content in many passes over the input packets. Some of the first research into applying multiple-keyword pattern matching algorithms was done by Fisk and Varghese [37], who proposed the use of the Aho-Corasick [2] algorithm and a new algorithm they named the Set-wise Boyer-Moore-Horspool (SBMH) algorithm. In this section we briefly look at their algorithm and research which probably significantly impacted Snort (hence its changes in version 2.0) and other applications like it such as virus scanning, firewalls, content distribution networks, layer-7 switches, and other NIDSs. Because at the time of their research Snort didn't use a multiple-keyword pattern matching approach, the results of adding AC and SBMH to Snort gave a massive boost to Snort's processing speed. Because AC has already been introduced herein (see Section 4.1), we begin by introducing SBMH.

The Boyer-Moore-Horspool (BMH) algorithm, Horspool's [46] variant of the Boyer-Moore algorithm (see Section 3.4 for details), is a well-known algorithm for single-keyword string matching that uses a suffix-based search approach. Although the Commentz-Walter (CW) [23] algorithm (see Section 4.2) did multiple-keyword string matching in sublinear time in the average case, the solution is rather complex. Fisk and Varghese propose SBMH which uses simplified ideas similar to CW and BMH.

The algorithm begins by constructing a trie using the reversed keywords. SBMH then builds a set-wise equivalent of the shift table (from BMH) by effectively using all the keywords' tables and taking the most conservative value for the final table of the whole keyword set. That is, the shift value for some input character (byte) c in the skip table is the minimum of all the values from all the individual keywords' tables at the same index for character c. This conservative use of the minimum of all the values is necessary to avoid skipping over potential matches. However, it also implies that the algorithm is limited to shifts no greater than the length of the shortest keyword in the set. This algorithm was later further explained with pseudocode by Navarro and Raffinot [63] as an alternative to the Commentz-Walter's B algorithm variant.

Because of its suffix-based search approach, SBMH achieves sublinear search time on average; albeit, with worst-case running time $O(n$ * the length of the longest keyword). Unfortunately, SBMH was not tested in Snort with specifically worst-case crafted packets, but rather with high workload packets [37]. Fisk and Varghese's experimental results showed that AC and SBMH performed far better than the single-keyword string matching approach used in Snort. Indeed, using the multiple-keyword string matching algorithms they suggested provided far more scalable solutions than that implemented at the time. Based on their experimental observations, Fisk and Varghese's final suggestions were to use Boyer-Moore on rule groups (see Chapter 2) of size 1, to use the new SBMH for rule groups of size 2 to 100, and for all other cases to use Aho-Corasick. Lastly, we also note that Coit et al. [20] did very similar research to that of Fisk and Varghese around the same time, suggesting the move to multiple-keyword string matching algorithms for the signature matching engine in Snort.

6.2 Current Solutions in Snort 2.6

Originally, Snort had no multiple-keyword pattern matching algorithm at all, using only the single-keyword Boyer-Moore [14] algorithm to search for signature patterns. Of course this made Snort very slow, and the developers quickly realized better solutions were available. In version 2.0 of Snort basic Aho-Corasick [2] (a multiple-keyword pattern matching algorithm) and the SFK Search algorithm (described below) were added to vastly improve the performance. Finally, version 2.2 of Snort included more pattern matching algorithms as configuration options of the signature matching engine. Those options remain the same in version 2.6[1]and are as follows:

1. **AC-Std** is the basic Aho-Corasick algorithm that was included in version 2.0. This is the default option in Snort 2.6.

2. **AC-Full** is an optimized Aho-Corasick algorithm that uses a full matrix state transition table.

3. **AC-Sparse** is an optimized Aho-Corasick algorithm that uses a sparse matrix state transition table.

4. **AC-Banded** is an optimized Aho-Corasick algorithm that uses a banded matrix state transition table.

5. **AC-Sparse-Banded** is an optimized Aho-Corasick algorithm that uses a sparse-banded matrix state transition table.

6. **Modified Wu-Manber** (MWM) is an algorithm derived from Wu-Manber [92].

[1]At the time of this writing, Snort is still in beta version 2.6 (RC2); although, in the final 2.6 release, no changes are expected to be made to the algorithms discussed herein.

7. **Low-Mem Trie** (also called SFK Search from version 2.0) is a low-memory alternative that builds a trie from the keyword set and effectively performs a Boyer-Moore style bad character shift search through the input until a possible match is found, whereupon a slower search using the trie is performed.

All of the above options except the first and the last are new additions as of version 2.2. They are used on rule groups which are essentially groups based first on protocol (TCP, UDP, ICMP or other IP) and subsequently source and destination ports if applicable (for TCP and UDP). Every rule group will thus have keyword sets or what can be thought of as a set of patterns. In Snort, the algorithms have integrated support to handle case insensitivity if needed, and the ability to test for the absence of patterns in packets as well. Both of these extensions are fairly trivial to add.

Each rule group stores its own preprocessed keyword set; thus it should be possible to use different search algorithm options from the list above for different rule groups. However, currently all groups get assigned the same algorithm option. This is an area we suggest for potential improvement in the future (see our AUTO option in Chapter 7). Certainly separate keyword sets may have diverse characteristics, and thus, have dissimilar requirements for algorithms. In fact, the MWM option accounts for this by adjusting its approach based on the size of the keyword set.

The Modified Wu-Manber (MWM) option actually uses multiple passes of a single-keyword string matching algorithm when the keyword set is of size four or less. The algorithm used is the Horspool [46] variant of Boyer-Moore (see Section 3.4). When there are at least five keywords in the set a multiple-keyword string matching algorithm is used that resembles Wu-Manber. The original Wu-Manber algorithm (recall from Section 4.3) uses a table called SHIFT that holds appropriate shifts for every possible two bytes (two was the standard Wu-Manber block length). This SHIFT

table resembles a Boyer-Moore bad character shift table and in Snort's MWM this (bad character shift table) is its name. Unlike the original Wu-Manber, MWM may use a block length of one or two bytes as its indices into the tables it uses.

Furthermore, MWM implements another shift table, the bad word shift table, which is optionally usable in addition to the bad character shift table; it is hard-coded to be used when the shortest keyword length is greater than one. If the shortest keyword length is one then neither of the shift tables are used, and a shift of length one is always performed followed by a lookup into the prefix table just as the Wu-Manber algorithm does as its third step (see Section 4.3).

MWM also implements a pattern-match tracking mechanism using extra memory that Snort is usually configured to keep anyhow. Although we have not seen results showing the benefits of this, it is documented with its implementation that it helps MWM's worst-case behaviour. Unfortunately, configuring Snort to use the MWM option is potentially unsafe because all the algorithms used in the MWM option have quadratic worst-case behaviour. This means they are particularly susceptible to algorithmic complexity attacks (see Section 5.1.5).

AC-Full is the probably the best NIDS option involving Aho-Corasick, as it uses a full matrix state transition table; thus, it has the least performance overhead in determining the next state upon the triggering of the state machine's transitions. It uses more memory than the other newly added options, but at about one quarter to one half that of the AC-Std option. It provides for up to 2^{16} states. Markedly, it is the fastest AC option in practice [65], as well as being the fastest overall that provides a linear bounded worst-case running time under situations such as an algorithmic complexity attack.

Unless the slight increase in the memory usage is of concern, we feel that AC-Full is currently the best option to safely use in a Snort NIDS, especially if it is used

inline in the network where it can drop packets as an intrusion prevention system. Furthermore, we feel that where memory is a concern, configuration of one of the other new Aho-Corasick options is the best way to go. Even so, it does not take a great deal of intuition to believe that memory will become less of a concern and algorithmic attacks will become increasingly frequent on NIDSs, requiring them to work as quickly as possible. Although the lower memory options improve caching performance in benchmark tests, it does not help when the algorithms are used in Snort. Snort source code comments indicate that unfortunately, after a pattern match test has been performed Snort moves on to doing so many other things that once it returns to do another pattern match test the cache is voided.

In an application such as a NIDS, saving memory only to cause slower processing seems imprudent given that the goal of a NIDS is performance high enough that all packets get inspected. Otherwise, packets may get dropped from the frame buffer due to overflows during peak times of overwhelming traffic floods (caused by denial of service attacks for example).

6.3 Piranha

Piranha [8] is an enhancement made to Snort's signature matching engine in the form of a new algorithm entirely. It is quite simple to understand, based on the observation that if the rarest substring of a pattern does not appear, then neither will the whole pattern [8]. The algorithm works on rule groups which is how Snort groups keywords or patterns into sets.

The implementation of Piranha uses four-byte (or 32-bit byte aligned) substrings because they fit easily into a four-byte integer. The algorithm finds all the substrings of all patterns and associates only one substring to each keyword from the set. The

substring from the keyword that will represent the keyword is always the rarest substring overall (the substring that appears the least in all of the keywords). After this straightforward pre-computation there are a set of substrings that are searched for in the input (packet headers and payloads) instead of searching directly for the keywords. However, certain substrings may represent more than one keyword from the set. In fact, the substrings are used as an index into a hash table that stores a list of keywords at each index. If when searching packets the four-byte window is scanned and a non-empty list is found in the hash table, this means that a substring is matched in the input and one of two possible cases will occur. Firstly, it could be a false positive or what Antonatos et al. [8] call a *collision*. This is determined by the failure of the test for the second case. Secondly, there may be a match of a represented keyword. To establish if one of the whole keywords that the substring represents is present in the input, the list of keywords stored under the substring in the hash table is searched for a match. Instead of directly doing a comparison between each keyword and the corresponding section of the input in the packet, Piranha first checks the last two bytes of the keyword (the suffix) to see if they match the two corresponding bytes in the input. This two-byte suffix check seems to allay many of the collisions by resolving them faster than doing something like a *memcmp* operation in the C programming language.

The two-byte suffix check conflicts with the advice of Wu and Manber [92]. They argue that in the English language, for example, identical suffixes are more common than prefixes, motivating their use of a prefix check by means of their PREFIX table where the first two bytes were used to further reduce collisions from their SHIFT table. There is apparently no absolute no right or wrong here—it completely depends on the keywords and the input data. However, we note that a fixed algorithm is easily attackable via an algorithmic complexity attack. Therefore, we suggest that instead

of fixing the algorithm to constantly check one end or a certain position that it be randomized to some extent. This prevents an attack from knowing how the algorithm will behave even if the attacker has access to the source code.

Another place that Piranha may be improved in regards to its security is in the selection of the substrings to represent keywords. For a keyword of length m, there will be $m - 4 + 1$ four byte substrings. Of all the substrings it is very possible that more than one tie as the rarest overall. Antonatos et al. [8] provide an example that demonstrates choosing the last of the substrings that tie as rarest. This would mean the keywords would usually get represented by substrings close to their end and that the substring may possibly be a suffix. This is an imprudent choice given that Piranha already tests the last two bytes (suffix) of the keyword and the corresponding input. A trivial improvement to this would be to choose the first of all substrings that tie as the rarest overall. But, in this case, addressing this potential problem by choosing another fixed option still leaves the algorithm open to attack by specially crafted input. A safer option would be to randomize the choice of substring amongst the rarest options. It is unclear if such a change would grow the size of the Piranha hash table in a way that would affect performance on average due to missed opportunity for caching. However, using some of the second to rarest possible substrings may be an option to keep the size of the table small by putting more than one keyword in the list at each hash table index.

The performance of Piranha in Snort version 2.2 fares very well in the test cases presented by Antonatos et al. [8]. The algorithm was mainly tested against Modified Wu-Manber for performance and showed a 10% - 23% improvement. With respect to memory requirements it was compared with Modified Wu-Manber, Low-Mem Trie, AC-Banded, an unspecified AC variant (AC-Std or AC-Full), and the two compressed Aho-Corasick inspired algorithms from the research of Tuck et al. [86]. To process

the full Snort rule set Piranha consumed 37MB of memory while the other algorithms used 45MB, 14MB, 96MB, 140MB, 20MB and 15MB respectively. However, Piranha was also the fastest option in terms of performance while processing the rule set (preprocessing).

Piranha was also examined and compared for its ability to handle attack traffic although the authors did not use exact worst cases so the results are not conclusive. In fact, it is not trivial to generate worst-case traffic especially considering that every algorithm has a unique worst case. The traffic that was tested was designed to be hard for Piranha and the other algorithms to process, rather than being specifically targeted towards Piranha's exact worst case or that of any particular algorithm. It may be that the traffic that is crafted to give the worst case for one algorithm is actually a good case for another. Thus, algorithms need to be designed to handle their worst-case scenarios well. Furthermore, the best and fairest attack-like comparison of algorithms' performance can only be done by comparing the actual worst cases for the individual algorithms. In the attack cases presented, Piranha was the fastest in two of the three cases. In the case where it was not the fastest the unspecified AC algorithm from Snort was the fastest. To exploit Piranha the worst case would likely contain traffic that causes as many collisions as possible. Ideally, an attacker would like to squeeze as many collisions as possible into one packet while maximizing the time it takes Piranha to identify each substring match as a false positive. This was not maximized in the Piranha worst-case tests the way an attacker would likely maximize the workoad through maliciously crafted input. However, using some of the randomization techniques mentioned above may indeed help to mitigate some of the slowing effects that Piranha exhibits under an attack. In fact, the idea of the randomization is to make the algorithmic complexity attack near impossible because the attacker would not know the substrings and other information needed to craft

worst-case traffic. In closing, we point out one side-effect of Piranha's approach is that it can not handle keywords smaller than four bytes in length.

6.4 Deterministic Memory Efficient String Matching

Tuck et al. [86] propose a strategy for deterministic memory efficient string matching that is a set of two additional string matching algorithms aimed for deployment within Snort, either in software or hardware. A nice trait of the algorithms is that they are deterministic, implying that their worst cases are bounded. In particular, they are linearly bounded in the size of the input they search through. The second algorithm presented builds on the first for even better memory efficiency.

Both algorithms are built on top of the basic non-optimized Aho-Corasick [2] algorithm. They can be thought of as variants of AC and as such we refer to them as AC-Bitmap and AC-Path. They use compression ideas from the strategies employed in IP address lookups on routers or other network devices. They also remark that both the IP lookup problem and string matching problem are alike in the sense that they are both longest prefix problems. The particular part of IP lookup algorithms that they take their compression ideas from is the Eatherton algorithm [34]. Use of these strategies makes AC-Bitmap and AC-Path particularly suitable for hardware—although this is not our focus—given that many network devices use the operations in the Eatherton algorithm, and as such network processors are optimized to perform some of the needed operations quickly. We begin by examining the simpler AC-Bitmap algorithm.

In the non-optimized AC automaton every state has 256 next state pointers and a failure pointer; this can be compressed to a bitmap—viewed as a bit-lookup table—of

256 bits with the regular failure pointer and a next state pointer. The number 256 corresponds to the alphabet size. The bitmap of course indicates whether or not a transition to the next state is valid by setting the appropriate bits in the bitmap during pre-computation. Because there may be more than one actual next state, a single next state pointer is insufficient. When searching if the bitmap indicates that the transition for the input byte is valid then the next state is set to be where the next state pointer points to plus the popcount from the bitmap at the bit checked. The *popcount* refers to the number of bits set (to 1) in the bitmap before the bit that was checked. For example, if the bitmap was `10101001...` and we check the fifth bit from the left—corresponding to an input byte of `0x04`, the fifth character in our alphabet—we see that it is set; therefore, the popcount is 2 because there are two bits set to 1 before (to the left of) the fifth bit. Using this popcount result pointer arithmetic is performed on the next state pointer to increment it by two which will point to the real next state. In this way the next state pointer is not really a pointer to the next state, but it is a base pointer to the space in memory where a list of all the state data structures is stored. Lastly, if the checked bit in the bitmap is not set (i.e. is 0) then we would simply follow the failure pointer, and set the next state equal to the state pointed to by it.

Where the regular size of a state in optimized AC is over 1KB, the structure for the states in the AC-Bitmap algorithm takes only 44 bytes [86]. This vastly decreased memory requirement improves cache behaviour as well as makes it feasible to store the entire automaton in fast SRAM in hardware implementations. The memory savings of AC-Bitmap come at the cost of checking a bit in a bitmap and potentially performing a popcount. The popcount is quite expensive on normal processors not designed for this. To minimize the work Tuck et al. [86] keep running sums of every 32 bits in the bitmap, but a single popcount on up to 31 bits is still needed. Moreover,

this costs a little bit of extra memory to maintain the sums. Naturally, it is not obligated to maintain sums every 32 bits if the memory savings are necessary, and on the contrary it could just as easily switch to running sums every 16 or even 8 bits for better performance and use more memory.

The AC-Path algorithm is built on top of AC-Bitmap, and thus, provides even further compression to minimize the memory requirement. It is based on the observation that the Aho-Corasick automaton, like a trie, is often dense near the root and sparse near the leafs or final states. It often has a single chain of states to a final state. Although Tuck et al. [86] were inspired by the Eatherton algorithm, the idea of path compression in tries is much older. A well-known path compressed trie that has a similar style to the AC-Path automaton is the PATRICIA trie (or PATRICIA tree) [60]. Furthermore, there is a large body of work done on the idea of trie compression because it is well known that standard tries have many nodes with only one child [47, 56, 38, 48, 64].

The AC-Path algorithm further compresses the size of the automaton by reducing the number of states. Because this algorithm builds on the previous one, it remains a requirement that the structures of all the states are the same size so that the pointer arithmetic may work properly. Normally, each state recognizes only one valid character from the input; however, the reduction in number of states happens by effectively adding more than one byte inside a state meaning that a state is not the result of matching a single byte but a string of bytes. For example, instead of a path in the automaton that matches the string `ABCD` with four states (one state per character) the AC-Path compression converts this path of states into one state that matches the entire string. The added overhead in processing time for the AC-Path algorithm now comes from the fact that each of the four separate states in uncompressed format had each of their own failure pointers. This must now be accounted for in the single

compressed state. Furthermore, when transitioning to a state via a failure transition it may now end up at a compressed state where a prefix of the string it represents is already matched, and therefore, it needs to somehow make a failure transition pointer capable of pointing to a location within the string of the state it points to. In practice Tuck et al. [86] report that AC-Path achieves an additional compression of about 2.5 times over AC-Bitmap, which already achieves a compression of about 25 times over the optimized AC automaton in Snort (the Aho-Corasick variant is not given although we believe it is likely the AC-Full variant).

Memory savings motivate this choice of algorithm for any application of Snort concerned with memory. The experimental results show that on average the Modified Wu-Manber option of Snort performs the fastest followed by an unspecified AC variant (we assume AC-Full) which performs about 30% to 45% slower depending on the processor. AC-Bitmap and AC-Path are respectively about 4% and 6% slower than the AC variant, which seems like a small price to pay for the memory savings. In the synthetic worst-case traffic test the AC variant performed the fastest having close to the same processing speed as its average case. AC-Bitmap and AC-Path seemed to slow down a little, being both approximately 10% to 30% slower than the Snort AC variant. The Modified Wu-Manber algorithm was easily exploited in its quadratic worst-case nature and performed approximately 500% (or more) slower than the AC variant depending on the processor.

Prior to the proposal of AC-Bitmap and AC-Path, the memory consumed by algorithms amenable to hardware was too great to be feasible. However, now with this low memory approach an implementation in hardware is possible, and further work has been done on porting the Aho-Corasick algorithm to hardware [83]. Furthermore, these algorithms offer excellent alternatives to the AC-Sparse and AC-Sparse-Banded options within Snort. Although no direct comparison between AC-Bitmap, AC-Path,

AC-Sparse, and AC-Sparse-Banded has been done, we believe that AC-Bitmap and AC-Path will outperform both Snort options as well as consume even less memory based on the experimental results presented in Tuck et al. [86].

6.5 Chapter Summary

In this chapter we focused on Snort by presenting background on Snort and the current version's approaches. Moreover, we present a couple recent approaches that aim to improve Snort's performance and memory usage. In Chapter 7 we observe how the various algorithm choices compare as we also present our own approaches and look intensively into implementing algorithms in Snort's signature matching engine.

Chapter 7

Implementing and Comparing Pattern Matching Algorithms for Snort

This chapter aims to present our observations about Snort, and specifically, its existing pattern matching approaches and the new approaches we add to it. Several pattern matching algorithms are implemented within Snort already for the purpose of providing configuration time options to the NIDS administrator with respect to the search method. That is, Snort may be configured to use a particular algorithm when searching the packet payloads for the patterns contained in its signatures. We have implemented, and furthermore, herein introduce a few new options. In addition to adding new algorithm options inside Snort, an important contribution of this chapter is our discussion and comparison of the existing algorithm options within Snort. Throughout this thesis, it is our goal to provide an unbiased comparison of the work that has been done for use in Snort and other NIDSs. In this chapter we also discuss our findings regarding unexplored pattern matching avenues that we believe to be

useful to pursue further for both researchers in the NIDS field, and potentially for eventual use in NIDSs.

Section 7.1 combines the discussions on the algorithms analyzed in Chapter 5 with the current options available in Snort from Chapter 6 and provides an introduction to our research and findings about the new and existing algorithm options that are most relevant for implementation comparisons in Snort. Section 7.2 introduces the new pattern matching approaches that we have implemented for use in Snort which as mentioned in Section 4.5 is a modification of an existing algorithm. Section 7.3 compares the pattern matching algorithm options available in Snort our modified version of Snort. This includes a comparison of the apposite algorithms identified in Section 7.1 as well as the new options covered in Section 7.2.

7.1 Algorithms Applicable For Snort

We note from Table 5.1 and Section 5.2 that no one algorithm is ideal for all of the cases that a NIDS such as Snort must handle. Nevertheless, Snort works around this problem by allowing trade-offs in various areas. It is becoming generally accepted that for a NIDS, any pattern matching algorithm whose worst-case characteristics are not linearly bounded to the size of the search input are ruled out of consideration, as NIDSs can easily fall prey to algorithmic complexity attacks if they have poor worst-case behaviour. For this reason, although Snort provides an option to use the Modified Wu-Manber algorithm (see Section 6.2), we do not examine it further in this thesis. Though it remains an option in Snort, in our views, in adversarial environments its use should be discontinued entirely. Analogously, we do not consider any algorithms with super-linear worst-case running times or memory consumption.

Besides the aforementioned, the other algorithms that we do not examine are

those in Snort that are experimental. Namely, the AC-Sparse and AC-Sparsebands algorithms that are already available in Snort occasionally do not pick up all matches that they should in packet payloads. These algorithms have yet to be perfected by the engineers working on Snort.

Lastly, we do not implement the Commentz-Walter B1 algorithm for use specifically in Snort despite it providing a linear worst-case running time. There are several reasons for this. First, upon close examination the algorithm does not meet several other important requirements or desirable features for a NIDS (see Section 5.2). Secondly, the implementation of the algorithm is dauntingly complex—as it is already complex enough in theory—and to our knowledge there is no implementation of the algorithm available. Although, the Commentz-Walter B algorithm, which is even less suitable, is widely used in computer applications. In summary, we believe that other algorithms can provide all the benefits and more.

The algorithms that we do pursue as viable options in Snort are the AC-Std, AC-Full, and AC-Banded options. As discussed in Section 6.2 regarding the current version of Snort, we expect the AC-Full algorithm to perform best among these options. To these options already available in Snort, we add three new options. Two of these options named MBOM and MBOM2 are based on the existing Multiple Backward Oracle Matching (MultiBOM) algorithm [6] (see Section 4.5 where we introduce SBOM and MultiBOM). We discuss our implementation of these in Section 7.2. The third option, AUTO, that we introduce is a new kind of approach within Snort because it uses multiple algorithms. Recall that in Snort patterns are grouped by certain characteristics of the rules (signatures) they come from. Only one of these groups are searched for matches per packet received by Snort. Currently, all search method options configure Snort to use the same algorithm on all groups. We propose an approach that first establishes the groups of patterns, then determines which algo-

rithm is best—performance-wise—to use for that group. Specifically, we experiment with using the algorithm from the AC-Full option where the length of the shortest pattern in the group is less than three bytes. In fact, this is most of the groups. For all other groups, we use the algorithm from the new MBOM option. We believe that, in theory, this is the best approach to address the issue discussed at the start of this section of having no one perfect algorithms for all situations. Section 7.3 shows our implementation and test results in practice.

The current release of Snort has 4096 rules enabled by default; this is what we use for testing. In the preprocessing phase Snort forms 197 pattern groups from these signatures' patterns of which there are 16674. In the AUTO option, 16 groups of the 197 have a shortest pattern length of three or more; therefore, they use the algorithm from the MBOM option. Of the 16674 total patterns, 59 patterns fall into these 16 groups.

7.2 Adding the New Algorithms

Inside of Snort we propose and implement three additional search method options (pattern matching algorithms), the main one called MBOM coming from the name of the algorithm it is based on: Multiple Backward Oracle Matching (previously shortened as MultiBOM) [6]. Allauzen et al. [6] first proposed MultiBOM—albeit, without giving its details—in their literature on a related algorithm, Set Backward Oracle Matching (SBOM). SBOM and MultiBOM were first reviewed in this thesis in Section 4.5, although MultiBOM less so. Herein, we give the details of the implementation of our MBOM algorithm (our modification of MultiBOM for use in Snort); also, we no longer distinguish between MBOM and MultiBOM. To our knowledge this is the first published detailed description of any kind on the MBOM algorithm,

as previous literature [6] did not provide its details, but rather the details for SBOM.

As indicated in Section 4.5, MBOM uses two data structures in its search: an Aho-Corasick [2] trie or state machine and a factor oracle. Both structures are built from the entire set of keywords. The AC state machine is built as normal, but the factor oracle is built from the set of reversed patterns[1]. The construction method for the factor oracle is the same as the method given in Section 4.5, but the reversed patterns are used. To build the AC state machine we use the functions available through the AC-Std option in Snort. All of the actual patterns are only stored once with the AC state machine, as the AC state machine will be the structure that actually identifies the matches.

In Snort everything takes place on the pattern group (see Section 2.3.3) level and the life cycle of a pattern group proceeds as follows for preparing any given search method:

- The pattern group is initialized by creating an empty structure to hold the patterns in the group along with the extra variables necessary for the algorithm. This is done through a call to a function whose name ends in "New", such as "mbomNew" for example.

- Each pattern to belong to the group is added to the group by the rule classifier (see Section 2.3.3) through successive function calls. This function name usually ends in "AddPattern", such as "mbomAddPattern".

- The pre-computation for the search algorithm is done once in a single call to a function whose name ends in "Compile", such as "mbomCompile".

- At this point searching is permitted, through calls to a function whose name

[1] We discovered building it from the set of strings of length equal to the shortest pattern (as the SBOM algorithm does) is not acceptable, as some matches in the text could be missed.

ends in "Search", such as "mbomSearch". In Snort this is where the bulk of the processing happens as this function is typically used for every packet that passes through the NIDS.

- To terminate the life of the pattern group and free all memory associated with it, a function whose name ends in "Free" is called, such as "mbomFree". This is only done when the Snort process terminates.

We define the memory used by an algorithm option to be the memory still allocated after the compile function is finished and before the free function is called. During the search phase of an algorithm—in the search function—there are also usually some variables declared in memory on the stack, but this is typically a very small amount that we do not count. Most of the memory allocation is done in the function calls to add the patterns to each pattern group and in the compile function where the structures used in the search are built.

For MBOM, the pattern adding function simply passes along the pattern to the AC state machine by in turn invoking its pattern adding function. Furthermore, it maintains the minimum of all pattern lengths. In the compiling function the factor oracle is built using all of the patterns, which at that point, are all stored in the AC state machine. The C code for these functions can be found in Appendix A.

The MBOM search function called *mbomSearch*, is given in C code in Appendix A as well as in pseudocode below. The search is performed in a window of length equal to the length of the shortest pattern in the group. Within the search we keep a marker called the *critical position* that points to the character to the right of where the scanning with the AC state machine stopped; this starts at position 0. The search first starts by scanning right to left using the factor oracle (which can be substituted for a DAWG). The search is stopped when either we have scanned all the characters

back to the critical position, or there is a mismatch. A mismatch occurs when there is no valid transition in the factor oracle because the substring scanned thus far is not a substring of any pattern. On a mismatch, we reset the current state of the AC state machine to the initial state and the critical position to point to the character to the right of the mismatched character. If there was no mismatch the critical position stays the same. Either way the search then proceeds scanning with the AC state machine.

The current character scanned is always the one at the critical position; therefore, after every character scanned by the AC state machine the critical position is moved to the right by one. In this way, the scanning of the AC state machine always proceeds left to right. This scanning is stopped one of two ways. Either the scanning reaches the end of the search text input, or the critical position moves past the right side of the window while the length of the longest prefix matched in the AC state machine is less than the window length. That is, provided the AC state machine has matched a pattern or pattern prefix at least the length of the window length the scan may continue past the right edge of the window up until a character is reached whereby the state machine's state no longer identifies a prefix longer than the window length. The length of the longest prefix matched by the state machine is given by the depth of the current state, or in other terms, the length of the path back to the initial state.

After the AC state machine has stopped scanning, the window is shifted such that the left side aligns to the critical position less the length of the longest prefix matched in the AC state machine. After the shift, the cycle starts again by scanning with the factor oracle. Of course when the window passes the end of the text input to search, the search is finished. Matches are identified throughout the scanning with the AC state machine by checking for a terminal state after every transition (character scanned). In Snort the search is performed within an uppercase search text; thus, if a

match is identified for a pattern that is not case insensitive we also check the proper case pattern against the proper case text input.

The MBOM search algorithm presented here achieves a sublinear running time on average because the AC state machine does not have to scan every character of the input text. Characters in the input search text can be entirely skipped over every time the factor oracle scanning does not reach the critical position before stopping (when a mismatch occurs). The longer the window length the more probable it is that more characters are skipped over by running into a mismatch with the factor oracle. Furthermore, even in the worst case this search algorithm never scans any character more than twice (once by the oracle and once by the AC state machine). Thus, we achieve a sublinear running time on average while maintaining the important linear time worst-case bound on running time. We define the average case to be independent equiprobable characters in the text input to search. We outline this search algorithm in pseudocode in Algorithm 7.1.

In order to enable the best performance in the MBOM search option, the nodes of the factor oracle contain an array of 256 pointers to other possible nodes in the oracle. This number is due to the size of the alphabet we are forced to use. The array of pointers allows for fast constant time branching or transitioning within the factor oracle much the same way the AC-Std state machine does. This method is the best option for running time performance and simplicity, but can consume significant memory references if the size of the factor oracle gets too large. Each node or state within the oracle contains 1024 bytes just for these pointers.

Eliminating some of the memory consumption will most likely lead to a performance hit; we have explored one option in the implementation of the MBOM2, a variation of MBOM in which nodes are only represented by 16-bit integers (i.e. as numbers between 0 and $2^{16} - 1$). To implement the transitions, and thus, create the

Algorithm 7.1 Multiple BOM Multiple-Keyword Matching Algorithm

1: **procedure** MBOM_SEARCH(y, n)
  ▷ Input:
  ▷ $y \leftarrow$ array of n bytes representing the text input
  ▷ $n \leftarrow$ integer representing the text length
  ▷ Assume access to the oracle & AC state machine (ACSM) (preprocessing)
  ▷ oracle_next and acsm_next are transition functions
  ▷ is_terminal is equivalent to the AC output function o (see Section 4.1)

2:   $min \leftarrow$ length of shortest pattern (in pattern group) ▷ Also window size
3:   $critpos \leftarrow 0$
4:   $i \leftarrow 0$
5:   $j \leftarrow 0$
6:   **while** $i < n - min + 1$ **and** $critpos < n$ **do**

    ▷ Search in the factor oracle until hitting a mismatch:
7:     $j \leftarrow i + min - 1$
8:     $current \leftarrow$ oracle initial state

9:     **while** $j \geq critpos$ **and** ($current \leftarrow$ oracle_next($current, y[j]$)) $\neq \emptyset$ **do**
10:       $j \leftarrow j - 1$
11:     **end while**

12:     **if** $j \geq critpos$ **then** ▷ If it didn't make it all the way to the critpos
13:       $state \leftarrow$ AC initial state ▷ Reset ACSM
14:       $critpos \leftarrow j + 1$
15:     **end if**

    ▷ Search in the AC state machine:
16:     **while** $critpos < n$ **and** ($critpos < i + min$ **or** depth($state$) $\geq min$) **do**
17:       $state \leftarrow$ acsm_next($state, y[critpos]$) ▷ Scan a character
18:       $critpos \leftarrow critpos + 1$

19:       **if** is_terminal($state$) **then** ▷ If there are matches at this state
          ▷ Output location that matches start in y
20:         For each match: **output** $critpos$ – length of pattern matched
21:       **end if**
22:     **end while**

23:     $i \leftarrow critpos -$ depth($state$) ▷ Shift

24:   **end while**
25: **end procedure**

oracle structure in MBOM2, we use a hashtable for the whole structure to cut down on the memory wasted by the MBOM option in the arrays of pointers kept by every node. The keys used in the hashtable are pairs of a state number and a character from the alphabet, and the values stored in the hashtable are state numbers alone. In the hashtable a key—a state number and a character—maps to another state number to represent a transition from the state in the key to the other state. This saves memory by only allocating memory for transitions that exist in the factor oracle. In this MBOM2 option the size of the state is consequently variable, and depends on the number of transitions it has leaving it. The memory consumption of the factor oracle is now equal to the memory consumption of the hashtable, which should be greatly reduced. The trade-off is that our branching or transitioning is now dependant on the speed of a hashtable lookup. Although hashtable lookups are constant time on average in theory, in practice one should expect a decrease in performance (compared to the MBOM branching) because of the key hashing and possibly going through a list of entries in a hashtable bucket. We do not go into detail with respect to hashtables in this thesis. The actual hashtable's code was taken from Clark [17], who makes this implementation available under a BSD license available at http://www.cl.cam.ac.uk/c̃wc22/hashtable/. The code for the MBOM2 option is also given in Appendix A.

Other variations of the MBOM option that would save memory could potentially use one hashtable per node. Also it is possible to save memory in the leaf nodes (states) in the trie that becomes the factor oracle by not allocating an array of pointers or a hashtable for those nodes unless they have transitions from them. Other options for storing the transitions would be in linked lists or for faster access (but more complex choice) we could use balanced trees or skip-lists. We do not explore these options in this thesis as we believe that hashtables should give the fastest

transition (branching) speed compared the other options mentioned above. Further improvements suggested for our MBOM algorithm are given in Section 8.3.

The last search method option that we created in Snort is called AUTO. The AUTO option implements no pattern matching algorithm on its own, but rather uses the ones from the other Snort options. The purpose of the AUTO option is to address the problem introduced in Chapter 5 of there being no one algorithm to best suit all situations. The variance in the pattern groups is one aspect that is problematic, yet it can be addressed; hence, our implementation of the AUTO option. In Snort all pattern groups currently get assigned the same search method from the configured option, but using our AUTO option we change this. During preprocessing we add all patterns to a pattern group structure appropriate for the AC-Full algorithm, but at the compile function call to assemble the pattern matching structures the pattern group is examined. In particular, we look at the length of the shortest pattern, and if it exceeds two then the pattern group is compiled for the MBOM option, otherwise it is compiled for the AC-Full option. Thus, the AUTO option decides intelligently on a pattern group-by-group basis whether or not the MBOM or AC-Full algorithm would usually work best for pattern matching using the keyword set held in the pattern group. We expect that for keyword sets where the shortest length pattern is three or more that the MBOM algorithm will be able to process the input sublinearly by skipping over parts of the input; accordingly, the AUTO uses MBOM for pattern groups where the length of the shortest keyword above two. As mentioned previously, longer lengths of the shortest pattern translate to larger potential window shifts in the MBOM algorithm; thus, on average as the length grows, so does the speed of the MBOM algorithm. For cases where the shortest pattern length is one or two, it is senseless to use the MBOM algorithm because skipping over parts of the input text becomes impossible—and there is slight overhead to using the factor oracle in

addition to a AC state machine; thus, in these cases (pattern groups) we choose the fastest AC algorithm, AC-Full, being that AC is best suited to this case.

Under constant worst-case conditions both algorithms are linearly bounded, but in practice the MBOM algorithm would be slightly slower because of the slight overhead to using the factor oracle in addition to a AC state machine, as mentioned above. In the common case where large window shifts in MBOM result in skipping parts of the input text, the overhead should easily be more than compensated for by this sublinear behaviour, resulting in MBOM outperforming all linear behaving AC algorithms. By combining the MBOM and AC-Full algorithms in this way into AUTO, we expect that AUTO should be the most sensible and the fastest option in practice most of the time.

7.3 Evaluating Our Algorithms In Snort

The experimentation that compares all the Snort options including the new options is performed with two test files that contain traffic captures from the Shmoo group's Capture the Capture the Flag project [85]. In this section we compare memory usage amongst the options as well as the performance in processing the tcpdump-style (PCAP) [55] network traffic files. The performance measurement is taken with respect to the search time only. Therefore, the duration of the preprocessing and post-processing is not taken into account. Furthermore, we turned off all traffic preprocessors in Snort except for the flow preprocessor as to focus the running-time results specifically on the search algorithms. Our results are documented in Table 7.1 and Table 7.2.

Firstly, the Orange and Red times come from the test files used which were orange.cctf.tar.gz and red.cctf.tar.gz. These are the traffic captures from the orange

and red teams from the DEFCON 10 conference's Capture The Flag competition. Although The Shmoo Group provides these tarballs containing multiple PCAP data files, we used the mergecap tool that comes with tcpdump to merge the data files into single data files of the network traffic captures. The red team's data capture was 42.3 MB, and the orange team's totaled 365.1 MB.

Secondly, we ran our test processing files generating no alerts (using "`snort -A none ...`") to maximize speed and avoid measuring times unrelated to the pattern matching algorithms. The time results in the tables below are the average times after 50 executions. Furthermore, the testing was done on two separate modern machines for comparison and results verification purposes:

- Machine 1 (from Table 7.1) was a Pentium4 2.8 GHz, 1 GB RAM, SATA HD that reads at 150 MB/s, Fedora 2 Linux 2.6.8
- Machine 2 (from Table 7.2) was a Pentium4 2.4 GHz, 1 GB RAM, SCSI HD that reads at 320 MB/s, Debian Linux 2.6.8

Search Method	Orange Time (sec)	Red Time (sec)	Memory Usage (KB)
AC-Banded	9.50091637	1.28260256	37,845.03
AC-Standard	8.81949943	1.2242112	157,967.03
AC-Full	8.3009172	1.21818092	80,317.42
Auto	8.2276888	1.22182818	81,348.35
MBOM	12.43954745	1.40906625	247,358.81
MBOM2	27.20387846	2.29741921	163,126.28

Table 7.1: Average results for Snort search methods on machine 1

Our results in practice are generally unsurprising compared to our theoretical expectations. The MBOM options are built on top of the AC-Standard implementation, and hence consume the same amount of memory for their Aho-Corasick [2] state machine alone. The remainder of the memory is what was actually consumed for the factor oracle. This is only a very small amount for the MBOM2 option as expected

Search Method	Orange Time (sec)	Red Time (sec)	Memory Usage (KB)
AC-Banded	8.41199482	1.25546272	37,845.03
AC-Standard	12.54950718	1.35974246	157,967.03
AC-Full	7.38273514	1.22266912	80,317.42
Auto	7.41804276	1.19416742	81,348.35
MBOM	13.83654648	1.41934164	247,358.81
MBOM2	27.54345626	2.23155244	163,126.28

Table 7.2: Average results for Snort search methods on machine 2

because it uses a hashtable. Both options should be, and are, slightly slower than the AC options because the shortest pattern length is one for most of the pattern groups. This means that the search window is reduced to a length of one, and thus, skipping parts of the search text, and accordingly, achieving a sublinear running time becomes impossible. The running time for the MBOM option, nonetheless, shows in both tables that the overhead—over the AC-Standard option—is fairly minimal. This is especially true given that for the MBOM and MBOM2 results the window size was often one in many pattern groups—due to a shortest keyword of length one in many pattern groups—which is the worst case for the MBOM algorithm. With further improvements (see Section 8.3) the MBOM option overhead could be lessened, making performance better.

In regards to the MBOM2 option, although the memory savings are good, it is quite slow. This is due to the chosen hashtable's hash function for hashing the keys. It may be too slow, and not sufficiently good enough to the point where it is causing collisions. The hash function specifically, needs more experimentation done on it to find the point where it is complex enough to avoid many bucket collisions while still being as fast as possible in terms of its machine instructions. Further hashtable analysis is not pursued herein as it is left for future work, but we have an idea of the memory savings when implementing an automaton (in this case a factor oracle) using a hashtable.

Between the Aho-Corasick variants, the AC-Full option is consistently the best option for performance, and the AC-Banded offers good memory compression of over 50 percent while sacrificing only a little bit of that performance. We conclude that the AC-Standard option should no longer be used given that it consumes more memory than the other variants, and performs the worst on average on one of the test machines. The difference in the AC-Standard times between machines could be due to more (better) caching on the newer 2.8 GHz processor of machine 1. That is, it is better able to handle the larger memory load of the AC-Standard method.

If software-based NIDS deployers will see running time performance as a greater benefit than small memory consumption—as is often the case—we conclude that the AC-Full and AUTO options are the best choices. Naturally they run very close in performance because the AUTO option is actually using the AC-Full method for pattern groups when their shortest patterns are of length one or two. This represents 181 of the 197 pattern groups. Because the times for AUTO run quite close to those of AC-Full, for the MBOM usage of the other 16 groups, we see that the MBOM method is approximately the same speed as the AC-Full method (in two cases AUTO is slightly faster and in two cases slightly slower). With further improvements to MBOM as mentioned above it should clearly outperform AC-Full where it is used which is unfortunately not in many pattern groups.

In conclusion, this data set together with our implementations, suggest that the MBOM algorithm has as much merit in NIDSs as does the Aho-Corasick algorithm provided it is used with appropriate keyword sets. The key in this case was identifying that different algorithms fit different cases, and within Snort this means its pattern groups (that hold the keyword sets).

We learned that in the worst case MBOM performs very well even though it is not quite as good as the AC variants, and on average MBOM could be faster than

the Aho-Corasick methods so long as the window size is large enough which is why we have chosen a length of three. Lastly, we believe that the approach we use in AUTO is a serious contender in pattern matching algorithms for the general class of software-based NIDSs.

7.4 Chapter Summary

In this chapter we examined the search method options in Snort that we believe to be the best for finding pattern matching in the context of a NIDS. Our contributions include adding three options which showcase a kind of pattern matching algorithm previously unseen in Snort and in any NIDS to our knowledge. Furthermore, we detailed the implementation of the MBOM search method option and algorithm, and we give an in-depth explanation of it in pseudocode and C code (in Appendix A). Finally, we presented an enlightening comparison of the introduced search method options along with the best existing options already present in Snort.

Chapter 8

Further Discussion and Concluding Remarks

8.1 Pattern Matching Algorithms in Other Security Applications

Multiple-pattern matching is widely used in many other applications such as virus detection, spam detection, content scanning, and filtering. In this section we briefly touch on a couple of such computer security related applications.

8.1.1 Antivirus Software

Perhaps the most renowned security application is antivirus software. First-generation virus string scanning—which may be the most obvious and simple virus detection method—is handled quite similarly to what we see in a NIDS like Snort [82]. String scanning uses an extracted sequence of bytes (string) from a virus that is unlikely to appear in clean (non-virus) programs or files [82]. This string is used as a virus

signature the same way extracted sequences of bytes from network packets code for a signature in a Snort detection rule. Virus signatures are organized into databases as are NIDS signatures. Indeed, the databases can contain thousands of signatures just as Snort's does. Given the long-term existence of antivirus software it is noteworthy to contrast how antivirus software performs its scanning. Certainly, antivirus software uses more advanced techniques than string scanning for detection, but nevertheless, string scanning remains a heavily used and powerful technique [82].

Fast scanning using plain single-keyword pattern matching algorithms such as the Boyer-Moore algorithm [14] are evidently not fast enough for antivirus software [82]. What is more, some software may support the use of single- and multiple-character wildcards in their signatures, and consequently, they demand more sophisticated algorithms. Although, using wildcards causes a performance penalty. Hashing and the use of lookup tables—not so different than in Wu-Manber [92]—have been implemented by many antivirus researchers as a method that speeds up string scanning algorithms [82, 1, 70]. Typically these algorithms use 16-bit or 32-bit words as an index into the hashtable. Other techniques include the use of bookmarks or check bytes, top-and-tail scanning, and entry-point and fixed-point scanning [82], but some of these techniques are not portable to NIDS when the whole payload must be searched, as is customary. One interesting concept here is coding an expected location or skip offset along with the signature as Matrawy et al. combine a pattern and location [57]. Naturally the difficulty with searching at a fixed or calculated offset is that in network traffic, packet manipulation can throw off the detection if the attacker can move the malicious pieces around within the payload.

While some antivirus software does use hashing-based algorithms, there are others such as ClamAntiVirus (ClamAV) [52] that do, indeed, use the Aho-Corasick algorithm [2]. In fact, its implementation is much the same as that of the AC-Std search

method in Snort, using a trie structure with a 256-element lookup array [58]. Given the newer implementations of the Aho-Corasick algorithm in Snort and the research on further optimizations for the algorithm in NIDSs, we expect that the open-source antivirus project researchers will quickly harness the benefits in algorithm optimizations, if they have not already. At least in the published works of the open-source and academic communities antivirus pattern matching research can potentially be useful to NIDS researchers and vice-versa. As well, AVFS [58], a true on-access antivirus system, was also able to achieve speeding up the pattern matching within the ClamAV search engine because it reduces a factor in its performance to logarithmic from linear in the number of patterns to search for. The issue of scalability for the antivirus pattern matching algorithms is just as important as for signature-based NIDS pattern matching algorithms. In fact, there are many more ClamAV virus signatures than there are Snort signatures.

Lastly, Salmela et al. [73] offer further suggestions to improve search algorithms' performance in antivirus and intrusion detection systems. They present new methods that improve algorithm performance in their tests by, in effect, creating a larger alphabet by grouping bytes together and looking at 16-bit or 32-bit words as alphabet characters. This is quite clever using the sublinear style algorithms that they have chosen because it creates more mismatches, and hence, more and larger shifts. Unfortunately, as we have seen in this thesis, these kinds of pattern matching algorithms—and others—also raise other concerns; one such important factor is the length of the minimum size pattern. In general, the smaller this length is, the worse the performance and memory consumption is. Therefore, we feel this approach would not work well when the minimum length pattern in the pattern set is already small which is, for example, often the case in Snort's pattern groups.

8.1.2 Spam Detection Software

Some spam e-mail filters, like SpamAssassin [44], use pattern matching with signatures, but most do so in a very different way than a virus scanner or a NIDS. For example, in a NIDS if a signature is matched, then a certain associated action is taken, but when a spam signature is matched, often the rule associated with the signature just affects of the spam score of the message. The difference here is that a false positive spam identification for an e-mail message is taken more seriously (depending on the context). Thus, spam detection software, such as the well-known SpamAssassin [44], uses a score-tally approach to flag spam. Only after passing through the scoring phase where spam-like patterns are matched is the message marked definitively as spam or valid e-mail.

Most anti-spam software programs do not document what algorithms they use for pattern matching, but upon analysis of open-source software there are many that still use single-keyword pattern matching algorithms. We conjecture that the reasons for this could be that e-mail processing latency does not have the same high-speed processing demands compared to network traffic processing in NIDSs. Moreover, due to the score-tally approach used by some anti-spam applications there are fewer rule updates to the rule database, meaning that the number of rules scales much slower than in virus or NIDS signature databases. In summary, we believe that even if there are low performance demands on spam software, certain spam applications could benefit by pursuing some of the pattern matching research from the NIDS community (and others).

Lastly, the rule filtering method is not the only detection method employed. Namely, statistical filters that perform Bayesian-style probabilistic classification [42, 79] are also very popular in this genre of application.

8.2 Pattern Matching in Hardware

As NIDS interest has grown wider there has been a substantial amount of work to migrate signature matching engines to hardware. Most of the work in this area seems as if it has been inspired from the software-based signature detection engine of Snort.

To this effect, there have been several new architectures proposed along with modifications to the existing algorithm that Snort started using back in version 2.0, that is, the Aho-Corasick style multiple-pattern matching algorithm. The Aho-Corasick pattern matching algorithm lends itself well to trial hardware implementations because it is the only deterministic algorithm of its kind, which is of utmost importance for algorithms implemented in hardware. However, a difficulty faced when implementing this algorithm in hardware is finding an architecture to represent the Aho-Corasick state machine with a small amount of memory. There can often be a large number of transitions in one of these state machines. This is due to many factors like the variance in patterns and their lengths, and the large size of the alphabet to deal with. For example, a single-character pattern can cause all states to have a transition to it. Modifications like bitmap and path compression (see Section 3.4.4) [86] to the Aho-Corasick state machine have been seen to drastically reduce the space needed to represent it. Another hardware implementation that even further reduced the size of the state machine was achieved implementing transitions as prioritized rules that can contain wildcards [87].

Many hardware trial implementations also borrow ideas from the well-studied solutions for fast Internet protocol (IP) address lookup on routers. Because the state transitions can be kept in a table in hardware, searching for the correct transition to follow can be achieved quickly using previously developed techniques in hardware structures like: ternary content addressable memory (TCAM) [93], field pro-

grammable gate arrays (FPGA) [33, 87], and custom solutions in application specific integrated circuits (ASIC) [83, 87]. In these hardware solutions the processing speed of the system as a whole is tied chiefly to the speed of these technologies' ability to find the correct transition, follow it, and process any pattern matches associated with the new state. Recent experimental results for FPGA implementations targeted for NIDS have been able to process input streams at speeds well into the low multiple gigabits per second (Gbps) area, with calculations to escalate those processing speeds above 20 Gbps for ASIC implementations [87].

Another issue hardware-based NIDSs face is the update process of the signature database. Some hardware solutions have facilitated pattern updates while the NIDS is running. Software-based NIDSs like Snort and Bro [67, 66] have not implemented this. This may be because enabling this extra feature is probably more important for hardware devices where a total system reload or reboot may be more expensive than restarting the Snort daemon for example.

8.3 Future Work in the MBOM Snort Options

One point of interest for both the MBOM and MBOM2 search method options for Snort that we implemented is that the Aho-Corasick [2] state machine and algorithm that is used within them comes from the AC-Std option. The AC-Std option is the oldest implementation of the Aho-Corasick algorithm and state machine within Snort, and it consumes slightly more memory than the newer AC-Full option. The AC-Full option also outperforms the AC-Std implementation by a small margin in our experimentation. The distinguishing difference in the AC-Full option is the implementation of a matrix as the state table [65]. Furthermore, the matrix state table format allows for further compression; a working example of this is implemented in the AC-Banded

option. One can picture this state transition matrix by thinking of the rows as current states, the columns as transition events (this is next input character from the input text to the matching algorithm), and the contents of each cell in the matrix as the next state. We expect a considerable improvement in memory usage and possibly performance should the new MBOM options be implemented using the AC-Full state machine in place of using the one from AC-Std. The visible disadvantage of this is that the AC-Full pattern matching code is more obscure, and hence, possibly not the best first option for a good understanding of the MBOM implementation. However, as a next step we foresee that using the new AC-Full state machine is likely a better choice. Moreover, it is also possible to implement the factor oracle using a similar matrix. Given the improvement of AC-Full over AC-Std this may indeed improve performance or memory consumption with respect to the factor oracle used in MBOM. Given that our AUTO option uses the MBOM algorithm, these suggestions would also improve the performance and memory usage for AUTO, which is our fastest and one of our recommended options.

8.4 Concluding Remarks

We have presented an array of existing multiple-pattern matching algorithms and explored their characteristics. We have also provided background, pseudocode, and clear explanations of the algorithms. In particular, we pursue research and elucidate the MBOM algorithm. To our knowledge, it is not explained completely—or in English—in other literature and available code. Furthermore, although the Aho-Corasisk [2] algorithm is used directly in NIDS algorithms, its consideration as a sub-component to another algorithm, such as MBOM, is a novel contribution within the NIDS community.

Another algorithm that we investigate in our research is the Commentz-Walter [23, 24] B1 algorithm. Again, to our knowledge, this algorithm has not received much attention at all in general—despite its ability to outperform the simpler B variant. Its consideration within past NIDS research is especially lacking given its linearly bounded worst-case running time. The B1 algorithm has apparently not been considered in NIDS literature to our knowledge, since all references to Commentz-Walter discuss the B variant—usually by implication of non-linearly bounded worst-case behaviour—instead of distinguishing the B variant.

In presenting all the algorithms and their characteristics we also compare and contrast the algorithms in general, and also with a focus on the context of signature matching in NIDSs. By examining the requirements for NIDSs like Snort we have revealed which of the available traits provided by multiple-pattern matching algorithms are most desired and which algorithms are indeed suited to fulfill the requirements. In doing so, we discussed and summarized that very few existing pattern matching approaches are actually appropriate for NIDSs.

Covering this broad range of pattern matching algorithms we discovered that the previously untested approach implemented in our MBOM search option for Snort is quite suitable for certain cases in software-based NIDS signature matching engines. Our AUTO option takes advantage of this fact and makes case-by-case decisions on which algorithm to use given the pattern group in Snort's signature matching engine. In Snort, we showed that this algorithm has merit in the cases when it is used on pattern groups where the shortest pattern length is not too small. Specifically, our AUTO algorithm shows promise because it was able to outperform all other Snort options in half of our test cases.

Appendix A

Modifications to Snort

This appendix covers the modifications made to Snort version 2.6 for the introduction of the MBOM, MBOM2, and AUTO search method options. In addition to the C code and details documented herein, a source code package that accompanies this thesis work is available from `ftp://jameskelly.net/mcs/`. The package contains not only the modified version of Snort 2.6, but also the testing scripts and tcpdump files used in the comparison experimentation from Section 7.3. Finally, the accompanying source code package also includes the modified *make* files that are not included here.

A.1 Adding the New Search Method Options

In this section we document the code changes to allow for the insertion of the new search methods. In addition, the `mpse.c` file was completely reworked to enable the AUTO option which makes an intelligent decision about what is the best algorithm on a pattern group-by-group basis. Using the global AUTO option as the search method changes pattern groups with the shortest pattern of length less than three to use the AC-Full option on that pattern group. For all other pattern groups the

AUTO option switches to use the MBOM option.

Changes to Snort source file `fpcreate.c`:

```
/*
** fpcreate.c (snippet)
**
** Search method is set using "config detect: search-method ac | mwm | auto | etc..."
*/
int fpSetDetectSearchMethod( char * method )
{
    LogMessage("Detection:\n");

    if( !strcasecmp(method,"ac-std") )                                    10
    {
       fpDetect.search_method = MPSE_AC ;
       LogMessage(" Search-Method = AC-Std\n");
       return 0;
    }
    if( !strcasecmp(method,"ac") )
    {
       fpDetect.search_method = MPSE_ACF ;
       LogMessage(" Search-Method = AC-Full\n");
       return 0;                                                          20
    }
    if( !strcasecmp(method,"acs") )
    {
       fpDetect.search_method = MPSE_ACS ;
       LogMessage(" Search-Method = AC-Sparse\n");
       return 0;
    }
    if( !strcasecmp(method,"ac-banded") )
    {
       fpDetect.search_method = MPSE_ACB ;                                30
       LogMessage(" Search-Method = AC-Banded\n");
       return 0;
    }
    if( !strcasecmp(method,"ac-sparsebands") )
    {
       fpDetect.search_method = MPSE_ACSB ;
       LogMessage(" Search-Method = AC-Sparse-Bands\n");
       return 0;
    }
                                                                          40
    if( !strcasecmp(method,"mwm") )
    {
       fpDetect.search_method = MPSE_MWM ;
       LogMessage(" Search-Method = Modified Wu-Manber\n");
       return 0;
    }
```

```
    if( !strcasecmp(method,"lowmem") )
    {
        fpDetect.search_method = MPSE_LOWMEM ;                                   50
        LogMessage(" Search-Method = Low-Mem Trie\n");
        return 0;
    }

    if( !strcasecmp(method,"mbom") )
    {
        fpDetect.search_method = MPSE_MBOM ;
        LogMessage(" Search-Method = Multiple Backwards Oracle Matching\n");
        return 0;
    }                                                                            60

    if( !strcasecmp(method,"mbom2") )
    {
        fpDetect.search_method = MPSE_MBOM2 ;
        LogMessage(" Search-Method = Multiple Backwards Oracle Matching v2\n");
        return 0;
    }

    if( !strcasecmp(method,"auto") )
    {                                                                            70
        fpDetect.search_method = MPSE_AUTO ;
        LogMessage(" Search-Method = Automatic (Smart Mode)\n");
        return 0;
    }
    return 1;
}
```

Changes to Snort source file `sfutil/mpse.h`:

```
/*
** mpse.h
**
** Multi-Pattern Search Engine
**
*/

#ifndef _MPSE_H
#define _MPSE_H
                                                                                 10
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "bitop.h"
```

```
/*
*   Move these defines to a generic Win32/Unix compatability file,
*   there must be one somewhere...
*/                                                                20
#ifndef CDECL
#define CDECL
#endif

#ifndef INLINE
#define INLINE inline
#endif

#ifndef UINT64
#define UINT64 unsigned long long                                 30
#endif


/*
*  Pattern Matching Methods
*
*  Added: MBOM and MBOM2
*  Fixed: AUTO which was previously useless
*/
#define MPSE_MWM 1                                                40
#define MPSE_AC  2
#define MPSE_KTBM 3
#define MPSE_LOWMEM 4
#define MPSE_AUTO 5
#define MPSE_ACF 6
#define MPSE_ACS 7
#define MPSE_ACB 8
#define MPSE_ACSB 9
#define MPSE_MBOM 10
#define MPSE_MBOM2 11                                             50

/*
** PROTOTYPES
*/
void * mpseNew(int method);

void   mpseFree(void * pv);

int    mpseAddPattern(void * pv, void * P, int m, unsigned noCase,
         unsigned offset, unsigned depth, void * ID, int IID);    60

void   mpseLargeShifts(void * pv, int flag);

int    mpsePrepPatterns(void * pv);

void   mpseSetRuleMask(void *pv, BITOP * rm);
```

```
int     mpseSearch(void * pv, unsigned char * T, int n,
         int (*action) (void * id, int index, void * data), void * data);
                                                                        70
UINT64 mpseGetPatByteCount();

void   mpseResetByteCount();

int    mpsePrintDetail(void * obj);

int    mpsePrintSummary();

#endif
```

Changes to Snort source file `sfutil/mpse.c`:

```
/*
**   mpse.c
**
**   An abstracted interface to the Multi-Pattern Matching routines,
**   thats why we're passing 'void *' objects around.
**
*/

#ifdef HAVE_CONFIG_H
#include "config.h"                                                     10
#endif

#include "bitop.h"
#include "mwm.h"
#include "acsmx.h"
#include "acsmx2.h"
#include "sfksearch.h"
#include "mbom.h"
#include "mbom2.h"
#include "mpse.h"                                                       20

#include "profiler.h"
#ifdef PERF_PROFILING
#include "snort.h"
PreprocStats mpsePerfStats;
#endif


// NEW: these two must correspond
#define AUTO_DEFAULT MPSE_ACF                                           30
#define AUTO_DEFAULT_AC ACF_FULL


static UINT64 s_bcnt=0;
```

```
typedef struct _mpse_struct {

  int     method;
  void * obj;
                                                                  40
}MPSE;

void * mpseNew(int method)
{
   MPSE * p;

   p = (MPSE*)malloc( sizeof(MPSE) );
   if( !p ) return NULL;

   p->method=method;                                              50
   p->obj =NULL;
   s_bcnt =0;

   switch( method )
   {
     case MPSE_MWM:
        p->obj = mwmNew();
        return (void*)p;
     case MPSE_AC:
       p->obj = acsmNew();                                        60
       return (void*)p;
     case MPSE_AUTO:
       p->obj = acsmNew2();
       if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,AUTO_DEFAULT_AC);
       return (void*)p;
     case MPSE_ACF:
       p->obj = acsmNew2();
       if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,ACF_FULL);
       return (void*)p;
     case MPSE_ACS:                                               70
       p->obj = acsmNew2();
       if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,ACF_SPARSE);
       return (void*)p;
     case MPSE_ACB:
       p->obj = acsmNew2();
       if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,ACF_BANDED);
       return (void*)p;
     case MPSE_ACSB:
       p->obj = acsmNew2();
       if(p->obj)acsmSelectFormat2((ACSM_STRUCT2*)p->obj,ACF_SPARSEBANDS);   80
       return (void*)p;
     case MPSE_KTBM:
     case MPSE_LOWMEM:
        p->obj = KTrieNew();
       return (void*)p;
     case MPSE_MBOM:
```

```
        p->obj = mbomNew();
        return (void*)p;
      case MPSE_MBOM2:
        p->obj = mbomNew2();                                              90
        return (void*)p;
      default:
        return 0;
    }
}


void mpseFree(void * pvoid)
{
  MPSE * p = (MPSE*)pvoid;                                                100

  switch( p->method )
    {
      case MPSE_AC:
        if(p->obj)acsmFree(p->obj);
        free(p);
        return;
      case MPSE_ACF:
      case MPSE_ACS:
      case MPSE_ACB:                                                      110
      case MPSE_ACSB:
        if(p->obj)acsmFree2(p->obj);
        free(p);
        return;
      case MPSE_MWM:
        if(p->obj)mwmFree(p->obj);
        free( p );
        return;
      case MPSE_KTBM:
      case MPSE_LOWMEM:                                                   120
        return; //no free? - JK
      case MPSE_MBOM:
        if(p->obj) mbomFree((MBOM_STRUCT *)p->obj);
        free(p);
        return;
      case MPSE_MBOM2:
        if(p->obj) mbomFree2((MBOM_STRUCT2 *)p->obj);
        free(p);
        return;
      case MPSE_AUTO:                                                     130
        // Shouldn't get here if compiled mpsePrepPatterns must be called
        // Because method is always reset to something else after compile
        free(p);
        return;
      default:
        return;
    }
```

```
}

int mpseAddPattern(void * pvoid, void * P, int m,                            140
     unsigned noCase, unsigned offset, unsigned depth, void* ID, int IID)
{
  MPSE * p = (MPSE*)pvoid;

  switch( p->method )
  {
    case MPSE_AC:
      return acsmAddPattern( (ACSM_STRUCT*)p->obj, (unsigned char *)P, m,
             noCase, offset, depth, ID, IID );
    case MPSE_ACF:                                                           150
    case MPSE_ACS:
    case MPSE_ACB:
    case MPSE_ACSB:
    case MPSE_AUTO:
      return acsmAddPattern2( (ACSM_STRUCT2*)p->obj, (unsigned char *)P, m,
             noCase, offset, depth, ID, IID );
    case MPSE_MWM:
      return mwmAddPatternEx( p->obj, (unsigned char *)P, m,
             noCase, offset, depth, (void*)ID, IID );
    case MPSE_KTBM:                                                          160
    case MPSE_LOWMEM:
      return KTrieAddPattern( (KTRIE_STRUCT *)p->obj, (unsigned char *)P, m,
             noCase, ID );
    case MPSE_MBOM:
      return mbomAddPattern( (MBOM_STRUCT *)p->obj, (unsigned char *)P, m,
             noCase, offset, depth, (void*)ID, IID );
    case MPSE_MBOM2:
      return mbomAddPattern2( (MBOM_STRUCT2 *)p->obj, (unsigned char *)P, m,
             noCase, offset, depth, (void*)ID, IID );
    default:                                                                 170
      return -1;
    break;
  }
}

void mpseLargeShifts (void * pvoid, int flag)
{
  MPSE * p = (MPSE*)pvoid;

  switch( p->method )                                                        180
  {
    case MPSE_MWM:
      mwmLargeShifts( p->obj, flag );
      break;
    default:
      return;
  }
}
```

```
// uncomment to use mbom2 in function below:                          190
// #define AUTO_MAX_STATES 8192

int mpsePrepPatterns(void * pvoid)
{
  MPSE * p           = (MPSE *)pvoid;
  ACSM_STRUCT2 * acsm = NULL;
  MBOM_STRUCT * mbom = NULL;
  ACSM_PATTERN2 * plist = NULL;

#ifdef AUTO_MAX_STATES                                                200
  MBOM_STRUCT2 * mbom2 = NULL;
#endif

  switch( p->method )
  {
     case MPSE_AC:
       return acsmCompile((ACSM_STRUCT*)p->obj);
     case MPSE_ACF:
     case MPSE_ACS:
     case MPSE_ACB:                                                   210
     case MPSE_ACSB:
       return acsmCompile2((ACSM_STRUCT2*)p->obj);
     case MPSE_MWM:
       return mwmPrepPatterns(p->obj);
     case MPSE_KTBM:
     case MPSE_LOWMEM:
       return KTrieCompile((KTRIE_STRUCT *)p->obj);
     case MPSE_MBOM:
       return mbomCompile((MBOM_STRUCT *)p->obj);
     case MPSE_MBOM2:                                                 220
       return mbomCompile2((MBOM_STRUCT2 *)p->obj);
     case MPSE_AUTO:
       acsm = (ACSM_STRUCT2*)p->obj;
       if(acsm != NULL) {
         if(acsm->minLen > 2) { // shortest pattern is length 3 or above

#ifdef AUTO_MAX_STATES
           // use mbom because it's faster on avg
           if(acsm->acsmNumStates > AUTO_MAX_STATES) {
             // use mbom2 to save mem                                 230
             mbom2 = p->obj = mbomNew2();
             p->method = MPSE_MBOM2;
             // move patterns into new struct
             for (plist = acsm->acsmPatterns; plist != NULL; plist = plist->next) {
               mbomAddPattern2(mbom2, plist->casepatrn, plist->n, plist->nocase,
                           plist->offset, plist->depth, plist->id, plist->iid);
             }
             acsmFree2(acsm);
             return mbomCompile2(mbom2);
```

```
            }                                                               240
#else
            // otherwise use normal mbom
            mbom = p->obj = mbomNew();
            p->method = MPSE_MBOM;
            // move patterns into new struct
            for (plist = acsm->acsmPatterns; plist != NULL; plist = plist->next) {
              mbomAddPattern(mbom, plist->casepatrn, plist->n, plist->nocase,
                         plist->offset, plist->depth, plist->id, plist->iid);
            }
            acsmFree2(acsm);                                                250
            return mbomCompile(mbom);
#endif
          }
          else {

            p->method = AUTO_DEFAULT; // go on using the ACSM
            return acsmCompile2(acsm);

          }
        }                                                                   260
      default:
        return 1;
  }
}

void mpseSetRuleMask(void * pvoid, BITOP * rm)
{
  MPSE * p = (MPSE*)pvoid;

  switch( p->method )                                                       270
  {
     case MPSE_MWM:
       mwmSetRuleMask( p->obj, rm );
     break;

     default:
       return ;
     break;
  }
}                                                                           280

int mpsePrintDetail(void * pvoid)
{
  MPSE * p = (MPSE*)pvoid;

  switch( p->method )
  {
     case MPSE_AC:
      return acsmPrintDetailInfo( (ACSM_STRUCT*) p->obj );
     case MPSE_ACF:                                                         290
```

```
    case MPSE_ACS:
    case MPSE_ACB:
    case MPSE_ACSB:
     return acsmPrintDetailInfo2( (ACSM_STRUCT2*) p->obj );
    case MPSE_MWM:
     return 0;
    case MPSE_LOWMEM:
      return 0;
    case MPSE_MBOM:
      mbomPrintDetailInfo((MBOM_STRUCT *)p->obj); break;                300
    case MPSE_MBOM2:
      mbomPrintDetailInfo2((MBOM_STRUCT2 *)p->obj); break;
    default:
      return 1;
  }

  return 0;
}

int mpsePrintSummary(void * pvoid)                                      310
{
   acsmPrintSummaryInfo();
   acsmPrintSummaryInfo2();
   mbomPrintSummaryInfo();
   mbomPrintSummaryInfo2();
   return 0;
}

int mpseSearch(void * pvoid, unsigned char * T, int n,
      int ( *action )(void*id, int index, void *data), void * data)     320
{
  MPSE * p = (MPSE*)pvoid;
  int ret;
  PROFILE_VARS;

  s_bcnt += n;

  switch( p->method )
  {
    case MPSE_AC:                                                       330
      PREPROC_PROFILE_START(mpsePerfStats);
      ret = acsmSearch( (ACSM_STRUCT*) p->obj, T, n, action, data );
      PREPROC_PROFILE_END(mpsePerfStats);
      return ret;

    case MPSE_ACF:
    case MPSE_ACS:
    case MPSE_ACB:
    case MPSE_ACSB:
      PREPROC_PROFILE_START(mpsePerfStats);                             340
      ret = acsmSearch2( (ACSM_STRUCT2*) p->obj, T, n, action, data );
```

```
      PREPROC_PROFILE_END(mpsePerfStats);
      return ret;

    case MPSE_MWM:
      PREPROC_PROFILE_START(mpsePerfStats);
      ret = mwmSearch( p->obj, T, n, action, data );
      PREPROC_PROFILE_END(mpsePerfStats);
      return ret;
                                                                    350
    case MPSE_LOWMEM:
      PREPROC_PROFILE_START(mpsePerfStats);
      ret = KTrieSearch( (KTRIE_STRUCT *)p->obj, T, n, action, data );
      PREPROC_PROFILE_END(mpsePerfStats);
      return ret;

    case MPSE_MBOM:
      PREPROC_PROFILE_START(mpsePerfStats);
      ret = mbomSearch( (MBOM_STRUCT *)p->obj, T, n, action, data );
      PREPROC_PROFILE_END(mpsePerfStats);                           360
      return ret;

    case MPSE_MBOM2:
      PREPROC_PROFILE_START(mpsePerfStats);
      ret = mbomSearch2( (MBOM_STRUCT2 *)p->obj, T, n, action, data );
      PREPROC_PROFILE_END(mpsePerfStats);
      return ret;

    case MPSE_AUTO:
      // should never happen                                        370
    default:
      //PREPROC_PROFILE_START(mpsePerfStats); // take out -JK
      return 1;

  }
}


UINT64 mpseGetPatByteCount()
{                                                                   380
  return s_bcnt;
}

void mpseResetByteCount()
{
    s_bcnt = 0;
}
```

A.2 The MBOM Option

The MBOM search method option is the Snort implementation of the Multiple Backward Oracle Matching (MultiBOM) algorithm [6] which we introduced at the end of Section 4.5. The details of the MBOM along with pseudocode for the search function are given in Section 7.2.

New Snort source file `sfutil/mbom.h`:

```
/*
**   MBOM.H (MultiBOM - or Multi Backwards Oracle Matching)
**
**   Version 1.0
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
                                                                10
#include "acsmx.h"

#ifndef MBOM_H
#define MBOM_H

#ifdef WIN32

#ifdef inline
#undef inline
#endif                                                          20

#define inline __inline

#endif

/*
*   DEFINES and Typedef's
*/
#define ALPHABET_SIZE 256
                                                                30
#define MBOM_VERBOSE 1
#define MBOM_NON_VERBOSE 0

#ifndef MBOM_EASYTYPES
#define MBOM_EASYTYPES

typedef unsigned int uint32_t;
typedef unsigned short uint16_t;
```

```
typedef unsigned char uint8_t;
                                                                          40
enum {
  MBOM_ORACLE, // keep first (0) entry default
  MBOM_DAWG,
};

#endif


/* A state/node in a DAWG/Oracle/Trie */
                                                                          50
typedef struct _mbom_node {
  /* allocate 256 pointers to other nodes for constant time branching in the automaton */

  struct _mbom_node * next_states[ALPHABET_SIZE];
  struct _mbom_node * supply_state;

  /* 256 bit table where 1s = extended transition at same index in next_states */
  /* If next_states[i] != NULL and 0 == bit i in the table is 0 then this node
   * owns the node at next_states[i] and is responsible for deleting it */
                                                                          60
  uint8_t          extendedTransitions[ALPHABET_SIZE / 8];
  uint16_t id;

} MBOM_NODE; /* SIZE: 1038 B (actual memory consumption depends on compiler settings) */


/*
*   MultiBOM Matcher Struct - one per group of pattterns
*/
typedef struct {                                                          70

        ACSM_STRUCT * acsm;     /* an Aho-Corasick Std state machine */
        MBOM_NODE * initialState; /* root node */
        uint32_t      mbomSize;      /* number of states/nodes */
        uint32_t      mbomNumTrans; /* number of transitions */
        uint32_t      mbomNumPatterns; /* number of patterns in the list */
        uint8_t       mbomFormat;   /* the automaton format either an Oracle or a DAWG */
        uint16_t      minLen;         /* length of the shortest pattern */
        uint16_t      matches;
                                                                          80
}MBOM_STRUCT;

/*
*   Prototypes
*/

MBOM_STRUCT * mbomNew();

int  mbomAddPattern(MBOM_STRUCT * mbom, unsigned char * pat, int n,
```

```
                 int nocase, int offset, int depth, void * id, int iid);            90

int  mbomCompile(MBOM_STRUCT * mbom);

int  mbomSearch(MBOM_STRUCT * mbom, unsigned char * T, int n,
                 int (*Match)(void * id, int index, void * data), void * data);

void mbomFree(MBOM_STRUCT * mbom);

int  mbomSelectFormat(MBOM_STRUCT * mbom, int format);
                                                                                    100
void mbomSetVerbose(int n);

void mbomPrintDetailInfo(MBOM_STRUCT * mbom);

void mbomPrintSummaryInfo();

#endif
```

New Snort source file `sfutil/mbom.c`:

```
/*
**  $Id$
**
**  mbom.c
**
**  Multi-Pattern Search Engine
**
**  MultiBOM - or Multi Backwards Oracle Matching
**
**  Version 1.0                                                                     10
**
**  Copyright (C) 2006 James Kelly jamesjameskelly.net
**
**  Reference: (Original MultiBOM proposal) – IN FRENCH
**  C. Allauzen and M. Raffinot. Oracle des facteurs d'un ensemble de mots.
**  Technical Report IGM 99-11, Institut Gaspard Monge, Universite de
**  Marne-la-Vallee, France, 1999.
**
**  Reference: (BEST REFERENCE FOR SBOM and how to build a factor oracle)
**  G. Navarro and M. Raffinot. Flexible Pattern Matching in Strings,               20
**  Practical On-line Search Algorithms for Texts and Biological Sequences.
**  Cambridge University Press, Cambridge, UK, 2002
**
**  Reference: (BEST REFERENCE FOR MultiBDM)
**  M. Crochemore and W. Rytter. Text Algorithms. Oxford University Press, 1994.
**  Pages 140-143 -Example in book has a mistake in it; one pattern is not matched-
**
**  Reference:
**  M. Raffinot. On the multi backward DAWG matching algorithm (MultiBDM). In
```

```
**  R. Baeza-Yates, editor, WSP'97: Proceedings of the 4th South American Work-     30
**  shop on String Processing, pages 149{165, Valparaiso, Chile, Nov. 1997.
**  Carleton University Press.
**
**  Version 1.0 Notes - James Kelly:
**
**  1) Finds all occurrences of all patterns within a text.
**
**  2) Currently supports only the use of a factor oracle; however, MultiDAWG
**     uses the same approach with a DAWG (Directed Acyclic Word Graph)
**                                                                                  40
**  3) MBOM is an implementation of MultiBOM from first reference. It
**     is for use in Snort and uses Snort's standard version of its
**     Aho-Corasick state machine (acsmx.h/c).
**
**  4) MBOM doesn't take much extra memory compared to Snort's standard
**     Aho-Corasick state machine pattern matcher; however, the running time
**     will greatly be *enhanced* (faster) because MBOM is average case
**     (and worst case) optimal. That is, it's sublinear (wrt text length)
**     on average and linear (wrt text length) in the worst case. The
**     average case is defined as only indepedent equiprobable characters           50
**     appearing in the search text. The MBOM algorithm executes at most
**     2n inspections of search text characters where the search text
**     length is n.
**
**  5) MBOM uses a window size of length equal to the minimum length
**     pattern. Therefore, shifts are limited by this window size.
**     Thus, it is not/hardly worth using the MBOM algorithm unless
**     the minimum length pattern is at least of length 3. Note that
**     for those cases the Aho-Corasick algorithm would be faster.
**                                                                                  60
**
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>


#include "mbom.h"                                                                   70

//#define DEBUG_MBOM

/*
* facilitates: memory checks
*/
#define MEMASSERT(p,s) if(!p){printf("MBOM-No Memory: %s!\n",s);exit(0);}

/*
* Keep this for stats:                                                              80
```

```
*/
static int max_memory = 0;

/*
* toggle verbose for all instances of MBOM
*/
static int s_verbose = MBOM_NON_VERBOSE;

/*
* Keep this summary for stats:                                    90
*/
typedef struct mbom_summary_s
{
      unsigned num_states;
      unsigned num_transitions;
      unsigned num_patterns;
      unsigned num_groups;

}mbom_summary_t;
                                                                  100
static mbom_summary_t summary={0,0,0,0};


/* QUEUE STUFF:
*  A Queue is needed to do a breadth-first traversal
*  of our trie to make a factor oracle.
*/

/*
*    Simple QUEUE NODE                                            110
*/
typedef struct _qnode
{
  MBOM_NODE * node; //data
  uint8_t     character; //data
  MBOM_NODE * parent; //data
  struct _qnode *next;
}
QNODE;
                                                                  120
/*
*    Simple QUEUE Structure
*/
typedef struct _queue
{
  QNODE * head, *tail;
  int count;
}
QUEUE;
                                                                  130
/*
```

```
*   Initialize the queue
*/
static void
queue_init (QUEUE * s)
{
  s->head = s->tail = 0;
  s->count= 0;
}
                                                                        140
/*
*  Add Tail Item to queue (FiFo/LiLo)
*/
static void queue_add (QUEUE * s, MBOM_NODE * node,
                        MBOM_NODE * parent, uint8_t character)
{
  QNODE * q;

  if (!s->head)
  {                                                                     150
      // don't count this in summary it will be deleted (it's tmp only)
      // a queue is never kept during the search only used for
      // precomputation/preprocessing purposes.
      q = s->tail = s->head = (QNODE *) malloc (sizeof (QNODE));
      MEMASSERT (q, "queue_add");
      q->node      = node;
      q->parent    = parent;
      q->character = character;
      q->next = NULL;
  }                                                                     160
  else
  {
      q = (QNODE *) malloc (sizeof (QNODE)); //don't count this in summary
      q->node      = node;
      q->parent    = parent;
      q->character = character;
      q->next = NULL;
      s->tail->next = q;
      s->tail = q;
  }                                                                     170
  s->count++;
}


/*
*  Remove Head Item from queue
*/
static void queue_remove (QUEUE * s, MBOM_NODE ** node,
                           MBOM_NODE ** parent, uint8_t * character)
{                                                                       180
  void * data;
  QNODE * q;
```

```
  data = NULL;

  if (s->head)
  {
     q      = s->head;
     *node    = q->node;
     *parent  = q->parent;                                  190
     *character = q->character;
     s->head = s->head->next;
     s->count--;

     if( !s->head )
     {
        s->tail = NULL;
        s->count = 0;
     }
     free (q);                                              200
  }
}

/*
*  Return items in the queue
*/
static int
queue_count (QUEUE * s)
{
  return s->count;                                          210
}


/*
* Free the queue
*/
static void
queue_free (QUEUE * s)
{
  QNODE * q;                                                220

  while (s->head)
    {
     q = s->head;
     s->head = q->next;
     s->count--;
     free (q);

     if( !s->head ) {
       s->tail = NULL;                                      230
       s->count = 0;
     }
    }
```

```
}


/* MBOM STUFF */

/*
* Case Translation Table                                             240
*/
static unsigned char xlatcase[256];

/*
* Init Case Translation Table
*/
static void init_xlatcase()
{
  int i;
  for (i = 0; i < 256; i++)                                          250
    {
      xlatcase[i] = toupper(i);
    }
}

/*
* measure memory allocations
*/
static void * MBOM_MALLOC (uint32_t size)
{                                                                    260
  void * p;
  p = malloc (size);
  if (p) {
    max_memory += size;
  }
  return p;
}

/*
* measure memory deallocations                                       270
*/
static void MBOM_FREE (void * p, uint32_t size)
{
  if (p) {
    free (p);
    max_memory -= size;
  }
}

/*                                                                   280
* toggle between verbose mode on/off with 1/0
*/
void mbomSetVerbose(int n)
{
```

```
  s_verbose = n;
}

/*
*  Select the desired storage mode
*/                                                                          290
int mbomSelectFormat(MBOM_STRUCT * mbom, int format)
{
  switch( format )
  {
    case MBOM_ORACLE:
    case MBOM_DAWG:
      mbom->mbomFormat = MBOM_ORACLE; // only support this currently
      break;
    default:
      return -1; //doesn't even matter right now (only one version coded)   300
  }

  return 0;
}

/*
*  Create a new MultiBOM Matcher struct
*/
MBOM_STRUCT * mbomNew()
{                                                                           310
  MBOM_STRUCT * p;

  init_xlatcase();

  p = (MBOM_STRUCT *) MBOM_MALLOC(sizeof (MBOM_STRUCT));
  MEMASSERT (p, "mbomNew");
  memset (p, 0, sizeof (MBOM_STRUCT));

  p->acsm = acsmNew();
  MEMASSERT (p->acsm, "mbomNew (acsm)");                                    320

  ++(summary.num_groups);

  return p;
}

/*
*  Add a pattern to the list of patterns for this instance
*/
int mbomAddPattern(MBOM_STRUCT * mbom, unsigned char * pat, int n, int nocase,   330
                int offset, int depth, void * id, int iid)
{
  if(n <= 0) {
    printf("Illegal pattern length found of: %d\n", n);
    exit(0);
```

```
  }

  if(mbom->minLen == 0 || mbom->minLen > n) {
    mbom->minLen = n; // keep track of the length of the shortest pattern
  }                                                                           340

  acsmAddPattern(mbom->acsm, pat, n, nocase, offset, depth, id, iid);
  ++(mbom->mbomNumPatterns);
  ++(summary.num_patterns);
  return 0;
}

#ifdef DEBUG_MBOM
/*
*   Prints out the factor oracle structure:                                   350
*
*   This prints out the F.O. the same way as v2.0
*   for comparison purposes (they should be the same)
*/
static void printMbom(MBOM_STRUCT * mbom)
{
  QUEUE q;
  int j;
  MBOM_NODE * current, * parent;
  uint8_t currentCharacter;                                                   360

  printf("\nMBOM structure:\n");

  /* use queue to facilitate a breadth first traversal over the node/states
   * of the trie to print them */
  queue_init(&q);

  for(j = 0; j < ALPHABET_SIZE; ++j) {
    if(mbom->initialState->next_states[j] != NULL) {
      // enqueue the node itself, its parent, and the character between them on the transition   370
      queue_add(&q, mbom->initialState->next_states[j], mbom->initialState, j);
      printf("t(%d,%x) = %d\n", mbom->initialState->id, j,
        mbom->initialState->next_states[j]->id);
    }
  }

  while(queue_count(&q)) {

    queue_remove(&q, &current, &parent, &currentCharacter);
                                                                              380
    /* Enqueue all children nodes of current */
    for(j = 0; j < ALPHABET_SIZE; ++j) {
      if(current->next_states[j] != NULL) {
        // enqueue the node itself, its parent, and the character between them on the transition
        queue_add(&q, current->next_states[j], current, j);
            printf("t(%d,%x) = %d\n", current->id, j, current->next_states[j]->id);
```

```
      }
    }
  }
                                                                     390
  queue_free(&q);
  printf("\n");
}
#endif

/*
* Helper used by the function below it (mbomCompile)
*/
static MBOM_NODE * newMbomState()
{                                                                    400
  MBOM_NODE * node;

  node = (MBOM_NODE *) MBOM_MALLOC(sizeof(MBOM_NODE));
  MEMASSERT (node, "newMbomState");
  memset(node, 0, sizeof(MBOM_NODE)); // zero values are defaults and all pointers are NULL

  return node;
}

/*                                                                   410
 * Helper fnc used by mbomCompile
 * Sets a bit to 1 in the table
 */
static void setBit(uint8_t * bitTable, uint8_t index)
{
  uint8_t j = 0;
  uint8_t bitPos = index % 8; // is the place of the bit to set in the (index/8)th byte
  uint8_t bitMask = 1;

  for(j = 0; j < bitPos; ++j) {                                      420
    bitMask <<= 1;
  }

  bitTable[index / 8] |= bitMask; // set bit
}

/*
 * Helper fnc used by deleteMbomNode and in turn mbomFree
 * Returns 1 if bit is set to 1 in table otherwise 0
 */                                                                  430
static uint8_t getBit(uint8_t * bitTable, uint8_t index)
{
  uint8_t j = 0;
  uint8_t bitPos = index % 8; // is the place of the bit to set in the (index/8)th byte
  uint8_t bitMask = 1;

  for(j = 0; j < bitPos; ++j) {
```

```
    bitMask <<= 1;
  }
                                                                        440
  if((bitTable[index / 8] & bitMask) == 0) { // get bit
    return 0;
  }
  return 1;
}

/*
*   Compile (Construct) the automaton to be used for this pattern matcher
*
*   Currently this function always builds a factor oracle,              450
*   but a DAWG could also be used
*
*   The resulting factor oracle recognizes at least all of the factors
*   of the pattern set P. It's construction time should be O(|P|) (linear).
*
*   For instructions on how to build this see the algorithm references/notes above
*/
int mbomCompile(MBOM_STRUCT * mbom)
{
  int           j;                                                      460
  ACSM_PATTERN * plist;
  MBOM_NODE * current, * new, * parent;
  uint8_t       currentCharacter;
  QUEUE         q; // temp for Breadth-First Traversal

  /* Create Trie: */
  /* ———— */

  mbom->initialState = newMbomState(); // Initial State
  ++(mbom->mbomSize);                                                   470
  mbom->initialState->id = mbom->mbomSize;

  for (plist = mbom->acsm->acsmPatterns; plist != NULL; plist = plist->next) {
    current = mbom->initialState;
    j = plist->n - 1; //start at the end of the patttern because we're entering it reversed

    while(j >= 0 && current->next_states[plist->patrn[j]] != NULL) {
      current = current->next_states[plist->patrn[j]];
      --j;
    }                                                                   480

    while(j >= 0) {
      current = (current->next_states[plist->patrn[j]] = newMbomState());
      --j;
      ++(mbom->mbomNumTrans); // Add Transition
      ++(mbom->mbomSize); // Add State
      current->id = mbom->mbomSize;
    }
```

```
}
                                                                            490

/* Build Factor Oracle From Trie: */
/* ------------------------------ */

/* We need to create external transitions with a breadth first traversal */

// use queue to facilitate a breadth first traversal over the node/states
// of the trie to make the factor oracle
queue_init(&q);
                                                                            500
for(j = 0; j < ALPHABET_SIZE; ++j) {
  if(mbom->initialState->next_states[j] != NULL) {
    // enqueue the node itself, its parent
    // and the character between them on the transition
    queue_add(&q, mbom->initialState->next_states[j], mbom->initialState, j);
  }
}

while(queue_count(&q)) {
                                                                            510
  queue_remove(&q, &current, &parent, &currentCharacter);

  /* Process current node */
  // new moves ("up") towards the root/initialState

  new = parent->supply_state;

  while(new != NULL && new->next_states[currentCharacter] == NULL) {

    // Add an external transition                                           520
    new->next_states[currentCharacter] = current;
    ++(mbom->mbomNumTrans); // Add Transition

    // set bit in bit table to indicate this is an extended transition
    setBit(new->extendedTransitions, currentCharacter);

    new = new->supply_state;
  }

  if(new != NULL) {                                                         530
    current->supply_state = new->next_states[currentCharacter];
  }
  else {
    current->supply_state = mbom->initialState;
  }

  /* Enqueue all children nodes of current */
  for(j = 0; j < ALPHABET_SIZE; ++j) {
    if(current->next_states[j] != NULL) {
```

```
      // enqueue the node itself, its parent                                    540
      // and the character between them on the transition
      queue_add(&q, current->next_states[j], current, j);
    }
  }
}

queue_free(&q);

/* Tell the ACSM to compile itself too */
/* ---------------------------------- */                                        550
acsmCompile(mbom->acsm);

/* Accrue Summary State Stats */
summary.num_states += mbom->mbomSize;
summary.num_transitions += mbom->mbomNumTrans;

#ifdef DEBUG_MBOM
  printMbom(mbom);
  mbomPrintDetailInfo(mbom);
#endif                                                                          560
  return 0;
}


/** Tc is declared once outside of this function is a pointer
 ** into all converted uppercase text characters/bytes
 **/
#define MBOM_MAX_TEXT 65536
static unsigned char Tc[MBOM_MAX_TEXT]; // should be more than enough space for snort
                                                                                570
/*
*  Search Function
*/
int mbomSearch(MBOM_STRUCT * mbom, unsigned char *Tx, int n,
         int (*Match) (void * id, int index, void *data),
         void *data)
{
  int nfound   = 0; /* num of patterns found */
  int min      = mbom->minLen; // minimal length of patterns (also the window size)
  int i        = 0; // i is the position of the window on the text             580
  int critpos = 0; // position of the input head of the ACSM
  int j        = 0; // tmp (may go to -1 tmply)
  int end      = n - min + 1; // last valid i + 1
  int windowEnd = min - 1;
  MBOM_NODE * current = NULL;

  int state        = 0; /* ACSM current state*/
  ACSM_PATTERN    * mlist; /* tmp list of patterns at a terminal state */
  ACSM_STATETABLE * states = mbom->acsm->acsmStateTable;
                                                                                590
```

```
// Tc is declared once outside of this function is a pointer
// into all converted uppercase text characters/bytes

if(n > MBOM_MAX_TEXT) {
  printf("mbom Search unperformed because text was too long");
  exit(0);
}

// Case conversion of text
for (j = 0; j < n; ++j) {                                                    600
  Tc[j] = xlatcase[ Tx[j] ];
}

while(i < end && critpos < n) {

  // Here's the ACSM has scanned up to but not including Tc[critpos]
  // We scan with the oracle back to and including Tc[critpos]

  j = i + windowEnd; // last char in window
  current = mbom->initialState;                                              610

  // Search for factor mismatch in the oracle/dawg:

  while(j >= critpos && (current = current->next_states[Tc[j]]) != NULL) {
      --j;
  }

  if(j >= critpos) { //if it didn't make it all the way to the critpos
    state = 0; // reset ACSM
    critpos = j + 1;                                                         620
  }

  // Search with ACSM between indexes critpos "up to" n-1:

  while(critpos < n && (critpos < i + min || states[state].depth >= min)) {

    state = states[state].NextState[Tc[critpos]]; // scan one character
    ++critpos;

    if(states[state].MatchList != NULL) { // if this state is terminal      630

      /* Go through the patterns that match at this state */

      for(mlist=states[state].MatchList; mlist != NULL; mlist = mlist->next) {

        /* j = location that match starts in Tx */
        j = critpos - (uint16_t)mlist->n;

        /* obviously faster for patterns that are case insensitive */
        if(mlist->nocase) {                                                  640
          ++nfound; ++(mbom->matches);
```

```
          if(Match (mlist->id, j, data))
            return nfound;
        }
        else {
          if(memcmp(mlist->casepatrn, Tx + j, mlist->n) == 0) {
            ++nfound; ++(mbom->matches);
            if(Match (mlist->id, j, data))
              return nfound;
                                                                        650
          }
        }

      } //end for
    } //end if
  } //end while

  /* shift by critpos - length of longest prefix matched */
  i = critpos - states[state].depth; // SHIFT WINDOW
 }                                                                      660
 return nfound;
}

/*
 * Helper fnc used by mbomFree
 * Free node (recursive helper)
 */
static void deleteMbomNode(MBOM_NODE * node)
{
 int j;                                                                 670

 if(node == NULL) {
   return;
 }

 /* delete all children */
 for(j = 0; j < ALPHABET_SIZE; ++j) {
   // check that it is not an extended transition
   // if it is then it "belongs" to another MBOM_NODE
   if(node->next_states[j] != NULL && (getBit(node->extendedTransitions, j) == 0)) {  680
     deleteMbomNode(node->next_states[j]);
   }
 }

 /* delete node */
 MBOM_FREE(node, sizeof (MBOM_NODE));
}

/*
*  Free all memory                                                      690
*/
void mbomFree(MBOM_STRUCT * mbom)
```

```
{
  deleteMbomNode(mbom->initialState); // deletes all states and transitions

  acsmFree(mbom->acsm); // deletes the ACSM

  MBOM_FREE(mbom, sizeof (MBOM_STRUCT));

  --(summary.num_groups);                                                    700
}

static int ins_num = 0;


/*
*  Prints information about a mbom matcher instance
*/
void mbomPrintDetailInfo(MBOM_STRUCT * mbom)
{                                                                            710
    char * sf[]= {"Factor Oracle", "DAWG (Directed Acyclic Word Graph)"};

    printf("+--[Pattern Matcher:Multi Backward Oracle Matching (MultiBOM)");
    printf(" Instance Info]------\n");
    printf("| Instance Number : %u\n", ++ins_num);
    printf("| Alphabet Size : %u Chars\n", ALPHABET_SIZE);
    printf("| Size of State : %u bytes\n", (int)(sizeof(MBOM_NODE)));
    printf("| Storage Format : %s\n", sf[mbom->mbomFormat]);
    printf("| Shortest Pat Len : %u\n", (unsigned int)mbom->minLen);
    printf("| Num States   : %u\n", (unsigned int)mbom->mbomSize);              720
    printf("| Num Transitions : %u\n", (unsigned int)mbom->mbomNumTrans);
    printf("| Num Patterns : %u\n", (unsigned int)mbom->mbomNumPatterns);
    printf("| State Density : %.1f%%\n",
      100.0*(double)mbom->mbomNumTrans/(mbom->mbomSize * ALPHABET_SIZE));
    printf("| All MBOMs' Memory: %.2fKbytes\n", (float)max_memory/1024 );
    printf("+--------------------------------");
    printf("----------------------------------");
    printf("----------------------\n\n");
    printf("+--------------------");
    printf(" AHO-CORASICK STATE MACHINE INFO FOLLOWS: ");                      730
    printf("----------------------\n\n");

    acsmPrintDetailInfo(mbom->acsm);
}

/*
 *  Global sumary of all mbom info and all state machines built during this run
 */
void mbomPrintSummaryInfo()
{                                                                            740
  // this IF is for mpsePrintSummary (which doesn't check which method is in use)
  if(summary.num_states > 0) {
    printf("+--[Pattern Matcher:Multi Backward Oracle Matching (MultiBOM)");
```

```
    printf(" Overall Summary]−−−−\n");
    printf("| Alphabet Size : %u Chars\n",ALPHABET_SIZE);
    printf("| Size of State  : %u bytes\n",(int)(sizeof(MBOM_NODE)));
    printf("| Num States    : %u\n",summary.num_states);
    printf("| Num Transitions : %u\n",summary.num_transitions);
    printf("| Num Groups  : %u\n",summary.num_groups);
    printf("| Num Patterns : %u\n",summary.num_patterns);                      750
    printf("| State Density : %.1f%%\n",
      100.0*(double)summary.num_transitions / (summary.num_states * ALPHABET_SIZE));
    printf("| Memory Usage : %.2fKbytes\n", (float)max_memory/1024 );
    printf("+−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−");
    printf("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−");
    printf("−−−−−−−−−−−−−−−−−−−−\n\n");
    printf("+−−−−−−−−−−−−−−−−−−");
    printf(" AHO−CORASICK STATE MACHINE SUMMARY FOLLOWS: ");
    printf("−−−−−−−−−−−−−−−−−−−−\n\n");
                                                                               760
    acsmPrintSummaryInfo();
  }
}



//#define MBOM_MAIN

#ifdef MBOM_MAIN
                                                                               770
#include <time.h>

/*
*  Text Data Buffer
*/
unsigned char text[2048];

/*
*   A Match is found
*/                                                                             780
int MatchFound (void* id, int index, void *data)
{
  printf("MATCH:%s\n", (char *) id);
  return 0;
}

/*
* MAIN (for testing purposes)
*/
int main (int argc, char **argv)                                               790
{
  int i, nc, nocase = 0;
  MBOM_STRUCT * mbom;
  char * p;
```

```
clock_t start, stop;

if (argc < 3) {
  fprintf (stderr,"\nUsage: %s search−text pattern +pattern... [flags]\n",argv[0]);
  fprintf (stderr," flags: −nocase −verbose\n");
  fprintf (stderr," use a + in front of pattern for single case insensitive pattern\n\n");   800
  exit (0);
}

mbom = mbomNew();

if(!mbom) {
  printf("mbom−no memory\n");
  fflush(stdout);
  exit(0);
}                                                                                          810

if(s_verbose) {
  printf("Parsing Parameters...\n");
  fflush(stdout);
}

strcpy (text, argv[1]);

for(i = 1; i < argc; ++i) {
                                                                                           820
  if(strcmp(argv[i], "−nocase") == 0) {
    nocase = 1;
  }
  if(strcmp (argv[i], "−verbose") == 0) {
    s_verbose = MBOM_VERBOSE;
  }
}

for (i = 2; i < argc; ++i) {
    if (argv[i][0] == '-') /* a switch */                                                  830
      continue;

    p = argv[i];

    if ( *p == '+') {
        nc=1;
        ++p;
    }
    else {
        nc = nocase;                                                                       840
    }

    mbomAddPattern(mbom, p, strlen(p), nc, 0, 0, (void*)p, i - 2);
}
```

```
  if(s_verbose) { printf("Patterns added\n"); fflush(stdout); }

  start = clock();
  mbomCompile(mbom);
  stop = clock();                                                          850

  if(s_verbose) {
    printf("Patterns compiled in (%f seconds)\n", ((double)(stop-start))/CLOCKS_PER_SEC);
    fflush(stdout);
    mbomPrintDetailInfo(mbom);
    printf("\n");
    mbomPrintSummaryInfo();
    printf("\nSearching text...\n");
    fflush(stdout);
  }                                                                        860

  start = clock();
  mbomSearch(mbom, text, strlen(text), MatchFound, (void *)0 );
  stop = clock();

  if(s_verbose)
   printf ("Done search in (%f seconds)\n", ((double)(stop-start))/CLOCKS_PER_SEC);

  mbomFree(mbom);
                                                                           870
  if(s_verbose) printf ("Done cleaning\n");

  return 0;
}
#endif /* include main program */
```

A.3 The MBOM2 Option

The MBOM2 search method option is the implementation of the same algorithms as in the MBOM option (see Section A.2); however, using a hashtable as discussed in Section 7.2. The implementation of the standard hashtable is not documented here. It is in entirely new files included with the source code package available from `ftp://jameskelly.net/mcs/`.

New Snort source file `sfutil/mbom2.h`:

```
/*
**   MBOM.H (MultiBOM - or Multi Backwards Oracle Matching)
```

```
**
**   Version 2.0
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
                                                                    10
#include "acsmx.h"
#include "hashtable.h"

#ifndef MBOM2_H
#define MBOM2_H

#ifdef WIN32

#ifdef inline
#undef inline                                                       20
#endif

#define inline __inline

#endif

/*
*   DEFINES and Typedef's
*/
#define ALPHABET_SIZE 256                                           30

#define MBOM_ROOT 1

#define MBOM_VERBOSE 1
#define MBOM_NON_VERBOSE 0

#ifndef MBOM_EASYTYPES
#define MBOM_EASYTYPES

typedef unsigned int uint32_t;                                      40
typedef unsigned short uint16_t;
typedef unsigned char uint8_t;

enum {
  MBOM_ORACLE, // keep first (0) entry default
  MBOM_DAWG,
};

#endif
                                                                    50
typedef struct hashtable HASHTABLE;

typedef uint16_t MBOM_STATE;
```

```
/* A state/node in a DAWG/Oracle/Trie */

typedef struct {
  MBOM_STATE from_state;
  uint8_t  character;
} MBOM_KEY;                                                                60

// MBOM_VALUE is just a MBOM_STATE as next_state

/*
*   MultiBOM Matcher Struct - one per group of pattterns
*/

typedef struct {
  /* instead of allocating 256 pointers to other nodes for constant time
   * branching in the automaton we use a hashtable */                      70

  HASHTABLE * transitions;
  ACSM_STRUCT * acsm;  /* an Aho-Corasick Std state machine */

  uint32_t    mbomSize;       /* number of states/nodes */
  uint32_t    mbomNumTrans; /* number of transitions */
  uint32_t    mbomNumPatterns; /* number of patterns in the list */
  uint8_t     mbomFormat;   /* the automaton format either an Oracle or a DAWG */
  uint16_t    minLen;         /* length of the shortest pattern */
                                                                           80
}MBOM_STRUCT2;

/*
*   Prototypes
*/

MBOM_STRUCT2 * mbomNew2();

int  mbomAddPattern2(MBOM_STRUCT2 * mbom, unsigned char * pat, int n,
                  int nocase, int offset, int depth, void * id, int iid);  90

int  mbomCompile2(MBOM_STRUCT2 * mbom);

int  mbomSearch2(MBOM_STRUCT2 * mbom, unsigned char * T, int n,
                  int (*Match)(void * id, int index, void * data), void * data);

void mbomFree2(MBOM_STRUCT2 * mbom);

int  mbomSelectFormat2(MBOM_STRUCT2 * mbom, int format);
                                                                           100
void mbomSetVerbose2(int n);

void mbomPrintDetailInfo2(MBOM_STRUCT2 * mbom);
```

```
void mbomPrintSummaryInfo2();

// make these available for the hashtable memory tracking:

void * MBOM_MALLOC2(uint32_t size);
                                                                                110
void * MBOM_REALLOC2(void * p, uint32_t new_size, uint32_t difference);

void  MBOM_FREE2(void * p, uint32_t size);

#endif
```

New Snort source file sfutil/mbom2.c:

```
/*
**   $Id$
**
**   mbom2.c
**
**   Multi-Pattern Search Engine
**
**   MultiBOM - or Multi Backwards Oracle Matching
**
**   Version 2.0                                                                 10
**
**   Copyright (C) 2006 James Kelly jamesjameskelly.net
**
**   Reference: (Original MultiBOM proposal) – IN FRENCH
**   C. Allauzen and M. Raffinot. Oracle des facteurs d'un ensemble de mots.
**   Technical Report IGM 99-11, Institut Gaspard Monge, Universite de
**   Marne-la-Vallee, France, 1999.
**
**   Reference: (BEST REFERENCE FOR SBOM and how to build a factor oracle)
**   G. Navarro and M. Raffinot. Flexible Pattern Matching in Strings,          20
**   Practical On-line Search Algorithms for Texts and Biological Sequences.
**   Cambridge University Press, Cambridge, UK, 2002
**
**   Reference: (BEST REFERENCE FOR MultiBDM)
**   M. Crochemore and W. Rytter. Text Algorithms. Oxford University Press, 1994.
**   Pages 140-143 *Example in book has a mistake in it; one pattern is not matched*
**
**   Reference:
**   M. Raffinot. On the multi backward DAWG matching algorithm (MultiBDM). In
**   R. Baeza-Yates, editor, WSP'97: Proceedings of the 4th South American Work–   30
**   shop on String Processing, pages 149{165, Valparaiso, Chile, Nov. 1997.
**   Carleton University Press.
**
**   Version 1.0 Notes – James Kelly:
**
**   1) Finds all occurrences of all patterns within a text.
```

```
**
** 2) Currently supports only the use of a factor oracle; however, MultiDAWG
**    uses the same approach with a DAWG (Directed Acyclic Word Graph)
**                                                                          40
** 3) MBOM is an implementation of MultiBOM from first reference. It
**    is for use in Snort and uses Snort's standard version of its
**    Aho-Corasick state machine (acsmx.h/c).
**
** 4) MBOM doesn't take much extra memory compared to Snort's standard
**    Aho-Corasick state machine pattern matcher; however, the running time
**    will greatly be *enhanced* (faster) because MBOM is average case
**    and worst case optimal. That is, it's sublinear (wrt text length)
**    on average and linear (wrt text length) in the worst case. The
**    average case is defined as only indepedent equiprobable characters    50
**    appearing in the search text. The MBOM algorithm executes at most
**    2n inspections of search text characters where the search text
**    length is n.
**
** 5) MBOM uses a window size of length equal to the minimum length
**    pattern. Therefore, shifts are limited by this window size.
**    Thus, it is not/hardly worth using the MBOM algorithm unless
**    the minimum length pattern is at least of length 3. Note that
**    for those cases the Aho-Corasick algorithm would be faster.
**                                                                          60
** New Version 2.0 Notes - James Kelly:
**
** 1) This version uses a hashtable and there is no trie or nodes. It is
**    all virtual in the hashtable which of course saves a lot (tons) of
**    memory. For comparison for the Snort default rule DB MBOM v1.0
**    would take 14331.21 KB of memory + 157366.49 KB for the Aho-Corasick
**    State Machine - ACSM), but MBOM v2.0 takes 548.70Kbytes + the same
**    for the ACSM. In the memory usage of the factor oracle there's a
**    difference of 26:1 (ratio)!
**                                                                          70
** 2) Still only supports only the use of a factor oracle; however, MultiDAWG
**    uses the same approach with a DAWG (Directed Acyclic Word Graph).
**
** 3) States in the factor oracle are represented by a uint16_t therefore we
**    are limited to 2^16 states. That should be plenty considering the factor
**    oracle's depth is cut off at the length of the shortest pattern. It should
**    be easy to change it to a uint32_t if needed, but of course this will
**    increase memory cost per state as well.
**
** 4) The hashtable holds a state id and character as a key, and another state  80
**    id as the value. The character is the label on the transition between the
**    two states.
**
** 5) MBOM v1.0 stored the supply state in the NODE which meant it was kept around
**    after pre-computation, but it actually isn't needed. In this version the
**    memory to hold the supply function (supply states) is only allocated during
**    precomputation (the compile routine). Before the search phase it is deleted.
```

```
**
**
*/                                                                      90

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "mbom2.h"

//#define DEBUG_MBOM2
                                                                        100
/*
* facilitates: memory checks
*/
#define MEMASSERT(p,s) if(!p){printf("MBOM-No Memory: %s!\n",s);exit(0);}

/*
* Keep this for stats:
*/
static int max_memory = 0;
                                                                        110
/*
* toggle verbose for all instances of MBOM2
*/
static int s_verbose = MBOM_NON_VERBOSE;

/*
* Keep this summary for stats:
*/
typedef struct mbom_summary_s
{                                                                       120
     unsigned  num_states;
     unsigned  num_transitions;
     unsigned  num_patterns;
     unsigned  num_groups;

} mbom_summary_t;

static mbom_summary_t summary={0,0,0,0};

                                                                        130
/* QUEUE STUFF:
*  A Queue is needed to do a breadth-first traversal
*  of our trie to make a factor oracle.
*/

/*
*   Simple QUEUE NODE
*/
```

```
typedef struct _qnode
{                                                                    140
  MBOM_STATE * state; //data
  MBOM_KEY * parent; //data
  struct _qnode *next;
} QNODE;

/*
*   Simple QUEUE Structure
*/
typedef struct _queue
{                                                                    150
  QNODE * head, *tail;
  int count;
} QUEUE;

/*
*  Initialize the queue
*/
static void
queue_init (QUEUE * s)
{                                                                    160
  s->head = s->tail = 0;
  s->count= 0;
}

/*
*  Add Tail Item to queue (FiFo/LiLo)
*/
static void queue_add (QUEUE * s, MBOM_STATE * state, MBOM_KEY * parent)
{
  QNODE * q;                                                         170

  if (!s->head)
  {
     // don't count this in summary it will be deleted (it's tmp only)
     // a queue is never kept during the search only used for
     // precomputation/preprocessing purposes.
     q = s->tail = s->head = (QNODE *)malloc(sizeof (QNODE));
     MEMASSERT (q, "queue_add");
     q->state   = state;
     q->parent  = parent;                                            180
     q->next    = NULL;
  }
  else
  {
     q = (QNODE *) malloc (sizeof (QNODE)); //don't count this in summary
     q->state     = state;
     q->parent    = parent;
     q->next      = NULL;
     s->tail->next = q;
```

```
    s->tail       = q;                                              190
  }
  ++(s->count);
}


/*
 *  Remove Head Item from queue
 */
static void queue_remove (QUEUE * s, MBOM_STATE ** state, MBOM_KEY ** parent)
{                                                                   200
  void * data;
  QNODE * q;

  data = NULL;

  if (s->head)
  {
    q       = s->head;
    *state  = q->state;
    *parent = q->parent;                                            210
    s->head = s->head->next;
    --(s->count);

    if( !s->head )
    {
      s->tail = NULL;
      s->count = 0;
    }
    free (q);
  }                                                                 220
}

/*
 *   Return # of items in the queue
 */
static int
queue_count (QUEUE * s)
{
  return s->count;
}                                                                   230


/*
 *  Free the queue
 */
static void
queue_free (QUEUE * s)
{
  QNODE * q;
                                                                    240
```

```
  while (s->head)
    {
      q = s->head;
      s->head = q->next;
      s->count--;
      free (q);

      if( !s->head ) {
        s->tail = NULL;
        s->count = 0;                                                        250
      }
    }
}

/** HASHTABLE STUFF **/

DEFINE_HASHTABLE_INSERT(insert_node, MBOM_KEY, MBOM_STATE);
DEFINE_HASHTABLE_SEARCH(get_node, MBOM_KEY, MBOM_STATE);
DEFINE_HASHTABLE_REMOVE(remove_node, MBOM_KEY, MBOM_STATE);
                                                                             260
/*
* Returns the hash value of a key:
*/

static unsigned int
hashFromKey(void *ky)
{
  MBOM_KEY * k = (MBOM_KEY *)ky;
  //uint32_t i = k->from_state;
  //uint32_t j = k->character;                                               270

  //return (i ^ j); // (fastest)
  return (((k->from_state << 7) | (k->from_state >> 5)) ^ k->character); // (faster)
  //return (((i << 7) | (i >> 5)) ^ j) + (i * 17) + (j * 4901); //4901 = 13*29 (good)
}

/*
*  Check equality between two keys:
*
*  DO NOT USE return (0 == memcmp(k1, k2, sizeof(MBOM_KEY))) in this function   280
*  unless you have checked the sizeof value for the key
*
*  Our key is actually 3B but sizeof(MBOM_KEY) returns 4B !!
*  this is due to structure field packaging in gcc which screws it up big time
*  because sometimes get_node would return NULL when it shouldn't
*
*/
static int
equalKeys(void *k1, void *k2)
{                                                                            290
  MBOM_KEY * kk1 = (MBOM_KEY *)k1, * kk2 = (MBOM_KEY *)k2;
```

```
  return (kk1->from_state == kk2->from_state && kk1->character == kk2->character);
}


/* MBOM STUFF */

/*
* Case Translation Table
*/                                                                      300
static unsigned char xlatcase[256];

/*
* Init Case Translation Table
*/
static void init_xlatcase()
{
  int i;
  for (i = 0; i < 256; i++)
    {                                                                   310
      xlatcase[i] = toupper(i);
    }
}

#ifndef MBOM_MEM_STATS
#define MBOM_MEM_STATS
/*
* measure memory allocations
*/
void * MBOM_MALLOC2 (uint32_t size)                                    320
{
  void * p;
  p = malloc (size);
  if (p) {
    max_memory += size;
  }
  return p;
}

/*                                                                      330
* measure memory reallocations
* (only used by the hashtable in special circumstances where low on mem)
*/
void * MBOM_REALLOC2 (void * p, uint32_t new_size, uint32_t difference)
{
  realloc (p, new_size);
  if (p) {
    max_memory += difference;
  }
  return p;                                                             340
}
```

```
/*
* measure memory deallocations
*/
void MBOM_FREE2 (void * p, uint32_t size)
{
  if (p) {
    free (p);
    max_memory −= size;                                                     350
  }
}
#endif

/*
* toggle between verbose mode on/off with 1/0
*/
void mbomSetVerbose2(int n)
{
  s_verbose = n;                                                            360
}

/*
*   Select the desired storage mode
*/
int mbomSelectFormat2(MBOM_STRUCT2 * mbom, int format)
{
  switch( format )
  {
    case MBOM_ORACLE:                                                       370
    case MBOM_DAWG:
      mbom−>mbomFormat = MBOM_ORACLE; // only support this currently
      break;
    default:
      return −1; //doesn't even matter right now (only one version coded)
  }

  return 0;
}
                                                                            380
/*
* Create a new MultiBOM Matcher struct
*/
MBOM_STRUCT2 * mbomNew2()
{
  MBOM_STRUCT2 * mbom;

  init_xlatcase();

  mbom = (MBOM_STRUCT2 *) MBOM_MALLOC2(sizeof (MBOM_STRUCT2));              390
  MEMASSERT (mbom, "mbomNew");
  memset (mbom, 0, sizeof (MBOM_STRUCT2));
```

```
  mbom->transitions = create_hashtable(16, hashFromKey, equalKeys);
  MEMASSERT (mbom->transitions, "mbomNew (HT)");

  mbom->acsm = acsmNew();
  MEMASSERT (mbom->acsm, "mbomNew (acsm)");

  ++(summary.num_groups);                                                    400

  return mbom;
}

/*
*  Add a pattern to the list of patterns for this instance
*/
int mbomAddPattern2(MBOM_STRUCT2 * mbom, unsigned char * pat, int n, int nocase,
                int offset, int depth, void * id, int iid)
{                                                                            410
  if(n <= 0) {
    printf("Illegal pattern length found of: %d\n", n);
    exit(0);
  }

  if(mbom->minLen == 0 || mbom->minLen > n) {
    mbom->minLen = n; // keep track of the length of the shortest pattern
  }

  acsmAddPattern(mbom->acsm, pat, n, nocase, offset, depth, id, iid);        420
  ++(mbom->mbomNumPatterns);
  ++(summary.num_patterns);
  return 0;
}

#ifdef DEBUG_MBOM2
/*
*  Prints out the factor oracle structure:
*
*  This prints out the F.O. the same way as v1.0                             430
*  for comparison purposes (they should be the same)
*/
static void printMbom2(MBOM_STRUCT2 * mbom)
{
  QUEUE q;
  MBOM_STATE * next_state;
  MBOM_KEY tmpKey, * key;
  int j;

  printf("\nMBOM structure:\n");                                             440

  /* use queue to facilitate a breadth first traversal over the node/states
   * of the trie to print them */
  queue_init(&q);
```

```
  tmpKey.from_state = MBOM_ROOT;

  for(j = 0; j < ALPHABET_SIZE; ++j) {
    tmpKey.character = j;
                                                                              450
    if((next_state = get_node(mbom->transitions, &tmpKey)) != NULL) {
      //key = (MBOM_KEY * )malloc(sizeof(MBOM_KEY));
      //key->character = j;
      //key->from_state = MBOM_ROOT;
      queue_add(&q, next_state, NULL);
      printf("t(%d,%x) = %d\n", MBOM_ROOT, j, *next_state);
    }
  }

  while(queue_count(&q)) {                                                    460

    queue_remove(&q, &next_state, &key);
    tmpKey.from_state = *next_state;

    for(j = 0; j < ALPHABET_SIZE; ++j) {
      tmpKey.character = j;
      if((next_state = get_node(mbom->transitions, &tmpKey)) != NULL) {
        queue_add(&q, next_state, NULL);
        printf("t(%d,%x) = %d\n", tmpKey.from_state, j, *next_state);
      }                                                                       470
    }
  }

  queue_free(&q);
  printf("\n");
}
#endif

/*
*  Compile (Construct) the automaton to be used for this pattern matcher     480
*
*  Currently this function always builds a factor oracle,
*  but a DAWG could also be used
*
*  The resulting factor oracle recognizes at least all of the factors
*  of the pattern set P. It's construction time should be O(|P|) (linear).
*
*  For instructions on how to build this see the algorithm references/notes above
*/
int mbomCompile2(MBOM_STRUCT2 * mbom)                                        490
{
  int             j;
  ACSM_PATTERN * plist;
  MBOM_STATE current = 0, * cur = NULL, * next_state = NULL; // states
  QUEUE          q; // temp for Breadth-First Traversal
```

```
MBOM_KEY   * key, tmpKey, * parent;
MBOM_STATE * supplyFnc;  /* Only used in precomputation */

/* Create Trie: */
/* ———— */                                                          500

++(mbom->mbomSize); // Initial State

for (plist = mbom->acsm->acsmPatterns; plist != NULL; plist = plist->next) {
  current = MBOM_ROOT; // Initial State
  j = plist->n - 1; //start at the end of the patttern because we're entering it reversed

  tmpKey.from_state = current;
  tmpKey.character = plist->patrn[j];
                                                                    510
  while(j >= 0 && (next_state = get_node(mbom->transitions, &tmpKey)) != NULL) {
    tmpKey.from_state = current = *next_state;
    tmpKey.character = plist->patrn[--j];
  }

  while(j >= 0) {
    key = (MBOM_KEY *)MBOM_MALLOC2(sizeof(MBOM_KEY));
    MEMASSERT(key, "mbomCompile K");
    key->from_state = current;
    key->character = plist->patrn[j];                               520

    next_state = (MBOM_STATE *)MBOM_MALLOC2(sizeof(MBOM_STATE));
    MEMASSERT(next_state, "mbomCompile V");
    *next_state = ++(mbom->mbomSize); // Add State
    ++(mbom->mbomNumTrans); // Add Transition

    insert_node(mbom->transitions, key, next_state);
    current = *next_state;
    --j;
  }                                                                 530
}

/* Build Factor Oracle From Trie: */
/* ------------------------------ */

/* We need to create external transitions with a breadth first traversal */

// don't count this memory because it will deleted after during this fnc
supplyFnc = malloc((mbom->mbomSize + 1) * sizeof(MBOM_STATE));
// supply fnc 0 = NULL/NOTHING                                      540
memset(supplyFnc, 0, (mbom->mbomSize + 1) * sizeof(MBOM_STATE));

// use queue to facilitate a breadth first traversal over the
// node/states of the trie to make the factor oracle
queue_init(&q);
```

```
tmpKey.from_state = MBOM_ROOT;
for(j = 0; j < ALPHABET_SIZE; ++j) {
  tmpKey.character = j;
                                                                                550
  if((next_state = get_node(mbom->transitions, &tmpKey)) != NULL) {
    parent = (MBOM_KEY * )malloc(sizeof(MBOM_KEY));
    parent->character = tmpKey.character;
    parent->from_state = tmpKey.from_state;
    queue_add(&q, next_state, parent);
  }
}

while(queue_count(&q)) {
                                                                                560
  queue_remove(&q, &cur, &parent);

  // Process current node
  // tmpKey.from_state moves ("up") towards the root/initialState

  tmpKey.from_state = supplyFnc[parent->from_state];
  tmpKey.character = parent->character;

  while(tmpKey.from_state != 0 && get_node(mbom->transitions, &tmpKey) == NULL) {
                                                                                570
    // Add an external transition
    key = (MBOM_KEY *)MBOM_MALLOC2(sizeof(MBOM_KEY));
    MEMASSERT(key, "mbomCompile K2");
    key->character = tmpKey.character;
    key->from_state = tmpKey.from_state;

    next_state = (MBOM_STATE *)MBOM_MALLOC2(sizeof(MBOM_STATE));
    MEMASSERT(next_state, "mbomCompile V2");
    *next_state = *cur;
                                                                                580
    insert_node(mbom->transitions, key, next_state);
    ++(mbom->mbomNumTrans); // Add Transition

    tmpKey.from_state = supplyFnc[tmpKey.from_state];
  }

  if(tmpKey.from_state != 0) {
    next_state = get_node(mbom->transitions, &tmpKey);
    supplyFnc[*cur] = *next_state;
  }                                                                             590
  else {
    supplyFnc[*cur] = MBOM_ROOT;
  }

  free(parent); // no longer needed (tmp only)

  /* Enqueue all children nodes of current */
```

```
    tmpKey.from_state = *cur;
    for(j = 0; j < ALPHABET_SIZE; ++j) {
      tmpKey.character = j;                                                   600
      if((next_state = get_node(mbom->transitions, &tmpKey)) != NULL) {
        parent = (MBOM_KEY * )malloc(sizeof(MBOM_KEY));
        parent->character = tmpKey.character;
        parent->from_state = tmpKey.from_state;
        queue_add(&q, next_state, parent);
      }
    }
  }

  queue_free(&q);                                                             610
  free(supplyFnc); // wasn't counted in memory usage

  /* Tell the ACSM to compile itself too */
  /* ---------------------------------- */
  acsmCompile(mbom->acsm);

  /* Accrue Summary State Stats */
  summary.num_states += mbom->mbomSize;
  summary.num_transitions += mbom->mbomNumTrans;
                                                                              620

#ifdef DEBUG_MBOM2
  printMbom2(mbom);
  mbomPrintDetailInfo2(mbom);
#endif

  return 0;
}

                                                                              630
/** Tc is declared once outside of this function is a pointer
 ** into all converted uppercase text characters/bytes
 **/
#define MBOM_MAX_TEXT 65536
static unsigned char Tc[MBOM_MAX_TEXT]; // should be more than enough space for snort

/*
*  Search Function
*/
int mbomSearch2(MBOM_STRUCT2 * mbom, unsigned char *Tx, int n,               640
         int (*Match) (void * id, int index, void *data),
         void *data)
{
  int nfound  = 0; /* num of patterns found */
  int min     = mbom->minLen; // minimal length of patterns (also the window size)
  int i       = 0; // i is the position of the window on the text
  int critpos = 0; // position of the input head of the ACSM
  int j       = 0; // tmp
```

```
int end      = n - min + 1; // last valid i + 1
int windowEnd = min - 1;
MBOM_STATE current = 0;
MBOM_STATE * tmp = 0;
HASHTABLE * trans = mbom->transitions;
MBOM_KEY tmpKey;

int state       = 0; /* ACSM current state*/
ACSM_PATTERN    * mlist; /* tmp list of patterns at a terminal state */
ACSM_STATETABLE * states = mbom->acsm->acsmStateTable;

// Tc is declared once outside of this function is a pointer
// into all converted uppercase text characters/bytes

if(n > MBOM_MAX_TEXT) {
  printf("mbom Search unperformed because text was too long");
  exit(0);
}

// Case conversion of text

for (j = 0; j < n; ++j) {
  Tc[j] = xlatcase[ Tx[j] ];
}

while(i < end && critpos < n) {

  // Here's the ACSM has scanned up to but not including Tc[critpos]
  // We scan with the oracle back to and including Tc[critpos]

  j = i + windowEnd;
  current = MBOM_ROOT;

  // Search for factor mismatch in the oracle/dawg:
  tmpKey.character = Tc[j];
  tmpKey.from_state = current;
  tmp = get_node(trans, &tmpKey);

  while(j >= critpos && tmp != NULL) {
    --j;
    tmpKey.character = Tc[j];
    tmpKey.from_state = *tmp; // new current
    tmp = get_node(trans, &tmpKey);
  }

  if(tmp == NULL) { //if it didn't make it all the way to the critpos
    state = 0; // reset ACSM
    critpos = j + 1;
  }

  // Search with ACSM between indexes critpos to n-1:
```

```
  while(critpos < n && (critpos < i + min || states[state].depth >= min)) {          700

    state = states[state].NextState[Tc[critpos]]; // scan one character
    ++critpos;

    if(states[state].MatchList != NULL) { // if this state is terminal

      /* Go through the patterns that match at this state */

      for(mlist=states[state].MatchList; mlist != NULL; mlist = mlist->next) {      710

        /* j = location that match starts in Tx */
        j = critpos - mlist->n;

        /* obviously faster for patterns that are case insensitive */
        if(mlist->nocase) {
          ++nfound;
          if(Match (mlist->id, j, data))
            return nfound;
        }                                                                          720
        else {
          if(memcmp(mlist->casepatrn, Tx + j, mlist->n) == 0) {
            ++nfound;
            if(Match (mlist->id, j, data))
              return nfound;
          }
        }

      } //end for
    } //end if                                                                     730
  } //end while

  /* shift by critpos - length of longest prefix matched */
  i = critpos - states[state].depth; // SHIFT WINDOW
 }

 return nfound;
}

                                                                                   740
/*
*  Free all memory
*/
void mbomFree2(MBOM_STRUCT2 * mbom)
{
 hashtable_destroy(mbom->transitions, 1); // deletes all states and transitions

 acsmFree(mbom->acsm); // deletes the ACSM

 MBOM_FREE2(mbom, sizeof(MBOM_STRUCT2));                                           750
```

```
  --(summary.num_groups);
}

/*
*  Prints information about a mbom matcher instance
*/
void mbomPrintDetailInfo2(MBOM_STRUCT2 * mbom)
{
    char * sf[]= {"Factor Oracle", "DAWG (Directed Acyclic Word Graph)"};                760

    printf("+--[Pattern Matcher:Multi Backward Oracle Matching (MultiBOM)");
    printf(" Instance Info]------\n");
    printf("| Alphabet Size : %u Chars\n", ALPHABET_SIZE);
    printf("| Size of State : %u bytes\n", (int)(sizeof(MBOM_STATE)));
    printf("| Storage Format : %s\n", sf[mbom->mbomFormat]);
    printf("| Shortest Pat Len : %u\n", (unsigned int)mbom->minLen);
    printf("| Num States  : %u\n", (unsigned int)mbom->mbomSize);
    printf("| Num Transitions : %u\n", (unsigned int)mbom->mbomNumTrans);
    printf("| Num Patterns : %u\n", (unsigned int)mbom->mbomNumPatterns);                770
    printf("| State Density : %.1f%%\n",
      100.0*(double)mbom->mbomNumTrans/(mbom->mbomSize * ALPHABET_SIZE));
    printf("| All MBOMs' Memory: %.2fKbytes\n", (float)max_memory/1024 );
    printf("+-------------------------------");
    printf("---------------------------------");
    printf("---------------------\n\n");
    printf("+-------------------");
    printf(" AHO-CORASICK STATE MACHINE INFO FOLLOWS: ");
    printf("---------------------\n\n");
                                                                                         780
    acsmPrintDetailInfo(mbom->acsm);
}

/*
 *  Global sumary of all mbom info and all state machines built during this run
 */
void mbomPrintSummaryInfo2()
{
  // this IF is for mpsePrintSummary (which doesn't check which method is in use)
  if(summary.num_states > 0) {                                                           790
    printf("+--[Pattern Matcher:Multi Backward Oracle Matching (MultiBOM)");
    printf(" Overall Summary]----\n");
    printf("| Alphabet Size : %u Chars\n",ALPHABET_SIZE);
    printf("| Size of State  : %u bytes\n",(int)(sizeof(MBOM_STATE)));
    printf("| Num States   : %u\n",summary.num_states);
    printf("| Num Transitions : %u\n",summary.num_transitions);
    printf("| Num Groups  : %u\n",summary.num_groups);
    printf("| Num Patterns : %u\n",summary.num_patterns);
    printf("| State Density : %.1f%%\n",
      100.0*(double)summary.num_transitions / (summary.num_states * ALPHABET_SIZE));    800
    printf("| Memory Usage : %.2fKbytes\n", (float)max_memory/1024 );
```

```
    printf("+-----------------------------");
    printf("-----------------------------------");
    printf("--------------------\n\n");
    printf("+-----------------");
    printf(" AHO-CORASICK STATE MACHINE SUMMARY FOLLOWS: ");
    printf("--------------------\n\n");

    acsmPrintSummaryInfo();
  }                                                          810
}



//#define MBOM2_MAIN

#ifdef MBOM2_MAIN

#include <time.h>
                                                             820
/*
*  Text Data Buffer
*/
unsigned char text[2048];

/*
*   A Match is found
*/
int MatchFound (void* id, int index, void *data)
{                                                            830
  printf("MATCH:%s\n", (char *) id);
  return 0;
}

/*
* MAIN (for testing purposes)
*/
int main (int argc, char **argv)
{
  int i, nc, nocase = 0;                                     840
  MBOM_STRUCT2 * mbom;
  char * p;
  clock_t start, stop;

  if (argc < 3) {
    fprintf (stderr,"\nUsage: %s search-text pattern +pattern... [flags]\n",argv[0]);
    fprintf (stderr," flags: -nocase -verbose\n");
    fprintf (stderr," use a + in front of pattern for single case insensitive pattern\n\n");
    exit (0);
  }                                                          850

  mbom = mbomNew2();
```

```
if(!mbom) {
  printf("mbom−no memory\n");
  exit(0);
}

if(s_verbose) {
  printf("Parsing Parameters...\n");                              860
}

strcpy (text, argv[1]);

for(i = 1; i < argc; ++i) {

  if(strcmp(argv[i], "−nocase") == 0) {
    nocase = 1;
  }
  if(strcmp (argv[i], "−verbose") == 0) {                         870
    s_verbose = MBOM_VERBOSE;
  }
}

for (i = 2; i < argc; ++i) {
    if (argv[i][0] == '-') /* a switch */
      continue;

    p = argv[i];
                                                                   880
    if ( *p == '+') {
        nc=1;
        ++p;
    }
    else {
        nc = nocase;
    }

    mbomAddPattern2(mbom, p, strlen(p), nc, 0, 0, (void*)p, i - 2);
}                                                                  890

if(s_verbose) printf("Patterns added\n");

start = clock();
mbomCompile2(mbom);
stop = clock();

if(s_verbose) {
   printf("Patterns compiled in (%f seconds)\n", ((double)(stop-start))/CLOCKS_PER_SEC);
   mbomPrintDetailInfo2(mbom);                                     900
   printf("\n");
   mbomPrintSummaryInfo2();
   printf("\nSearching text...\n");
```

```
  }

  start = clock();
  mbomSearch2(mbom, text, strlen(text), MatchFound, (void *)0 );
  stop = clock();

  if(s_verbose)                                                           910
    printf ("Done search in (%f seconds)\n", ((double)(stop-start))/CLOCKS_PER_SEC);

  mbomFree2(mbom);

  if(s_verbose) printf ("Done cleaning\n");

  return 0;
}

#endif /* include main program */                                         920
```

A.4 Changes to the AC-Std Option

There was only a minor change made to the implementation of the AC-Std option in order to keep track of the minimum length pattern added to the group as well as the depth of every state in the Aho-Corasick state machine. We do not include the source code files here because the changes are minor and simple and the files are lengthy.

Bibliography

[1] VET Anti-virus, Nov 2005. http://www.vet.com.au.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[3] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: A new structure for pattern matching. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99, Theory and Practice of Informatics (Brno, 1999)*, volume 1725 of *Lecture Notes in Computer Science*, pages 291–306. Springer-Verlag, 1999. In *Proceedings of the 26th Seminar on Current Trends in Theory and Practice of Informatics*, Milovy, Czech Republic, November 1999.

[4] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle, suffix oracle. Technical Report IGM 99-08, Institut Gaspard Monge, Université de Marne-la-Vallée, France, 1999.

[5] C. Allauzen, M. Crochemore, and M. Raffinot. Oracle des facteurs, oracle des suffixes. Technical Report IGM 99-08, Institut Gaspard Monge, Université de Marne-la-Vallée, France, 1999.

[6] C. Allauzen and M. Raffinot. Oracle des facteurs d'un ensemble de mots. Technical Report IGM 99-11, Institut Gaspard Monge, Université de Marne-la-Vallée, France, 1999.

[7] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, J. P. Anderson Co., Fort Washington, PA, USA, 1980.

[8] S. Antonatos, M. Polychronakis, P. Akritidis, K. G. Anagnostakis, and E. P. Markatos. Piranha: Fast and memory-efficient pattern matching for intrusion detection. In *Proceedings 20th IFIP International Information Security Conference (SEC 2005)*, May 2005.

[9] A. Apostolico and M. Crochemore. String pattern matching for a deluge survival kit. *Handbook of massive data sets*, pages 151–194, 2002.

[10] A. Apostolico and Z. Galil. *Pattern Matching Algorithms*. Oxford University Press, New York, USA, 1997.

[11] R. Baeza-Yates and G. Navarro. *Text Searching: Theory and Practice*. Springer, Berlin, Germany, 2004.

[12] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In *SIGIR '89: Proceedings of the 12th annual international ACM SIGIR conference on research and development in information retrieval*, pages 168–175, New York, NY, USA, 1989. ACM Press.

[13] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, New York, NY, USA, 2002.

[14] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[15] A. Z. Broder. Some applications of Rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[16] C. Charras and T. Lecroq. Exact string matching algorithms – Animations in Java. Electronic Publication, Laboratoire d'Informatique de Rouen a l'Universite de Rouen, Facultdes Sciences et des Techniques. Jan. 1997. `http://www-igm.univ-mlv.fr/~lecroq/string/index.html`.

[17] C. Clark. C hash table – source code for a hash table data structure in C, Mar 2006. Computer Laboratory of the University of Cambridge. `http://www.cl.cam.ac.uk/~cwc22/hashtable/`.

[18] L. Cleophas, B. W. Watson, and G. Zwaan. A new taxonomy of sublinear keyword pattern matching algorithms. Technical Report CS-TR 04-07, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, Mar 2004.

[19] L. Cleophas, G. Zwaan, and B. W. Watson. Constructing factor oracles. Technical Report CS TR 04-01, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, Jan. 2004.

[20] C. J. Coit, S. Staniford, and J. McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of Snort. In *2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, June 2001.

[21] E. Cole, R. L. Krutz, and J. Conley. *Network Security Bible*. John Wiley & Sons, Inc., New York, NY, USA, 2005.

[22] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. In *SODA '91: Proceedings of the Second Annual ACM-SIAM Sym-*

posium on Discrete Algorithms, pages 224–233, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[23] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium on Automata, Languages and Programming*, pages 118–132, London, UK, 1979. Springer-Verlag.

[24] B. Commentz-Walter. A string matching algorithm fast on the average. Technical Report 79.09.007, IBM Heidelberg Scientific Center, 1979.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2002.

[26] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan. 1998.

[27] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. Technical Report 93-03, Institut Gaspard Monge, Université de Marne-la-Vallée, France, 1993.

[28] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte.* Vuibert, 2001.

[29] M. Crochemore, T. Lecroq, A. Czumaj, L. Gasieniec, S. Jarominek, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. In *STACS '92: Proceedings of the 9th Annual Symposium on Theoretical Aspects*

of Computer Science, volume 577 of *Lecture Notes in Computer Science*, pages 589–600, London, UK, 1992. Springer-Verlag.

[30] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[31] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium*, pages 29–44, Aug. 2003.

[32] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[33] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *Hot Interconnects*, pages 44–51, Stanford, CA, USA, Aug. 2003.

[34] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[35] J.-J. Fan and K.-Y. Su. An efficient algorithm for matching multiple patterns. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):339–351, 1993.

[36] M. Fisk and G. Varghese. Applying fast string matching to intrusion detection. Technical Report in preparation, successor to UCSD TR CS2001-0670, University of California at San Diego, La Jolla, CA, USA, 2001.

[37] M. Fisk and G. Varghese. Fast content-based packet handling for intrusion detection. Technical Report CS2001-0670, University of California at San Diego, La Jolla, CA, USA, 2001.

[38] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[39] A. Gal, C. W. Probst, and M. Franz. Average case vs. worst case – margins of safety in system design. In *New Security Paradigms Workshop (NSPW 2005)*, Lake Arrowhead, CA, USA, 2005.

[40] Z. Galil. On improving the worst case running time of the Boyer-Moore string matching algorithm. *Communications of the ACM*, 22(9):505–508, Sept. 1979.

[41] F. Gong. Deciphering detection techniques: Part II anomaly-based intrusion detection. Mar 2003. `http://www.mcafee.com/us/local_content/white_papers/wp_ddt_anomaly.pdf`.

[42] J. Graham-Cumming. The POPFile Website, Mar 2006. `http://popfile.sourceforge.net//`.

[43] M. Gregg. *CISSP Exam Cram 2*. Que, New York, NY, USA, 2005.

[44] T. A. Group. The SpamAssassin Website, Nov 2005. `http://spamassassin.apache.org/`.

[45] M. Haertel. Gnugrep-2.0. *Usenet archive comp.sources.reviewed*, 3, July 1993.

[46] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.

[47] J. ichi Aoe, K. Morimoto, and T. Sato. An efficient implementation of trie structures. *Software Practice & Experience*, 22(9):695–721, 1992.

[48] J.-P. Iivonen, S. Nilsson, and M. Tikkanen. An experimental study of compression methods for functional tries. *Workshop on Algorithmic Aspects of Advanced Programming Languages (WAAAPL'99), Part of PLI'99*, 19(7), 1999.

[49] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, Mar 1987.

[50] S. Kim and Y. Kim. A fast multiple string-pattern matching algorithm. In *Proc. of 17th AoM/IAoM Conference on Computer Science*, Aug. 1999.

[51] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[52] T. Kojm. The Clam AntiVirus Website, Nov 2005. `http://www.clamav.net/`.

[53] J. Koziol. *Intrusion Detection with Snort (1st Edition).* SAMS, May 2003.

[54] B. Laing and J. Alderson. How-to guide – Implementing a network-based intrusion detection system. 2000. `http://www.snort.org/docs/iss-placement.pdf`.

[55] Libpcap. Tcpdump/libpcap, Nov 2005. The Tcpdump Group. `http://www.tcpdump.org`.

[56] K. Maly. Compressed tries. *Communications of the ACM*, 19(7):409–415, 1976.

[57] A. Matrawy, P. C. van Oorschot, and A. Somayaji. Mitigating network denial-of-service through diversity-based traffic management. In *Applied Cryptography and Network Security: Third International Conference, ACNS 2005*, volume 3541 of *Lecture Notes in Computer Science*, pages 104–121, New York, USA, Jun 2005. Springer.

[58] Y. Miretskiy, A. Das, C. Wright, and E. Zadok. Avfs: An on-access anti-virus file system. In *13th USENIX Security Symposium*, pages 73–88, San Diego, CA, USA, Aug. 2004.

[59] J. H. Morris and V. R. Pratt. A linear pattern-matching algorithm. Technical Report TR 40, University of California, Berkley, CA, USA, 1970.

[60] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[61] G. Navarro. NR-grep: a fast and flexible pattern-matching tool. *Software – Practice and Experience*, 31(13):1265–1312, 2001.

[62] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithms (JEA)*, 5(4), 2000.

[63] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings, Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, Cambridge, UK, 2002.

[64] S. Nilsson and M. Tikkanen. Implementing a dynamic compressed trie. In K. Mehlhorn, editor, *The Second Workshop on Algorithm Engineering (WAE '98)*, pages 25–36, Max-Planck-Institut fr Informatik, Saarbrcken, Germany, 1998. Chapman & Hall, Ltd.

[65] M. Norton. Optimizing pattern matching for intrusion detection. Sep 2004. `http://www.sourcefire.com/products/downloads/secured/sf_OPMforID.pdf`.

[66] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[67] V. Paxson. The Bro Website, Nov 2005. `http://bro-ids.org`.

[68] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Aiken Computation Laboratory, 1981.

[69] M. Raffinot. On the multi backward DAWG matching algorithm (MultiBDM). In R. Baeza-Yates, editor, *WSP'97: Proceedings of the 4th South American Workshop on String Processing*, pages 149–165, Valparaiso, Chile, Nov. 1997. Carleton University Press.

[70] R. Riordan. Polysearch: An extremely fast parallel search algorithm. In *Fifth International Computer Virus and Security Conference*, pages 631–640, New York, NY, USA, 1992.

[71] R. Rivest. The MD5 Message-Digest Algorithm, RFC 1321, Apr. 1992.

[72] M. Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[73] L. Salmela, J. Tarhio, and J. Kytoki. Multi-pattern string matching with q-grams. *To appear in ACM Journal of Experimental Algorithmics (JEA)*, 2006.

[74] R. Sekar, T. F. Bowen, and M. E. Segal. On preventing intrusions by process behavior monitoring. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, pages 29–40, Berkeley, CA, USA, 1999. USENIX Association.

[75] A. Somayaji and S. Forest. Automated response using system-call delays. In *9th Usenix Security Syposium*, pages 185–197, Aug. 2000.

[76] Sourcefire. Snort 2.0 detection revisited. Apr 2004. `http://www.sourcefire.com/products/downloads/secured/sf_snort20_detection_rvstd.pdf`.

[77] Sourcefire. The Snort Website, Nov 2005. `http://www.snort.org`.

[78] Sourcefire Inc. The Sourcefire Website, Nov 2005. `http://www.sourcefire.com`.

[79] SpamBayes. The SpamBayes Website, Mar 2006. `http://spambayes.sourceforge.net/index.html/`.

[80] M. Strebe. *Network Security Foundations: Technology Fundamentals for IT Success*. Sybex, New York, NY, USA, 2004.

[81] D. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[82] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, New York, NY, USA, 2005.

[83] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 112–122, Washington, DC, USA, 2005. IEEE Computer Society.

[84] M. Tenase. The great ids debate: Signature analysis versus protocol analysis. *Infocus*, Feb 2003. `http://www.securityfocus.com/infocus/1663`.

[85] The Shmoo Group. The Shmoo Group Website, Nov 2005. `http://www.shmoo.com`.

[86] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of the IEEE Infocom Conference*, pages 333–340. IEEE, Mar. 2004.

[87] J. van Lunteren and T. Engbersen. High-performance pattern-matching engine for intrusion detection - a new approach for fast programmable accelerators. In *Hot Chips 17*, Palo Alto, CA, USA, 2005. Stanford University.

[88] B. Watson. The performance of single-keyword and multiple-keyword pattern matching algorithms. Technical Report CS TR 94-19, Department of Computing Science, Eindhoven University of Technology, 1994.

[89] B. W. Watson and L. Cleophas. Spare parts: a C++ toolkit for string pattern recognition. *Software Practice and Experience*, 34(7):697–710, 2004.

[90] B. W. Watson and G. Zwaan. A taxonomy of sublinear multiple keyword pattern matching algorithms. *Science of Computer Programming*, 27(2):85–118, 1996.

[91] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, 1992.

[92] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science. Chung-Cheng University, 1994.

[93] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *ICNP '04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04)*, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.