國 立 交 通 大 學

電 信 工 程 學 系 碩 士 班

碩 士 論 文

高效能字串比對演算法及其實現

An Efficient Pattern Matching Algorithm and Its Implementation

研究生　：梁嘉旂

指導教授：李程輝 教授

中 華 民 國 九 十 六 年 一 月

高效能字串比對演算法及其實現

An Efficient Pattern Matching Algorithm and Its Implementation

研 究 生： 梁嘉旂　　Student: Chia-Chi Liang
指導教授： 李程輝 教授　　Advisor: Prof. Tsern-Huei Lee

國 立 交 通 大 學

電 信 工 程 學 系 碩 士 班

碩 士 論 文

A Thesis
Submitted to Institute of Communication Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Communication Engineering
January 2007
Hsinchu, Taiwan, Republic of China.

中 華 民 國 九 十 六 年 一 月

高效能字串比對演算法及其實現

學生: 梁嘉旂　　　　　　　　　　　　　指導教授: 李程輝 教授

國立交通大學電信工程學系碩士班

中文摘要

因為可以準確判斷，因此字串比對在一般入侵偵測系統中扮演著相當重要的角色，被廣泛應用到網路安全設備來偵測攻擊或病毒。在現今有許多知名的字串比對演算法中，因為 AC 演算法可以同時比對多個字串並保證在最壞情況的效能，因此被廣泛使用。然而，原始的 AC 演算法有兩項缺點需要改進，其中一項是記憶體需求量，另一項是工作輸出量。因為原始的 AC 演算法在一個運算週期只能處理一個字元，無法滿足現在高速網路，所以本研究延伸 AC 演算法，使其可以處理多的字元，以達到工作輸出量的改進。在演算法裡，全部的字樣及欲比對之字串都分成 K 份，有 K 組比對引擎同時做比對，一個運算週期共處理 K 個字元，所以工作輸出量增加至 K 倍。我們實作在 Xilinx FPGA 上，當 K=4 的時候可以達到 4.5Gbps 的工作輸出量。然而考慮實作的情況，因為每個運算週期都必須讀取相當多的位元並不理想，所以根據所提的演算法，將字樣依規則分成幾個組別，使用編號方式做延伸改進。除此之外，考量記憶體的資源，我們可以複製較少份的查表資料，使其在多個運算週期內計算完 K 組比對結果。換言之，我們可以達到一個運算週期處理多個字元，並且使用較少的記憶體，根據任何硬體考量可以做修正，適用於各種的字樣的字串比對演算法。

An Efficient Pattern Matching Algorithm and Its Implementation

Student: Chia-Chi Liang Advisor: Prof. Tsern-Huei Lee

Institute of Communication Engineering

National Chiao Tung University

Abstract

Because of its accuracy, pattern matching technique has recently been applied to Internet security applications such as intrusion detection/prevention, anti-virus, and anti-malware. Among the various pattern matching algorithms, the Aho-Corasick (AC) can match multiple pattern strings simultaneously with worst-case performance guarantee and thus is widely adopted. However, the throughput performance of the original AC may not be satisfactory for high speed environments because only one symbol is processed in an operation cycle. In this paper we present an extension of the AC algorithm where multiple symbols are processed in an operation cycle to improve throughput performance. In our proposed scheme, all pattern strings, and the input text string as well, are divided into K substrings, if K symbols are processed in an operation cycle. Moreover, K pattern search engines are employed to scan the text substrings in parallel. As a result, the throughput performance can be improved by K times. We implemented our proposed pattern matching scheme with Xilinx FPGA and achieved more than 4.5Gbps throughput for $K = 4$. As we need to access so many bits for PMV per cycle, we have presented an extension of our algorithm with output index. We separate patterns into several groups and PMV can be replaced by index. Considering the memory resource of hardware, we can duplicate fewer tables by processing it in several cycles. As a result it's flexible to deal with any given rule set in all situations.

誌謝

首先，感謝我的指導教授—李程輝教授。從大三做專題到研究所的這段時間，在教授的引領之下，讓我接觸感興趣的題目著手研究，並且給予充分的信任和鼓勵，以及每一次詳細的討論和中肯的建議。在教授的教誨下，不論是做研究的正確態度或方法，都讓我獲益良多。

感謝網路技術實驗室的謝景融學長、黃迺倫學姐和黃郁文學長，實驗室不管大小事情都可以找你們解決，並且帶著大家研究的熱情，有這些帶領的學長姐真的很幸福。也要感謝一起奮鬥的同學們：建成、登煌和柏庚，每次趕進度的時候都是你們在旁邊加油打氣，有你們的支持陪伴，在實驗室的生活更充實。

最後，我要特別感謝我的父親與母親，在您們的無微不至的照顧下，讓我毫無後顧之憂做任何事情，從小的諄諄教誨才造就今日的我，您們的關心和信任給我最大動力。

謹將此論文獻給我最愛的親友。

Contents

中文摘要 **i**

English Abstract **ii**

誌謝 **iii**

Contents **iv**

List of Tables **vi**

List of Figures **vii**

Chapter 1 Introduction **1**

1.1 Background 1

1.2 Motivation 2

Chapter 2 Network Intrusion Detection Systems **5**

2.1 Introduction 5

2.2 Snort 6

2.2 ClamAV 7

Chapter 3 Pattern Matching **10**

3.1 Aho Corasick Algorithm 10

3.2 The Bit-Split Aho Corasick Algorithm 13

3.2 Suffix Based Traversing Algorithm 15

Chapter 4 A High-Performance and Memory-Efficient Pattern Matching Algorithm 18

4.1 Conception 18

4.2 Our Proposed Algorithm 19

4.2 Extension of Our Proposed Algorithm 25

Chapter 5 Comparison and Experimental Results 29

5.1 Experimental Results 29

5.2 Results with Output Index 31

Chapter 6 Conclusions 34

Bibliographies 36

List of Tables

Table 4-1. Conditions for pattern string *S* to be matched 21

Table 5-1. The result of our proposed algorithm with output index 32

List of Figures

Figure 2-1. Rule Chain logical structure 7

Figure 2-2. A fragment of the ClamAV trie structure 8

Figure 3-1. AC trie constructed from {he,she,his,hers} 11

Figure 3-2. Sequences of state transition for input string=ushers 12

Figure 3-3. The fourth sub-trie with m=2 14

Figure 3-4. Bit-split search engine with 4 tiles 15

Figure 3-5. SBT architecture 16

Figure 3-6. Reducing table size using indirect pointers 17

Figure 4-1. Example of reassembling two parts of a string 19

Figure 4-2. Example of trie Gp* for {he, she, his, hers} with $k = 2$ 20

Figure 4-3. Example of all the matches for S = abcdefghij and $K = 6$. 23

Figure 4-4. The String Matching Machine Architecture 24

Figure 4-5. The trie with output index 26

Figure 4-6. Example of combining all groups with more index numbers 28

Figure 5-1. Comparison of total memory requirement 30

Figure 5-2. Number of state with different m and K 30

Figure 5-3. Comparison of total memory requirement 33

Chapter 1

Introduction

1.1 Background

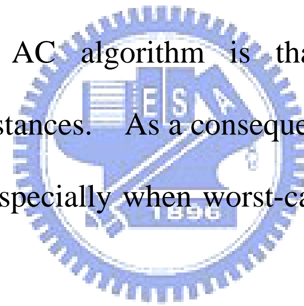
The progress of network speed and technology makes network security to be an important issue in network application. Because Internet is accessible to everyone, more and more users highly rely on the correct operation of networks. There are many security incidents such as eavesdropping, intrusion, and virus/worms that caused great damage and economic loss to our society. Nowadays, we do some protection in host, and furthermore adding more efficient mechanism in network edge devices has attracted much attention in recent years.

Now we have two kinds of technologies of detection for intrusions and virus. The first one is behavior anomaly. The concept of behavior anomaly is to establish a profile of normal behavior and identify a host to be abnormal if its behavior does not conform to the profile. It may result in a high false positive rate because the normal behavior profile is difficult to specify. The second one is based on packet content, and it checks if there is something abnormal by pattern matching. Those attacks are usually represented by simple strings or regular expressions. It is possible to detect any malware as long as its signature is available.

Pattern matching has been an important technique in information retrieval and text editing for many years. Recently, it has been applied to Internet security for signature matching to detect virus, worms, intrusion, etc. The function of pattern

matching is to search for predefined patterns in packet payloads. Since pattern occurrence may happen at any position of the payload, it is very time consuming. Because of the rapid advances of computer and network technologies, it becomes increasingly desirable for a high-performance pattern matching module that achieves at least 10Gbps throughput.

There are some well-known pattern matching algorithms such as Knuth-Morris-Pratt (KMP) [2], Boyer-Moore (BM) [3], and Aho-Corasick (AC) [4]. The KMP and BM algorithms are efficient for single pattern matching but are not scalable for multiple patterns. The AC algorithm pre-processes the patterns and builds a finite automaton which can match multiple patterns simultaneously. Another advantage of the AC algorithm is that it guarantees deterministic performance under all circumstances. As a consequence, the AC algorithm is widely adopted in various systems, especially when worst-case performance is an important design factor.



1.2 Motivation

Unfortunately, the original AC algorithm processes only one symbol per operation cycle which limits the maximum throughput to LC bps where L and C are, respectively, the size of a symbol and the clock rate for processing each symbol. In most Internet security applications each symbol is a byte and thus $L = 8$. Consequently, the maximum achievable throughput is only 1.6Gbps if C = 200 MHz. This is obviously not enough for high speed networks. To achieve high throughput performance, a variant of the AC algorithm, suffix based traversing (SBT) algorithm,

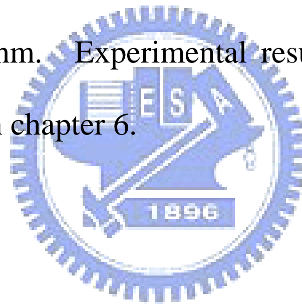
where multiple symbols are processed in every operation cycle was proposed in [5]. The basic idea is to collect all possible *k*-step transitions for $1 \le k \le w$ from every state, where *w* is the number of symbols processed in an operation cycle. Assuming that $w = 2^n$, the number of table lookups for the longest path is equal to *n*. According to the evaluation, it has to satisfy $w \ge 64$ bytes in order to achieve 10 Gbps throughput.

Another problem of the AC algorithm is the potential huge amount of memory space requirement. If every symbol is a byte, then the number of possible input is 2^8=256 which means an array of 256 entries is required for every state. In other words, with the straightforward implementation for 1M states, the required memory space is about 1G bytes if every state is represented by 4 bytes. There are various compression techniques to reduce the space requirement. A clever bit-splitting idea which can largely reduce the space requirement was proposed in [6]. In this scheme, an *L*-bit symbol is divided into *L*/*m* equal-size sub-symbols. For example, if $L = 8$ and $m = 2$, then every symbol is divided into 4 sub-symbols and each sub-symbol is composed of 2 bits. The sub-symbols derived from the same positions of the symbols in all pattern strings form a group. As a result, there are *L*/*m* groups of sub-symbols. A sub-trie is built for each group of sub-symbols. It is clear that the number of possible inputs reduces from 2^L for the original trie to 2^m for each sub-trie. Consequently, the space requirement reduces from $O(2^L)$ to $O(2^m L/m)$, which is significant because in most applications $L = 8$ and one can choose $m = 2$. Since we adopt the bit-splitting idea in our implementation, details of this scheme is described in Section II.

In this paper, we present a generalization of the AC algorithm where multiple symbols are processed in each operation cycle. Different from the solution proposed

in [5], our approach divides every pattern string into K sub-strings and construct tries for sub-strings, where K is the number of symbols processed in each operation cycle. The text string is also divided into K sub-strings. Besides, we use K search engines to scan text sub-strings in parallel. Experimental results show that our scheme can achieve more than 4.5Gbps throughput for $K = 4$ and in general requires less space than the scheme proposed in [5].

The rest of this paper is organized as follows. We introduce two well-known network intrusion detection systems, snort and ClamAV in chapter 2. Chapter 3 contains a review of the original AC algorithm, the bit-split AC algorithm and suffix based traversing algorithm. In chapter 4, we present our generalization of the high-performance AC algorithm. Experimental results are discussed in chapter 5. Finally, we draw conclusion in chapter 6.

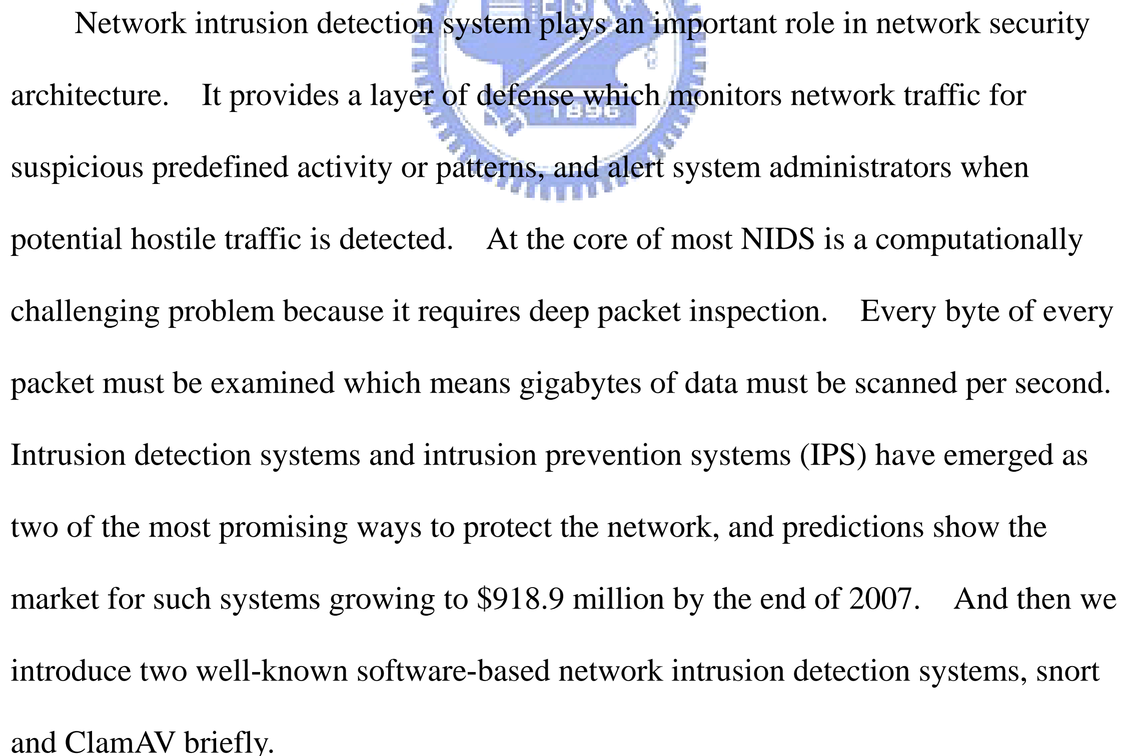


Chapter 2

Network Intrusion Detection Systems

Network Intrusion Detection Systems (NIDS) have become widely recognized as powerful tools for identifying, deterring and deflecting malicious attacks over the network. This chapter includes the brief of two well-known software-based network intrusion detection systems, snort and ClamAV.

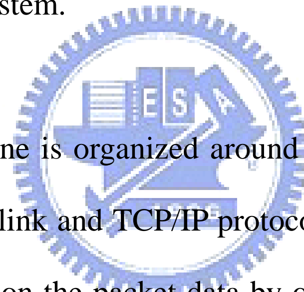
2.1 Introduction



Network intrusion detection system plays an important role in network security architecture. It provides a layer of defense which monitors network traffic for suspicious predefined activity or patterns, and alert system administrators when potential hostile traffic is detected. At the core of most NIDS is a computationally challenging problem because it requires deep packet inspection. Every byte of every packet must be examined which means gigabytes of data must be scanned per second. Intrusion detection systems and intrusion prevention systems (IPS) have emerged as two of the most promising ways to protect the network, and predictions show the market for such systems growing to $918.9 million by the end of 2007. And then we introduce two well-known software-based network intrusion detection systems, snort and ClamAV briefly.

2.2 Snort

Snort is a libpcap-based packet sniffer and logger that can be used as a lightweight network intrusion detection system. It features rules based logging to perform content pattern matching and detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and much more. Snort has real-time alerting capability, with alerts being sent to syslog, Server Message Block (SMB) "WinPopup" messages, or a separate "alert" file. Snort's architecture is focused on performance, simplicity, and flexibility. There are three primary subsystems that make up Snort: the packet decoder, the detection engine, and the logging and alerting subsystem.



The packet decoder engine is organized around the layers of the protocol stack present in the supported data-link and TCP/IP protocol definitions. Each subroutine in the decoder imposes order on the packet data by overlaying data structures on the raw network traffic. These decoding routines are called in order through the protocol stack, from the data link layer up through the transport layer, finally ending at the application layer. Snort provides decoding capabilities for Ethernet, SLIP, and raw (PPP) data-link protocols.

In the detection engine, Snort maintains its detection rules in a two dimensional linked list of what are termed Chain Headers and Chain Options as in Figure 2-1. These are lists of rules that have been condensed down to a list of common attributes in the Chain Headers, with the detection modifier options contained in the Chain Options. These rule chains are searched recursively for each packet.

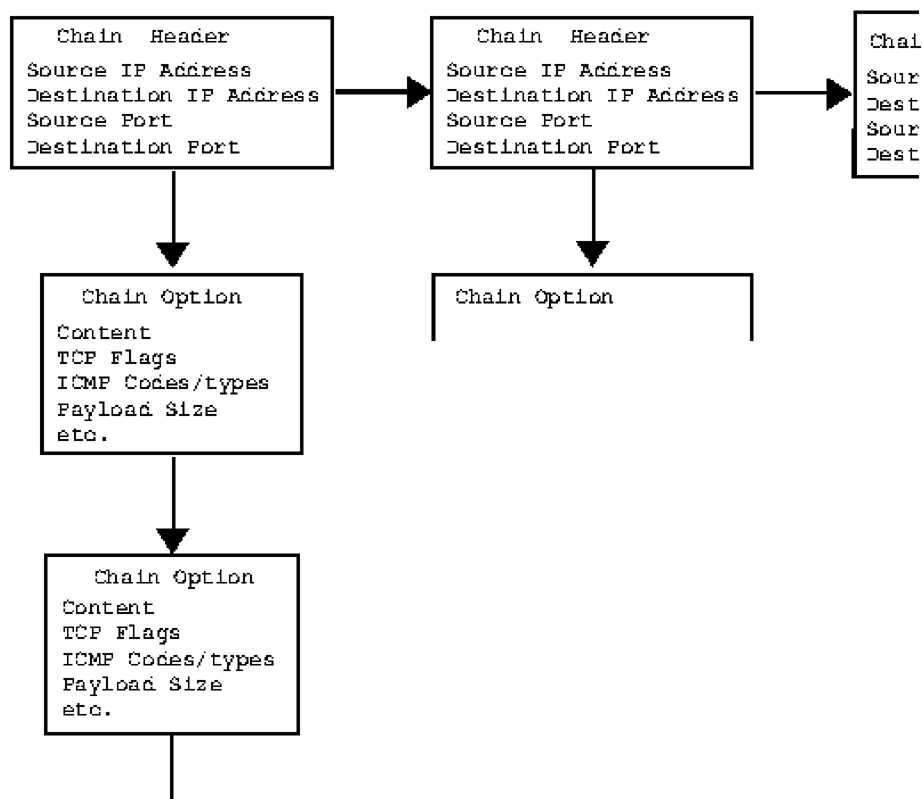


Figure 2-1 : Rule Chain logical structure

There are currently three logging and five alerting options in The logging/alerting Subsystem. The logging options can be set to log packets in their decoded, human readable format to an IP-based directory structure, or in tcpdump binary format to a single log file. Alerts may either be sent to syslog, logged to an alert text file in two different formats, or sent as WinPopup messages using the Samba smbclient program.

2.3 ClamAV

Clam AntiVirus is a GPL anti-virus toolkit for UNIX. The main purpose of this software is the integration with mail servers for attachment scanning. The package

provides a flexible and scalable multi-threaded daemon, a command line scanner, and a tool for automatic updating via Internet. The programs are based on a shared library distributed with the Clam AntiVirus package, which you can use with your own software. Most importantly, the virus database is kept up to date. It is the most widely used open-source anti-virus scanner available. Currently, it can detect over 35000 viruses, worms, and trojans, including Microsoft Office and Mac Office macro viruses. Then it also built-in supports for many kinds of compressed files, mail files, and compressed portable executable files.

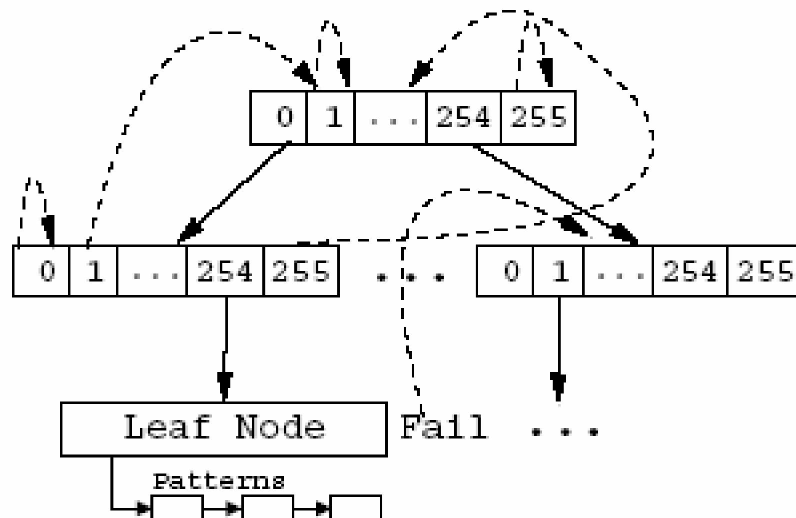


Figure 2-2 : A fragment of the ClamAV trie structure. Solid lines show success transitions; dashed lines show failure ransitions.

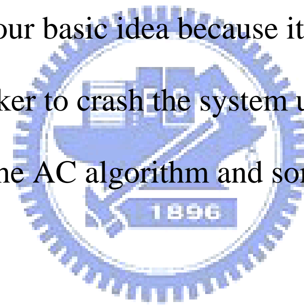
ClamAV uses a variation of the Aho Corasick pattern matching algorithm. To quickly look up each character read from the input, ClamAV constructs a 256-way trie structure as shown in Figure 2-2. The memory usage of ClamAV depends on how deep the trie is. The deeper the trie, the more nodes are created. Since the Aho Corasick algorithm builds an automaton with a depth equal to the longest pattern, the memory usage would be unacceptably large because some patterns are over 2KB.

ClamAV modifies the Aho-Corasick algorithm so that the trie is constructed only to some maximum height, and all patterns beginning with the same prefix are stored in a linked list under the appropriate leaf node. The maximum trie height is restricted by the length of the shortest pattern, which is currently two bytes. ClamAV's performance suffers whenever a node with a large number of patterns with the same prefix is encountered during input matching.

Chapter 3

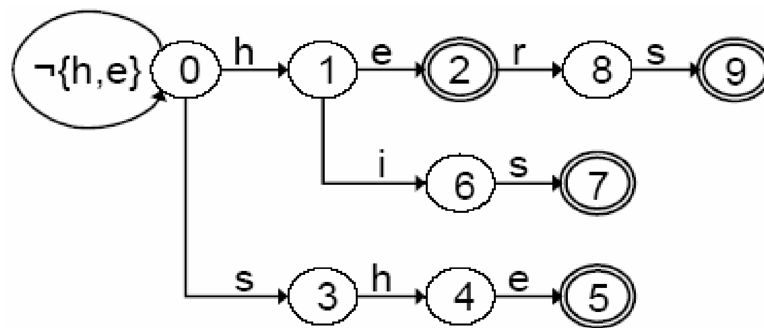
Pattern Matching

According to some report [11], the pattern matching module can consume up to 70% of CPU computation power in an intrusion detection system. But it is still applied in network security devices to detect attacks or virus because of its accuracy. Patterns are like fingerprints so we can check if it has something wrong. We can cut out a segment from the virus program that is unique to represent the virus. A number of algorithms have been proposed for pattern matching in network security. We choose AC algorithm for our basic idea because it can guarantee the worst-case performance that protect attacker to crash the system using weakness of the algorithm. In this chapter, we introduce the AC algorithm and some of its improvements.



3.1 Aho Corasick Algorithm

Aho Corasick Algorithm pre-processes the patterns and builds a finite automaton which can match multiple patterns simultaneously. Every state of the trie represents that the suffix of input matches the longest prefix of some pattern up to this time. Figure 3-1 shows an example of the trie that we have four patterns he, she, his and hers. In AC algorithm, we look up table for next state according to a current state and the input symbol. The same process is repeated. It reports a match in final state when it matches a pattern. AC algorithm has to build three tables in advance: goto function, g(state,t); failure function, f(state); output function, output(state). The behavior of the pattern matching state machine is dictated by these three functions.



(a) Goto function.

i	1	2	3	4	5	6	7	8	9
$f(i)$	0	0	0	1	2	0	3	0	3

(b) Failure function.

i	*output* (i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

(c) Output function.

1896

Figure 3-1 : The state machine is original AC trie constructed from 4-string rule set, {he,she,his,hers}

The goto function maps a pair consisting of a state and an input symbol into a state or the failure message. The current symbol is received by goto function, and the current state is transferred to next state. The state machine starts with an empty root node which is the default non-matching state, state 0.

If it fails in state transition, it will look up failure function. It continues to process from the return state of failure function. The failure function maps a state into a state. If failure happens, the failure function points current state to the longest prefix of that state which also leads to a valid state in the trie. We can see that it can

also eliminate all failure transitions by pre-computing the next state for every character from every state in the machine. We can use a next function to replace the original goto function and failure function, and it is applied in bit-split AC algorithm and SBT algorithm.

In the course of processing, output function outputs all strings that has been matched. The output function formalizes this concept by associating a set of keywords (possibly empty) with every state. The same process repeats until all symbols of text string are processed.

For example, the patterns are he, she, his, and hers, and the text string is ushers. Figure 3-2 shows the sequences of state transition for this example. At the beginning, the state is root node 0, and it is transferred to state 3 and 4 according to goto function. Consider the operation cycle when state is 4 and input symbol is e. Since g(4,e)=5 , output(5) indicates that it has found the keywords she and he at the end of position four in the text string. In state 5 on input symbol r, the machine makes two state transitions in its operating cycle. Since g(5, r) =fail, it lookup again for g(2,r) according to f(5)=2, and the next state is 8. At the end, it is transferred to state 9 and output hers because of output(9).

input		u		s		h		e		r		s	
state	0		0		3		4		5		8		9
f(state)									2				

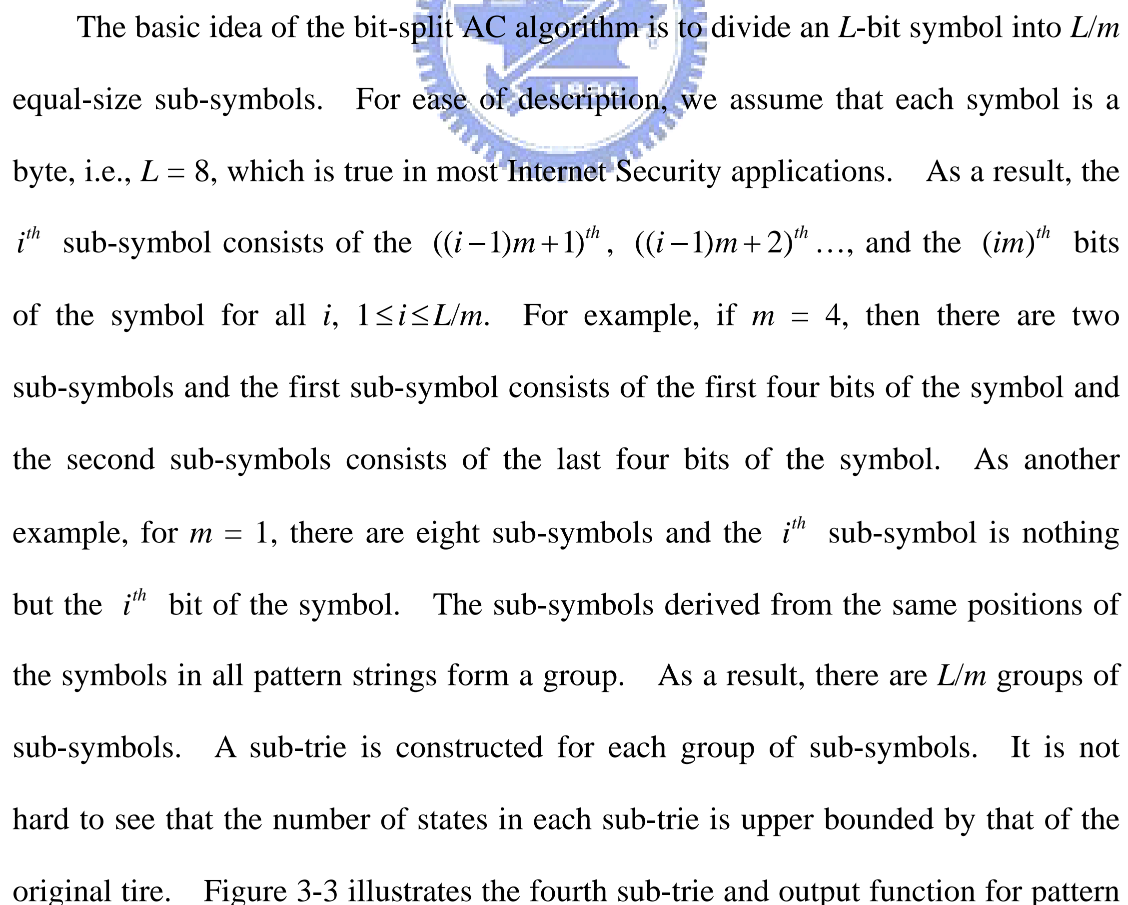
Figure 3-2: Sequences of state transition for input string=ushers

In processing an input of length n makes exactly n goto transitions. And the

total number of failure transitions must be at least one less than the total number of goto transitions. Therefore, the total number of state transitions is less than 2*n*. In addition, next function combining goto function and failure function makes it need exactly *n* state transitions for text string of *n* symbols. As we mentioned, it can guarantee the worst-case performance.

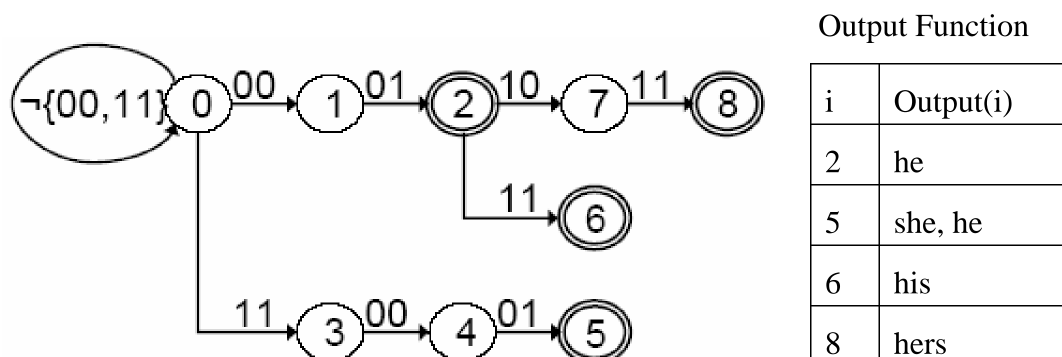
3.2 The Bit-Split Aho-Corasick Algorithm

In this section, we review the bit-split AC algorithm which was proposed in [6]. As mentioned before, the bit-split AC algorithm can largely reduce the memory requirement.



The basic idea of the bit-split AC algorithm is to divide an *L*-bit symbol into *L*/*m* equal-size sub-symbols. For ease of description, we assume that each symbol is a byte, i.e., *L* = 8, which is true in most Internet Security applications. As a result, the ith sub-symbol consists of the $((i-1)m+1)^{th}$, $((i-1)m+2)^{th}$ …, and the $(im)^{th}$ bits of the symbol for all *i*, $1 \le i \le L/m$. For example, if *m* = 4, then there are two sub-symbols and the first sub-symbol consists of the first four bits of the symbol and the second sub-symbols consists of the last four bits of the symbol. As another example, for *m* = 1, there are eight sub-symbols and the ith sub-symbol is nothing but the ith bit of the symbol. The sub-symbols derived from the same positions of the symbols in all pattern strings form a group. As a result, there are *L*/*m* groups of sub-symbols. A sub-trie is constructed for each group of sub-symbols. It is not hard to see that the number of states in each sub-trie is upper bounded by that of the original tire. Figure 3-3 illustrates the fourth sub-trie and output function for pattern

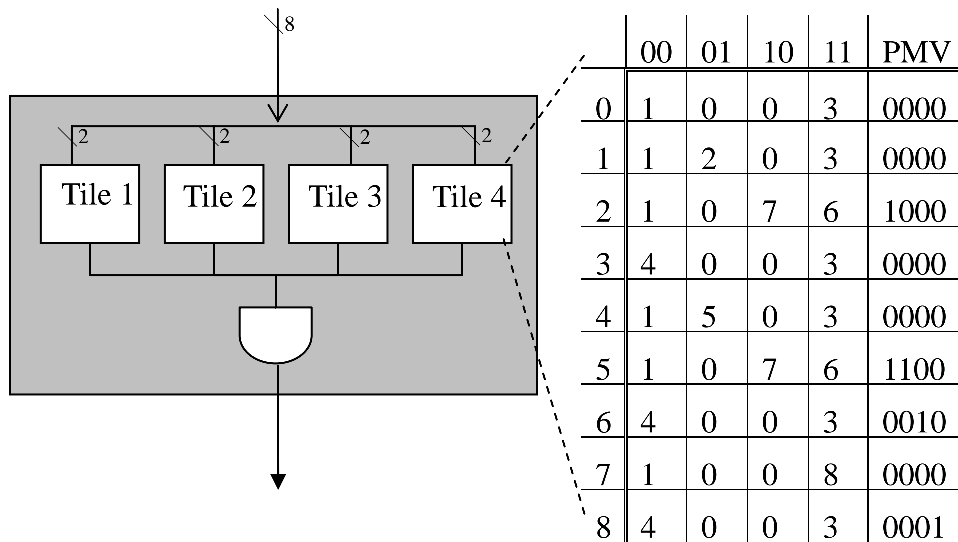
strings {he, she, his, hers} with $m = 2$.



i	Output(i)
2	he
5	she, he
6	his
8	hers

Figure 3-3 : The fourth sub-trie with m=2 from the group of sub-symbol, {00 01, 11 00 01, 00 01 11, 00 01 10 11}.

To maintain the same throughput performance as the original AC algorithm, the bit-split scheme uses *L/m* search engines to scan the input text concurrently. Let $T = t_1t_2t_3...$ be the input text. Moreover, let t_i^j denote the jth sub-symbol of t_i. In the bit-split scheme, tile *j* takes $t_1^jt_2^jt_3^j...$ as its input and processes sub-symbol t_i^j in the ith operation cycle. We say tile *j* finds a match of pattern string $S = s_1s_2...s_n$ in operation cycle *i* if $s_1^js_2^j...s_n^j = t_{i-n+1}^jt_{i-n+2}^j...t_i^j$. It is obvious that pattern string *S* is matched by the original AC algorithm in operation cycle *i* if and only if (iff) tile *j* finds a match of *S* in operation cycle *i* for all *j*, $1 \le j \le L/m$. To check whether or not all tiles find matches of the same pattern string in the same operation cycle, the pattern strings are numbered and a partial match vector (PMV) is associated to every state of each sub-trie. The length of the PMV is *f* bits if there are *f* pattern strings. Consider any particular j^{th} sub-trie. The kth bit of the PMV associated to state *x* is a 1 iff pattern string *k* is matched in state *x*. See Figure 3-4 for examples of PMVs.



	00	01	10	11	PMV
0	1	0	0	3	0000
1	1	2	0	3	0000
2	1	0	7	6	1000
3	4	0	0	3	0000
4	1	5	0	3	0000
5	1	0	7	6	1100
6	4	0	0	3	0010
7	1	0	0	8	0000
8	4	0	0	3	0001

Figure 3-4 : The left side is a search engine with 4 tiles. The right side shows the structure of a tile. Each row in the table is a state and each state has 4 possible next state and a PMV. It transits state according to 2-bit input sub-symbol, and output PMV of current state.

1896

Since every symbol is divided into L/m sub-symbols, the number of possible inputs reduces from 2^L for the original trie to 2^m for each sub-trie. Consequently, the space requirement reduces significantly from $O(2^L)$ to $O(2^m L/m)$. The bit-split idea is adopted in our implementation.

3.3 Suffix Based Traversing Algorithm

Suffix Based Traversing (SBT) algorithm was introduced in 2004 that multiple symbols are processed in an operation cycle. To achieve high performance, it processes w symbols concurrently with pipeline where $w = 2^n$. At first, it encodes all useful strings by index numbers. Lookup table, C_2, C_4… are used to calculate the

longest suffix. C_2 returns the index number of the longest suffix of a 2-byte string. With two index numbers of C_2, it can encode the index number of a 4-byte string. It will be encoded into'0'when the symbols are useless for these patterns. It uses index numbers instead of symbols to avoid too many possible inputs while matching.

For all possible positions in which it may match a pattern, it collects all possible k-step transitions for $1 \leq k \leq w$ from every state. It reports a match if any one of the k-step states is a final state. Figure 3-5 shows an example with w=4. The left side of the figure is responsible for indexing the input symbols. The right side looks up the table, NS_1, NS_2, NS_4,…, by index number. And it adopt pipeline to calculate states for 1 to 4 steps transitions.

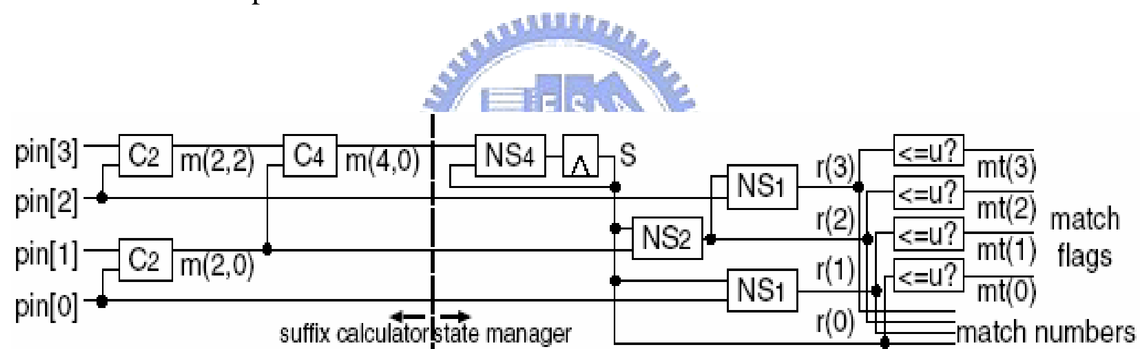


Figure 3-5 : SBT architecture

As we mentioned before, every state needs 256 entries for table lookup while we process an 8-bit symbol at a time. But for 2 or more bytes are processed, it needs 65536 entries or more. So if we index the useful ones, we can save much unnecessary entries for table lookup. Moreover, according to the characteristic of the table, it can be reduced by storing a default transition because many entries are the same for every useful index.

Table size can be reduced using indirect pointers as shown in Figure 3-6. The

2D-table is decomposed into rows, and then packed into the linear array. Each entry of the linear array has a row number to identify the owner row of the entry. When the 2D-table lookup fails, the default value of each column is returned. Note that unused elements may exist in the linear array because of fragmentation of the free area.

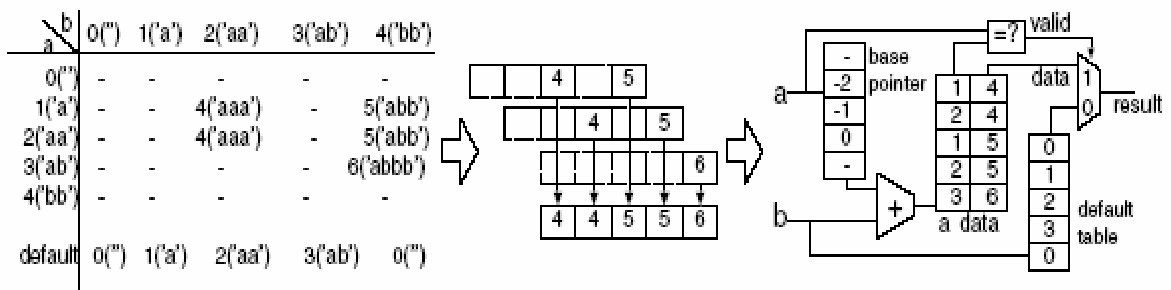


Figure 3-6 : Reducing table size using indirect pointers

Chapter 4

A High-Performance and Memory-Efficient Pattern Matching Algorithm

In this section, we present our proposed high-performance and memory-efficient pattern matching algorithm. In our proposed algorithm, K symbols are processed at a time. It is clear that the original AC algorithm corresponds to $K = 1$. For simplicity, we assume that every symbol is a byte throughout this paper.

4.1 Conception

Take a pattern string, abcdef, for example. How do we find it out quickly in such a long text string with AC algorithm? It's straightforward to use more than one symbol for table lookup per operation cycle. But however the table size will grow increasingly while we have much more possible inputs. In order to solve this problem, the table should be compressed properly. The memory requirement is a problem in original AC algorithm furthermore we have larger lookup table so we have to deal with it pretty well. The basic idea we thought is that not to add any more possible inputs for table lookup. But it can process more than one byte every cycle with original table size in AC. And we can adopt so many table compression researches to solve the problem of memory requirement easily.

We let many scan engines work in parallel, and each one is responsible for one

part of text string according to the position of every byte in text string. It finds a match of a string when all engines find every part of a string. After that, we can try to reassemble all information of every engine's report. For example, the first engine finds "ace" and "bdf" is found by the second engine. The two strings can be reassembled and then we can find it may be a full string "abcdef". But we can not make sure if we only know finding all parts of a string. Figure 4-1 shows the problem that we should check if the text string is matched. How to tell that the string is "abcdef" or "badcfe" is the issue that we'll discuss in detail later. But we approach our goal that every engine processes only one byte every cycle for less table size. And the functionality is equivalent to a huge state machine of processing more than one byte per operation cycle.

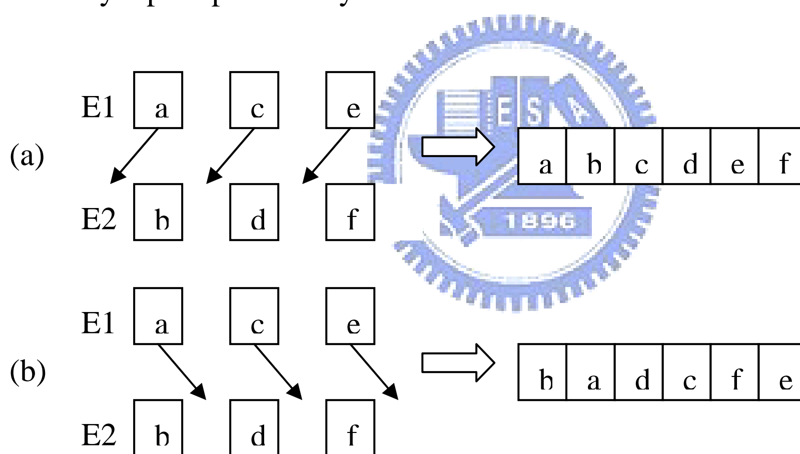


Figure 4-1 : Example of reassembling two parts of a string.

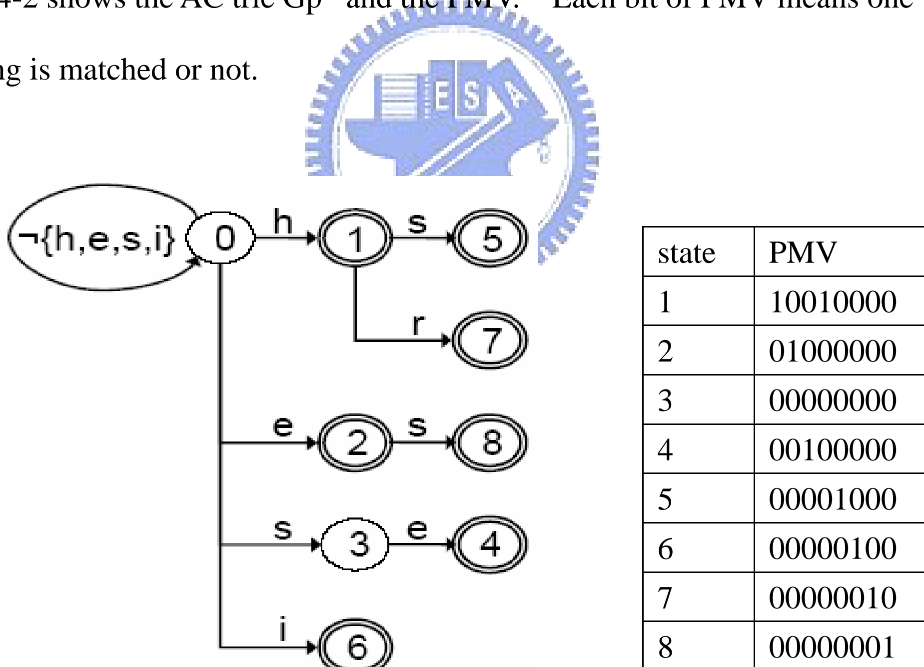
4.2 Our Proposed Algorithm

4.2.1 Pre-process

To achieve high performance, we need to pre-process every pattern string. Let P represent the set of all pattern strings. For every pattern string $S \in P$, we divide S

into K substrings as follows. Assume that $S = s_1s_2...s_n$ and $n = qK + r$, where q is the quotient and r is the remainder of n divided by K. String S is divided into K substrings so that the ith string, denoted by S_i, is given by $s_is_{K+i}...s_{qK+i}$ if $i \le r$ or $s_is_{K+i}...s_{(q-1)K+i}$ if $i > r$. Let P^* be the set of all substrings derived from the strings in set P. The AC trie, denoted by G_{P^*}, is constructed for all the substrings in P^*.

Let's see some example with K = 2 and patterns are he, she, his and hers. But right now, he is divided into h and e, two substrings. And these two states constructed by h and e are both final states. She is divided into se and h and so on. Figure 4-2 shows the AC trie Gp* and the PMV. Each bit of PMV means one substring is matched or not.



state	PMV
1	10010000
2	01000000
3	00000000
4	00100000
5	00001000
6	00000100
7	00000010
8	00000001

Figure 4-2 : Example of trie Gp* for {he, she, his, hers} with k = 2.

Let $T = t_1t_2...t_m$ denote the text string to be scanned and $m = q'K + r'$, where q' and r' are, respectively, the quotient and the remainder of m divided by K. The text string T is also divided into K substrings T_1, T_2, …, and T_K so that the ith text

substring $T_i = t_i t_{K+i} \ldots t_{q'K+i}$ if $i \le r'$ or $t_i t_{K+i} \ldots t_{(q'-1)K+i}$ if $i > r'$. There are K parallel pattern search engines, denoted by E_1, E_2, …, and E_K, each with its own AC trie that is the same as G_{P^*}. Search engine E_i scans text substring T_i.

4.2.2 Match Conditions

There are various conditions for the pattern string S to be matched. As an example, for $r = 0$, search engine E_i finds a match of substring S_i at the end of operation cycle C for all i, $1 \le i \le K$, is one possible condition. As another example, for $r = 1$, pattern string S is matched if search engine E_i finds a match of substring S_{i-1} at the end of operation cycle C for all i, $3 \le i \le K$, and search engines E_1 and E_2 find matches of substrings S_K and S_1, respectively, at the end of operation cycle C+1. The complete conditions for pattern string S to be matched are summarized in Table 4-1. The proof of the correctness of the listed conditions is straightforward and thus is omitted.

1896

Table 4-1. Conditions for pattern string S to be matched.

Cases	Conditions
$q = 0, r > 0$	(a) $\exists l$, $1 \le l \le K$-r+1, E_{l+i} finds a match of S_{i+1}, $0 \le i \le r$-1, at the end of operation cycle C Or (b) $\exists l$, K-r+2 $\le l \le K$, E_{l+i} finds a match of S_{i+1}, $0 \le i \le K$-l, at the end of operation cycle C and E_j finds a match of $S_{K-l+1+j}$, $1 \le j \le r$-K+l-1, at the end of operation cycle C+1

$q > 0$, $r = 0$	(a) E_i finds a match of S_i, $1 \le i \le K$, at the end of operation cycle C Or (b) $\exists l$, $2 \le l \le K$, E_{l+i} finds a match of S_{i+1}, $0 \le i \le K\text{-}l$, at the end of operation cycle C and E_j finds a match of $S_{K-l+1+j}$, $1 \le j \le l\text{-}1$, at the end of operation cycle C+1
$q > 0$, $r > 0$	(a) E_i finds a match of S_{r+i} if $1 \le i \le K\text{-}r$ or S_{i-K+r} if $K\text{-}r\text{+}1 \le i \le K$ at the end of operation cycle C Or (b) $\exists l$, $r\text{+}1 \le l \le K$, E_{l+i} finds a match of S_{i+1+r}, $0 \le i \le K\text{-}l$, at the end of operation cycle C and E_j finds a match of $S_{K-l+1+r+j}$ if $1 \le j \le l\text{-}r\text{-}1$ or $S_{j-l+r+1}$ if $l\text{-}r \le j \le l\text{-}1$ at the end of operation cycle C+1 Or (c) $\exists l$, $2 \le l \le r$, E_{l+i} finds a match of S_{i+1+r} if $0 \le i \le K\text{-}r\text{-}1$ or $S_{i-K+1+r}$ if $K\text{-}r \le i \le K\text{-}l$, at the end of operation cycle C and E_j finds a match of $S_{r+1-l+j}$, $1 \le j \le l\text{-}1$, at the end of operation cycle C+1

Figure 4-3 illustrates an example of all the matches for S = abcdefghij and K = 6. In this example, we have S_1 = ag, S_2 = bh, S_3 = ci, S_4 = dj, S_5 = e and S_6 = f. According to the position of S in text string, there are totally six possible situations for pattern string S to be matched.

	(1)	(2)	(3)	(4)	(5)	(6)
E_1	a g	x f	x e	x d j	x c i	x b h
E_2	b h	a g	x f	x e x	x d j	x c i
E_3	c i	b h	a g	x f x	x e x	x d j
E_4	d j	c i	b h	a g x	x f x	x e x
E_5	e x	d j	c i	b h x	a g x	x f x
E_6	f x	e x	d j	c i x	b h x	a g x

Figure 4-3 : Example of all the matches for S = abcdefghij and K = 6.

Note that the conditions for pattern string S to be matched for the case $q > 0$ and $r > 0$ are complicated. Fortunately, it can be simplified if the substrings are renumbered as follows. Given $n = qK + r$, define $S_i^{'} = S_{(i+r) \bmod K}$. After the renumbering, the conditions for pattern string S to be matched for the case $q > 0$ and $r > 0$ become (a) E_i finds a match of $S_i^{'}$, $1 \le i \le K$, at the end of operation cycle C or (b) $\exists l$, $2 \le l \le K$, E_{l+i} finds a match of $S_{i+1}^{'}$, $0 \le i \le K\text{-}l$, at the end of operation cycle C and E_j finds a match of $S_{K-l+1+j}^{'}$, $1 \le j \le l\text{-}1$, at the end of operation cycle C+1, the same as those for the case $q > 0$ and $r = 0$. It operates well as we rotate the substrings' numbers. As for the case $q = 0$ and $r > 0$, it means that pattern length is smaller than K. The same conditions can be applied as long as every search engine reports a match for null substring S_i, $r+1 \le i \le K$, unconditionally at the end of every operation cycle. As a consequence, we have unified conditions for pattern string S to be matched.

4.2.3 Architecture

Figure 4-4 shows the architecture of our implementation. To process K symbols every operation cycle, we have K search engines and each one is received one byte per operation cycle. The bit-split idea is adopted to reduce memory requirement. So there are $8/m$ state machines (i.e., $m = 2$) for every search engine in our implementation. Each state machine scans m-bit sub-symbols of text string with its own sub-trie.

Note that since each pattern string is divided into K sub-strings, it is necessary to distinguish which sub-string is found when a search engine reports a match. As a result, the length of every PMV is Kf bits if there are f pattern strings. The $(4i+k)^{th}$ bit of a PMV associated to state x is a 1 iff the kth sub-string of pattern string i is matched in state x. As well as that in bit-split AC, sub-string S is matched in a search engine if and only if tile j finds a match of S for all j, $1 \le j \le 8/m$ in that search engine. According to all search engines' outputs, we can verify match conditions as we mentioned in section 4.2.2.

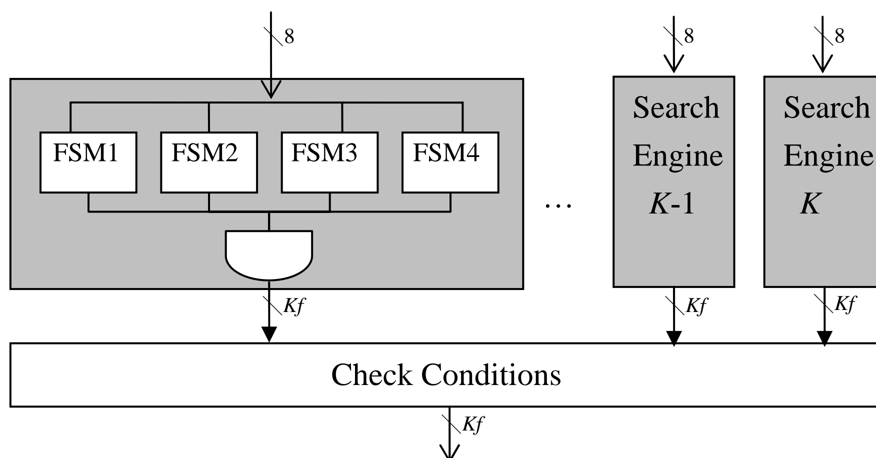


Figure 4-4 : The String Matching Machine Architecture

4.3 Extension of Our Proposed Algorithm

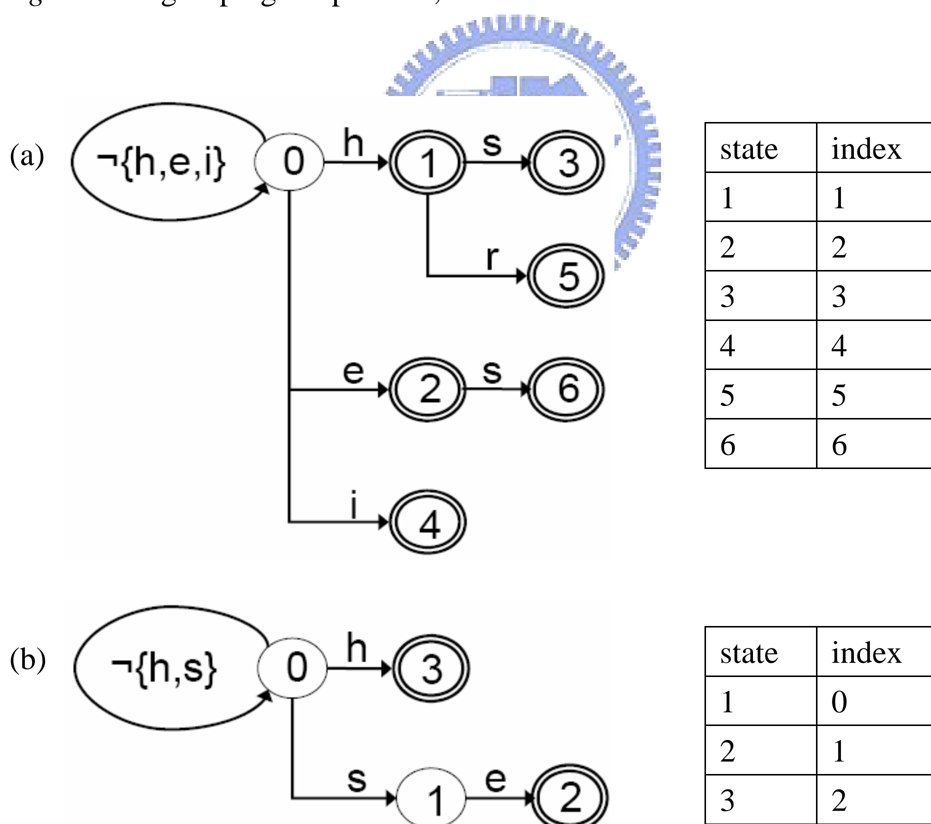
4.3.1 Problem of PMV

To implement our proposed algorithm, we need Kf-bit PMV to indicate every part of every string matched or not. However, we have K search engines and each one consist of $8/m$ finite state machines. We have to access $8K^2f/m$ bits in every operation cycle. It's not a good solution especially when K is large. First it consumes much power to access so many bits every operation cycle. Moreover it needs much wire and logic circuit to verify match conditions. If implementing thousands of patterns, we should stored lookup table in external memory as it may be very large. But the width of bus is not enough completely. It has some problem for implementation in practice.

The simplest way that we can solve the problem is to use index number in stead of PMV. But it has some necessary reasons to use so many bits PMV. As we let the AC trie divided into many sub-tire, each match in one sub-trie is just a part of a string. We have to use all the match information to check so we can not discard any one. However the state in the sub-trie usually represents two or more strings matched partially. Using PMV is the straightforward and simplest answer. To avoid using too many bits in PMV, we group the pattern strings into several sets. We need as many matching machines as the number of sets and every matching machine is responsible for a set of pattern strings.

4.3.2 Output Index

First, we construct our AC trie Gp* as we mentioned before. From the result of first construction, we decide every pattern string to be in which set. If one state indicates more than one string matched partially, we should separate these strings into different sets. To make sure that every pattern string is not in the set of which some state of the state machine represents itself and other strings, we can list all mutual exclusion pattern strings for a pattern string S in exc(S) while pre-processing. According to the exclusion list, we can separate all patterns easily. Roughly, if there are not more than g bits 1 in PMV of a state, the number of sets is g or a little more than g. After grouping the patterns, we can construct a trie for each set.



state	index
1	1
2	2
3	3
4	4
5	5
6	6

state	index
1	0
2	1
3	2

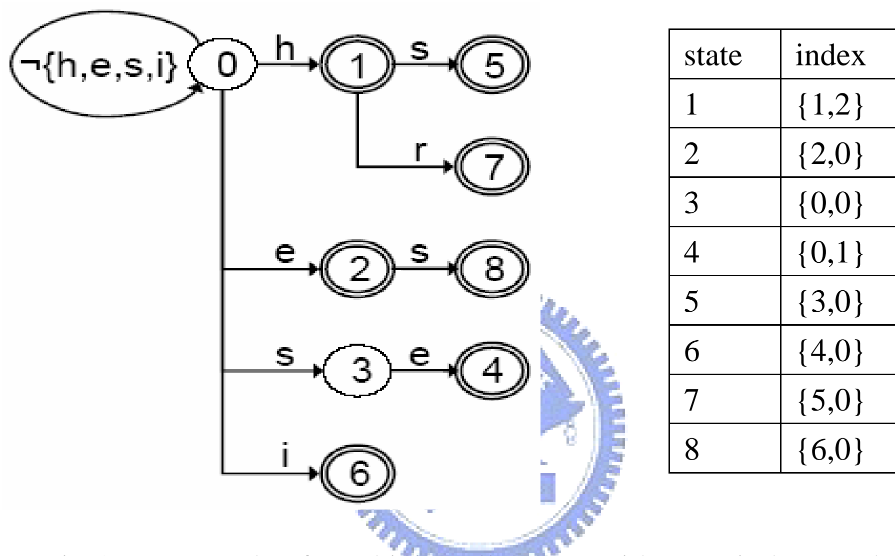
Figure 4-5(a) : The trie and output index for {he, his, hers} and 4-5(b) : The trie and output index for {she}

For example, we can record exc(he)=she and exc(she)=he because state 1 indicates one substring of he and one substring of she concurrently in Figure 4-2. With the exc function listed, he, his, and hers are grouped in the first set and she is grouped in another one. Figure 4-5(a) is the trie constructed for the first set and Figure 4-5(b) is for the second one. PMV for every state can be replaced by a simple index number. We just consider a complete AC trie Gp* for taking a simple example. But adopting bit-split idea, we should list all exclusion correlations depending on all sub-trie. And for simplicity, we ignore the special case that one state indicates more than one part of a string.

No state indicates more than one string is the only rule that we should conform to. We have many workable solutions, but which one we should take is difficult to decide. The basic concept is to let as more states to be reused as possible such that it requires fewer states. Furthermore we let every machine have about the same number of states that will avoid huge amount of memory requirement for some large FSM. For example, we can group he and hers for the first set, and group she and his for the second one. And it requires less memory than that we group in Figure 4-5. Actually g and table size depend on pattern strings and K. The best separating method is not established yet with complex analysis. This problem constitutes a future work.

When K increases, the number of groups increases too. Thus we have so many matching machines. To keep all matching machines active at the same time, we have to access so many bits for next state lookup. To save the number of bits accessed and simplify the architecture, we can combine all the matching machines. In Figure 4-6, the trie is the same as that when patterns are not grouped. But we use

two index numbers instead of PMV by concatenating the index numbers in Figure 4-5(a) and Figure 4-5(b). This problem is just like how we can encode the PMV. Although every state has more bits for index number, the number of total states is less than that of separating the trie. It needs fewer bits of index number than that of PMV whatever g is. However the larger K is, the larger g is, and the more verification circuit we need. We think it's tradeoff between K and g.



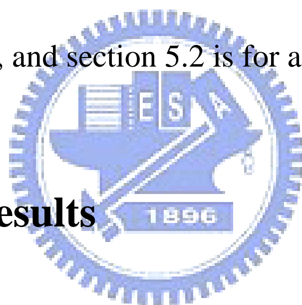
state	index
1	{1,2}
2	{2,0}
3	{0,0}
4	{0,1}
5	{3,0}
6	{4,0}
7	{5,0}
8	{6,0}

Fig 4-6 : Example of combining all groups with more index numbers

Chapter 5

Comparison and Experimental Results

We generated HDL files of our proposed multi-symbol AC pattern matching algorithm for FPGA with given rules. The tables, next function and PMV, are implemented using on-chip 18Kbit Block RAM. We implemented our algorithm with a Xilinx Virtex2 Pro-30 FPGA, and it is evaluated using timing analyzer of Xilinx ISE6.1i. We compare our result with [5] for two rule sets, set 1 of about 1000 characters and set 2 of about 2200 characters. Section 5.1 shows the result of our proposed algorithm with PMV, and section 5.2 is for algorithm with output index.



5.1 Experimental Results

We randomly select 64 pattern strings of length 14-16 bytes from ClamAV [1] virus signatures. Different values of m are tried to compare the memory space requirement. Figure 5-1 compares the memory requirements of our proposed algorithm with the one presented in [5]. It can be seen that our proposed algorithm with m = 1 or 2 in general requires less memory space than the scheme proposed in [5] as long as $K \geq 8$. Moreover, the amount of memory requirement is the least when K = 16.

From Figure 5-1, we can see the amount of memory requirement is least when K = 16. It is mainly because we choose virus signatures of length 14-16 bytes. When

a pattern string is divided into 16 substrings, every substring is of length 0 or 1. As a result, the depth of each sub-trie is only one and many states are reused by different substrings. Therefore, K = 16 requires the least amount of memory space since it is proportional to the number of states.

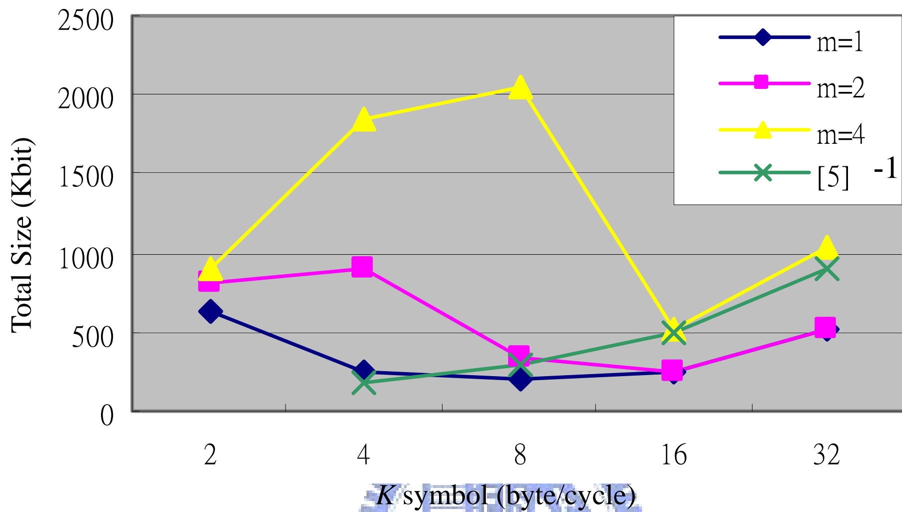


Figure 5-1 : Comparison of total memory requirement

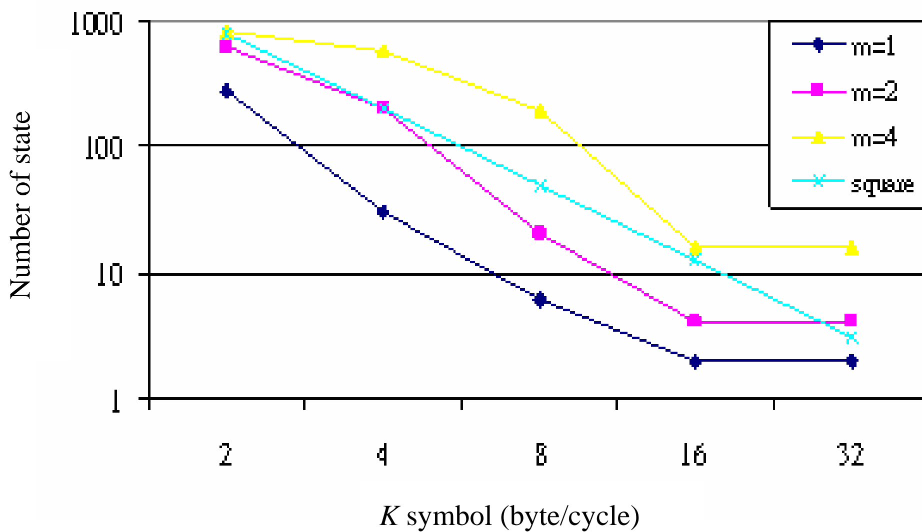


Figure 5-2 : Number of state with different m and K

As we have twice K, we have twice PMV bits and twice search engines. The table should be about 4 times the size. However, the state is also reduced because

each state indicates more number of prefix of rules while the depth of the trie is half. Roughly, the table will be reduced if the number of state becomes less than one fourth. Figure 5-2 shows the number of state with different *K* and *m*. In Figure 5-2, we have a curve, square, for comparison.

When K = 32, the number of state is reduced no more. The table should grow to 4 times while the half of 32-bit PMV is redundant. We use 16-bit PMV instead of 32-bit PMV as we have known each rule is not longer than 16 bytes at first. So the table grows in proportion to *K*. Table size is half of that in [5] when $K \geqq 16$ although we duplicate tables for each search engine.

The clock rate of our synthesized logic can be operated higher than 140 MHz. In other words, our proposed algorithm can easily achieve more than 4.5Gbps throughput with K = 4. Obviously, the throughput can be further improved as long as we increase the value of *K*. The tradeoff is larger space requirement and more complicated verification logic. We believe that K = 8 can be implemented with state-of-art FPGA development platform to achieve throughput higher than 10Gbps.

5.2 Result with Output Index

We have four rule sets for experiment, and each one is consist of 16-byte, 32-byte, 64-byte, 128-byte pattern strings. We randomly select about 2200-byte pattern strings for each rule set from ClamAV virus signatures. Different values of *K* and pattern length are tried to compare the memory space requirements and *g*. Table 5-1 shows the complete result for all situations. It can be seen that it needs the least

memory in 16-byte rule set with $K = 4$, 32-byte rule set with $K = 8$, and 64-byte rule set with $K = 16$. The results of 16-byte rule set with $K = 8$ and 16 and 32-byte rule set with $K = 16$ are not shown in table because g is too large. The exclusion correlations are complicated such that it's difficult to separate the strings because of large g. Furthermore it needs much more verification circuit so we skip the cases.

Table 5-1. The result of our proposed algorithm with output index

	Pattern Length	16	32	64	128
K=4	Number of state	414	1225	1610	1743
	Memory requirement(Kbit)	278	666	759	697
	g	8	3	4	1
K=8	Number of state	x	304	1157	1493
	Memory requirement(Kbit)	x	408	1277	1337
	g	x	8	5	2
K=16	Number of state	x	x	282	948
	Memory requirement(Kbit)	x	x	776	1759
	g	x	x	10	3

Figure 15 compares the memory requirement of our proposed algorithm with the one presented in [5]. It can be seen that our proposed algorithm with output index needs less memory than that proposed in [5] for 16-byte rules with $K = 4$, 32-byte rules with $K = 8$, 64-byte rules with $K = 16$. The curve arises rapidly in [5] as it needs much more tables when K increases. Table size is reduced by decomposing 2D-table into rows, and packing into the linear array. The number of index increases

with *K* such that there are much more possible inputs and table is more difficult to be reduced. But for our result, the curve will decrease in some value of *K* according to the pattern length. Although the performance of result seems case by case, it is quite good if we use proper *K* value. In this case, we use less *K* for less verification circuit such that many states are not reused. Without consideration of verification circuit, we can use larger *K*, and the number of states will be reduced much more.

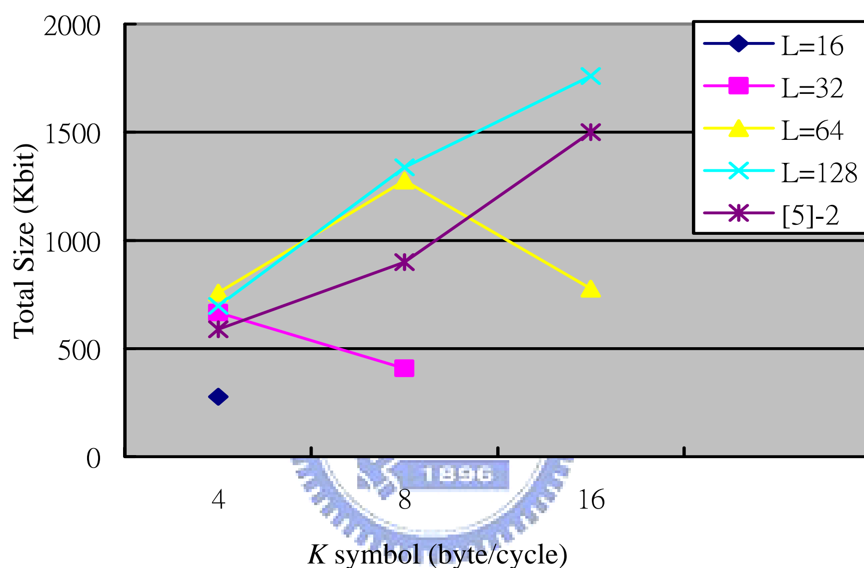


Figure 5-3 : Comparison of total memory requirement

We can optimize for a given rule set according to its pattern length. But for long pattern, we need use large *K* for optimization and we have to duplicate *K* lookup tables. Considering the memory resource, the memory requirement may be too large. To solve this problem, we can process *K/t* bytes per operation cycle with duplicating *K/t* tables. We can register the results and the after *t* cycles we can get the same result as we process *K* byte in an operation cycle. Each table can be reduced well by proper *K* value, but we don't have to duplicate so many tables. As a result it's flexible to deal with any given rule set in all situations.

Chapter 6

Conclusions

In this paper, we have presented an extension of the Aho-Corasick pattern matching algorithm where multiple symbols are processed in an operation cycle. In our proposed scheme, K search engines are employed to scan the input text substrings in parallel to improve system performance. Since every pattern string is divided into K substrings, it is possible for each individual search engine to output a false positive. Thus some verification logic is needed to eliminate false positives. We showed that the match conditions and the verification logic can be simplified if the substrings are appropriately renumbered (if needed). Experiments with Xilinx FPGA development platform reveal that one can achieve more than 4.5Gbps throughput performance with $K = 4$. Compared with a related scheme, our proposed algorithm achieves slightly better throughput performance and requires less memory space.

To implement our proposed algorithm, we need Kf-bit PMV to indicate every part of every string matched or not. We have presented in this paper an extension of our pattern matching algorithm with output index. We group the pattern strings into g sets and we need g matching machines of which each is responsible for a set of pattern strings. We show how to separate pattern strings well and combine all the matching machines for simpler architecture and fewer bits of next state for table lookup. For our experiment result, we show the relations with pattern length, memory requirement, K and g. We get quite good performance in some value of K depending on pattern length which is much better than the one in the related scheme.

Considering the memory resource of hardware, we can duplicate fewer tables by processing it in several cycles. As a result it's flexible to deal with any given rule set in all situations.

At last, we got our goal, not adding any more possible inputs for each table lookup and processing multi bytes every cycle. Dividing a string into many sub-strings makes it possible to process multi bytes per cycle. However it's the top level to improve the structure of AC trie. As we mentioned before, we can adopt any table compression researches to solve the problem of memory requirement.

Bibliography

[1] Clam anti virus signature database, www.clamav.net.

[2] D. E. Knuth, J. H. Morris, and V. R. Pratt, “Fast pattern matching in strings,” TR CS-74-440, Stanford University, Stanford, California, 1974.

[3] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” Communications of the ACM, Vol. 20, October 1977, pp. 762-772.

[4] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” Communications of the ACM, Vol. 18, June 1975, pp. 333-340.

[5] Y. Sugawara, M. Inaba and K. Hiraki, “Over 10Gbps string matching mechanism for multi-stream packet scanning systems,” Field Programmable Logic and Application, Vol. 3203, Sep. 2004, pp. 484-493.

[6] L. Tan and T. Sherwood, “A high throughput string matching architecture for intrusion detection and prevention,” 32nd Annual International Symposium on Computer Architecture, pp. 112-122, 2005

[7] L. Tan and T. Sherwood, “Architectures for Bit-Split String Scanning in Intrusion Detection,” IEEE Micro, Vol.26, pp. 110-117, 2006

[8] M. Roesch, “Snort – lightweight intrusion detection for networks,” Proceedings of LISA’99, 13th Administration Conference, Seattle Washington, USA, 1999

[9] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok, “Avfs: An On-Access Anti-Virus File System,” proceedings of the 13th USENIX Security Symposium, 2004

[10] open source IDS/IPS snort, www.snort.org

[11] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” IEEE Infocom 2004, pp. 333-340.

[12] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, “Deep packet inspection using parallel bloom filters,” Symposium on High-Performance Interconnect (HotI), Stanford, CA, pp. 44-51, Aug. 2003.

[13] T. H. Lee and J. C. Liang, “A high-performance memory-efficient pattern matching algorithm and its implementation,” IEEE Tencon, Hong-Kong, 2006.