國 立 交 通 大 學

電信工程研究所

碩 士 論 文

深度封包檢測使用進階 Aho-Corasick 演算法

Deep Packet Inspection with The Enhanced Aho-Corasick Algorithm

研究生：机奕璉

指導教授：李程輝 教授

中 華 民 國 九十九 年 六 月

深度封包檢測使用進階 Aho-Corasick 演算法

Deep Packet Inspection with The Enhanced Aho-Corasick Algorithm

研 究 生： 机奕璉
指導教授： 李程輝 教授

Student: Yi-Lien Chi
Advisor: Prof. Tsern-Huei Lee

國 立 交 通 大 學

電 信 工 程 研 究 所

碩 士 論 文

A Thesis
Submitted to Institute of Communication Engineering
College of Electrical and Computer Engineering
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Communication Engineering
June 2010
Hsinchu, Taiwan, Republic of China.

中 華 民 國 九 十 九 年 六 月

深度封包檢測使用進階 Aho-Corasick 演算法

學生: 机奕璉　　指導教授: 李程輝 教授

國立交通大學

電信工程研究所碩士班

中文摘要

因為字串比對的準確性，使其技術近年來被廣泛運用到網際網路應用上，其中，Snort 為最具彈性與精確性的偵測軟體之一。Snort 是一套開放原始碼的網路入侵預防與入侵檢測軟體，使用以特徵值(signature-based)和通訊協定的偵測方式，加上 Snort 規則語言(rules language)，搭配正規表示式(Perl compatible regular expression-PCRE)資料庫透過正規表示式字串比對，來達到流量封包辨識目的。其不僅單純檢測網路封包的表頭(header)，更依據封包內容(payload)做比對，檢查其是否與所設定的網路安全規範一致，這過程稱深度封包檢測(deep packet inspection)，效果會比傳統偵測方式僅檢測封包表頭更具安全性。有一著名正規表示式比對的演算法稱 Aho-Corasick 演算法，不僅可以同時比對多字串並保證在各情形下有合理的效能。我們提出一個方法延伸 Aho-Corasick 演算法，可以將 Snort PCRE 部分，依其特徵規則式有系統地建造特徵正規表示式比對圖，實驗數據顯示可得到合理的效能及較少的記憶體需求量。

關鍵字：深度封包檢測、網路安全、字串比對、正規表示式

Deep Packet Inspection with The Enhanced Aho-Corasick Algorithm

Student: Yi-Lien Chi Advisor: Prof. Tsern-Huei Lee

Institute of Communication Engineering
National Chiao Tung University

Abstract

Snort is an open source and free network intrusion prevention system (NIPS) and network intrusion detection system (NIDS) clever of performing packet logging and real-time traffic analysis on IP networks. Snort can also deal with deep packet inspection (DPI) which is an effective security measure that checks not only the packet headers but also the packet content. It uses Perl Compatible Regular Expression (PCRE) library for checking regular expressions which is replacing explicit string patterns as the pattern matching language of choice in many deep packet scanning applications. For regular expression, there is a famous pattern matching algorithm named Aho-Corasick (AC) which can match multiple patterns simultaneously and guarantee deterministic performance under all circumstances. We provide a method to extend the AC algorithm, and use this scheme to systematically construct a signature matching system which can indicate the ending position in a finite input string for the occurrence of Snort rules signatures that are specified by regular expressions. Use extended AC algorithm on Snort PCRE yields acceptable throughput performance and memory requirement.

Keywords: deep packet inspection, network security, string matching, regular expression

誌　謝

誠摯的感謝指導教授 李程輝老師，在研究所的求學過程中悉心地指導我，使我在兩年的研究所生涯獲益匪淺。在您的教誨下，我學習到了做研究應有的態度和嚴謹的思維，在做研究和撰寫論文的過程中，成長不少。您對學問的嚴謹態度更是學生做學問的優良典範；且您平時的親和力及幽默感，更拉近了師生間的距離，讓實驗室氣氛溫暖又歡樂，開心自己可以加入這個團體。

感謝NTL實驗室，博士班-迺倫學姊、孟諭學長、郁文學長、景融學長、瑋哥學長；已畢業學長-大頭、鈞鈞、松松、阿信和丹奇；同窗-韋儒、菜人、KV、曉薇、阿祥、熊仔、阿倫；學弟妹們一大票真的列舉不完；朋友大餅、蔣阿蕾、邱阿智、程敬智、李天琴。感謝各位在我的課業、生活以及研究上不吝嗇地給我最大的指教和關懷，使我一路茁壯，碩士生涯過的很充實愉快，充滿了各種美好的回憶，謝謝每一個曾經伴我成長的戰友們，我以你們為傲。

最後，更要特別感謝我的父親机德茂先生與母親周秀珍女士，謝謝你們對我從小無微不至的養育照顧與支持，讓我可以無後顧之憂地完成學業。感謝我的兄長机亮燁先生以及各位親朋好友，謝謝你平時對我的關懷和勉勵，也謝謝男友黃郁文，默默地支持我、陪伴我、鼓勵我。由衷的感謝每一個為我付出的人，因為你們，才能讓我求學之路走得堅定踏實，我的成就與驕傲全因您們而得，將一切的榮耀都奉獻給大家!

謹將此論文獻給所有愛我與我愛的人

2010年6月 於風城交大

Contents

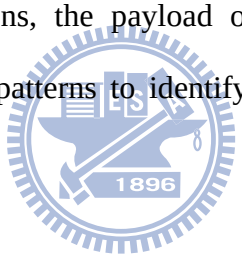
List of Tables

List of Figures

Chapter 1.

Introduction

From before until now, most security detection device only can examine the packet headers, so Layer-7 intrusions can go through these device undetected. For such problems, deep packet inspection (DPI) is an effective security measure checks which not only the packet headers but also the packet content. Packet content scanning (also known as Layer-7 filtering or payload scanning) is very important to network intrusion detection system (NIDS) and network intrusion detection prevention (NIDP) applications. In these applications, the payload of packets in a traffic stream is matched against a given set of patterns to identify specific classes of applications, viruses, and protocol definitions.



Snort is an open source and free network intrusion prevention system (NIPS) and network intrusion detection system (NIDS) clever of performing packet logging and real-time traffic analysis on IP networks. It can also deal with deep packet inspection (DPI) which is an effective security measure checks which not only the packet headers but also the packet content.

Currently, regular expression used to specify virus signatures are often simple ones and flexibility for describing information than exact strings, so it is replacing explicit string patterns as the pattern matching language of choice in many deep packet scanning applications.

According to [1], the deep packet inspection are the most expensive parts of Snort (a popular open source IDS) [2], accounting for 21% and 31% of the execution time. In [3], there is a table to show that memory requirements using traditional ways, which are prohibitively high for many patterns used in packet scanning applications. I will list the table out in Chapter 2. The Snort-like systems are usually specified the signatures using simple rule-based language. So, the IDS use a scheme to check whether any rule matches an incoming packet. The concept of Snort will be reviewed roughly in Chapter 2.

Much research has focused on improving the performance of signature matching component of Snort. Snort uses Perl Compatible Regular Expression (PCRE) library for checking regular expressions. The regular expressions are also checked for the rules whether string matching has succeeded.

When security attacks become more complicated, regular expressions are much more expressive than plain strings were used to specify their signatures. It is well known that a regular expression can be recognized with a non-deterministic finite automaton (NFA), which can be transformed into a deterministic finite automaton (DFA) so it is equivalent. There are some famous algorithms [4], [5] to construct an NFA recognizing a given regular expression. However, NFA-based solutions are often inefficient on a processor with limited parallelism. Hardware accelerators were proposed to achieve high performance [6].

To be aimed at regular expression, there has a famous pattern matching algorithms named Aho-Corasick (AC). The AC algorithm can match multiple patterns simultaneously and guarantee deterministic performance under all circumstances.

Besides, we provide a method to extend the AC algorithm and use this scheme to systematically construct a signature matching system which can indicate the ending position in a finite input string for the occurrence of Snort rules signatures that are specified by regular expressions. The scheme of AC algorithm and the extend AC algorithm will be sketched briefly in Chapter 3.

In [7], an idea is similar to the failure transition of the AC algorithm, which was proposed to reduce the number of state transitions. In this way, the space requirement of a DFA is also reduced. Although the idea works for selected sets of regular expressions, it still has the risk of resulting in a huge number of states. Therefore, the purpose of the method to extend the AC algorithm is to present a high-performance, reasonable memory requirement signature matching system for simple regular expressions and plain strings that can be efficiently implemented on general-purpose processors.

The rest of this paper is organized as follows. In Chapter 2, we introduce some background about Snort and regular expression. In Chapter 3, we review the related works, which is about how the enhanced Aho-Corasick algorithm works. In Chapter 4, we present the PCRE handling with our proposed enhanced Aho-Corasick algorithm. Experimental results are provided in Chapter 5. Finally, we draw conclusion in Chapter 6.

Chapter 2.

Background

2.1. Snort Overview

2.1.1. Operation Mode

Snort with intrusion detection related has four modes:

1. **Sniffer mode**

   Sniffer the packets content in network, and display the packets content on monitor.

2. **Packet Logger mode**

   Record the sniffer packets content into hard disc.

3. **Network Intrusion Detection System mode (NIDS mode)**

   Analyze the packets content. If there has matched the rules which is made by user, it will take reaction.

4. **Inline mode**

   Capture the packets from Iptables instead from Libpcap. If these packets matched Snort rules, this rules corresponding reaction then act to let these packets pass or throw away.

2.1.2. Snort Operation Architecture

Snort system architecture has four parts and shown in Figure 1.

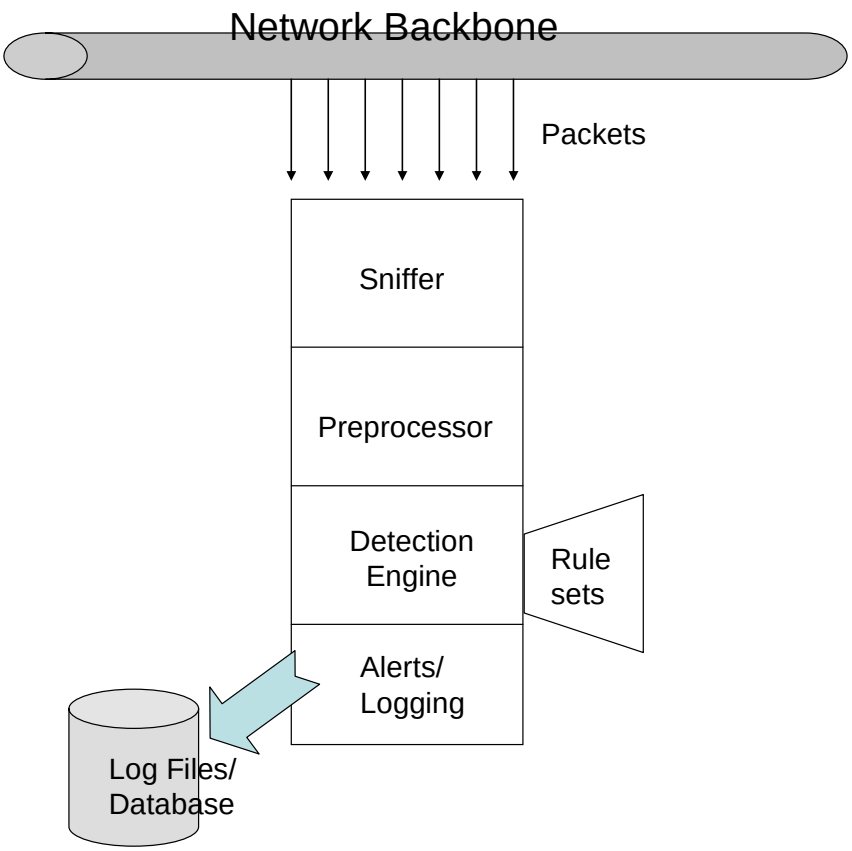


Figure 1. Snort system architecture.

1. **Sniffer**

   Detect and capture packets.

2. **Preprocessor**

   Base on TCP/IP protocol to filter the packets and to analysis the reassociated packets. Snort is used 'Libpcap' to capture the network packets, and can set the packets filter to catch designated packets.

3. **Detection engine**

   Snort system take the detection rules to form a tow dimension linking structure, and use inserted way to organize rule library, which means to divide intrusion

behavior into different parts.

4. **Alerts / Logging system**

   When intrusion detection system detect the threat, it will alarm and record in log file. The IDS use TCPDUMP form to record the alarm message, and send the alarm message to Syslog to notify network security management.

2.1.3. Snort Rule Language

According to [8], following will introduce the rule language of Snort. To specify signatures, Snort uses a simple rule-based language. Snort signatures are written in a configuration file which is read when Snort starts up. After starting up, the signature file consists of several variable declarations and rules, and the value of the variable is instated in the rules for signature matching. The rules consist of a *rule header* and a *rule body* in Figure 2.

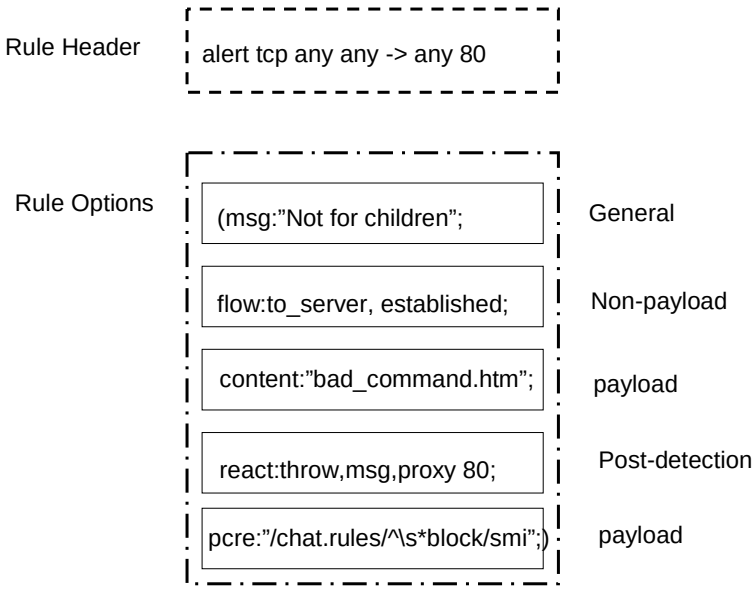


Figure 2. Snort rule header and rule body example.(From [9])

◆ **Rule Header**

The rule header consists of *action*, *protocol*, *ip addresses*, *ports*, and *direction operator*.

- Rule actions

  Specify the action like alerting or logging that Snort should perform when a rule matches a packet. Common action is in following:

  1.alert : provide warning message and log in file.

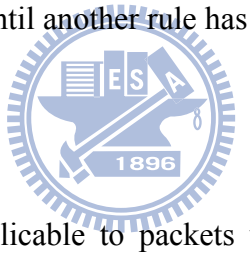
  2. log : record packets

  3.pass : ignore packets

  4.drop : notify iptables and throw the packets awey

  5.activate : provide warning message and act another rule

  6.dynimic : wait until another rule has been executed



- Rule protocols

  Each rule is applicable to packets which belonging to a particular protocol like TCP, UDP, ICMP, or IP.

- Rule IP and port

  According to TCP and UDP rules, the header defined the source and destination ip addresses and port fields for which the rule is to be applied.

  Snort uses *any* for one of these field means that the rule will match for any value in a packet. In other words, *any* can mean arbitrary addresses or determined addresses. For example, 140.113.13.118. Also, Snort rules can use '!' to indicate 'not' what kind of network ip addresses. For example, !140.113.13.0/24 is indicate not from

140.113.13.1 to 140.113.13.255 this range ip addresses.

Port can present in many way. If use *any* means arbitrary port, and assigned port like telnet port is 23 and http port is 80 so on. Snort rule also have ':' to present designated port range. Following have three instances:

1.

log udp any any -> 140.113.13.0/24 1:1024 log udp

This means traffic coming from any port and destination ports ranging from 1 to 1024.

2.

log tcp any any ->140.113.13.0/24 :3000

This means log tcp traffic from any port going to ports less or equal to 3000.

3.

log tcp any :1024 -> 140.113.13.0/24 20:

This means log tcp traffic from privileged ports less than or equal to 1024 going to ports greater than or equal to 20.

- Rule direction

The fields to the left of the direction operator (->) are the source fields, while the ones on the right hand side are for the destination. An alternative operator which is called bidirectional operator (<>), indicates that the rule is to be applied to both directions of the flow.

For example:

alert tcp 140.113.13.118 80 -> 140.113.13.0/24 any

This 140.113.13.118 is source ip address, and 80 is source prot.

The direction operator '->' means the packet is from left to right. This 140.113.13.0/24 is destination ip address, and *any* is destination port.

◆ **Rule Options**

Rule options are the most important parts of Snort intrusion detection engine. There are four classifications as following and in Table 1:

- general

  Provide information that related to the rule, and this option has no relationship with intrusion detection.

- payload

  Matching the content in packets.

- non-payload

  Matching all protocol fields.

- post-detection

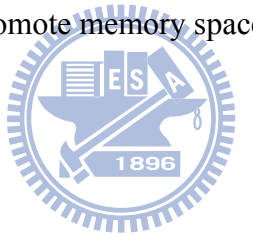
  When packets content match Snort rules, it will take other reaction.

Table 1. Rule option keywords (From[10])

Type	Keywords
general	msg、reference、gid、sid、rev、classtype、priority、metadata
payload	**content**、nocase、rawbytes、depth、offset、distance、within、http_client_body、http_cookie、http_header、http_method、http_uri、fast_pattern、uricontent、urilen、isdataat、**pcre**、bype_test、byte_jump、

	ftpbounce、asn1、cvs
non-payload	fragoffet、ttl、tos、id、ipopts、fragbits、dsize、flags、flow、flowbits、seq、ack、window、itype、icode、icmp_id、icmp_seq、rpc、ip_proto、sameip、stream_size
post-detection	logto 、session 、resp 、react 、tag 、activates 、activated_by、count

In rule option keywords, the most significant words are *content* and *pcre*, which are concerned with whether regular expression string matching is precise or not. According to this reason, we focus on *pcre* to achieve regular expression matching scheme using our algorithm to promote memory space and throughput.



2.2. Regular Expression Overview

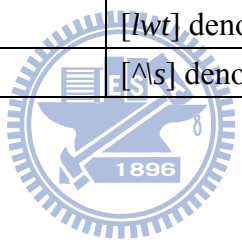
2.2.1 Regular Expression Patterns

Regular expressions also referred to as *regex* or *regexp*, which provide a brief and flexible meaning for matching strings from text, such as particular characters, words, or patterns of characters. A regular expression describes a set of strings without enumerating them explicitly, and it is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

In addition, regular expression, often called a *pattern*, is an way that describes a set of strings. They are usually used to give a concise description of a set, without having to list all elements. According to [3], Table 2 lists the common features of regular expression patterns used in packet payload scanning. For example, take consideration to a regular expression from the Linux L7-filter [11] for detecting Yahoo traffic: "^(*ymsg*|*ypns*|*yhoo*).?.?.?.?.?.?.?[*lwt*].**xc0**x80*". This pattern matches any packet payload that starts with *ymsg, ypns,* or *yhoo*, followed by seven or fewer arbitrary characters, and then a letter *l, w* or *t*, and some arbitrary characters, and finally the ASCII letters *c0* and *80* in the hexadecimal form.

Table 2. Features of Regular Expressions

Syntax	**Meaning**	**Example**
^	Pattern to be matched at the start of the input	^XY means the input starts with XY. A pattern without '^', e.g., XY, can be matched anywhere in the input.
|	OR relationship	*X*|*Y* denotes X or *Y*.
.	A single character wildcard	
?	A quantifier denoting one or less	W? denotes W or an empty string.
*	A quantifier denoting zero or more	*W** means an arbitrary number of Ws.
{}	Repeat	*Q*{100} denotes 100 *Q*s.
{m,n}	Matches the preceding element at least *m* and not more than *n* times.	Z{3,5}denotes ZZZ, ZZZZ, or ZZZZZ
[]	A class of characters	[*lwt*] denotes a letter *l*, *w*, or *t*.
[^]	Anything but	[^\s] denotes any character except \s.



Metacharacters mean escaped with a backslash is reversed for some characters in the POSIX Extended Regular Expression (ERE) syntax. In Table 3, it will list extended regular expression symbol and meaning. A backslash causes the metacharacter to be treated as a literal character. Additionally, support is removed for *n* backreferences.

Table 3. Features of Extended Regular Expression

Syntax	**Meaning**	**Example**
+	Matches the preceding element one or more times.	op+ matches "op", "opp", "oppp", and so on.

2.2.2 Regular Expression Patterns Using DFA Space

For regular expressions, finite automata are a natural formalism. Here are two main categories: *Deterministic Finite Automaton* (DFA) and *Nondeterministic Finite Automaton* (NFA).

A DFA consists of a finite set of input symbols, which denoted as Σ, a transition function δ [12], and a finite set of states. Σ contains the 2^8 symbols from the extended ASCII code in networking applications. Beside the states, there is a single start state Q_0 and a set of accepting states, which we call final state. The transition function δ takes an input symbol and a state as functions and returns a state. A major feature of DFA is that at any time, there is only one active state in the DFA. But an NFA works multiple states simultaneously. Otherwise, an NFA similar to a DFA except that the δ function, maps from a state and a symbol to a set of new states.

2.2.3 DFA Analysis for Snort PCRE Parts

In this section, we introduce the regular expressions used in deep packet payload scanning by[13]. Snort assumed the Perl-compatible regular expression (PCRE) syntax. More precisely, this present the features of the regular expressions included in Snort IDS. For example, alert tcp any ->(pcre:"/^PASS\s*\n/smi";) is a Snort rule and has introduced in section 2.1.3. Based on the above rule, Snort will detect any packet with payload including a string that matches the "/^PASS\s*\n/smi" regular expression. Because "/^PASS\s*\n/smi" is the content of packets, this means deep packet inspection (DPI). Take from the famous features of a strict definition of regular expressions, PCRE have more features such as constrained repetitions and several flags. Table 4 lists the PCRE basic syntax supported by our regular expression pattern matching algorithm.

Table 4. Snort-PCRE Basic Syntax (From [13])

Feature	Description
a	All ASCII characters, excluding meta-characters, match a single instance of themselves
[\^$.—?*+()	Meta-characters. Each one has a special meaning
.	Matches any character except new line
\?	Backslash escapes meta-characters, returning them to their literal meaning
[abc]	Character class. Matches one character inside the brackets. In this case, equivalent to (a|b|c)
[a-fA-F0-9]	Character class with range.
[^abc]	Negated character class. Matches every character except each non-Meta character inside breackets.
RegExp*	Kleene Star. Matches zero or more times the regular expression.
RegExp+	Plus. Matches one or more times the regular expression.
RegExp?	Question. Matches zero or one times the regular expression.
RegExp{N}	Exactly. Matches N times the regular expression.
RegExp{N, }	AtLeast. Matches N times or more the regular expression.
RegExp{N,M}	Between. Matches between N and M times the regular expression.
\xFF	Matches the ASCII character with the numerical value indicated by the hexadecimal number FF.
\000	Matches the ASCII character with the numerical value indicated by the octal number 000.
\d, \w and \s	PCRE Shorthand character classes matching digits 0-9, word characters (letters and digits) and whitespace, respectively.
\n, \r and \t	Match an LF character, CR character and a tab character respectively.
(RegExp)	Groups regular expressions, so operators can be applied.
RegExp1RegExp2	Concatenation. Regular Expression 1, followed by Regular Expression 2
RegExp1 | RegExp2	Union. Regular Expression 1 or Regular Expression 2.
^RegExp	Matches Regular Expression 1 only if at the beginning of the string.
RegExp$	Dollar. Matches Regular Expression only if at the end of the string.
(?=RegExp), (?!RegExp), (?<=text), (?<!text)	Lookaround. Without consuming characters, stops the matching if the RegExp inside does not match.
(?(?=RegExp) then |else)	Conditional. If the lookahead succeeds, continues the matching with the then RegExp. If not, with the else RegExp.
\1, \2. . . \N	Backreferences. Have the same value as the text matched by the corresponding pair of capturing parethesis, from 1st through Nth.

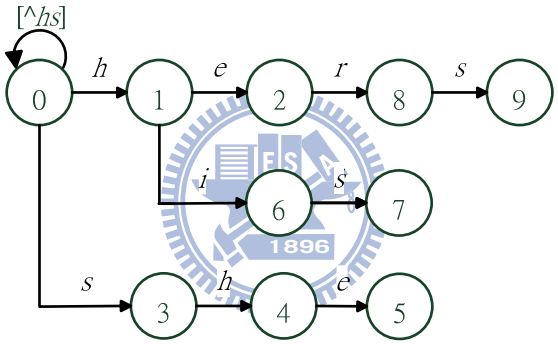
Flags	Description
i	Regular Expression becomes case insensitive.
s	Dot matches all characters, including newline.
m	^ and $ match after and before newlines.

Chapter 3.

Related Works

3.1. The Aho-Corasick Algorithm

The Aho-Corasick (AC) algorithm is dictated by three functions: a goto function *g*, a failure function *f*, and an output function *output*. Figure 3 shows the three functions for the pattern set *Y* = {*he*, *she*, *his*, *hers*} [7][14][15].



(a)

R	1	2	3	4	5	6	7	8	9
f(*R*)	0	0	0	1	2	0	3	0	3

(b)

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

(c)

Figure 3. (a) goto function, (b) failure function, and (c) output function for *Y* = {*he*, *she*, *his*, *hers*}.

Some definitions are needed. Let S_1S_2 represent concatenation of strings S_1 and S_2. We say S_1 is a prefix and S_2 is a suffix of the string S_1S_2. Moreover, S_1 is a proper prefix if S_2 is not empty. Likewise, S_2 is a proper suffix if S_1 is not empty. String S is said to represent state *P* on a goto graph if the shortest path from the start state to state *P* spells out S. Throughout this paper, the representing string of state P is denoted by S^P. The start state is represented by the empty string ε. The length of string *S* is denoted by $|S|$.

Note that there might be a self-loop at the start state of a goto graph. However, it becomes a tree after removing the self-loop, if exists. In the following definitions, we ignore the self-loop. We call state *P* the parent of state *R* and state *R* the child of state *P* if there exists a symbol σ such that $g(P,\sigma)=R$. State R is said to be a descendent of state *P* and state P an ancestor of state R if S^P is a proper prefix S^R. The tree which consists of state P and all its descendant states is called the sub-tree of *P*.

One state, numbered 0, is designated as the start state. The goto function *g* maps a pair (state, input symbol) into a state or the message *fail*. For the example shown in Figure 3, we have *g*(0, *h*) = 1 and $g(1,\sigma)$ = *fail* if σ is not *e* or *i*. State 0 is a special state which never results in the *fail* message. With this property, one input symbol is processed by the AC algorithm in every operation cycle.

The failure function *f* maps a state into a state and is consulted when the outcome of the goto function is the *fail* message. We have *f*(*P*) = *R* iff S^R is the longest proper suffix of S^P that is also a prefix of some pattern. The output function maps a state

into a set (could be empty) of patterns. The set *output*(*P*) contains a pattern if the pattern is a suffix of S^P.

Let *P* be the current state and σ the current input symbol. Also, let X denote the input string. Initially, the start state is assigned as the current state and the first symbol of X is the current input symbol. An operation cycle of the AC algorithm is defined as follows.

1. If $g(P, \sigma) = R$, the algorithm makes a state transition such that state *R* becomes the current state and the next symbol in X becomes the current input symbol. If $output(R) \neq \varnothing$, the algorithm emits the set *output*(*R*). The operation cycle is complete.
2. If $g(P, \sigma) = fail$, the algorithm makes a failure transition by consulting the failure function *f*. Assume that $f(P) = R$. The algorithm repeats the cycle with *R* as the current state and σ as the current input symbol.

The procedures to construct the goto, failure, and output functions are described in Algorithms AC1 and AC2 below [7]. The goto function and the failure function are constructed in Algorithms 1 and 2, respectively. The output function is partially constructed in Algorithm 1 and completed in Algorithm 2.

Algorithm AC1. Construction of the goto function.

Input. Set of keywords $Y = \{y_1, y_2, ..., y_k\}$.

Output. Goto function *g* and a partially computed output function *output.*

Method. We assume *output*(*P*)=∅ when state *P* is first created, and $g(P, \sigma) = fail$ if σ is undefined or if $g(P, \sigma)$ has not yet been defined. The procedure *enter*(*y*) inserts into the goto graph a path that spells out *y*.

begin

newstate ← *0*

for *i* ← 1 until *k* **do** $enter(y_i)$

for all σ such that $g(0, \sigma) = fail$ **do** $g(0, \sigma) \leftarrow 0$

end

procedure $enter(a_1 a_2 ... a_m)$:

begin

$state \leftarrow 0; j \leftarrow 1$

while $g(state, a_j) \neq fail$ **do**

begin

$state \leftarrow g(state, a_j)$

$j \leftarrow j + 1$

end

for *p* ← *j* **until** *m* **do**

begin

newstate ← *newstate* + 1

$g(state, a_p) \leftarrow newstate$

state ← *newstate*

end

$output(state) \leftarrow \{a_1 a_2 ... a_m\}$

end

Algorithm AC2. Construction of the failure function.

Input. Goto function *g* and output function *output* from Algorithm 1.

Output. Failure function *f* and output function *output*.

Method.

```
begin
    queue ← empty
    for each σ such that g(0, σ) = P ≠ 0 do
        begin
            queue ← queue ∪ {P}
            f(P) ← 0
        end
while queue ≠ empty do
        begin
            let R be the next state in queue
            queue ← queue - {R}
            for each σ such that g(R, σ) = P ≠ fail do
                begin
                    queue ← queue ∪ {P}
                    state ← f(R)
                    while g (state, σ) = fail do state ← f(state)
                    f(P) ← g(state, σ)
                    output(P) ←output(P) ∪ output(f(P))
                end
        end
end
```

3.2. Enhancing The Aho-Corasick Algorithm

Let RE_1, RE_2, …, and RE_n be n regular expressions that contain * operators only. Further, let RE_{n+1}, RE_{n+2}, …, and RE_{n+m} be m regular expressions, each of them contains at least one $\{min, max\}$ operator. We construct in this section the signature matching system for RE_1, RE_2, …, RE_n, RE_{n+1}, RE_{n+2}, …, and RE_{n+m}. An important fact in finding a match for $RE = RE^1 * RE^2$, where RE^1 and RE^2 are plain strings or simple regular expressions, is that, once RE^1 was matched before, a match of RE is found if RE^2 is matched. Therefore, we need to remember whether or not RE^1 was matched before. We use different goto graphs to implicitly memorize such information. Our proposed signature matching system consists of a pre-filter and a verification module, which are described separately below. With a pre-filter, the space complexity is largely reduced and the throughput performance can be significantly improved.

3.2.1. Pre-filter

The pre-filter is designed based on the well-known Bloom filters which guarantee no false negative. Given block size k, there are m-k+1 membership query module. Recall that $p_i^1 p_i^2 ... p_i^m$ are the first m symbols of pattern P_i. The sub-string $p_i^1 p_i^2 ... p_i^k$ is a member stored in the first membership query module, the sub-string $p_i^2 p_i^3 ... p_i^{k+1}$ is a member stored in the second membership query module, …, and the sub-string $p_i^{m-k+1} p_i^{m-k+2} ... p_i^m$ is a member stored in the $(m-k+1)^{th}$ (or the last) membership query module. For convenience, these membership query modules are denoted by MQ_1, MQ_2, …, and MQ_{m-k+1}. The h^{th} bit of MQ_j is set to 1 iff there

exists pattern P_i such that $h = hash(p_i^j p_i^{j+1} ... p_i^{j+k-1})$. Every membership query module reports 1 if the query result is positive or 0 otherwise.

Again, a search window W of length m is used during scanning. Initially, W is aligned with T so that the first symbol of T, i.e., t_1, is at the first position of W. The last k symbols in W, i.e., $t_{m-k+1}t_{m-k+2}...t_m$ at this moment, are used to query MQ_1, MQ_2, …, and MQ_{m-k+1}. Let qb_i be the report of MQ_i and $QB = qb_1 qb_2 ... qb_{m-k+1}$ denote the bitmap of current query result. We observe that not only current query result but also previous ones are useful for filtering. Therefore, we introduce the stateful concept in pre-filter design. That is, current query result and previous ones are utilized to determine how many symbols in the text can be skipped in our pre-filter design. Note that no additional queries are required to implement the stateful concept. In our implementation, we use a master bitmap of size $m-k+1$ bits to accumulate results obtained from previous queries. Let $MB = mb_1 mb_2 ... mb_{m-k+1}$ represent the master bitmap. Initially, the master bitmap contains all 1's, i.e., $mb_i = 1$ for all i, $1 \le i \le m-k+1$. After a query result is fetched, we perform $MB = MB \oplus QB$, where $\oplus$ is the bitwise AND operation. A suspicious sub-string is found and the verification engine is consulted if $mb_{m-k+1} = 1$. The advancement of W is $m-k+1$ positions if i $mb = 0$ for all i, $1 \le i \le m-k+1$ positions if $mb_r = 1$ and $mb_i = 0$ for all i, $r < i \le m-k$. If W is decided to be advanced by g positions, MB is right-shifted by g bits and filled with 1's for the holes left by the shift. Figure 4. shows the architecture with master bitmap (stateful) for $m = 6$ and $k = 3$.

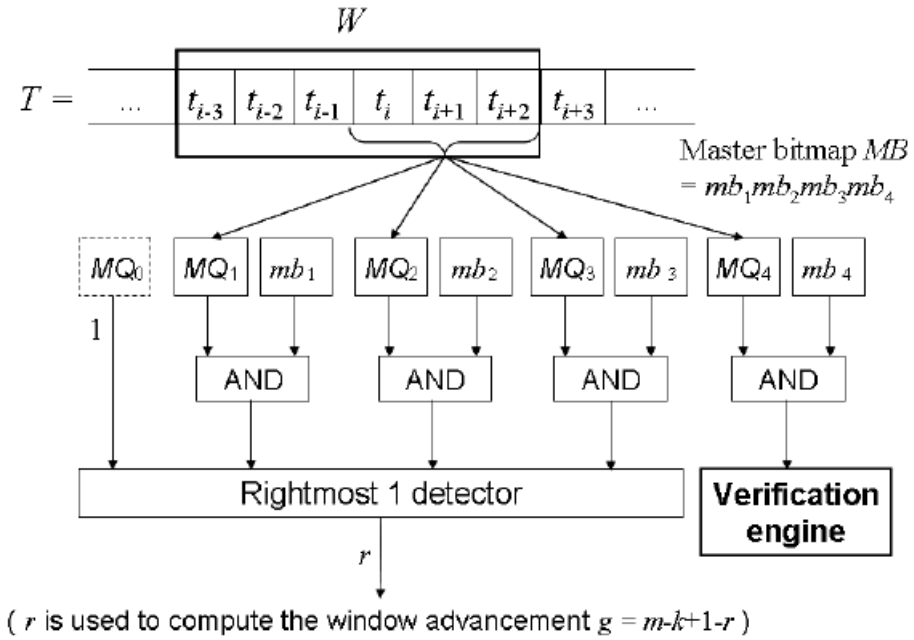


Figure 4. The stateful pre-filter architecture for $m = 6$ and $k = 3$.

3.2.2. Verification Module

The verification module is an extension of the AC algorithm. We describe constructions of the goto function, the failure function, the output function, and the signature matching machine separately.

- **The goto function**

A regular expression which contains at least one $\{min, max\}$ operator is fragmented by the $\{min, max\}$ operators. For example, regular expression $RE = S_1 * S_2 * S_3 \{min_1, max_1\} S_4 * S_5 \{min_2, max_2\} S_6$ is fragmented into $S_1 * S_2 * S_3$, $S_4 * S_5$, and S_6. Let re_{n+k}, $1 \le k \le m$, represent the first fragment of RE_{n+k} and $Y = \{RE_1, ..., RE_n, re_{n+1}, ..., re_{n+m}\}$. Define SRE_k as the string derived from RE_k (if $1 \le k \le n$) or re_k (if $n+1 \le k \le n+m$) by removing all the * operators. We shall construct multiple goto graphs using suffixes of SRE_k, $1 \le k \le n+m$.

Let $Z_0 = \{SRE_1, ..., SRE_n, SRE_{n+1}, ..., SRE_{n+m}\}$ and $\mathbf{G_0}$ be the goto graph

constructed with the strings contained in Z_0. The self-loop at the start state, if exists, is deleted. Consider a regular expression $RE \in Y$. Assume that $RE = S_1 * S_2 * ... * S_{J+1}$. We call states Q_i, $1 \le i \le J$, on graph $\mathbf{G_0}$ with $S^{Q_i} = S_1 S_2 ... S_i$ switching states. These J switching states are said to be contributed by RE or they belong to RE. Note that it is possible for a switching state to belong to multiple regular expressions. Define $SRE - S^{Q_i} = S_{i+1} ... S_{J+1}$. If string $SRE - S^{Q_i}$ is included in constructing a goto graph $\mathbf{G}$, states Q'_j, $1 \le j \le J - i$, on graph $\mathbf{G}$ with $S^{Q'_j} = S_{i+1} ... S_{i+j}$ are switching states on graph $\mathbf{G}$. These switching states also belong to RE. It is not hard to see that, for the switching state Q'_j on graph $\mathbf{G}$, there is a switching state on graph $\mathbf{G_0}$ represented by $S_1 ... S_{i+j}$. We call this switching state on graph $\mathbf{G_0}$ the corresponding switching state of Q'_j. In this paper, we shall use Q^* to denote the corresponding switching state of a switching state Q. We have $Q^* = Q$ if switching state Q is on graph $\mathbf{G_0}$. Construction of other goto graphs is as follows.

Assume that there are a total of M distinct switching states on graph $\mathbf{G_0}$. Let $Q_1, Q_2, ...,$ and Q_M denote the switching states. A binary flag FQ_i is associated with state Q_i. The flag $FQ_i = 1$ iff the string representing state Q_i was found. The possible values of $(FQ_1, FQ_2, ..., FQ_M)$ are called configurations. Clearly, there are 2^M possible values for $(FQ_1, FQ_2, ..., FQ_M)$. We say a configuration is feasible if it is possible to occur during scanning. A goto graph is constructed for each feasible configuration. In general, not all the 2^M possible configurations are

feasible. The goto graph $\mathbf{G_0}$ corresponds to the all-zero feasible configuration $C_0 = \mathbf{0} = (0, 0, \ldots, 0)$. We call goto graph $\mathbf{G_0}$ the Level 0 graph. Graph $\mathbf{G_0}$ is used to construct Level 1 goto graphs, which in turn are used to construct Level 2 goto graphs, and so on. In the construction procedure shown below, the function **Construct_Goto_Graph(**$\mathbf{G}$, Z**)** is to construct goto graph $\mathbf{G}$ with the strings in Z using Algorithm AC1, except that the self-loop at the start state, if exists, is removed. The goto graph $\mathbf{G_i}$, with corresponding feasible configuration C_i, is constructed with the strings contained in set Z_i. The set Z_0 is the input to the construction procedure. Some states are marked as fork states because, as will become clear in sub-section B.4, a process is forked whenever a fork state is visited. State R on goto graph $\mathbf{G_0}$ is a fork state iff $S^R = SRE_{n+k}$ for some k, $1 \le k \le m$. Similarly, state R on goto graph $\mathbf{G_i}$ $(i \ge 1)$ is a fork state iff $S^R = SRE_{n+k} - S^Q$ is a string in Z_i, where Q is a switching state on graph $\mathbf{G_0}$ that is contributed by RE_{n+k}.

Procedure Goto(Z_0**)**

$i = 0$ /* index of goto graphs */
$I = 0$ /* level of goto graphs */

$C_0 = \mathbf{0}$

$Configurations\_in\_Level[I] = \{C_0\}$

Construct_Goto_Graph($\mathbf{G_0}$, Z_0**)**

Mark the fork states on graph $\mathbf{G_0}$

$Graphs\_in\_Level[I] = \{\mathbf{G_0}\}$

while (1)

$J = I + 1$

$Configurations\_in\_Level[J] = \varnothing$

$Graphs\_in\_Level[J] = \varnothing$

For every $\mathbf{G} \in Graphs\_in\_Level[I]$ with corresponding configuration C

For every switching state Q on graph $\mathbf{G}$

Determine the corresponding switching state Q^* on graph $\mathbf{G_0}$

Set_Flags(C', Q^*) /* set $FQ_j = 1$ if S^{Q_j} is a prefix of S^{Q^*} */

$C'' = C \oplus C'$ /* $\oplus$ denotes the bitwise OR operation */

If $C'' \neq C_j$ for all j, $0 \le j \le i$ /* a new feasible configuration */

$i{+}{+}$

$C_i = C''$

$Configurations\_in\_Level[J] =$

$Configurations\_in\_Level[J] \cup \{C_i\}$

Find_Strings(Z_i, C_i) /* determine Z_i */

` **Construct_Goto_Graph**($\mathbf{G_i}$, Z_i)

Mark the fork states on graph $\mathbf{G_i}$

$Graphs\_in\_Level[J] = Graphs\_in\_Level[J] \cup \{\mathbf{G}_i\}$

If $Configurations\_in\_Level[J] = \varnothing$

Break

$I{+}{+}$

Set_Flags(C, Q)

$C = \mathbf{0}$

For every switching state Q_i

If S^{Q_i} is a prefix of S^Q

$FQ_i = 1$ /* FQ_i denotes the i^{th} bit of C */

Find_Strings(Z , C)

For every switching state Q_i such that FQ_i =1

Find $B(Q_i)$ the set of regular expressions that contribute state Q_i

For every $RE_j \in B(Q_i)$

$$Z = Z \cup \{SRE_j - S^{Q_i}\}$$

For every $SRE_j - S^{Q_k} \in Z$

If there exists $SRE_j - S^{Q_l} \in Z$ which is a proper suffix of $SRE_j - S^{Q_k}$

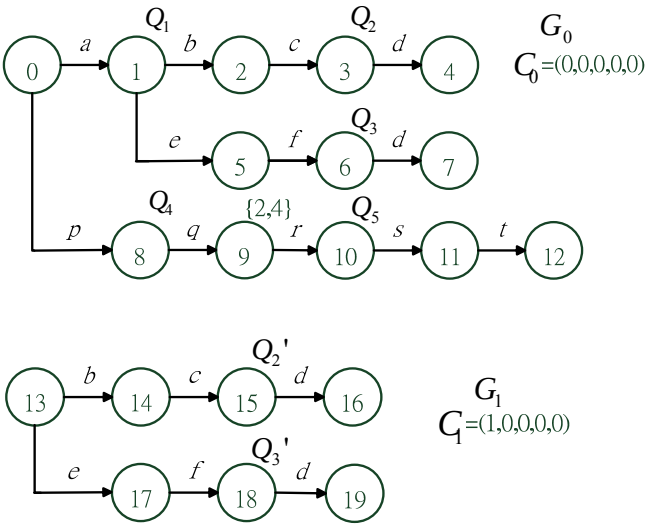
$$Z = Z - \{SRE_j - S^{Q_k}\}$$

Construction of the goto graphs for $Y = \{RE_1, ..., RE_n, re_{n+1}, ..., re_{n+m}\}$ is accomplished by the above procedure. The remaining work is to handle the other fragments of RE_{n+k} , $1 \le k \le m$. Again, we use RE_{n+1} = $S_1 * S_2 * S_3 \ \{min_1, max_1\} \ S_4 * S_5 \ \{min_2, max_2\} \ S_6$ as an example for explanation. Handling of the other fragments of RE_{n+1} is basically to repeat the above construction procedure assuming that there is only one regular expression $RE = S_4 * S_5 \ \{min_2, max_2\} \ S_6$. Consider handling of the second fragment $S_4 * S_5$. Two goto graphs are constructed: one for $\{S_4 S_5\}$ and another one for $\{S_5\}$. The start state on the goto graph constructed for $\{S_4 S_5\}$ is modified as follows. It is marked with $\{min_1, max_1\}$ and the self-loop, if exists, is not removed. The remaining fragments are handled the same as the second fragment. For differentiation, we shall use $\mathbf{T}_i$'s to represent the goto graphs constructed for the fragments other than the first

one of RE_{n+k}, $1 \le k \le m$. The construction of goto graphs is completed after all fragments of RE_{n+k}, $1 \le k \le m$, are processed.

Note that there is no Level 2 goto graph if the first string of any regular expression is not a prefix of the first string of any other regular expression. This is called non-overlapping condition. Under the non-overlapping condition, string S_i of $RE = S_1 * S_2 * ... * S_{J+1}$ appears exactly *i* times on *i* different goto graphs.

Figure 5. shows the goto graphs for $RE_1 = a*bc*d$, $RE_2 = a*ef*d$, $RE_3 = pqr*st$, and $RE_4 = p*q\{2,4\}u\{3,5\}vw*xy$. Note that there are five switching states and one fork state on graph $\mathbf{G_0}$. Switching state Q_1 is contributed by both RE_1 and RE_2. Therefore, strings *bcd* and *efd* are used to construct graph $\mathbf{G_1}$. Graphs $\mathbf{G_1}$ to $\mathbf{G_5}$ are Level 1 graphs while graph $\mathbf{G_6}$ is the only Level 2 graph and is generated by graph $\mathbf{G_2}$. Goto graph $\mathbf{T_0}$ is created by the second fragment of RE_4. Note that state 31 is a fork state and marked with $\{2,4\}$.



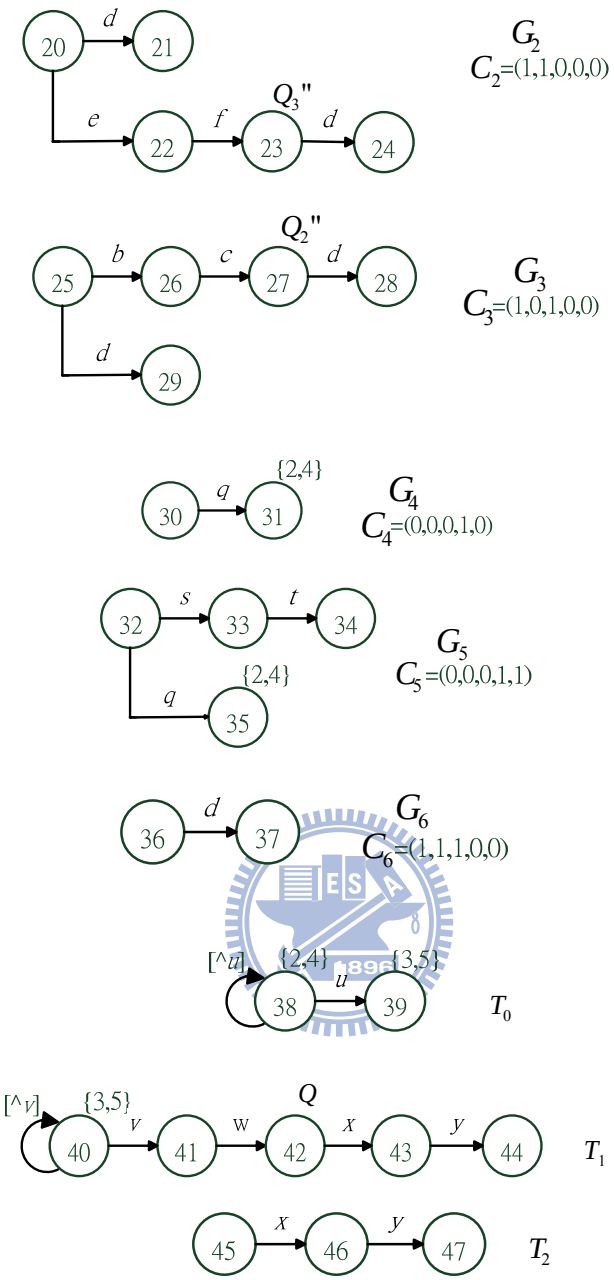


Figure 5. The goto graphs for $RE_1 = a*bc*d$, $RE_2 = a*ef*d$, $RE_3 = pqr*st$, and $RE_4 = p*q\{2,4\}u\{3,5\}vw*xy$.

- **The failure function**

For convenience, we call a goto graph whose start state is marked with some $\{min, max\}$ operator a $\{min, max\}-$graph. As an example, the goto graphs $\mathbf{T_0}$ and

$\mathbf{T_1}$ shown in Figure 5 are $\{min, max\}-$graphs. The failure functions for $non-\{min, max\}-$graphs and $\{min, max\}-$graphs are constructed with the following **Non-{min, max}_Failure** and **{min, max}_Failure** procedures, respectively. In the procedures, C represents the corresponding feasible configuration of graph $\mathbf{G}$ or $\mathbf{T}$. An additional state, called the END state, is added in constructing the failure function. As will be seen in Sub-section B.4, traversal on a goto graph ends if it enters the END state.

Figure 6(a) shows the failure function for the four regular expressions used in Figure 5. In this figure, the state number of the $(i, j)^{th}$ entry is $10*i+j$ and value 0 for $f(R)$ represents the END state. The symbol "-" means failure never occurs in that state. For example, failure never occurs in states 38 and 40.

$f(R)$	0	1	2	3	4	5	6	7	8	9
0	0	13	13	20	20	13	25	25	30	30
1	32	32	32	0	0	20	20	0	25	25
2	0	0	0	36	36	0	0	36	36	0
3	0	0	0	0	0	0	0	0	-	38
4	-	40	45	45	45	0	0	0		

(a)

R	4, 16, 21, 28	7, 19, 24, 29	12, 34	44, 47	37
$output(R)$	RE_1	RE_2	RE_3	RE_4	RE_1, RE_2

(b)

Figure 6. (a) The failure function and (b) the output function for the example regular expressions used for Figure 5.

- **The output function**

Consider some goto graph $\mathbf{G}$ constructed for Y. Assume that $RE_k = S_1 * S_2 * ... * S_{J+1}$, $1 \le k \le n$, and $S_{j+1}...S_{J+1}$ is included in constructing graph $\mathbf{G}$. We assign initially $output(P) = \varnothing$ for every state P on graph $\mathbf{G}$.

Let R be the state on graph $\mathbf{G}$ with $S^R = S_{j+1}...S_{J+1}$. The output function $output(R)$ is modified as $output(R) = output(R) \cup \{RE_k\}$.

Now consider a goto graph $\mathbf{T}$ constructed for some fragment of RE_{n+k}, $1 \le k \le m$. For every state P on graph $\mathbf{T}$, we assign $output(P) = \varnothing$. If graph $\mathbf{T}$ is constructed for the last fragment of RE_{n+k}, then $output(R)$ is modified for some state R. Assume that the last fragment of RE_{n+k} is $S_1 * S_2 * ... * S_{J+1}$ and graph $\mathbf{T}$ is constructed with $S_{j+1}...S_{J+1}$. The output function of state R on graph $\mathbf{T}$ is modified as

$output(R) = output(R) \cup \{RE_{n+k}\}$ if $S^R = S_{j+1}...S_{J+1}$

- **The signature matching machine**

During scanning, a set called $Active\_Graphs$ is maintained. When the pre-filter finds the starting position of a suspicious sub-string which may result in match of some signatures, concurrent traversals begin at the start states of all the goto graphs contained in $Active\_Graphs$ .Initially, we have $Active\_Graphs = \{\mathbf{G_0}\}$. Consider the traversal on a specific goto graph. A process is forked to traverse a $\{min, max\}-$graph if a fork state is visited. As an example, consider the goto graphs shown in Figure 4. A process is forked to traverse graph $\mathbf{T_0}$ if state 9, 31, or state 35 is visited. As another example, a process is forked to traverse graph $\mathbf{T_1}$ if state 39 is visited. Assume that the failure function is consulted in state R and $f(R)$ is the start state of some goto graph $\mathbf{G}$ or $\mathbf{T}$, different from the goto graph state R is on. In this case, graph $\mathbf{G}$ or $\mathbf{T}$ is added to $Active\_Graphs$ so that it will be traversed when succeeding suspicious sub-strings are found by the pre-filter. For

example, for the goto graphs shown in Figure 4, if the failure function is consulted in state 2, then graph $\mathbf{G_1}$ is added to $Active\_Graphs$. Traversal on a $non-\{min, max\}-$graph ends if a match is found or the failure function is consulted. Traversal on $\{min, max\}-$graph $\mathbf{T}$ is as follows. Let **{min, max}** be the mark of its start state. A counter ctr is maintained when traversing graph $\mathbf{T}$. The content of ctr is initialized to **min** and the next **min** symbols are skipped. The counter is increased by one if the current state is the start state of $\mathbf{T}$ and it returns to the same state after an input symbol is processed. Assume that the failure function is consulted in state *P*. If state $f(P)$ is also on graph $\mathbf{T}$, which implies state *P* is not on the sub-tree of any switching state, then ctr is updated as $ctr = ctr + |S^P| - |S^{f(P)}|$. We set $ctr = \mathbf{max} + 1$ if state $f(P)$ is on a different graph. The traversal ends iff a match is found or $ctr > \mathbf{max}$.

Assume that a particular goto graph is under traversal. RE_k, $1 \le k \le n$, is a candidate signature to be matched if some suffix of SRE_k is included in constructing the goto graph. Similarly, RE_{n+k}, $1 \le k \le m$, is a candidate signature to be matched if some suffix of the string obtained by removing the * operators of some fragment of RE_{n+k} is included in constructing the goto graph. Obviously, the number of candidates never increases during traversal for a given suspicious sub-string. The verification process ends if any signature is matched, the input string is exhausted, or all concurrent traversals end.

Chapter 4.

PCRE Handling with The Enhanced Aho-Corasick Algorithm

PCRE Rules Pattern Form

In this section, we divide the PCRE rules into six parts[3]. The division factor is focus on regular expression, so Table 5 will list the six parts of pattern features and the complexity of states. Definite strings generate DFAs of length linear to the number of characters in the string. If a pattern starts with '^', it originates a DFA of polynomial complexity with respect to the pattern length *k* and the length restriction *j*. From the existing content scanning rule sets is that the pattern length *k* is usually limited but the length restriction *j* maybe reach hundreds or even thousands. It will cause very large and high complexity of space. Therefore, Case 5 can effect in a large DFA because it has a element quadratic in *j*. This patterns starting with ".*" and having length restrictions, Case 6, cause the creation of DFA of exponential size.

Table 5. Analysis of patterns with *k* characters

Pattern features	Example	#of states
1.Explicit strings with k characters	^ABCD .*ABCD	K+1
2.Wildcards	^AB.*CD .*AB.*CD	K+1
3.Patterns with ^, a wildcard, and a length restriction j	^AB.{0,j}CD	*O(k*j)*
4. Patterns with ^, a wildcard, and a length restriction j(min=max=j)	^AB.{j}CD	K+j
5.Patterns with ^, a class of characters overlaps with the prefix, and a length restriction j	^A+[A-Z]{j}D	$O(k+j^2)$
6.Patterns with a length restriction j, where a wildcard or a class of characters overlaps with the prefix	.*AB.{j}CD .*A[A-Z]{j}D	$O(k+2^j)$

15

The following will show above six cases DFA graphs and our proposed signature matching system.

- PCRE Patterns Form- Case 1
  - ✓ The pattern features : Explicit strings with k characters, k is the pattern length.
  - ✓ Size of DFA : linear.
  - ✓ Number of states : k+1
  - ✓ Example: ^ABCD and .*ABCD on Figure 7.

    Notice that, if the next state is failed, we assume that it will terminate immediately. So, we do not show the failed path back to the stating state on the graphs.

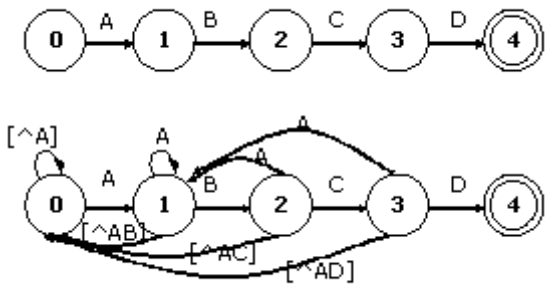


Figure 7. DFA of ^ABCD and .*ABCD

PCRE rules example(take from snort PCRE library[10]) after using the enhanced Aho-Corasick algorithm:

1.ftp.rules 3441 /^PORT/smi

2.backdoor.rules 12242 /^Start$/smi

Shown on Figure 8.

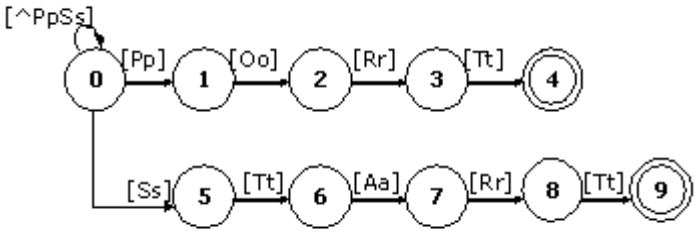


Figure 8. Snort PCRE rule example

- PCRE Patterns Form- Case 2
  - ✓ The pattern features : Wildcards.
  - ✓ Size of DFA : linear.
  - ✓ Number of states : k+1
  - ✓ Example: ^AB.*CD and .*AB.*CD on Figure 9.

    Notice that, if the next state is failed, we assume that it will terminate immediately. So, we do not show the failed path back to the stating state on the graphs.

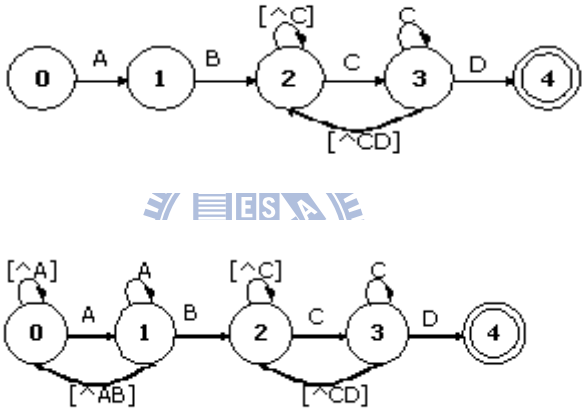


Figure 9. DFA of ^AB.*CD and .*AB.*CD

PCRE rules example(take from snort PCRE library[10]) after using the enhanced Aho-Corasick algorithm:

1.chat.rules 6182 /^\s*NOTICE/smi

2.smtp.rules 664 /^rcpt to\:\s*decode/smi

Shown on Figure 10.

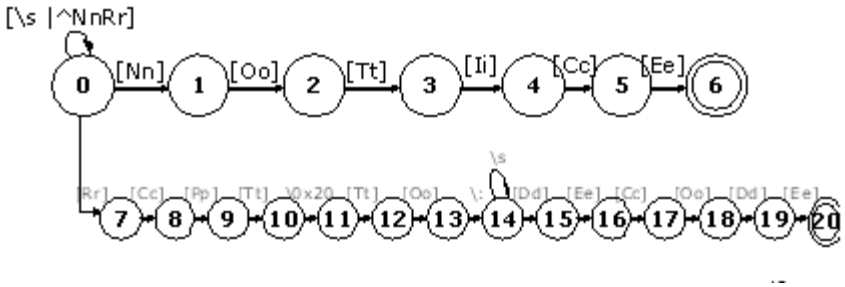


Figure 10. Snort PCRE rule example

- PCRE Patterns Form- Case 3
  - ✓ The pattern features : Patterns with ^, a wildcard, and a length restriction j.
  - ✓ Size of DFA : Polynomial.
  - ✓ Number of states : *O(k*j)*
  - ✓ Example: ^AB.{0,j}CD on Figure 11.

    Notice that, if the next state is failed, we assume that it will terminate immediately. So, we do not show the failed path back to the stating state on the graphs.

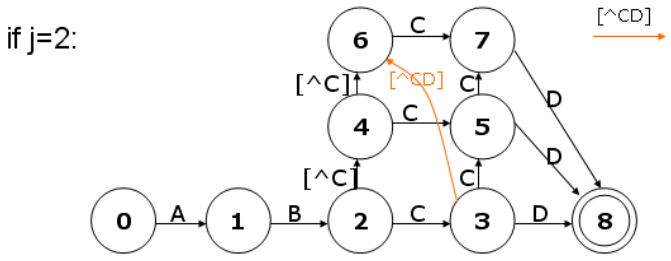


Figure 11. DFA of ^AB.{0,j}CD

PCRE rules example(take from snort PCRE library[10]) before using the enhanced Aho-Corasick algorithm:

1.ddos.rules 228 /^[0-9]{1,5}\x00/ [0 - 9] \x00

2.Shown on Figure 12.

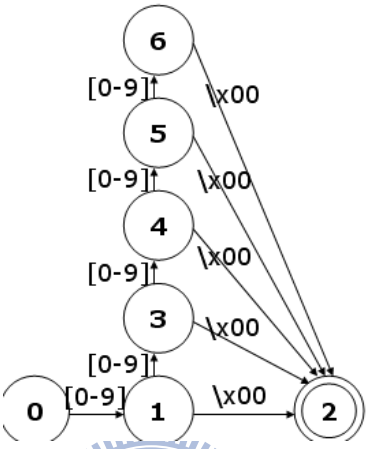


Figure 12. Snort PCRE rule example

PCRE rules example(take from snort PCRE library[10]) after using The Enhanced Aho-Corasick Algorithm:

1.ddos.rules 228 /^[0-9]{1,5}\x00/ [0 - 9] \x00

Shown on Figure 13.

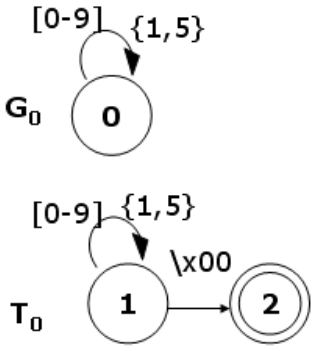


Figure 13. Snort PCRE rule example

- PCRE Patterns Form- Case 4
  - ✓ The pattern features : Patterns with ^, a wildcard, and a length restriction j(min=max=j).
  - ✓ Size of DFA : linear.
  - ✓ Number of states : $k+j$
  - ✓ Example: ^AB.{j}CD on Figure 14.

Notice that, if the next state is failed, we assume that it will terminate immediately. So, we do not show the failed path back to the stating state on the graphs.

if j=2:

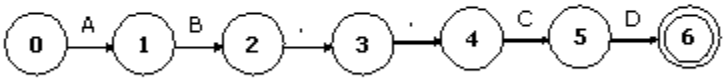


Figure 14. DFA of ^AB.{j}CD

- PCRE Patterns Form- Case 5
  - ✓ The pattern features : Patterns with ^, a class of characters overlaps with the prefix, and a length restriction j.
  - ✓ Size of DFA : quadratic.
  - ✓ Number of states : $O(k+j^2)$
  - ✓ Form example: ^A+[A-Z]{j}D
  - ✓ Example: ^B+[^\n]{3}D on Figure 15.

  Notice that, if the next state is failed, we assume that it will terminate immediately. So, we do not show the failed path back to the stating state on the graphs.

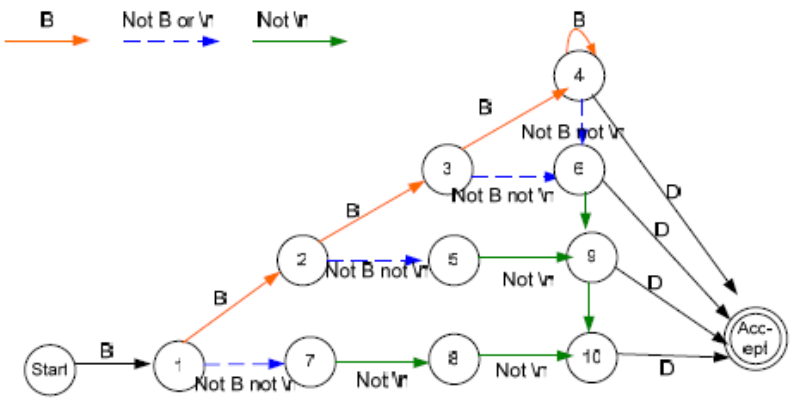


Figure 15. DFA of ^B+[^\n]{3}D

PCRE rules example(take from snort PCRE library[10]) after using The Enhanced Aho-Corasick Algorithm:

1.^B+[^\n]{3}D

Shown on Figure 16.

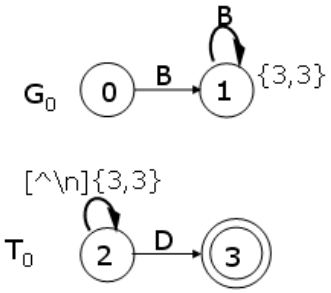


Figure 16. Snort PCRE rule example

Here, our proposed enhanced AC algorithm surely decrease total states of this Snort PCRE case. But notice that if the {min,max} number is larger than this figure example, it will create large number of fork graphs and take a lot of time to scan between these graphs.

To avoid creating so many graphs, we bring up an idea that using a *fork_counter* to count how many times the previously continuous character has happened.

For example, the pattern form is the same as previous figure 16, ^B+[^\n]{3}D. The example of this pattern is BYAAD or BABAD. Starting character is B, and next is Y or A which is not equaled to B. So the *fork_counter* is still keep the same(here is still 3 in Figure 17).

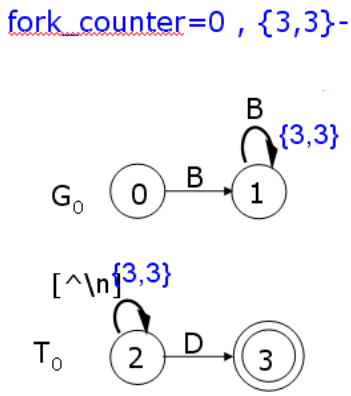


Figure 17. *fork_counter* to count previously continuous character.

In the same pattern form instance, BBAAD let the *fork_counter* become 1 because the second character B which is same as the first character.

See Figure 18, the value of min is countdown because original min value minus value of *fork_counter*.

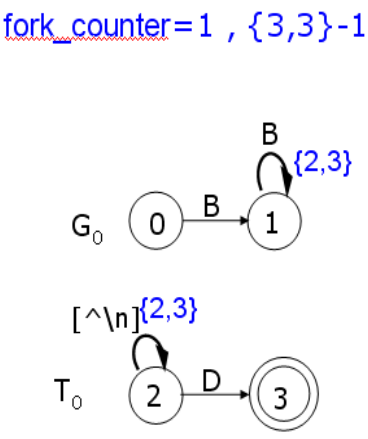
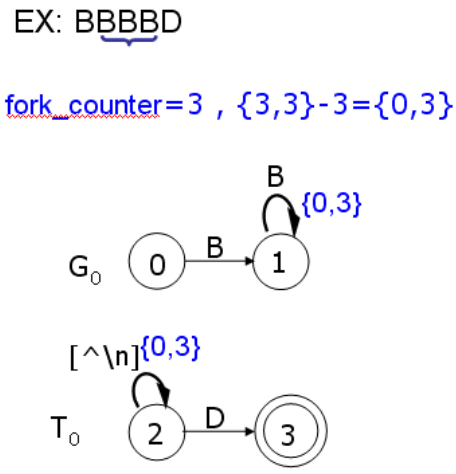
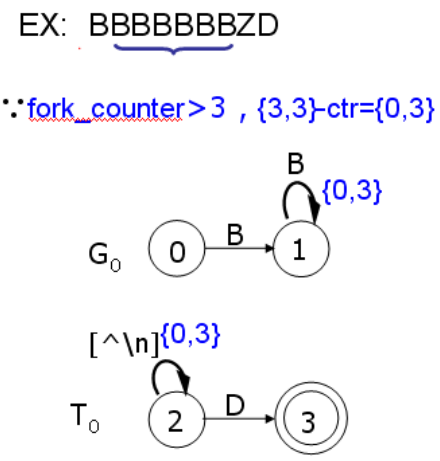


Figure 18. The value of min minus value of *fork_counter*.

But if the value of *fork_counter* is equal or larger than value of min? In this situation, the min value is become zero. In Figure 19(a), the same pattern form instance, BBBBD let the *fork_counter* become 3 because the number of second and later character B which is same as the first character. So the value of min minus value of *fork_counter* is equal to zero. In Figure 19(b), the value of *fork_counter* is larger than min value, so min value sets to zero.



(a)



(b)

Figure 19.Value of *fork_counter* is equal or larger than value of min

- PCRE Patterns Form- Case 6
  - ✓ The pattern features : Patterns with a length restriction j, where a wildcard or a class of characters overlaps with the prefix.
  - ✓ Size of DFA : exponential.
  - ✓ Number of states : $O(k+2^j)$
  - ✓ Form example: *AB.{j}CD and .*A[A-Z]{j}D
  - ✓ Example: .*A.{2}CD on Figure 20.

  Notice that, if the next state is failed, we assume that it will terminate immediately. So, we do not show the failed path back to the stating state on the graphs.

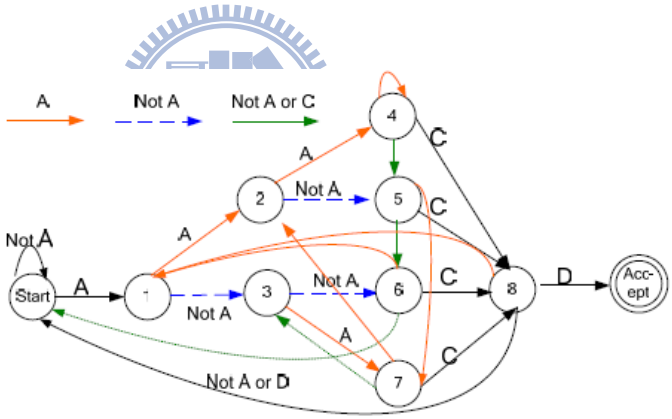


Figure 20. DFA of .*A.{2}CD

PCRE rules example(take from snort PCRE library[10]) after using The Enhanced Aho-Corasick Algorithm:

1..*A.{2}CD

Shown on Figure 21.

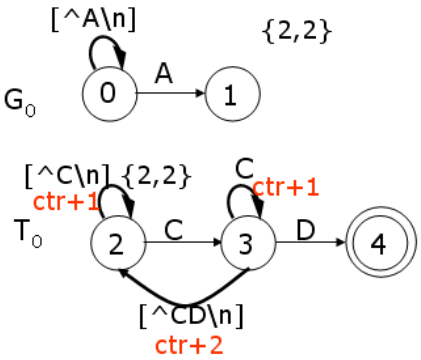


Figure 21. Snort PCRE rule example

Chapter 5.

Experimental Results

In this chapter, we present simulation results for the Snort PCRE rules parts using the enhanced Aho-Corasick algorithm.

There are divided into two sections which shown on Figure 22: Pre-filter and verification module. Using pre-filter to find the suspicious starting position from input file. Once the suspicious position has found, pre-filter pause at that position to transfer to verification module procedure then starting to run all active graph structure.

Figure 22. The Procedure of algorithm.

Programming Procedure

The programming flow is shown on Figure 23.

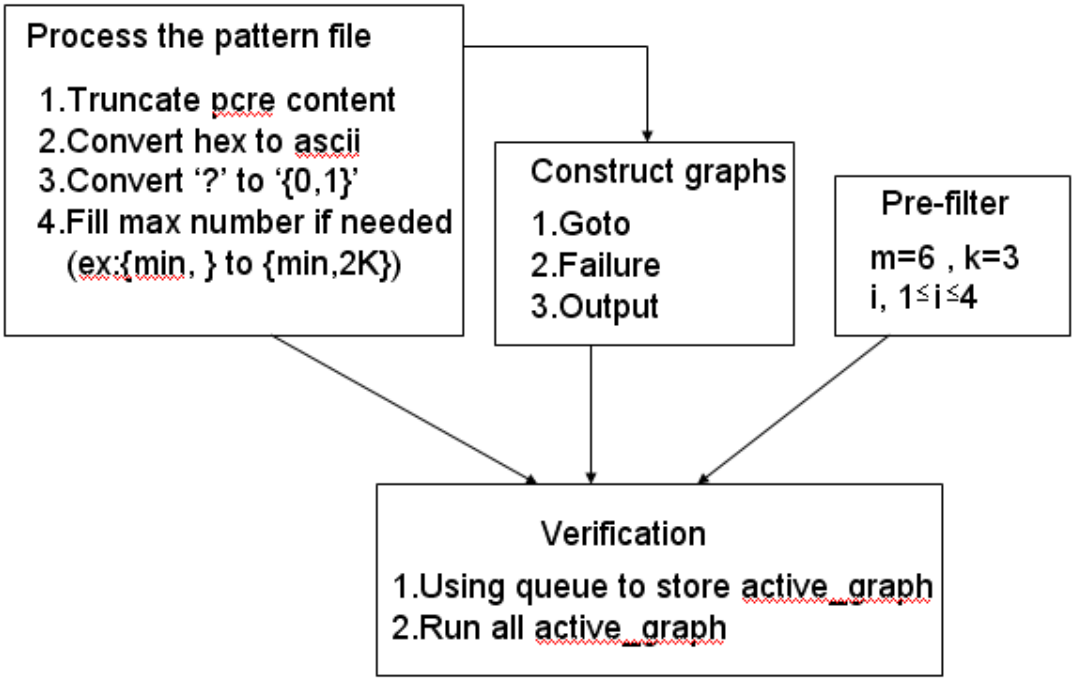


Figure 23. The Programming flow.

The beginning, we process the pattern file to make it become legitimate pattern rules. After process the rule file, take all rule file to construct graphs. When graphs construct completely, we first read in a clean file which means there has not exist any string that matched by PCRE rules. Simultaneously, the pre-filter will look for the suspicious position which matched any PCRE rule starting segment. Once find the matched rule position, turn into verification steps.

In our experiment, we use 11147 Snort PCRE rules to construct matching graphs. Figure 24 displays performance using our proposed signature matching system for

clean files of various sizes.

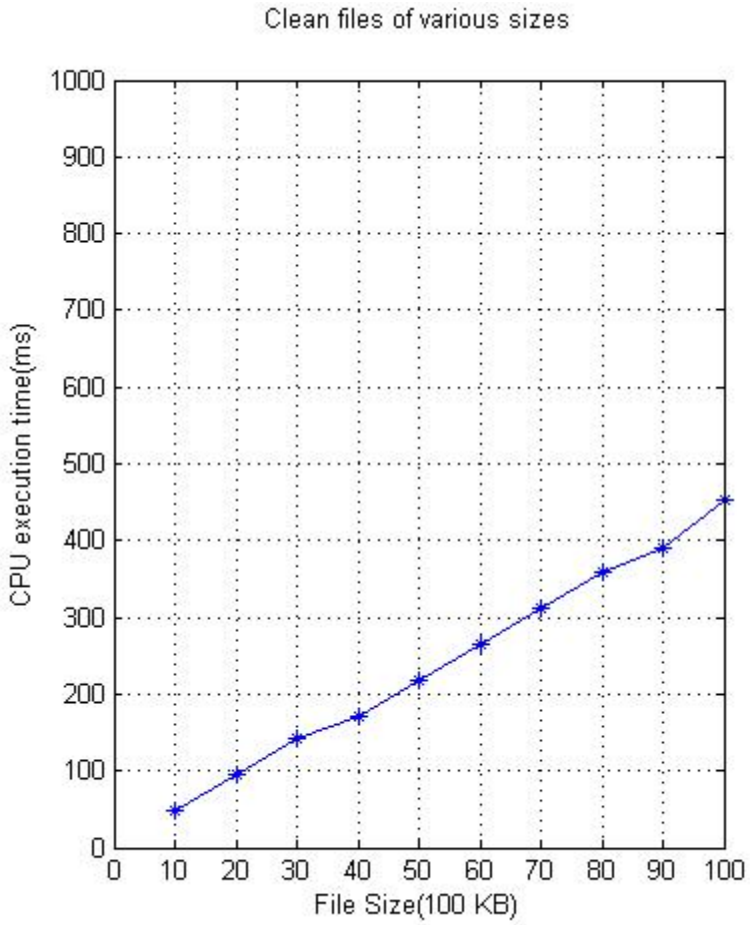


Figure 24. Performance using our proposed signature matching system for clean files of various sizes.

Figure 25 displays performance using our proposed signature matching system for a file with an inserted Snort PCRE rules at various positions.

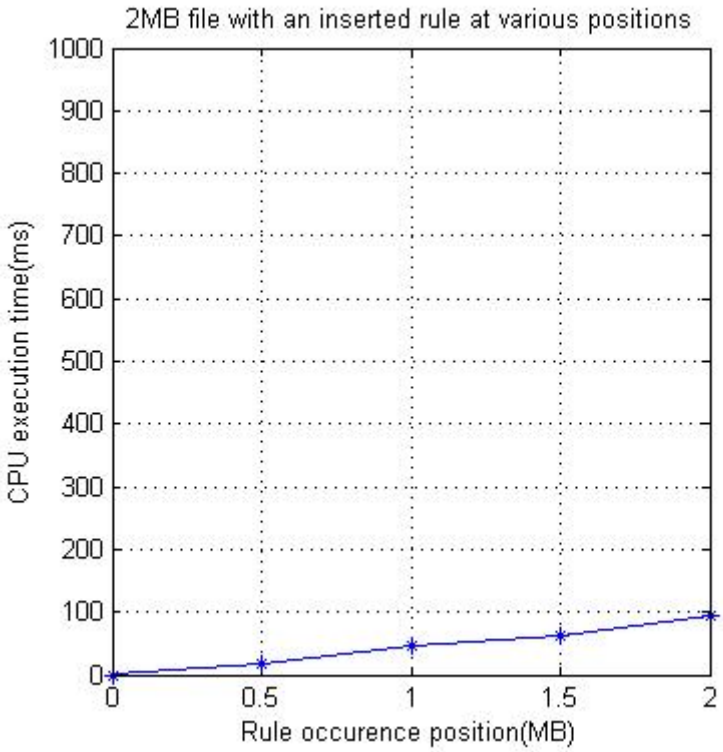


Figure 25. Performance using our proposed signature matching system for a file with an inserted Snort PCRE rules at various positions.

Chapter 6.

Conclusion

We have presented in this paper Snort PCRE rule to detect deep packet content using enhanced Aho-Corasick algorithm. Numerical results show that our proposed algorithm provides less regular expression matching states, that means, we use less memory space to apply PCRE matching.

In this way, the space requirement of a DFA is also reduced. Therefore, the purpose of the method to extend the AC algorithm is to present a high-performance, reasonable memory requirement signature matching system for simple regular expressions and plain strings that can be efficiently implemented on general-purpose processors.

Because this scheme is only simulated in our personal computer, how to implement on hardware like FPGA remains to be further studied.

Bibliography

[1] Fisk, M. and G. Varghese, *Fast Content Based Packet Handling for Intrusion Detection*, 2001.

[2] Roesch, Martin, ‘‘Snort – Lightweight Intrusion Detection for Networks,’’ *13th Systems Administration Conference*, USENIX, 1999.

[3] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, “Fast and memory-efficient regular expression matching for deep packet inspection,” in Proc. of Architectures for Networking and Communications Systems (ANCS), pp. 93-102, 2006.

[4] K. Thompson, “Programming techniques: Regular expression search algorithm,” Commun. ACM, 11(6):419-422, 1968.

[5] V. M. Glushkov, “The abstract theory of automata,” Russian Mathematical Surveys, 16:1-53, 1961.

[6] R. W. Floyd and J. D. Ullman, “The compilation of regular expression into integrated circuits,” Journal of ACM, vol. 29, no. 3, pp. 603-622, July 1982.

[7] T. H. Lee, “Enhancing the Aho-Corasick Algorithm for Signature Based Anti-Virus/Worm Applications,” ICCCN 2007.

[8] *Alok Tongaonkar, Sreenaath Vasudevan, and R. Sekar,* “Fast Packet Classification for Snort by Native Compilation of Rules,” (LISA ’08).

[9] 王聲浩，陳一瑋，林盈達，”攻擊、病毒與廣告信的辨識機制與套件，”2008

[10] http://www.snort.org.

[11] J. Levandoski, E. Sommer, and M. Strait, "Application Layer Packet Classifier for Linux." http://l7-filter.sourceforge.net/.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, 2001.

[13] Jo˜ao Bispo, Ioannis Sourdis, Jo˜ao M.P. Cardoso and Stamatis Vassiliadis , “Regular Expression Matching for Reconfigurable Packet Inspection,” supported by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project.

[14] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” Communications of the ACM, vol. 18, pp. 333–340, Jun. 1975.

[15] Tsern-Huei Lee, *IEEE,* and Nai-Lun Huang, ” A Pattern Matching Scheme with High Throughput Performance and Low Memory Requirement,” Submitted for publication.

[16] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching,” Technical Report, May 1994.