國立交通大學

電信工程研究所

碩士論文

字串比對在入侵偵測/防護系統上針對 Aho-Corasick 演算法的強化與實現

Enhancing the Aho-Corasick Algorithm for Signature Based Anti-Virus/Worm Implementations

研 究 生：李韋儒

指導教授：李程輝 教授

中華民國九十九年六月

字串比對在入侵偵測/防護系統上針對 Aho-Corasick 演算法的強化與實現

Enhancing the Aho-Corasick Algorithm for Signature Based Anti-Virus/Worm Implementation

研 究 生：李韋儒　　Student：Wei-Zoo Lee

指導教授：李程輝　　Advisor：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 Communications Engineering

College of Electrical and Computer Engineering

National Chiao Tung University

In partial Fulfillment of the Requirements for the Degree of

Master

in

Communications Engineering

June 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年六月

字串比對在入侵偵測/防護系統上針對 Aho-Corasick 演算法的強化與實現

學生: 李韋儒　　　　指導教授:李程輝教授

國立交通大學
電信工程研究所

摘要

因為現在網路的迅速成長，字串比對已經在防毒/防蟲當中被視為一種很重要的技術。目前相當有名的字串比對演算法：Aho-Corasick (AC)演算法，是一個能夠同時比對多重字串，並且在各種環境之下都能夠保證穩定的輸出表現的演算法。AC 演算法的發展是依照字串比對的方式，然而病毒/蠕蟲本身是可以利用正規表示式來表示。這篇論文裡，我們會將 AC 演算法作強化，用一種系統化的方式來實現這套延伸強化應用的 AC 演算法，以達到可以針對一般字串以及正規表示式作為表示的字串比對，並且能準確指出字串的來源以及在文件中出現之後到結束的位置。

關鍵字：網路安全，字串比對，正規表示式

Enhancing the Aho-Corasick Algorithm for Signature Based Anti-Virus/Worm Implementations

Student: Wei-Zoo Lee   Advisor: Prof. Tsern-Huei Lee

Department of Communication Engineering
National Chiao Tung University

ABSTRACT

Because of its accuracy, pattern matching is considered an important technique in anti-virus/worm applications. Among some famous pattern matching algorithms, the Aho-Corasick (AC) can match multiple patterns simultaneously and guarantee deterministic performance under all circumstances. However, the AC algorithm was developed for strings while virus/worm signatures could be specified by simple regular expressions. In this paper, we enhance the AC algorithm to systematically construct a signature matching system which can indicate the ending position in a finite input string for the occurrence of virus/worm signatures that are specified by strings or simple regular expressions. The regular expressions studied are those adopted in ClamAV for signature specification.

Keywords: network security, string matching, regular expression

誌　　謝

感謝交通大學電信工程研究所 NTL 實驗室的各位，郁文學長、景融學長、迺倫學姐、曉薇、建碩、奕璉、永昌、永祥，學弟妹國書、煜傑、順閔、謙和、建男、運良、晴嬅、承潔，還有已經畢業的俊德學長、松晏學長、鈞傑學長，感謝你們陪我度過我的研究所，以及在這之間所提供給我的任何意見跟想法。

特別感謝我的指導教授 李 程輝 博士，在我的學業、研究方面的指導讓我在研究所兩年中獲益匪淺。與迺倫學姐、建碩同學與奕璉同學在研究方面的相互討論更是讓我的研究能夠順利進行的一大助力。

最後感謝我的家人對我的付出與支持我才能走到今天。

謹將此論文獻給所有幫助過我的人

2010/06

目　　　錄

圖 目 錄

Chapter 1.

Introduction

Because of the rapid advances of computer and network technologies, modern computer viruses and worms can spread at a speed much faster than human-mediated responses. Fast and effective detection of viruses/worms as they are spreading is, therefore, necessary to prevent the majority of vulnerable systems from being infected and minimize the damage.

There are some well-known pattern matching algorithms such as Knuth-Morris-Pratt (KMP) [1], Boyer-Moore (BM) [2], and Aho-Corasick (AC) [3]. 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.

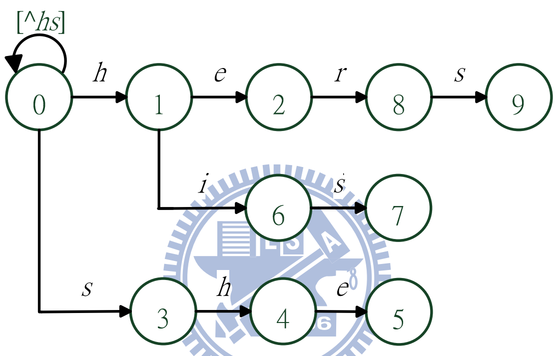
As security attacks become sophisticated, regular expressions which are much more expressive than plain strings were used to specify their signatures. Fortunately, the regular expressions used to specify virus/worm signatures are often simple ones. For example, the signatures defined in ClamAV [4] allow only plain strings and three operators: * (match any number of symbols), ? (match any symbol), and {*min*, *max*} (match minimum of *min*, maximum of *max* symbols). The AC algorithm was generalized to match such simple regular expressions in [5]. Actually, the AC algorithm can be extended to detect other types of attacks, such as injection attacks [6].

The purpose of this paper is to present an implementation of a high-performance and reasonable memory requirement signature matching system for plain strings and simple regular expressions. It can be directly applied to anti-virus/worm applications for matching exploit signatures or used as a matcher primitive for matching vulnerability signatures [7]. The proposed signature matching system consists of a pre-filter and a verification module. It has space complexity comparable to NFA-based solutions.

Chapter 2.

The Aho-Corasick Algorithm

The AC algorithm is a string matching algorithm which can match multiple patterns simultaneously. It is dictated by three functions: a goto function *g*, a failure function *f*, and an output function *output*. Fig. 1 shows the three functions for the pattern set *Y* = {*he, she, his, hers*} [9].



(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)

Fig. 1. (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. For example, string *her* represents state 8 in Fig. 1. The start state is represented by the empty string ε. Throughout this paper, the representing string of state P is denoted by S^P. The length of string *S* is represented by $|S|$.

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 Fig. 1, 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* if and only if (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 of patterns. (Note that the set could be empty.) The set *output*(*P*) contains a pattern if the pattern is a suffix of S^P.

The operation of the AC pattern matching machine is as follows. 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 [3]. 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 $output(P)=\varnothing$ 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
    $newstate \leftarrow 0$
    for $i \leftarrow 1$ 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 $p \leftarrow j$ **until** m **do**
begin
$newstate \leftarrow newstate + 1$
$g(state, a_p) \leftarrow newstate$
$state \leftarrow newstate$
end
$output(state) \leftarrow \{a_1a_2...a_m\}$
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 $\leftarrow$ empty
for each σ such that $g(0,\sigma) = P \neq 0$ **do**
begin
queue $\leftarrow$ queue $\cup \{P\}$
$f(P) \leftarrow 0$
end
while queue $\neq$ empty **do**
begin
let R be the next state in queue
queue $\leftarrow$ queue - $\{R\}$
for each σ such that $g(R,\sigma) = P \neq fail$ **do**
begin
queue $\leftarrow$ queue $\cup \{P\}$
$state \leftarrow f(R)$
while $g(state,\sigma) = fail$ **do** $state \leftarrow f(state)$
$f(P) \leftarrow g(state,\sigma)$
$output(P) \leftarrow output(P) \cup output(f(P))$
end
end
end

Chapter 3.

Problem Definition

We address in this paper the problem of detecting occurrence in a given input string for a group of plain strings and simple regular expressions. We focus on simple regular expressions because plain strings can be considered as special cases of simple regular expressions. As mentioned before, the studied regular expressions can only contain strings and three operators: *, ?, and $\{min, max\}$. It is assumed that every symbol is a byte. We only consider * and $\{min, max\}$ operators because consecutive ? operators can be replaced with a $\{min, max\}$ operator.

We shall construct a signature matching system that can indicate the ending position in a finite input string X for the occurrence of signature(s). Note that it is possible for multiple signatures to be matched simultaneously. As in the AC pattern matching machine, we use functions g, f, and *output* to represent, respectively, the goto function, the failure function, and the output function of the constructed signature matching system.

Chapter 4.

The Proposed Signature Matching System

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}. Let $RE = RE^1 * RE^2$, where RE^1 and RE^2 are plain strings or simple regular expressions. An important fact in finding a match for RE 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. Similar to the Wu-Manber (WM) algorithm [8], 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.

4.1 Pre-filter

The pre-filter is designed based on the well-known Bloom filters [9], [10] which guarantee no false negative. Given block size k, there are m-k+1 membership query module. Recall that $p_i^1 p_i^2 \ldots p_i^m$ are the first m symbols of pattern P_i. The sub-string $p_i^1 p_i^2 \ldots p_i^k$ is a member stored in the first membership query module, the sub-string $p_i^2 p_i^3 \ldots p_i^{k+1}$ is a member stored in the second membership query module, …, and the sub-string $p_i^{m-k+1} p_i^{m-k+2} \dots p_i^{m}$ 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 $h = hash(p_i^j p_i^{j+1} \dots p_i^{j+k-1})$. 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., $t_{m-k+1} t_{m-k+2} \dots t_m$ 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 = $qb_1 qb_2 \dots qb_{m-k+1}$ 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 = $mb_1 mb_2 \dots mb_{m-k+1}$ 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. Fig. 2 shows the architecture with master bitmap (stateful) for m = 6 and k = 3.

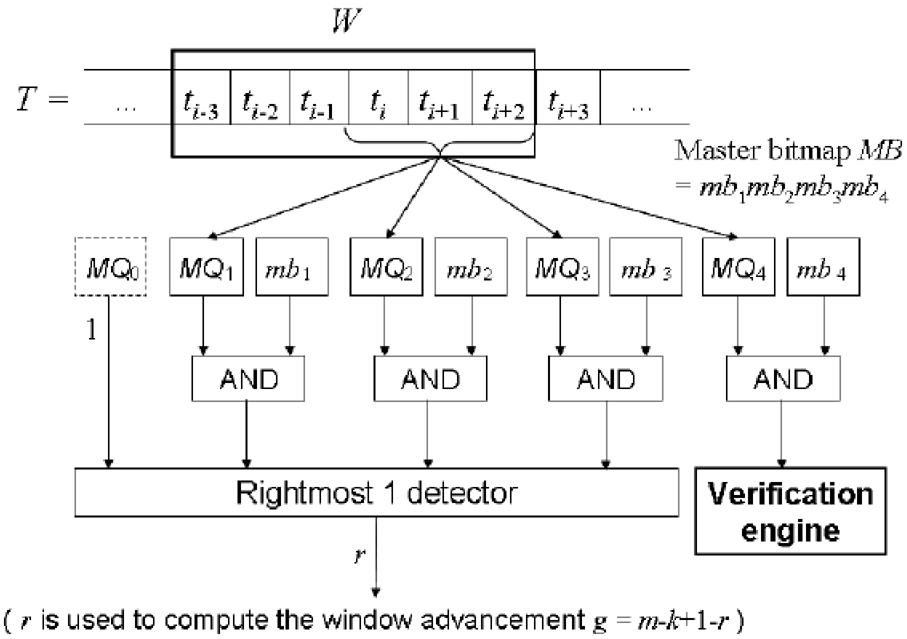


Fig. 2. The stateful pre-filter architecture for m = 6 and k = 3.

4.2 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.

4.2.1 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 $SRE_j - S^{Q_k} \in Z$

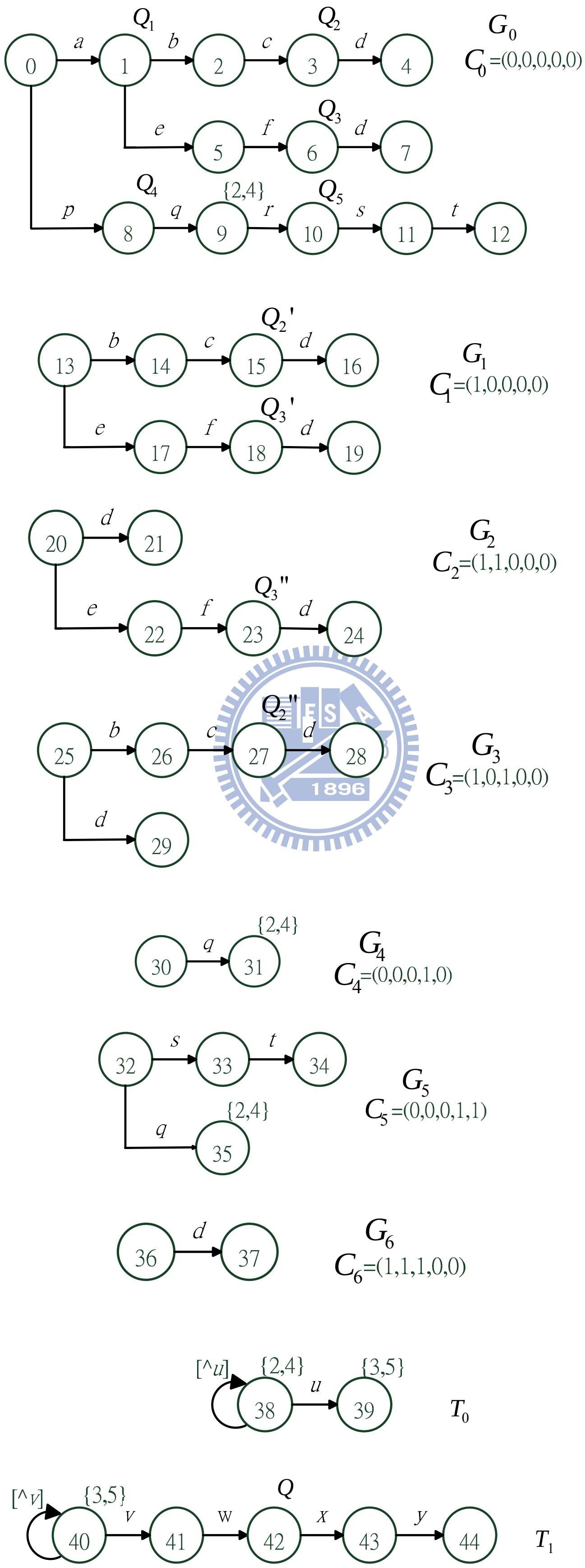
If there exists $SRE_j - S^{Q_l} \in Z$ which is a proper suffix of $SRE_j - S^{Q_k}$

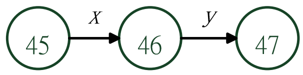
$Z = Z - \{SRE_j - S^{Q_k}\}$

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.

Fig. 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 $\{2,4\}$.





T_2

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

4.2.2 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.

Fig. 4(a) shows the failure function for the four regular expressions used in Fig. 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)

Fig. 4. (a) The failure function and (b) the output function for the example regular expressions used for Fig. 3.

4.2.3 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 $S^R = S_{j+1}...S_{J+1}$.

Note that, with the pre-filter and the fork states, we do not need to consider the case where a string which matches a regular expression contains a sub-string that matches another regular expression. Fig. 6(b) gives the output function of the states shown in Fig. 5. States with the same output function are shown in the same column. We have $output(R) = \varnothing$ if state R does not appear in the figure.

4.2.4 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 Fig. 5. A process is forked to traverse graph $\mathbf{T_0}$ if state 9, 31, or 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 Fig. 5, 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 $ctr > \mathbf{max}$.

Note that traversal on a $\{min, max\} - \text{graph}$ with mark $\{\mathbf{min, max}\}$ may take a long time to end if $\mathbf{max}$ is large. One possible remedy for this is to place the string that follows such a $\{\mathbf{min, max}\}$ operator into the pre-filter and let the traversal ends once it enters the start state. If $ctr \leq \mathbf{max}$ when the traversal ends, then the status, including $\mathbf{min}$, $\mathbf{max}$, ctr value, and the position of the last processed symbol, are saved. Moreover, the $\{min, max\} - \text{graph}$ is added to $Active\_Graphs$. The ctr value can be updated according to the saved status and the starting position of the next suspicious sub-string. The traversal on the $\{min, max\} - \text{graph}$ ends immediately if the starting position of the suspicious sub-string minus the position of the last processed symbol is smaller than $\mathbf{min}$ or the updated ctr value is greater than $\mathbf{max}$.

Assume that a particular goto graph is under traversal. RE_k, $1 \leq k \leq n$, is a candidate signature to be matched if some suffix of SRE_k is included in constructing the goto graph. Similarly, RE_{n+k}, $1 \leq k \leq m$, 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 5.

Data Structures

Consider a particular goto graph. In our proposed scheme, we classify states according to the number of child states. State P is said to be a branch state, a single-child state, or a leaf state, if it has at least two child states, exactly one child state, or no child state, respectively. Moreover, state P is said to be a final state if $output(P) \neq \varnothing$. Note that a leaf state is either a final state or a fork state or both. As shown in Fig. 5, the data structures for branch, single-child, and leaf states are different. The meanings of the first four bits of the first byte, denoted by $b_0b_1b_2b_3$, are the same for all data structures. Bit b_0=1 iff the state is a final state and bit b_1=1 iff the state is a fork state. Bits b_2b_3 indicate the type of the state and are equal to 00, 01, or 10 if the state is a leaf state, a single-child state, or a branch state, respectively. The rest four bits of the first byte are unused. The data structure consists of four bytes if b_0=1 regardless of the type of the state. In this case, bytes 2, 3, and 4 store the index of matched signatures. In the following, we only describe data structures for non-final states.

Final state

Final	Fork	Type	
Index of matched signatures: 3 bytes			

Leaf state

Final	Fork	00	
fork(*P*): 3 bytes			
min: 2 bytes			
max: 2 bytes			
f(*P*): 3 bytes			

Single-child state

Final	Fork	01	
σ : 1 byte			
f(*P*): 3 bytes			
R: 3 bytes			
fork(*P*): 3 bytes or empty			
min: 2 bytes or empty			
max: 2 bytes or empty			

Branch state

Final	Fork	10	
f(*P*): 3 bytes			
fork(*P*): 3 bytes or empty			
min: 2 bytes or empty			
max: 2 bytes or empty			
start index: 1 byte			
end index: 1 byte			
band values: 3(start index – end index +1) bytes			

Figure 5. Data structures for leaf, single-child, and branch states.

The data structure for non-final leaf state *P* consists of eleven bytes. Since state *P* is not a final state, it must be a fork state. Bytes 2, 3, and 4 store the start state of the goto graph to be traversed by a forked process. Let $\{min, max\}$ be the mark of the state. Bytes 5 and 6 store the *min* value and bytes 7 and 8 store the *max* value. The content of bytes 9, 10, and 11 represents the failure state $f(P)$. Note that $f(P)=0$ means the *END* state is entered when the failure function is consulted in state *P*.

Assume that state *P* is a single-child state and $g(P,\sigma) = R$. We allocate eight or fifteen bytes for state *P*. The second byte stores the symbol σ. Bytes 3, 4, and 5 store the failure state $f(P)$ and bytes 6, 7, and 8 store state *R*. The data structure is completed if state *P* is not a fork state. Otherwise, seven more bytes are needed. Bytes 9, 10, and 11 store the start state of the goto graph to be traversed by a forked process. Bytes 12 and 13 store the *min* value and bytes 14 and 15 store the *max* value of the mark.

Finally, assume that state *P* is a branch state. The data structure adopted is the banded-row format [11]. As an example, consider the sparse vector (0 0 0 5 4 0 0 0 9 0 7 0 0 0 0 0 0 0 0 0). The non-zero values occur in between the third (numbered from 0) and the tenth elements. Consequently, it can be represented as (3 10 5 4 0 0 0 9 0 7), where the first number indicates the start index and the second number denotes the end index, followed by eight band values. In our application, a non-zero band value represents the next state number and value zero means the failure function is to be consulted. To summarize, the data structure for non-final branch state *P* includes four or eleven bytes followed by the banded-row format. Bytes 2, 3, and 4 store the failure state $f(P)$. If state *P* is a fork state, then seven more bytes are needed. Bytes 5, 6, and 7 store the start state of the goto graph to be traversed by a forked process. Bytes 8 and 9 store the *min* value and bytes 10 and 11 store the *max* value of the mark. As for the banded-row format, there is one byte for the start index and another byte for the end index. Each band value takes three bytes.

For an input symbol σ which falls in the band with a non-zero band value *k*, it means that $g(P,\sigma)=k$. In case the input symbol σ falls outside the band or it falls in the band with a band value zero, it means $g(P,\sigma)=$ *fail*.

Since the goto graph $\mathbf{G_0}$ is likely to have a large number of states for a large signature set. As a result, to make the proposed signature matching system useful, it is necessary to reduce the memory space required by goto graphs. We modified the goto graph $\mathbf{G_0}$ such that the state number of $\mathbf{G_0}$ can be largely reduced.

There are many redundancies in the failure function, since many states may fail to the same state (say, the start state of a goto graph). But in the data structure we mention before, we store the failure function for each state. State *R* is said to be a first single-child state if it is a single-child state and its parent state is a branch state. Moreover, state *S* is said to be an explicit state if it is the start state, a branch state, a first single-child state, a switching state, a fork state, or a final state. We modified the goto graph $\mathbf{G_0}$ into a different way which is represented by explicit state only.

Assume that state *P* is a single-child state and is represented by string $\boldsymbol{S_1}$. State *R* is said to be a descendent state of state *P* if it is represented by $\boldsymbol{S_1 S_2}$, where $\boldsymbol{S_2}$ is a non-empty string. Furthermore, state *R* is said to be a descendent explicit state of state *P* if *R* is an explicit state and a descendent state of state *P*. State *R* is said to be the nearest descendent explicit state (NDES) of state *P* if state *R* is a descendent explicit state of state *P* and there is no other descendent explicit state of state *P* which is represented by string $\boldsymbol{S_1 W_1}$ where string $\boldsymbol{W_1}$ is a proper prefix of string $\boldsymbol{S_2}$. The data structure for the single-child state *P* includes *P*.pattern, *P*.distance, and *f*(*P*), where ***P*.pattern** and ***P*.distance** store, respectively, the identification of the pattern $\boldsymbol{P_l}$, and $|S_1S_2|$.

Only the goto graph $\mathbf{G_0}$ is modified, the original data structure is still needed. It doesn't make any different on branch state and leaf state (or final state). So we add an additional data structure shown if Fig. 6 for the first single-child state on $\mathbf{G_0}$. Bytes 2, 3, and 4 store the failure state $f(P)$. Bytes 5, 6, and 7 store the next explicit state it will enter according to the goto function. If it is not a fork state, bytes 8 and 9 store the ***P*.distance**. Bytes 10 and more (if needed) store the ***P*.pattern**. If it is a fork state, then seven more bytes are needed. Bytes 8, 9, and 10 store the start state of the goto graph to be traversed by a forked process. Bytes 11 and 12 store the *min* value and bytes 13 and 14 store the *max* value of the mark. Bytes 15 and 16 store the ***P*.distance**. Bytes 17 and more (if needed) store the ***P*.pattern**.

Non-branch, non-leaf explicit state

<table>
<tr><td>Final</td><td>Fork</td><td>11</td><td></td></tr>
<tr><td colspan="4">f(P): 3 bytes</td></tr>
<tr><td colspan="4">R: 3 bytes</td></tr>
<tr><td colspan="4">fork(P): 3 bytes or empty</td></tr>
<tr><td colspan="4">min: 2 bytes or empty</td></tr>
<tr><td colspan="4">max: 2 bytes or empty</td></tr>
<tr><td colspan="4">distance: 2 bytes</td></tr>
<tr><td colspan="4">σ : 1*(distance) bytes</td></tr>
</table>

Figure 6. Data structures for Non-branch, non-leaf explicit state.

Chapter 6.

Programming Schedule

In this section, we will describe the programming schedule in detail. There are six processes in this program, each of them has their own input and output. The main idea of this program is dictated by three parts: the matching machine construction, data compression, the scanning engine. Process 1 to 4 is the construction part, including the pre-filter, goto function, failure function, and output function. Process 5 handles the data compression. In this process, we combine the goto, failure and output function into a form of data structure we describe in section 5. Process 6 is the scanning part. In this process, we can really scan a file and show that if there is any pattern matched. Each process is described in the following statement in detail individually:

Process 1: Signature analysis

Inputs: Signature file

Outputs: **NumSignature**, **eacwp.pattern[NumSignature]**

Description:

Since we care about the regular expression, each signature is fragmented into several segments according to their operator. And we need to know how many segment does a signature has. If it's a plain string, it's obvious that it doesn't need to be fragmented, so the segment number must be one. For each segment, we not only store the actual string, but also other information, ex. Length, operator type following the segment. All this information will be stored under **eacwp.pattern**.

Process 2: Pre-filter construction

Inputs: eacwp.pattern

Outputs: Pre-filter, Advancement table

Description:

Let the windows length m=10, block size k=4. We hash the series of 4 bytes into 18 bits, hence the pre-filter has 2^18=262144 entries. Each hash result will reply a bitmap with the size of 8 bits. So the total size of the pre-filter is about 256k bytes. Note that the advancement table is used to look up the pre-filter's advance number. In that way, we don't need to do the online computing to get the advancement number.

Process 3: Goto graph procedure

Inputs: **eacwp.pattern**

Outputs: **Numstate**, goto function, output function, Configuration

Description:

During the construction of the graphs, we can also decide the output function. It's important for us to remember all the switching state and it's represented pattern sting, in that way, it's possible to get the all feasible configurations. Note that it's impossible to construct the failure function before we finish all the graph's constructions, since we need to know all feasible configurations and its corresponding goto graph when we build the failure function. And the fork transition is not completed yet. We only decide the fork transition on the goto graph $\mathbf{G_0}$ during the construction, but not all the other level's graph.

Process 4: Failure function procedure

Inputs: **Numstate**, goto function, Configuration

Outputs: failure function, fork transition

Description:

We finish the fork transition and build the failure function state by state in this process. After that, the pattern matching machine's construction is completely finished.

Process 5: Data compression

Inputs: goto function, failure function, output function, fork transition

Outputs: **eacwp.datastructure[NumState]**

Description:

Before we combine the three main functions and the fork transition into a special data structure, the modification of the goto graph $\mathbf{G_0}$ is needed. As we mentioned in section 5, in order to reduce the memory requirement, we represent the goto graph $\mathbf{G_0}$ in a different way. Note that this modification is only for memory reduction, the data structure is still suitable if we don't modify the goto graph $\mathbf{G_0}$. The data structure **eacwp.datastructure** is the only one we need in verification module.

Process 6: Scanning procedure

Inputs: **eacwp.datastructure**, pre-filter, Advancement table, Text file

Outputs: Matched Signature ID and starting position, if signature occurs in Text.

Description:

During the scanning process, we have to maintain an information : **Active_graph**. The procedure will be end if there is any pattern matched or the text file is finished.

The program can also apply on the internet. The only difference is that we need to modify the program for packet based. Since the original program will be end when the input file comes to the end if there is no pattern matched. But in the network, all the file transmission is based on the packet, in other words, we have to scan these packets in order to guarantee the whole completed file to be scanned. It means that the scanning process doesn't end until all the packets have been scanned. In order to continue the scanning process between each packet, we must to remember all the status about the scanning process. The status is including the state we are going to continue, and if it's during the traversal on $\{min, max\} - \text{graphs}$, the counter and the value of min and max are needed. Note that it's possible that the process will stop on multiple goto graphs when we finish a single packet's scanning. Not only the state information, pre-filter's window and its next advancement are both needed too. And the program will end if there is a pattern matched or all the packets are completed finished its own scanning process.

Chapter 7.

Experimental Result

In this section, we compare the performance of our proposed signature matching system with that of the ClamAV implementation and its enhancement [?]. Both throughput performance and memory requirement are compared. Programs are coded in C++ and the experiments are conducted on a PC with an Intel Pentium 4 CPU operated at 2.02GHz with 1.75GB of RAM.

We traced the ClamAV implementation, extracted the ideas, and re-wrote the codes for our experiments. In the ClamAV implementation, a trie of height two is constructed for the first two bytes of all patterns based on AC pattern matching machine. Effectively, patterns are grouped based on their first two bytes. The failure function for non-leaf states is eliminated because the next move function δ is adopted. The next move function δ is defined as $\delta(P,\sigma)=g(P,\sigma)$ if $g(P,\sigma)\neq fail$ or $\delta(P,\sigma)=\delta(f(P),\sigma)$ otherwise. When the first two bytes of some group are matched, a sequential search is performed for all patterns in the group. Different from our proposed scheme, a regular expression is fragmented by the three *, ?, and {*min*, *max*} operators. A data structure is maintained to indicate up to which fragment a regular expression had been matched and the position in the text of the last matched fragment. Consider a regular expression which consists of k fragments. Assume that the first e fragments had been matched and the e^{th} fragment ends at the i^{th} position of the text. Assume further that another fragment is matched at the j^{th} position. This newly matched fragment is discarded if it is not the $(e+1)^{th}$ fragment or i and j do not satisfy the condition specified by the operator which separates the e^{th} and the $(e+1)^{th}$ fragments. As an example, consider a regular expression $RE = sre_1 \, ? \, sre_2 \, \{2,4\} \, sre_3 \, \{3,5\} \, sre_4$. Assume that the first fragment sre_1 was matched at the i^{th} position of the text. If the second fragment sre_2 is matched at the $(i+|sre_2|+1)^{th}$ position, then the data structure will be updated to indicate that the first two fragments are matched and the position of the second fragment is matched at the $(i+|sre_2|+1)^{th}$ position. Assume that a fragment is further found at the j^{th} position, then the data structure is further updated only if it is the third fragment sre_3 and j satisfies $2 \le j\text{-}i\text{-}|sre_2|\text{-}|sre_3|\text{-}1 \le 4$. Otherwise, the newly matched fragment is discarded and the data structure remains intact.

As of November 2009, the ClamAV database has 30,385 signatures. Among these signatures, 1599 are regular expressions. After converting ? operators into {*min*, *max*} operators, there are ? regular expressions which contain at least one {*min*, *max*} operator. The shortest pre-filter pattern has only two bytes. To demonstrate the potential benefit of using a pre-filter, we discard a string which generates a pre-filter pattern of length shorter than 6. We eliminated 217 signatures based on this criterion.

In our simulations, we select $K = 6$ and $L = 3$ with four pre-filters. Let $t_j t_{j+1} \ldots t_{j+5}$ be the string contained in the search window. Since hash functions are not the focus of this paper, we use simple ones. The i^{th} hash function used in our experiments is simply $t_{j+4-i} t_{j+5-i} \otimes t_{j+5-i} t_{j+6-i}$, where $\otimes$ represents the bitwise exclusive-OR operation.

Fig 7 shows the comparison of CPU execution time for randomly generated files of various sizes without any signature occurrence. We call our proposed system eacwp for short. It can be seen that the CPU execution time is proportional to file size. The CPU time required by the ClamAV implementation is about 4 times of that required by eacwp. We expect the performance improvement to become larger as the number of signatures increases. The reason is that, in ClamAV implementation, the number of strings in a group with identical first two bytes increases as the number of signatures increases. Since the ClamAV implementation performs sequential search for strings in the same group, it consumes more CPU time to find the match in a larger group.

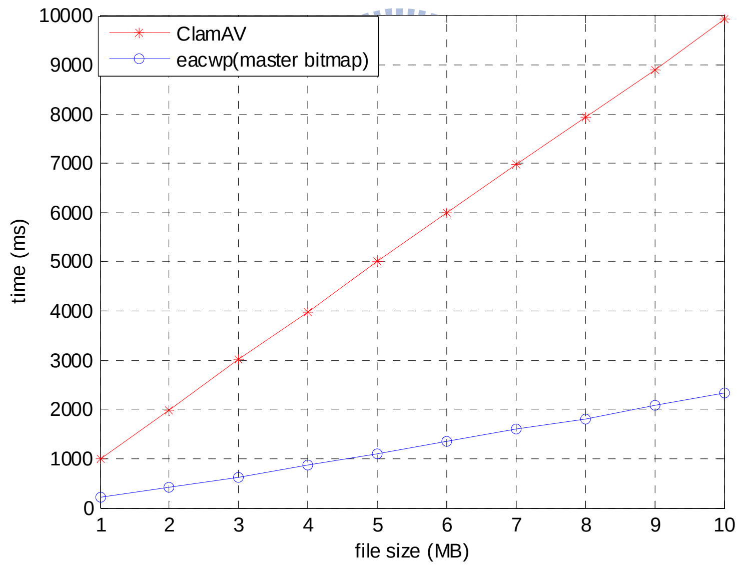


Figure 7. Performance comparison of ClamAV implementation and our proposed signature matching system for clean files of various sizes.

As for memory requirement, ClamAV implementation uses 3.57M bytes and eacwp uses about 5.7M bytes. The pre-filter requires 256K bytes and the verification module needs 5.5M bytes. We believe the amount of memory required by our proposed signature matching system is acceptable for practical systems.

Now we modify the pre-filter with a new value of K = 10 and L = 4. And we increase the hash value's bit number so that the collision due to the hash function will be reduced. So the size of the pre-filter will come to 1M bytes (20 entries, 2^20=1048576). Because of the difference of window size, we discard a string which generates a pre-filter pattern of length shorter than 10. We eliminated a little more, about 377 signatures based on this criterion. And one more difference is that we apply two pre-filters. Each pre-filter is built with its own hash function which is different from the other one. When the first pre-filter's query result consults the verification module, we apply the second pre-filter instead. The verification module is consulted iff the two pre-filters both consult the verification module. The memory requirement grows up a little, comes to 7.5M bytes. The pre-filter requires 2M bytes and the verification module needs 5.5M bytes. We expect the improvement will work on the performance's advancement. Fig 8 shows the result and confirms our expectation. The CPU time required by the ClamAV implementation is about more than 10 times of that required by modified eacwp.

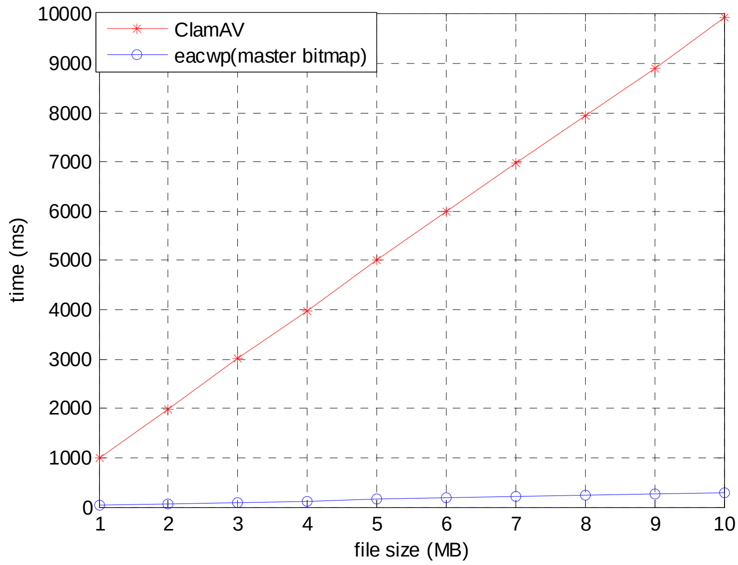


Figure 8. Performance comparison of ClamAV implementation and our proposed signature matching system for clean files of various sizes.

References

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

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

[3] 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.

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

[5] 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.

[6] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Joannidis, “Gnort: High performance network intrusion detection using graphics processors,” In Recent Advances in Intrusion Detection (RAID), 2008.

[7] J. Rejeb and M. Srinivasan, “Extension of Aho-Corasick algorithm to detect injection attacks,” SCSS (1) 2007.

[8] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching,” TR-94-17, 1994.

[9] B Bloom, “Space/time trade-offs in hash coding with allowable errors,” ACM, 13(7): 422–426, May 1970.

[10] A. Broder and M. Mitzenmacher, “Network applications of Bloom filters: a survey,” Internet Mathematics, vol. 1, no. 4, pp. 485–509.

[11] R. Smith, C. Estan, and S. Jha, “XFA: Fast signature matching with extended automata,” In IEEE Symposium on Security and Privacy, May 2008.