

國立交通大學

電信工程研究所

碩士論文

動態過濾器應用在偵測病毒特徵碼防毒軟體

Dynamic Pre-filter Designs for Signature Based Anti-Virus/Worm  
Applications

研究生：王廣煜

指導教授：李程輝 教授

中華民國 一零二年七月

動態過濾器應用在偵測病毒特徵碼防毒軟體  
Dynamic Pre-filter Designs for Signature Based  
Anti-Virus/Worm Applications

研究生：王廣煜

Student : Kuang-Yu Wang

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 College of Electrical and Computer Engineering

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science,

June 2013

Hsinchu, Taiwan, Republic of China

中華民國 一零二 年七月

# 動態過濾器應用在偵測病毒特徵碼防毒軟體

學生：王廣煜

指導教授：李程輝

國立交通大學電信工程研究所

## 摘 要

字串比對在病毒偵測的應用上是一門很重要的技術，因為字串比對的精確度比異常行為偵測來的高。目前有許多有名的字串比對演算法已經被提出，其中 Aho-Corasick (AC) 是一種可以同時比對多隻病毒的演算法。然而，AC 演算法偵測的對象是以普通字串表示的病毒，無法偵測以正規表示式表示的病毒。

在我們提出的字串比對系統中，主要是偵測正規表示式的病毒特徵碼，包含動態過濾器與驗證模組兩部分。動態過濾器的主要目的是快速移動到檔案可疑的病毒位置，它透過將相對應的字串資訊逐步加入系統中，可以避免不必要的字串資訊加入，增強效能。驗證模組是驗證動態過濾器找出來的可疑位置是否真的是病毒特徵碼的某一段，我們事先將病毒特偵碼分段建造狀態機，驗證模組只需要針對可能的狀態機進行追蹤，減少時間上的浪費。

**關鍵字：**Aho-Corasick 演算法、字串比對、正規表示式、動態過濾器



# Dynamic Pre-filter Designs for Signature Based Anti-Virus/Worm Applications

Student : Kuang-Yu Wang

Advisors : Prof. Tsern-Huei Lee

Institute of Communication Engineering  
National Chiao Tung University

## ABSTRACT

Pattern matching is an important technology in anti-virus/worm applications and is more accuracy than behavior anomaly. Many famous pattern matching algorithms have been presented in the past, and Aho-Corasick (AC) is one of the famous algorithms that can match multiple patterns simultaneously. However, the AC algorithm was developed for plain strings while virus/worm signatures could be specified by simple regular expressions.

Our proposed signature matching system which consists of a dynamic pre-filter and a verification module is designed for simple regular expressions detection. The main purpose of dynamic pre-filter is to quickly find the starting position of suspicious substrings which may result in match of some signatures. It can avoid unnecessary information by adding a few fragments of signature to enhance the performance. The verification module is used to verify whether there is any virus at suspicious position found by dynamic pre-filter. We built the state machine in advanced according to the fragments of signatures. The verification module only traces the possible state machine to save the time.

**Keywords:** Aho-Corasick algorithm, pattern matching, Regular expression, Dynamic pre-filter

# 誌 謝

感謝我的指導教授—李程輝教授。從專題到研究，他引導我學習新的知識、彙整資訊多方思考問題的所在，在這兩年的研究生活中，我從老師身上學習到了研究應該有的態度與思考，獲益良多。

感謝我的父母對我的栽培，他們對我的支持與鼓勵，讓我能不顧後顧之憂，全心學習知識。

感謝交通大學電信工程學系 NTL 實驗室的各位伙伴，學長姐們的熱心指導；與同儕之間的互相討論；學弟妹們的意見交流，讓我的研究能順利完成，謝謝。

感謝一路以來幫助我的所有好友，在我需要幫助時，適時地拉我一把，成為我能努力不懈、勇往直前的動力。

2013/07 王廣煜

# 目 錄

中文摘要		i
英文摘要		ii
誌謝		iii
目錄		iv
圖目錄		vi
表目錄		vii
Chapter1	Problem definitionIntroduction	1
Chapter2	Problem definition and related work	4
2.1	Problem definition	4
2.2	Pre-filter	5
2.2.1	HASH function	5
2.2.2	Pre-filter operation	6
2.3	Stateful Pre-filter	8
2.4	Static pre-filter with verification	9
2.4.1	Static pre-filter	10
2.4.2	Verification	11
Chapter3	Proposed Algorithm	15
3.1	Hash tables of dynamic pre-filter	15
3.2	Hash functions	18
3.3	Operation of dynamic pre-filter	19
Chapter4	Simulation	21
4.1	Performance comparisons between static pre-filter and dynamic pre-filter	21
4.2	Time comparison	22
4.2.1	Pre-filter module	23
4.2.2	Verification module	24

4.3	Memory requirement	26
Chapter5	Analysis	28
5.1	Average window advancement	28
5.2	Derivation of $P_{j,i}$	30
5.3	Numerical result	30
Chapter6	Conclusion	33
References		34





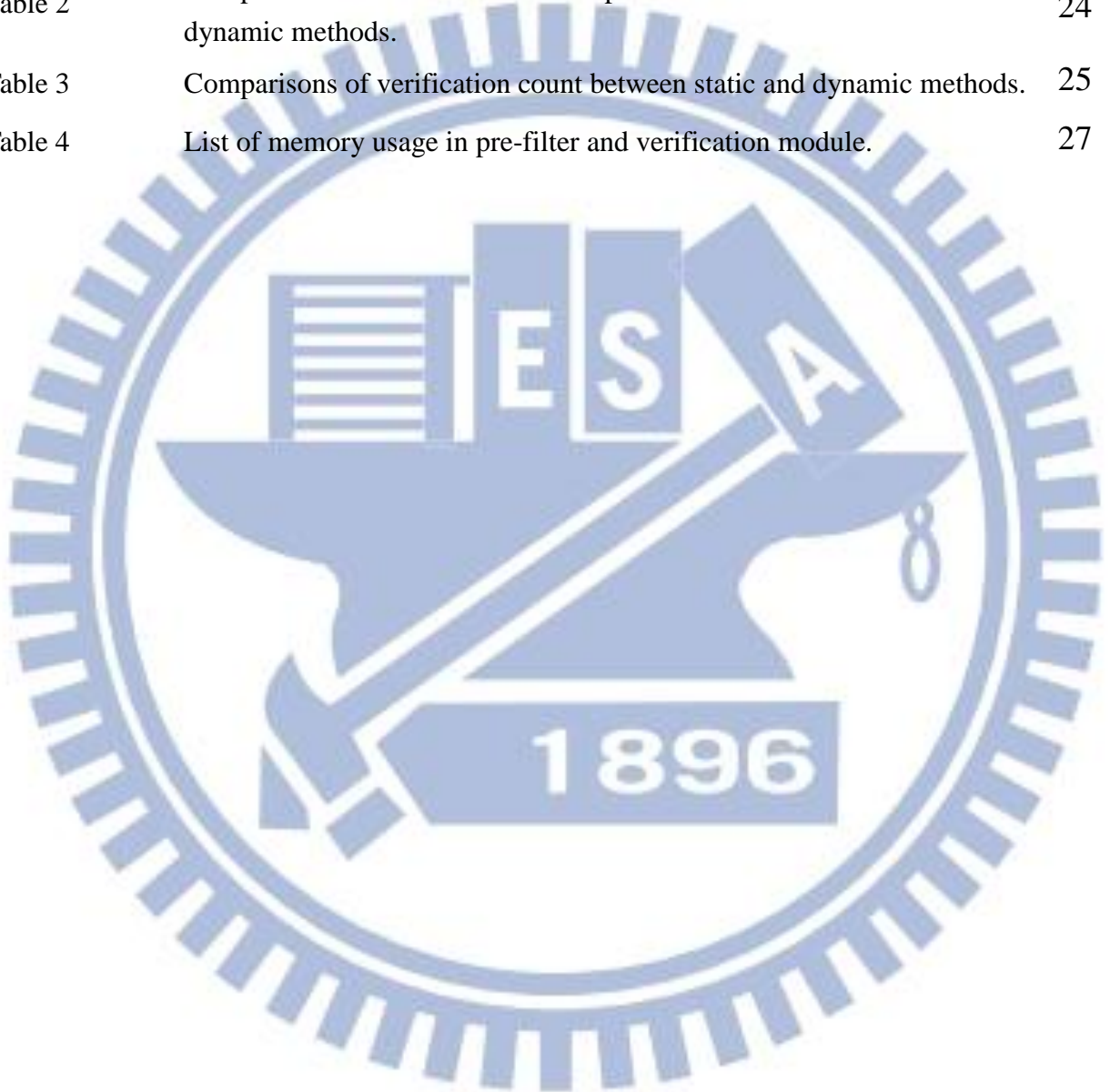
# 圖目錄

Fig. 1	Hash table	6
Fig. 2	Pre-filter operation	7
Fig. 3	The stateful pre-filter architecture for $W = 6$ and $k = 3$	9
Fig. 4	Example of Goto graphs for $RE_1 = abc$ , $RE_2 = ab*cd*e$ , $RE_3 = bc*ad*e$ , $RE_4 = pqr*vs$ , and $RE_5 = pq\{2,4\}qrqs\{3,5\}tu*vw*x\{2,6\}y$	12
Fig. 5	Comparison of hash table with static and dynamic methods for example $RE_1 = abcdefghi*12345$ and $RE_2 = uvwxyz*9876543210$	16
Fig. 6	Procedure of updating hash table using dynamic pre-filter.	17
Fig. 7	The diagram of two hash functions	18
Fig. 8	Two hash tables for different size of window	19
Fig. 9	Performances comparison of static and dynamic methods with a virus inserted in the file.	22
Fig. 10	Performances comparison of static and dynamic methods in pre-filter module with a virus inserted in the file.	23
Fig. 11	Performances comparison of static and dynamic methods in verification module with a virus inserted in the file.	25
Fig. 12	Performance comparison of static and dynamic methods with less memory requirement in dynamic method.	27
Fig. 13	Average window advancement with various number of fragments in hash table with $N=2^{20}$ .	31
Fig. 14	Fig. 14 Average window advancement with various number of fragments in hash table with $N=2^{15}$ .	32



# 表 目 錄

Table 1(a)	The failure function for the example used in Fig. 4.	13
Table 1(b)	The output function for the example used in Fig. 4.	13
Table 2	Comparisons of numbers of shift in pre-filter module between static and dynamic methods.	24
Table 3	Comparisons of verification count between static and dynamic methods.	25
Table 4	List of memory usage in pre-filter and verification module.	27



# Chapter 1.

## Problem definition Introduction

---

Two major techniques are used in virus detection. One is behavior anomaly and another is pattern matching. Behavior anomaly can detect virus when abnormal behavior occurs. For instance, an infected computer would have higher new connections rate than a normal computer would have. This abnormal behavior can be detected by observing the number of new connections [1]. However, behavior anomaly may create false positive if normal behavior is not well-defined in advanced. Pattern matching is another technique that is more accuracy than behavior anomaly. There are some significant patterns derived from malicious codes in packet. The idea of pattern matching is to find out whether there is a significant pattern hidden in the files.

Knuth-Morris-Pratt(KMP) [2], Boyer-Moore(BM) [3], Aho-Corasick(AC) [4] and Wu-Member(WM) [5] are famous pattern matching techniques. Bloom filter [6]-[12] is also a technique for pattern matching that is famous for its space-efficient probabilistic data structure. KMP and BM are efficiency only in single pattern detection. The AC and WM are designed for multiple patterns detection. However, AC algorithm may have the disadvantage about the huge memory requirement of constructing a two dimensional state transition table. Thus, some methods such as band-row format, AC-bnfa and bitmap data structure proposed for memory reduction.

The idea of pre-filter comes from shift table of WM. The purpose of pre-filter is to exclude the impossible position in the file. In other words, it can find the suspicious position in the file fast and verify suspicious position in verification module. In pre-filter module, it is realized that previous query result can accelerate the next query result to achieve high performance which is named stateful pre-filter [13]. In our paper, stateful pre-filter is applied to find the suspicious position in the file and is described in chapter 2.

Because of virus variation, the types of significant patterns become more complicated nowadays. Regular expressions (REs) can express significant patterns better than plain strings can do. The significant patterns expressed in REs are often simple. For instance, the patterns defined in Clam Anti-virus (ClamAV) [14] consist of plain strings and three operators :

- \* : match any number of symbols
- ? : match any symbol
- {min, max} : match minimum of min, maximum of max symbols

We separate the REs with \* operators. For convenience, we use “the first fragment of RE” to represent the substring in front of the first \* operator of RE and “the first string of fragment” to represent the substring in front of the first {min, max} operator of fragment.

To detect RE, some algorithms like generalized AC [15], ClamAV, extend finite automata (XFA) [16], and Snort [17]-[18] were proposed already. Although these methods can detect RE, the performance or memory requirement may be unacceptable. Even worse is that they may cause false positive or false negative. Our goal is to propose higher performance with acceptable memory requirement for REs matching. It can be simply implemented by applying stateful pre-filter and a verification engine. After we separate the \*



and {min, max} operator with appropriate method, the left substrings of REs can regard as plain strings. The only difference is that the substrings need to be matched in the order of REs. In our implementation, we only consider first fragments of every REs at first and consider entire fragments of every REs after any first fragment matches. However, it is not efficiency to consider entire fragments of every REs at the same time when the first fragment matched. It is better to reduce the effect caused by unnecessary fragments that would lower the performance. The proposed pattern matching system consists of a dynamic pre-filter module and a verification module. Dynamic pre-filter can only add information of a few fragments to accelerate the speed of search. It has CPU execution comparison with different methods in pre-filter module. The simulation result shows that dynamic method has better performance than the static method. The verification only verifies the possible state machine to save the time.

In the following section, we have problem definition and related work in chapter 2. In chapter 3, we describe the proposed methods for our pre-filter design. Then, we show the simulation result in chapter 4 and analysis our design in chapter 5. Conclusion is in chapter 6 to end the paper.

# Chapter 2.

## Problem definition and related work

---

### 2.1 Problem definition

There is a given database contained every significant patterns that derived from malicious code. We read the significant patterns from database first in order to construct a pattern matching system that can point out the starting position of significant pattern in the given file. Our system is consists of pre-filter module and verification module. In verification module, goto function, failure function, and output function processed as AC's three functions do. Besides, fork function is an extra function in verification module which is added to process the {min, max} operator.

ClamAV is an open source anti-virus, so we can get its database easily. Therefore, in our paper, we use the same database in ClamAV for our simulation. The type of significant patterns in ClamAV's database is simple regular expression. The simple regular expression consists of plain strings and three operators : \*, ?, and {min, max}. We can only consider the \* and {min, max} operators in our design. Because consecutive ? operators can be substituted for a {min, max} operator with min = max = number of ? operators.

## 2.2 Pre-filter

The information of significant patterns is stored in membership query modules (MQs). In pre-filter module, a window is used to scan the file and a block at the end of window is used to compare with the information of significant patterns in  $MQ_i$ ,  $1 \leq i \leq W-k+1$ , where  $W$  is window size,  $k$  is block size and  $P^1 = p_1^1 p_2^1 \dots p_W^1$ ,  $P^2 = p_1^2 p_2^2 \dots p_W^2$ , ...,  $P^n = p_1^n p_2^n \dots p_W^n$  are first  $W$ -byte of first string of each significant patterns where  $n$  is the number of significant patterns. Window size is chosen as the smallest length of first string of significant patterns because it will ignore some information of shorter patterns if window size is larger than smallest length of first string of significant patterns. Substrings of significant patterns  $P_1^j = p_1^j p_2^j \dots p_k^j$ ,  $P_2^j = p_2^j p_3^j \dots p_{k+1}^j$ , ...,  $P_i^j = p_i^j p_{i+1}^j \dots p_{i+k-1}^j$ , ...,  $P_{W-k+1}^j = p_{W-k+1}^j p_{W-k+2}^j \dots p_W^j$  are stored in  $MQ_i$  for comparison,  $1 \leq i \leq W-k+1$ ,  $1 \leq j \leq n$ .  $MQ_i$  reports 1 iff the substring in block is the same as one of substrings in  $MQ_i$  and reports 0, otherwise. According to the report of  $MQ_i$ , we can shift the window to appropriate position. It is complicated to compare with all substrings in  $MQ_i$  step by step. Hence, HASH table is used to store the report of  $MQ_i$  previously.

### 2.2.1 HASH function

The report of  $MQ_i$  can be calculated in advanced and store the report in hash table by using hash function HASH. The  $h^{th}$  bit of  $MQ_i$  reports 1 iff substring  $P_i^j = p_i^j p_{i+1}^j \dots p_{i+k-1}^j$  exists such that  $h = HASH(P_i^j)$ , otherwise reports 0. Fig 1 is a diagram of hash table. Each  $MQ_i$  is a bit array with size  $2^{8k}$  because the block size is  $k$ -byte =  $8k$ -bit, there are  $2^{8k}$  possible entries. Hash table is a combination of these bit arrays of  $MQ_i$ . Elements in  $MQ_0$  are all 1 for the default value when  $MQ_i$  are empty, where  $i > 0$ .



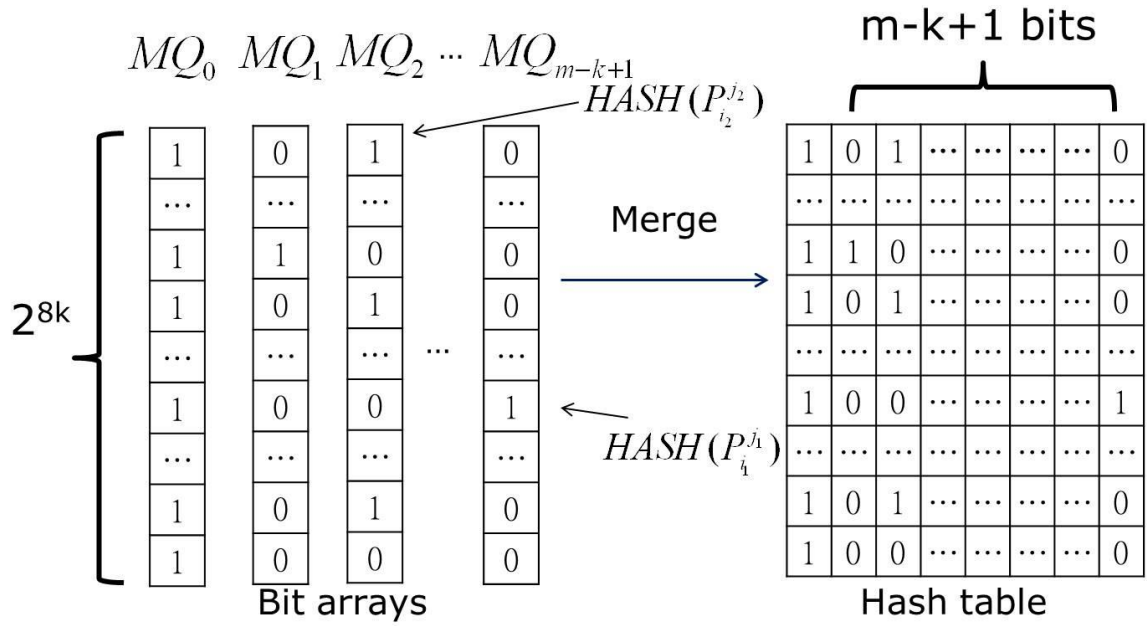


Fig. 1 Hash table

The advantage of using hash function not only can store the report previously but also can reduce the size of hash table. The number of input bits can be reduced by hash function easily. For instance, if input bits = 3 bytes = 24 bits, we can simply take first 16-bit and last 16-bit XOR to get 16-bit only. Although it would increase the probability of collision, it is worth saving lots of memory space.

### 2.2.2 Pre-filter operation

We take an example to show the pre-filter operation. Assume  $P^1 = \text{"abcdefghi"}$ ,  $P^2 = \text{"uvwxyz"}$  and text file is  $\text{"aaabcedfff"}$ . The window size  $W$  is chosen as the smallest length of significant patterns, i.e.,  $W=6$ . Then, all of significant patterns are truncated into the same length  $W$ . For this example,  $P^{1\prime} = \text{"abcdef"}$ ,  $P^{2\prime} = \text{"uvwxyz"}$ . Block exists at the end of window and the substrings in block will compare to the information in  $MQ_i$ . Therefore, block size  $k$  is chosen to be smaller than  $W$  definitely. We assume block size  $k = 3$ , there are  $W-k+1$

= 4  $MQ_i$  with different substrings in each. The substrings set of  $MQ_1$ ,  $MQ_2$ ,  $MQ_3$  and  $MQ_4$  are {abc, uvw}, {bcd, vwx}, {cde, wxy} and {def, xyz} respectively. Figure 2 shows the Pre-filter operation.

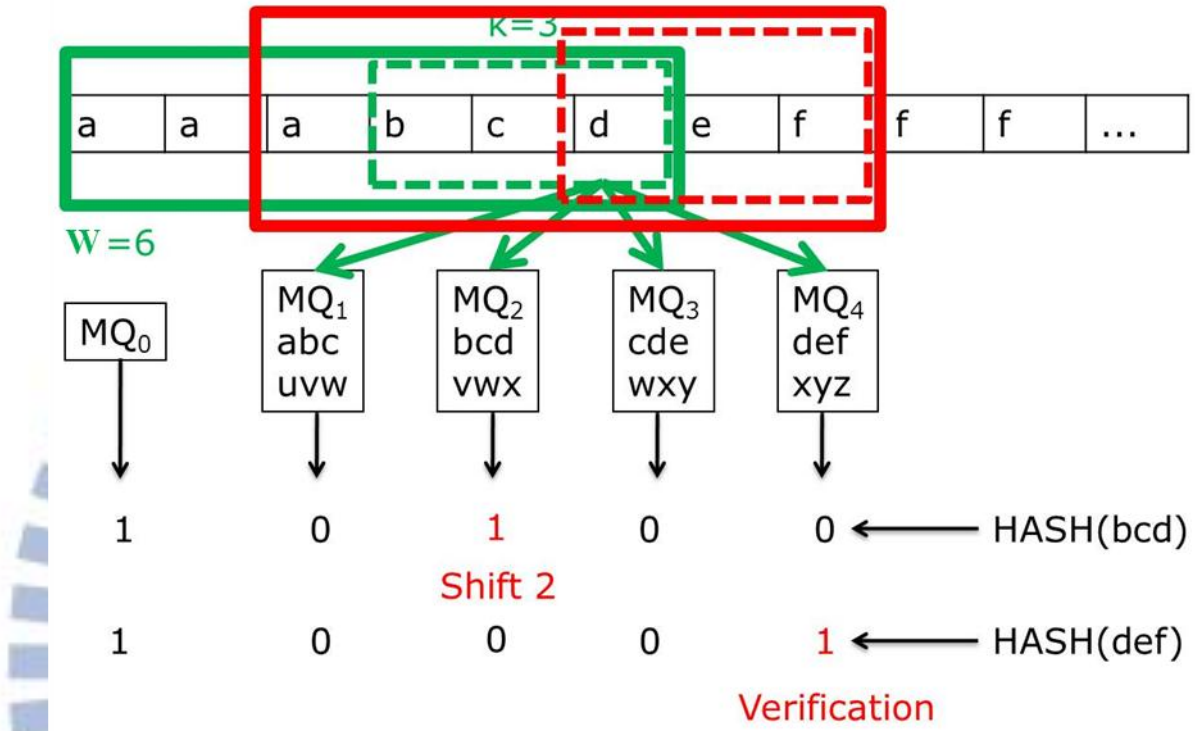


Fig. 2 Pre-filter operation

The window with small size starts at the head of file. The content of window is “aaabcd” and content of block with dash line is “bcd”. Then we process HASH (bcd) to get the report of  $MQ_i$ . It is obviously that  $MQ_2$  has substring ”bcd”, so the report is “10100”. After we get “10100”, we shift the window by 2. The window with large size is the next position. Again, the content of window is “abcdef” and content of block with dash line is “def”. Then we process HASH (def) to get the report of  $MQ_i$ . It is obviously that  $MQ_4$  has substring ”def”, so the report is “10001”. When the  $MQ_{W-k+1}$  reports 1, we enter the verification module and find the virus.

## 2.3 Stateful Pre-filter

In previous section, window of pre-filter shift according to the query report of hash function. Stateful pre-filter [13] was proposed to enhance the window advancement by calculating all of previous query reports.

In section 2.2, we mentioned stateless pre-filter. To show the different of stateless pre-filter and stateful pre-filter, we take an example to illustrate the difference. For stateless pre-filter, if the report of round 1 is “11010”, then the window would shift by 1 and if the report of round 2 is “10110”, then the window would shift by 1 again. For stateful pre-filter, if the same reports happened, it can have better window advancement than stateless pre-filter. For stateful pre-filter, the report of round 1 “11010” not only informs the window about the shift but also excludes the impossible starting position of significant patterns. In other words, the report “11010” shows that it is impossible to find the significant patterns after window shift by 2. Therefore, after the window shift by 1 in round 1, it is unnecessary to shift by  $2-1=1$  in round 2. It can be implemented by using a bitmap to memorize the state of pre-filter by calculating all of previous query reports. The bitmap shifts the bit as window shift and AND the next query report to exclude the impossible position. The bitmap is named Master Bitmap (MB) and acts as state of pre-filter. Fig. 3 shows the architecture of stateful pre-filter. After pick up the right most 1 as shift value, the window and MB' shift for next round. Stateful pre-filter is adopted in the following pre-filter module of this paper.



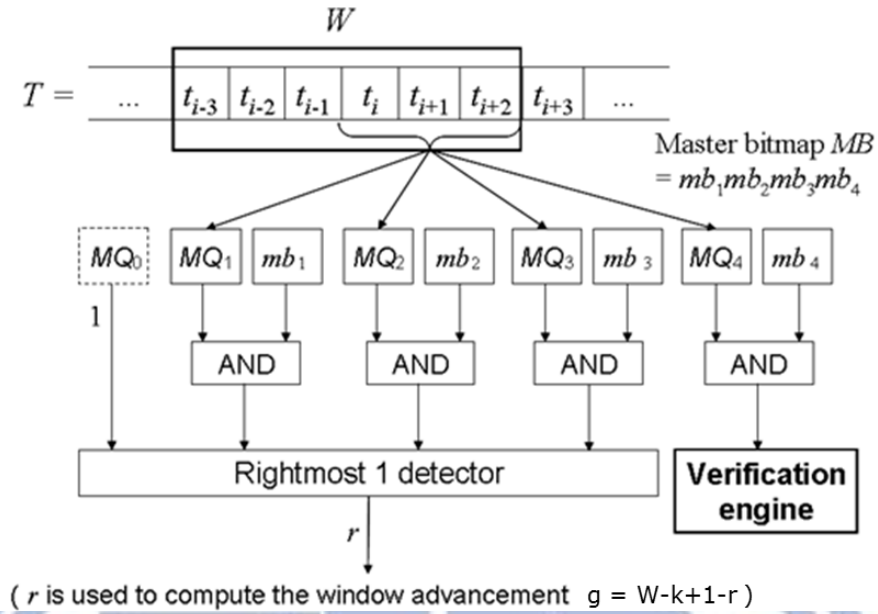


Fig. 3 The stateful pre-filter architecture for  $W = 6$  and  $k = 3$ .

## 2.4 Static pre-filter with verification

The method of using static pre-filters and verification are related to our work. The purpose of its pre-filters is to quickly find out where the suspicious substrings are. The type of virus is also simple regular expression. Regular expression (RE) is fragmented into substrings every  $*$  operator. Two pre-filters are used in pre-filter module, but only one of them process at a time. Each of pre-filters is responsible to the viruses of their database. Pre-filter 1 is responsible to the first string of first fragments of viruses so that the system can only use pre-filter 1 at beginning in order to save the resource. Because it is unnecessary to find the second, third or higher order fragment of viruses when there isn't any first fragment of virus matched before. Pre-filter 2 is responsible to the first string of every fragment of viruses. After any first fragment of viruses was found, pre-filter 2 begins processing. The verification module (verification engine) of this system is an extension of the AC algorithm that verifies the existence of virus at suspicious position. Several levels are created for fragments of REs with different order. The static pre-filter and verification are review separately below.

## 2.4.1 Static pre-filter

The pre-filter module is an extension of a bitmap-based stateful design [13]. Fragments are extracted from significant patterns in database to build two pre-filters. Regular expression is fragmented into substrings every \* operator. Let  $M$  denote the maximum number of \* operator in any REs. As a result, there are at most  $M + 1$  fragments for each RE. Let  $Y_i$ ,  $0 \leq i \leq M$ , be the set that contains the  $i^{th}$  fragments of all REs in database. All plain strings only contain exactly one fragment and are included in  $Y_0$ . Pre-filter 1 is prepared to detect first string in  $Y_0$  and Pre-filter 2 is prepared to detect first string in  $Y_0 \cup Y_1 \cup Y_2 \cup \dots \cup Y_M$ . The first string of fragments means the substring in front of first {min, max} operator if {min, max} operator exist, otherwise the fragment itself is the first string.

The  $W_1$ -byte prefix of every pre-filter 1 first string is used to construct Pre-Filter 1 where  $m_1$  is chosen to be the shortest length of the first strings in  $Y_0$ . A parameter  $k_1$  ( $\leq W_1$ ), called block size, is selected to build membership query modules. There are  $m_1 - k_1 + 1$  membership query modules, denoted by  $MQ_1^1$ ,  $MQ_2^1$ , ..., and  $MQ_{W_1 - k_1 + 1}^1$ . According to the method mentioned in chapter 2.2, we can calculate the hash table 1 for pre-filter 1.

The parameters for pre-filter 2 are denoted by  $W_2$  and  $k_2$ . Given these two parameters, there are  $W_2 - k_2 + 1$  membership query modules, denoted by  $MQ_1^2$ ,  $MQ_2^2$ , ..., and  $MQ_{W_2 - k_2 + 1}^2$ , for pre-filter 2. For proper operation, it is required that  $k_2 \leq W_2$ . Again, according to the method mentioned in chapter 2.2, we can calculate the hash table 2 for pre-filter 2.

For the example  $RE_1 = abc$ ,  $RE_2 = ab * cd * e$ ,  $RE_3 = bc * ad * e$ ,  $RE_4 = pqr * vs$ , and

$RE_5 = pq\{2,4\}qrqs\{3,5\}tu^*vw^*x\{2,6\}y$ , the sets of pre-filter 1 and pre-filter 2 first string of fragments are  $\{abc, ab, bc, pqr, pq\}$  and  $\{abc, ab, bc, pqr, pq, cd, ad, vs, vw, e, x\}$ , respectively. The parameter values can be chosen as  $W_1=2$ ,  $W_2=1$ , and  $k_1=k_2=1$ . Given the chosen parameter values, the strings  $\{ab, bc, pq\}$  are used to construct pre-filter 1 and the strings  $\{a, b, p, c, v, e, x\}$  are used to construct pre-filter 2.

## 2.4.2 Verification

The verification module is a modification of the generalized AC algorithm [15]. To process the simple REs, the generalized AC algorithm has a fork function for a  $\{\min, \max\}$  operator. The verification module also uses a fork function to handle the  $\{\min, \max\}$  operator as generalized AC does. The difference is that in verification module, multiple goto graphs are constructed for  $*$  operator. So that the information of processing pattern matching is remembered by traversing different goto graphs. There are four functions: goto, failure, output and fork that are described below.

### The goto function

A regular expression is fragmented into substrings every  $*$  operator. If  $M$  denote the maximum number of  $*$  operators in any RE, there are  $M + 1$   $\mathbf{G}$  graphs for each  $Y_i$ . The  $\mathbf{G}$  graph constructed for  $Y_i$  is called the Level  $i$   $\mathbf{G}$  graph and denoted by  $\mathbf{G}_i$ . Similarly, if  $N$  denote the number of  $\{\min, \max\}$  operators in  $Y_i$ , there are  $N$   $\mathbf{T}$  graphs in  $Y_i$ .  $\mathbf{T}$  graphs constructed for  $Y_i$  are called the Level  $i$   $\mathbf{T}$  graphs.

A goto graph, denoted by  $\mathbf{G}_0$ , is constructed with algorithm AC1 [4] for all of the first strings in  $Y_0$ . Note that self-loop is removed from starting state, is called  $\mathbf{G}_0$  graph. More



goto graphs are constructed for the remains parts in  $Y_0$ , are called Level 0 T graphs. The difference is that self-loop exists at starting state. Goto graphs of other levels can be constructed similarly. Figure 4 shows the goto graphs of previous example.

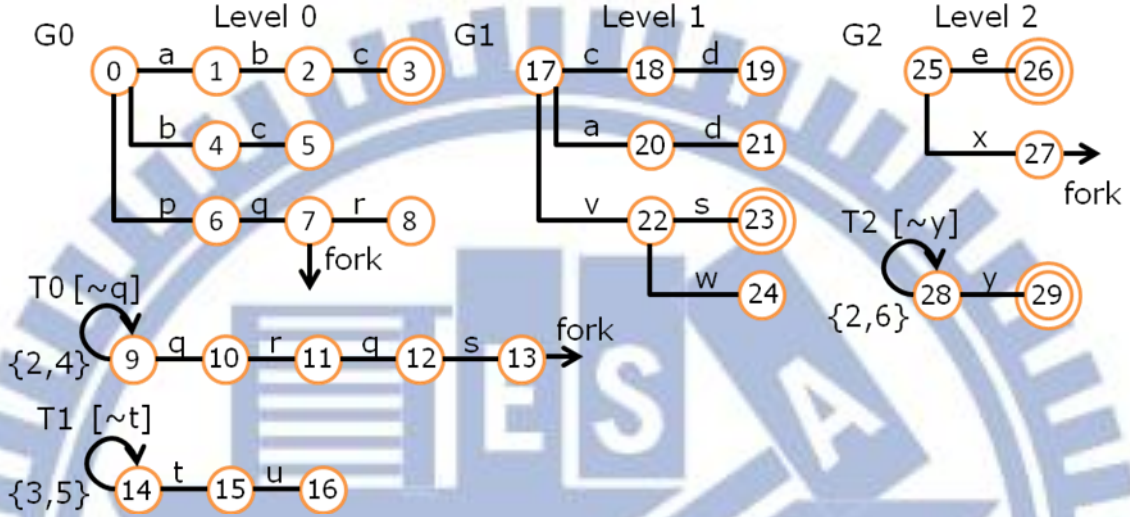


Fig. 4 Example of Goto graphs for  $RE_1 = abc$ ,  $RE_2 = ab*cd*e$ ,  $RE_3 = bc*ad*e$ ,  $RE_4 = pqr*vs$ , and  $RE_5 = pq\{2,4\}qrqs\{3,5\}tu*vw*x\{2,6\}y$ .

### The failure function

Consider every  $\mathbf{G}$  graphs, we assign  $f(P) = END$  for every state  $P$  on  $\mathbf{G}$  graphs. Because pre-filters are used previously, failure function is unnecessary in every  $\mathbf{G}$  graph. However, the  $\mathbf{T}$  graphs need failure function because the  $\{\min, \max\}$  operator would not fail if counter  $\leq$  max. The failure function of states on T graphs is computed with algorithm AC2 [4]. Table 1(a) shows the failure function for the example used in Fig. 4. In this table, the state number of the  $(i, j)^{th}$  entry is  $10*i + j$  and value 0 for  $f(P)$  represents the  $END$ . The symbol "-" means failure never occurs in this state.

$f(R)$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	-
1	9	9	10	9	-	14	14	0	0	0
2	0	0	0	0	0	0	0	0	-	28

(a)

State $S$	3	23	26	29
$output(S)$	$RE_1$	$RE_4$	$RE_2, RE_3$	$RE_5$

(b)

Table 1(a) The failure function and (b) the output function for the example used in Fig. 4.

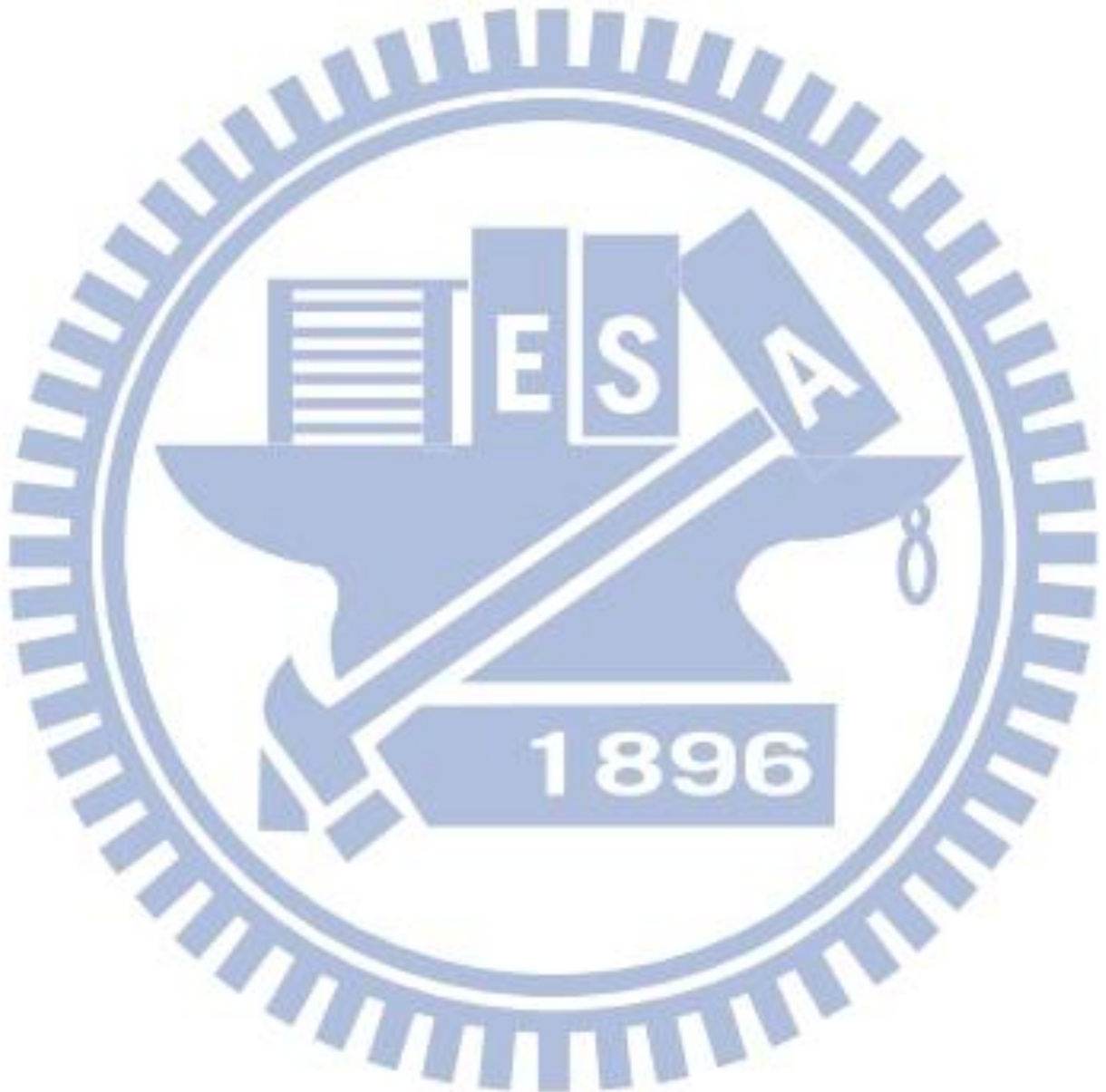
### The output function

Consider  $\mathbf{G}$  and  $\mathbf{T}$  graphs, we assign  $output(P) = \emptyset$  for every state  $P$  on  $\mathbf{G}$  and  $\mathbf{T}$  graphs. If state  $P$  is the end of some  $R_i$  in one of REs, then state  $P$  is a final state and  $output(P) = output(P) \cup \{i\}$ . On goto graphs, a final state is represented by a double circle. Table 1(b) shows the final states and the matched patterns for the example used in Fig. 4. The output of other state is empty. Similar to the information of final states, if state  $R$  is the end of some fragments, then state  $R$  is a fragment-end state and  $Leveloutput(R) = Leveloutput(R) \cup \{i\}$ .

### The fork function

The fork function is adopted to solve the  $\{\min, \max\}$  operator. Consider every  $\mathbf{G}$  graph, we assign  $F(P) = \emptyset$  for every state  $P$  on  $\mathbf{G}$  graphs. If state  $P$  is the first string of some fragments with  $\{\min, \max\}$  operator, then  $F(P)$  stores  $min = \mathbf{min}$ ,  $max = \mathbf{max}$ , and  $forked\_state =$  the start state of the  $\mathbf{T}$  graph constructed with the second string. Here,  $\mathbf{min}$  and  $\mathbf{max}$  are, respectively, the minimum and maximum values of the  $\{\min, \max\}$

operator which separates the first and the second strings in a same fragment. To complete a fragment, there would be some  $T$  graphs that have created one after another to solve all of  $\{\min, \max\}$  operators in a fragment. For convenience, a state with non-empty fork function is called a fork state.





# Chapter 3.

## Proposed Algorithm

---

In related work, we introduce a method consists of a static pre-filter module and a verification module. It is inefficient that pre-filter 2 of static pre-filter contains first string of all fragments in  $Y_0 \cup Y_1 \cup Y_2 \cup \dots \cup Y_M$  once first fragment of any regular expressions matched. Therefore, we figure out dynamic pre-filter method to solve this problem and achieve higher performance. Before first fragment of any REs matched, we only use pre-filter 1 to save the resource. The hash table of pre-filter 1 is a little different in dynamic method and is described below.

### 3.1 Hash tables of dynamic pre-filter

The length of REs in Fig. 4 is short, so we use another example in this chapter,  $RE_1 = abcdefghi*12345$  and  $RE_2 = uvwxyz*9876543210$ . In static pre-filter method, the sets of pre-filter 1 and pre-filter 2 first string of fragments are  $\{abcdefghi, uvwxyz\}$  and  $\{abcdefghi, uvwxyz, 12345, 9876543210\}$ , respectively. The parameter values can be chosen as  $W_1=6$ ,  $W_2=5$ , and  $k=k_1=k_2=3$ . Given the chosen parameter values, the strings  $\{abcdef, uvwxyz\}$  are used to construct hash table 1 of pre-filter 1 and the strings  $\{abcde, uvwxy, 12345, 98765\}$  are used to construct hash table 2 of pre-filter 2. Once any fragments in pre-filter 1 matched, pre-filter 2 is being substitute for pre-filter 1 in pre-filter module. However it's inefficiency to contain first string of all fragments at the same time in pre-filter 2 once the first fragment of any REs matched. Hence, we proposed the idea of dynamic pre-filter that can only add information of the next fragment according to what fragment matched before. For this

example, if “abcd~~fghi~~” matched, strings in hash table 2 of pre-filter 2 are only {abcde, uvwxy, 12345} that is less than static pre-filter method. Fig 5 is the comparison of hash table with static and dynamic methods, where  $H()$  represents the hash function.

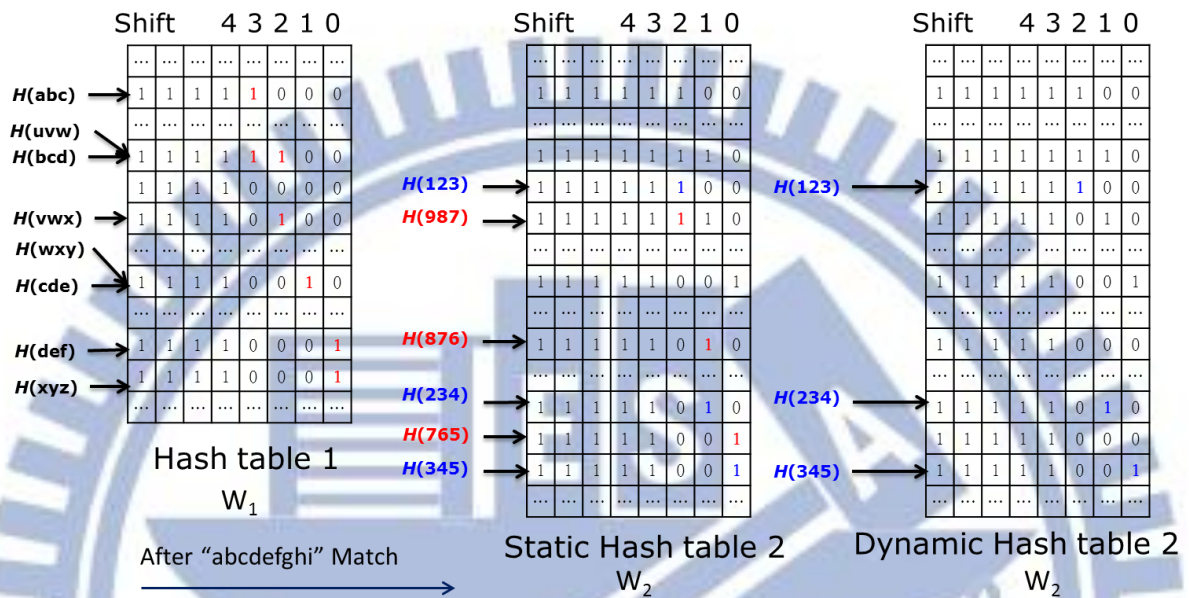


Fig. 5 Comparison of hash table with static and dynamic methods for example  $RE_1 = abcdefghi*12345$  and  $RE_2 = uvwxyz*9876543210$ .

As mentioned in related work, we stored the result of membership query modules in hash table. Therefore, we need to add information of the next fragment into hash table. However, the process of adding information needs to cut the  $m_2$ -byte prefix of the next fragment and stored the information into hash table  $W_2-k_2+1$  times. Both processes are burdensome. Fortunately, we can store the information into hash table first and only stored the hash value (index of table) at the state in verification module where previous fragment ends. Which result stored in hash table is valuable is the only thing we need to know. Hence, we use the first bit of hash table as our controlling bit. The first bit reports 1 when the result is valuable and reports 0 otherwise.

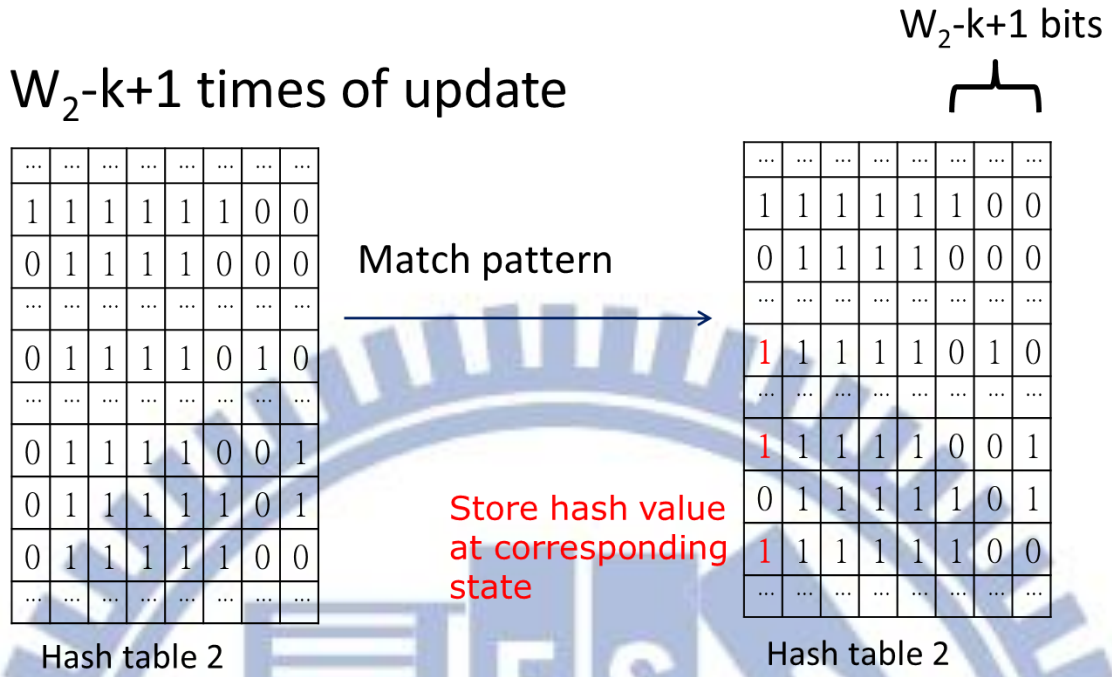


Fig. 6 Procedure of updating hash table using dynamic pre-filter.

Fig. 6 shows the procedure of updating hash table using dynamic pre-filter. In verification module, it is a state machine. Once we find the previous fragment of any REs, we must enter a fragment-end state with hash values in it. We use these hash values as index of hash table to update the information of the next fragment. The work of updating the hash table is simply OR the first bit.

Because the hash table 2 of static pre-filter contains all remaining fragments, the first bit of query result is always 1. It is obviously that dynamic pre-filter can decrease the burden of pre-filter 2. Although the information of fragments already exist in pre-filter 2, it is valuable only in the situation that the first bit of query report is 1. In other words, if first bit of query report is 0, the window of pre-filter 2 can shift maximum shift  $W_{2-k_2+1}$  which is faster than static pre-filter does.



### 3.2 Hash functions

Collision may be happened if two different substrings get the same hash value. To avoid the effect of collision, we use two hash functions in our pre-filter module, hash function A and hash function B.

For example, if substrings “bcd”、 “uvw” get the same hash value in hash function A, and substrings “cde”、 “uvw” get the same hash value in hash function B. If we only use hash function A,  $H_A(uvw)$  has information of two substrings that would decrease the shift of window. If two hash functions are used, we can AND the reports of two hash functions to get rid of the effect of collision. Fig. 7 is the diagram of two hash functions.

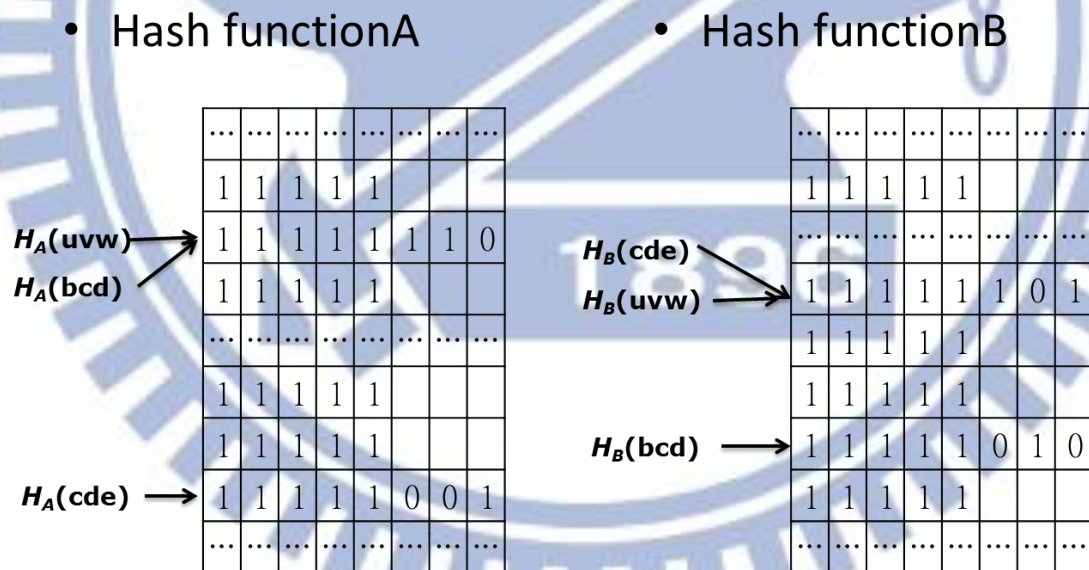


Fig. 7 The diagram of two hash functions.

### 3.3 Operation of dynamic pre-filter

The size of pre-filter 1 and pre-filter 2 would not be the same so we need two hash tables shown in Fig. 8. After first fragment of any REs is matched, we used hash table 2 for pre-filter 2 and add information of the next fragment using dynamic pre-filter method. The first bit is also used as controlling bit. The information of first strings of first fragments is also in hash table 2 and the controlling bit is initialized to 1. The difference is that the first strings of first fragments are truncated into length  $W_2$ . Hence, the reports of first strings of first fragments in hash table 2 are equal to the reports in hash table 1 shift right  $W_1 - W_2$  position.

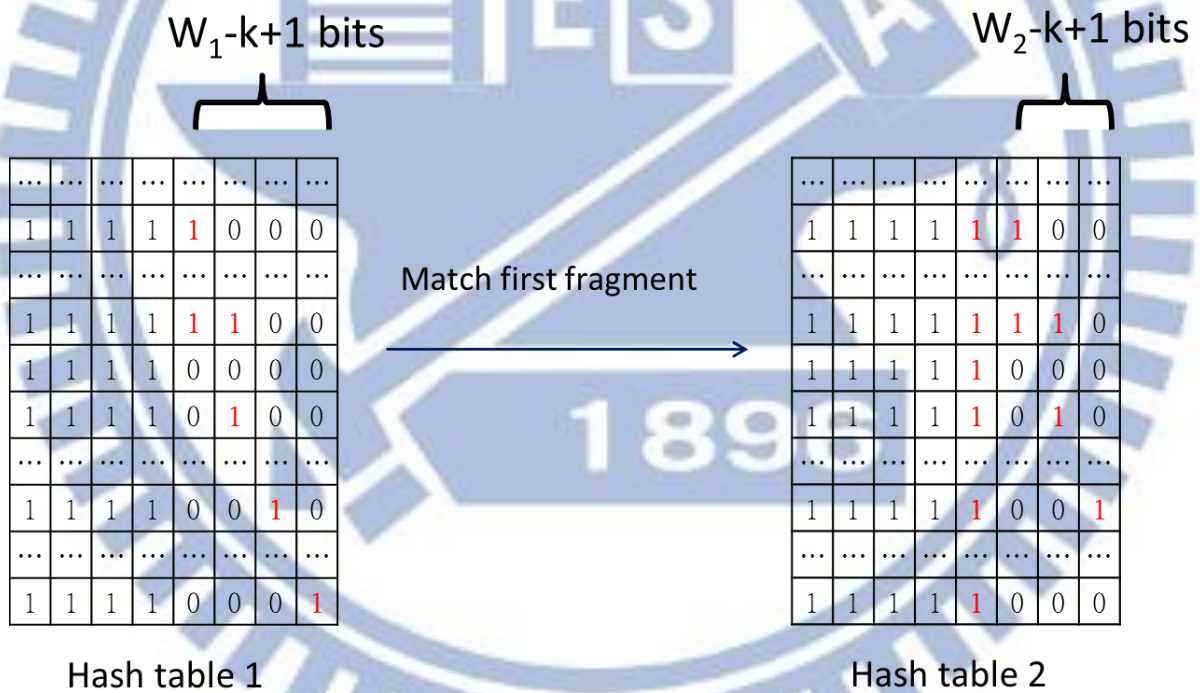


Fig. 8 Two hash tables for different size of window

The main idea of dynamic pre-filter is not only decreasing the effect of other fragments that is unnecessary but also shifting the maximum position with large window size. However, if we change the table once we encounter the first fragment of virus, the performance would

become worse tempestuously. Fortunately, we can use the length of next fragment which is going to be added in the hash table to distinguish whether we should change the table or not.

In our example, the large window size is 6, the small window size is 5 and block size is 3. According to the stateful pre-filter, the maximum shift of large window size is 4 and the maximum shift of small window size is 3. If the length of next fragment which is going to be added in the hash table is greater than 6, substitution of hash table is unnecessary. In other word, we can still shift the maximum shift with large window size. It can be implemented by adding the information of next fragment step by step which is the characteristic of dynamic pre-filter. If the length of next fragment which is going to be added in the hash table is smaller than 6, we use the hash table 2 of pre-filter 2 in pre-filter module.

There are two hash tables for two pre-filters and two hash functions for each pre-filter to decrease the effect of collision, so there are totally four hash tables to build. In dynamic method, we can calculate the first strings of first fragments and enable the first controlling bit. Also, we need to calculate the first strings of other fragments whose length are greater than or equal to  $W_1$  in advanced with controlling bit = 0. Once we need to add the fragment whose first string is greater than or equal to  $W_1$ , we need to update the controlling bit of both hash table 1 and hash table 2. If the length of first string of the next fragment is smaller than  $W_1$ , pre-filter module needs to change from hash table 1 to hash table 2, so we need to update both hash tables. The controlling bits of other entries that is never used are initialized with zero.



# Chapter 4.

## Simulation

---

### 4.1 Performance comparisons between static pre-filter and dynamic pre-filter

In this section, we compare the performance of our proposed signature matching system with static and dynamic methods in terms of throughput performances. Programs are coded in C++ and the experiments are conducted on a PC with an Intel Core2 Quad CPU operated at 2.83GHz with 4.00GB of RAM.

As mentioned in problem definition of chapter 2, we use the same database as ClamAV does. There are 30008 simple regular expressions. 1019 REs of them are REs with \* operator and 407 REs of them are REs with {min, max} operator. In database, the maximum number of \* operators in one simple RE is five which means there are six levels created for verification module. The maximum number of max and minimum number of min are 122 and 1 respectively. The minimum length of first string of level 0 is 10 which is also the window size of pre-filter 1. The window size of pre-filter 2 is 8. The block size is 4 and the size of hash table is  $2^{20} \times 1$  byte. We use two different hash functions and AND their query reports to avoid the false positive collision. One is to pick up the right 20 bits and left 20 bits XOR, the other is to pick up the right 20 bit only as reference of hash table.

We use pre-filter module to find the suspicious position. Figure 9 shows the comparison of CPU execution time for randomly generated files of various sizes with a virus in it. It can be seen that the CPU execution time is proportional to file size. The speed of dynamic pre-filter is about 32 % faster than the speed of static pre-filter. Therefore, dynamic pre-filter is better than static pre-filter.

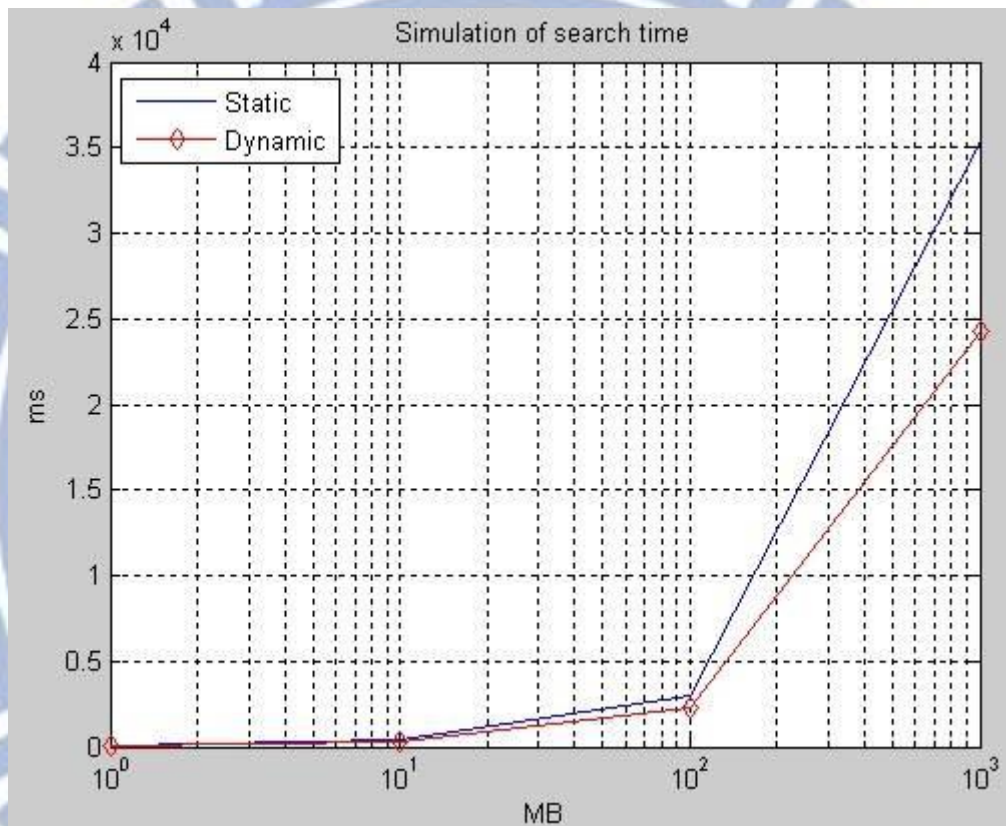


Fig. 9 Performances comparison of static and dynamic methods with a virus inserted in the file.

## 4.2 Time comparison

In this section, we compare the time in pre-filter module and in verification module. We do not take the time of reading the file into consideration because whatever algorithms need

to read the file first. Saving the time of reading the file may be the other topic that is out of our research.

### 4.2.1 Pre-filter module

The procedure in pre-filter module is as following. First, get the substrings from block. Second, put the substrings into hash function to get the report in hash table and AND with the master bitmap which acts as the state of pre-filter. Final, if  $MQ_{W-k+1}$  reports 1, then enters the verification module to verify, otherwise, shift the window according to the master bitmap. In above procedure, numbers of shift and usage amount of hash function are two major components that affect the time in pre-filter module the most. Fig. 10 shows the performance comparison of static and dynamic methods in pre-filter module only.

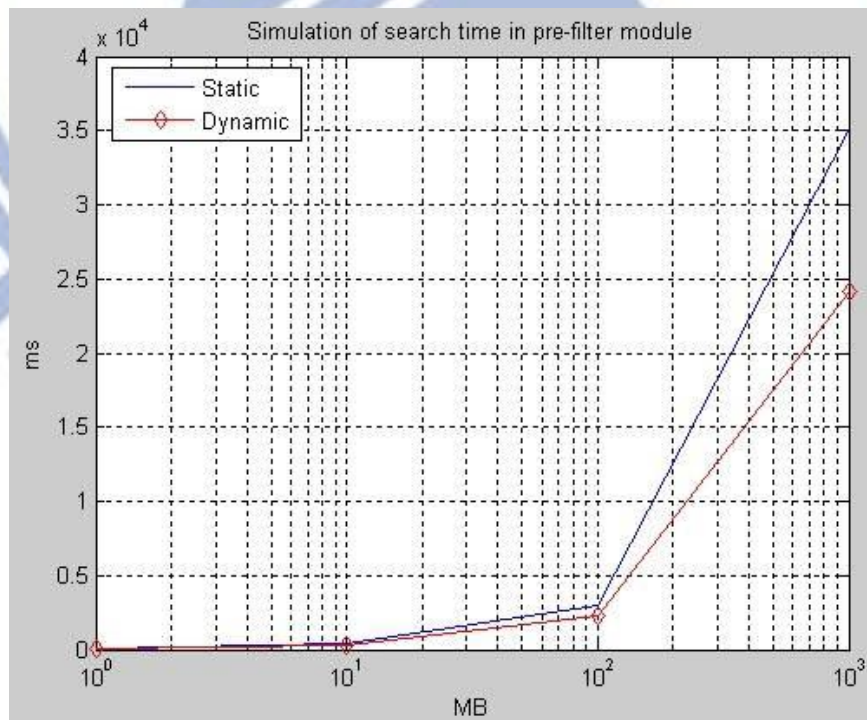


Fig. 10 Performances comparison of static and dynamic methods in pre-filter module with a virus inserted in the file.



Table 2 shows the comparisons of numbers of shift between static and dynamic methods. The dynamic pre-filter is better than static pre-filter because dynamic pre-filter can not only decrease the effect of other fragments that is unnecessary but also shift the maximum shift with large window size that lower the numbers of shift.

	1MB	10MB	100MB	500MB	1GB
Static	209,889	2,098,673	20,986,607	104,933,008	214,902,796
Dynamic	150,008	1,499,769	14,997,607	74,988,008	153,575,435

Table 2 Comparisons of numbers of shift in pre-filter module between static and dynamic methods.

#### 4.2.2 Verification module

Fig. 11 shows the performance comparison of static and dynamic methods in verification module only. It is obviously that CPU execution time in pre-filter module is the major part of search time, because our hash table is large enough to decrease the verification count. If hash table is small, the effect of collision increases. Hence, the probability of  $MQ_{w-k+1}=1$  increases and verification count increases.

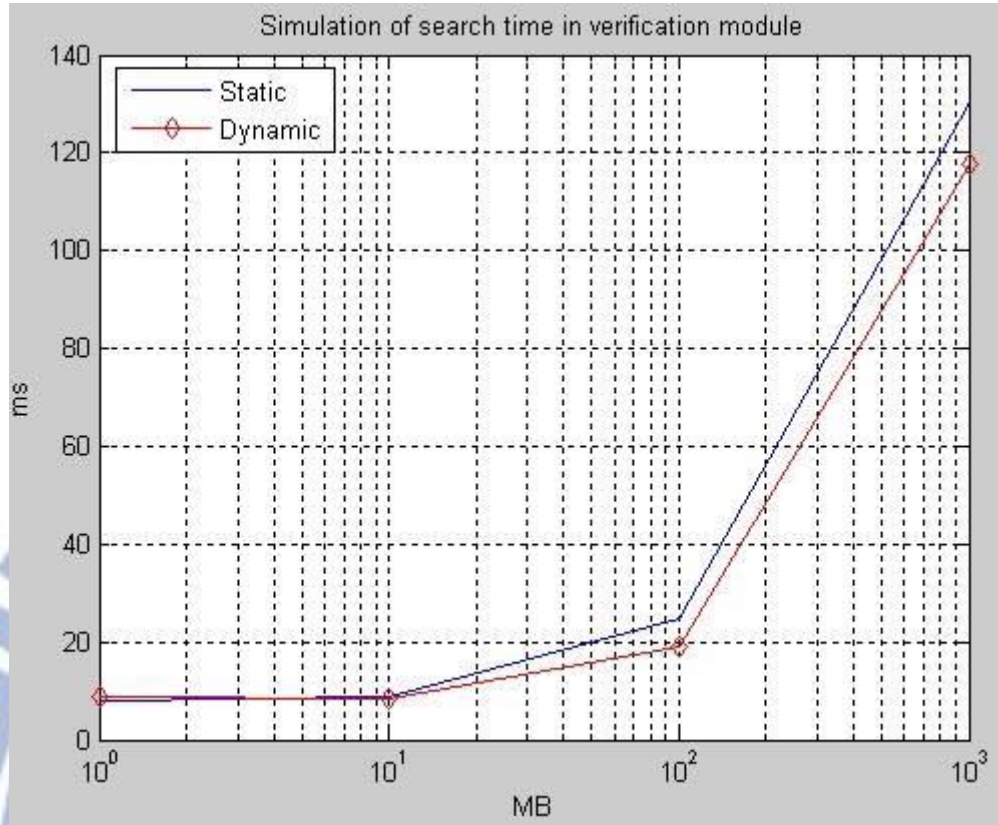


Fig. 11 Performances comparison of static and dynamic methods in verification module with a virus inserted in the file.

Table 3 shows the comparisons of verification count between all methods mentioned in chapter 3. The method using dynamic pre-filter have the fewer verification counts because dynamic pre-filter can decrease the effect of other fragments that is unnecessary. Therefore, the window of dynamic pre-filter would not stop for the fragments that are excluded from dynamic pre-filter. For example, some query reports whose  $MQ_{w-k+1}$  is 1 but first bit is 0 are excluded. The result of smaller verification count can save the time.

	1MB	10MB	100MB	500MB	1GB
Static	91	663	6,604	33,004	67,586
Dynamic	70	522	5,203	26,003	53,251

Table 3 Comparisons of verification count between static and dynamic methods.

The method of using dynamic pre-filter needs update in verification module. When we find a fragment, we enter a state. There are  $W-k+1$  hash values at this state. We update the hash table according to the hash value to add the information of next fragment into the dynamic pre-filter. Fortunately, the time of update is about 180ms/per  $10^6$  hash function which is less than the time we enhance.

### 4.3 Memory requirement

Dynamic Pre-filter needs update, so hash values are stored at corresponding states as extra memory compared with static method. In verification, it also needs some extra memory to store the index of REs to get the length of next fragment and maximum shift value or extra hash tables. Therefore, we list the memory usage of different parts as Table 4.

	variable	size	static	dynamic	purpose
Pre-filter	PreFilter1A	1,048,576			hash_tableA of prefilter1
	PreFilter2A	1,048,576			hash_tableA of prefilter2
	PreFilter1B	1,048,576			hash_tableB of prefilter1 (avoid collision)
	PreFilter2B	1,048,576			hash_tableB of prefilter2 (avoid collision)
verification	State	1,266,653			information of each state such as type or table index
	S_leaf	403,871		155,335	information of leaf state like ID and table index of hash_table
	Hash_value	70,840	X		hash_value of next fragment
	Store	652,407			store ID and endposition during scanning
	FSCTable	272,192			
	B2Table	61,750			
	B3Table	21,510			
	B4Table	12,620			
	B5Table	7,550			
	B256Table	1,010,688			
CompactedFile	2,020,509				all strings of Regular Expression



	Len	20,380	X		length of next fragment
Total			9,748,339	9,994,894	
extra				246,555	

Table 4 List of memory usage in pre-filter and verification module.

It is possible that dynamic method has better performance because of the extra memory usage. Hence, we have the simulation with the memory usage of static method is greater than the memory usage of dynamic method. We can decrease the size of hash table to make the memory usage of dynamic method lower than memory usage of static method. The difference of memory usage between dynamic and static is less than 256k bytes. Therefore, we decrease the size of hash table in pre-filter module to  $2^{19}$  for dynamic method to have less memory in dynamic method. The simulation result shows in Fig. 12. The dynamic method is still better.

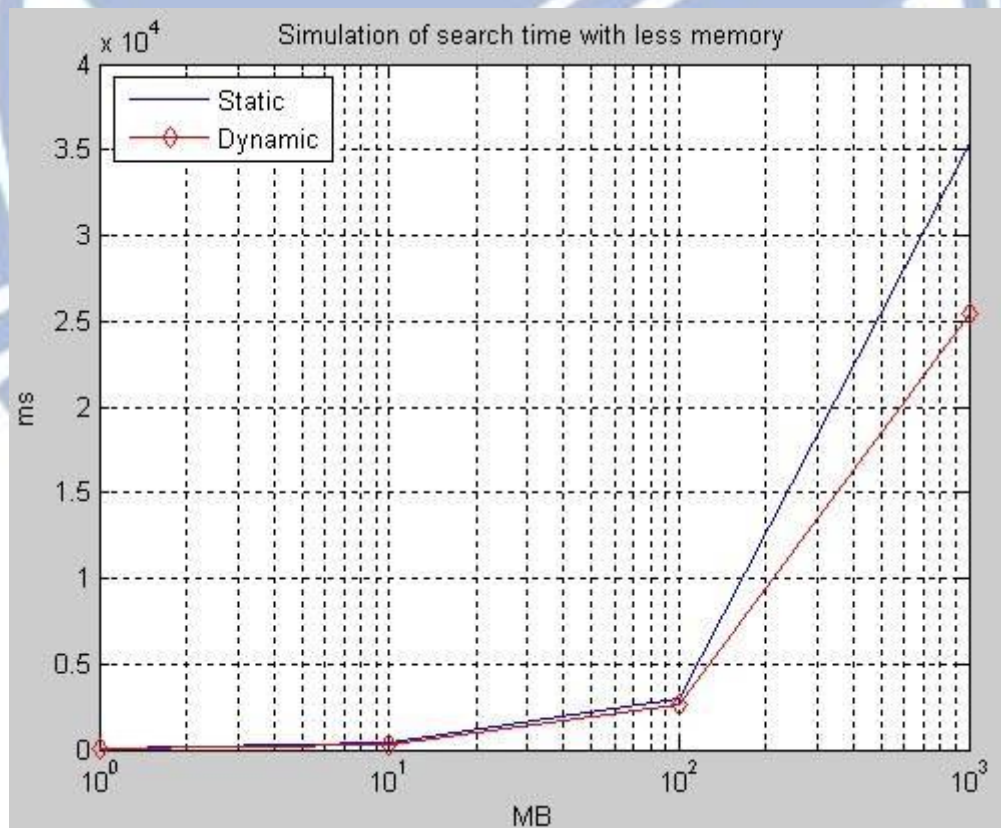


Fig. 12 Performance comparison of static and dynamic methods with less memory requirement in dynamic method.

# Chapter 5.

## Analysis

---

In analysis, we calculate the average window advancement with different number of fragments in hash table. Although in pre-filter module, we put the information of fragments into hash table in advanced, it does not work if the controlling bit is disabled. Thus, in dynamic method, we can regard number of fragments whose controlling bit is enabled as number of fragments in hash table.

### 5.1 Average window advancement

Before we calculate average window advancement, we need to define some variables that will use in the following. Let  $L$  be the number of hash functions used in a pre-filter,  $y$  represents number of fragments in hash table and  $N$  is the entries of hash table. We AND every query report from hash tables with different hash functions to get  $QB$ , so  $QB = QB_1 \& QB_2 \& QB_3 \& \dots \& QB_L = qb_1qb_2qb_3\dots qb_{W-k+1}$ . The parameter  $\rho$  represents the probability of  $qb_i=1, 1 \leq i \leq W-k+1$ , and  $\rho$  can be calculated by

$$\rho = [1-(1/N)^y]^L \quad (1)$$

Markov chain is adopted to analyze the average window advancement. In pre-filter module master bitmap needs to bitwise-AND with  $QB$  and window advances according to the result of  $MB \& QB$ . Let  $MB \& QB = mb_1mb_2\dots mb_{W-k+1}$ , then  $mb_{W-k+1}$  is used to determine whether the system needs to enter verification module or not that is irrelative to window

advancement. Therefore, states of Markov chain correspond to the value of  $mb_1mb_2\dots mb_{w-k}$ , i.e., there are  $2^{w-k}$  states on the Markov chain.

If  $X_l$  be the state after the  $l^{\text{th}}$  iteration of queries, then state transition probabilities from state  $i$  to state  $j$  can be defined as  $p_{j,i} = P(X_{l+1} = j | X_l = i)$  that will compute in next section.

Once  $p_{j,i}$  is given, we can calculate the stationary probability distribution  $\Pi = (\pi_0, \pi_1, \dots, \pi_{2^{w-k}-1})$  as following equation.

$$\begin{bmatrix} p_{0,0} & \cdots & p_{0,i} & \cdots & p_{0,2^{w-k}-1} \\ \vdots & \ddots & \vdots & & \vdots \\ p_{j,0} & \cdots & p_{j,i} & \cdots & p_{j,2^{w-k}-1} \\ \vdots & & \vdots & \ddots & \vdots \\ p_{2^{w-k}-1,0} & \cdots & p_{2^{w-k}-1,i} & \cdots & p_{2^{w-k}-1,2^{w-k}-1} \end{bmatrix} \begin{bmatrix} \pi_0 \\ \vdots \\ \pi_i \\ \vdots \\ \pi_{2^{w-k}-1} \end{bmatrix} = \begin{bmatrix} \pi_0 \\ \vdots \\ \pi_i \\ \vdots \\ \pi_{2^{w-k}-1} \end{bmatrix} \quad (2)$$

Note that summation of probability of every state equals to 1, i.e.,  $\pi_0 + \pi_1 + \dots + \pi_{2^{w-k}-1} = 1$ . Therefore, eq. (2) can turn into the following format.

$$\begin{bmatrix} p_{0,0} - 1 & \cdots & p_{0,i} & \cdots & p_{0,2^{w-k}-1} & 1 \\ \vdots & \ddots & \vdots & & \vdots & \vdots \\ p_{j,0} & \cdots & p_{j,i} - 1 & \cdots & p_{j,2^{w-k}-1} & 1 \\ \vdots & & \vdots & \ddots & \vdots & \vdots \\ p_{2^{w-k}-1,0} & \cdots & p_{2^{w-k}-1,i} & \cdots & p_{2^{w-k}-1,2^{w-k}-1} - 1 & 1 \\ 1 & \cdots & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} \pi_0 \\ \vdots \\ \pi_i \\ \vdots \\ \pi_{2^{w-k}-1} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \quad (3)$$

In eq. (3),  $\Pi$  can be computed by matrices operation. Let  $\bar{G}$  denote the average



window advancement and  $g_i$  represents the corresponding shift value for every state. Then  $\overline{G}_L$  can be calculated by

$$\overline{G}_L = \sum_{i=0}^{2^{W-k}-1} \pi_i g_i \quad (4)$$

## 5.2 Derivation of $P_{j,i}$

As mentioned above, state transition probabilities  $p_{j,i} = P(X_{l+1} = j | X_l = i)$ . Let  $i = i_{W-k-1} i_{W-k-2} \dots i_0$ ,  $j = j_{W-k-1} j_{W-k-2} \dots j_0$  and window advancement =  $g$  in state  $i$ . Let  $i' = i'_{W-k-1} i'_{W-k-2} \dots i'_0$  be the right-shifted master bitmap in state  $i$ . Then, there are two cases discuss below.

In case 1, if  $g=W-k+1$ , then  $i=0^{W-k}$  and  $i'=1^{W-k}$ . The state transition probabilities  $p_{j,i} = \rho^x (1-\rho)^{W-k-x}$ , where  $x$  is number of 1's in  $j$ . In case 2, if  $g < W-k+1$ , then  $i = i_{W-k-1} \dots i_g 10^{g-1}$  and  $i' = 1^g i_{W-k-1} \dots i_g$ . The state transition probabilities  $p_{j,i} = 0$  if there exist  $r$ ,  $0 \leq r \leq W-k-1$  such that  $i'_r = 0$  and  $j_r = 1$ , otherwise,  $p_{j,i} = \rho^{x_1} (1-\rho)^{W-k-x_1-x_2}$ , where  $x_1$  is number of 1's in  $j$  and  $x_2$  is number of 0's in  $i'$ .

## 5.3 Numerical result

We adopt the value of parameters used in chapter 4 to evaluate the average window advancement. Because the window size is different in two pre-filter, we have  $W=10$  and  $W=8$ . The other parameters are  $k=4$ ,  $N=2^{20}$ ,  $L=2$  and  $y$  is the number of fragments in hash table. Fig. 13 shows the average window advancement with different number of fragments in hash table.

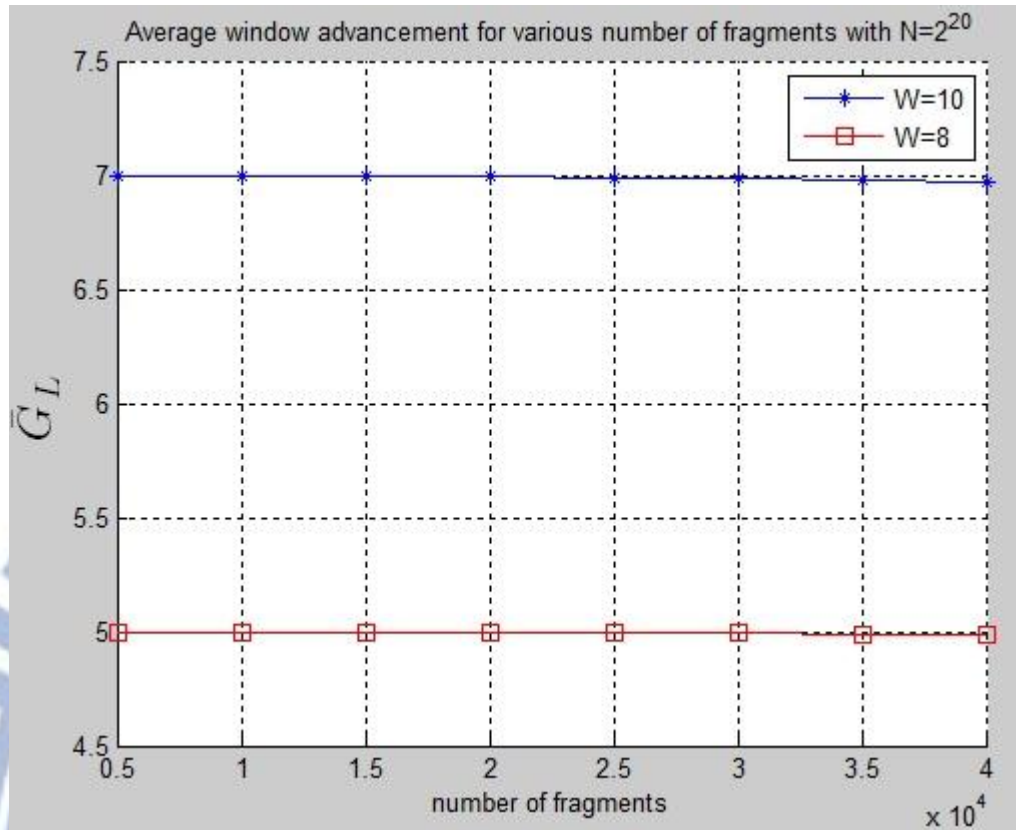


Fig. 13 Average window advancement with various number of fragments in hash table with  $N=2^{20}$ .

It is obviously that average window advancement is close to the maximum window advancement. It's because our hash table is large enough to handle the fragments up to 40000. Once a first fragment of RE was found, dynamic pre-filter can still perform the average window advancement with the better situation  $W=10$  if length of the next fragment is larger than or equal to 10.

If hash table is full of fragments, the average window advancement decrease as shown in Fig. 14. In Fig. 14, we decrease the size of hash table to make it full of fragments. However, even with the same window size in situation of Fig. 14, number of fragments in hash table of dynamic method is less than number of fragments in hash table of static method. The average window advancement is bigger with fewer fragments in hash table. In consequence, dynamic

method can perform better average window advancement than static method.

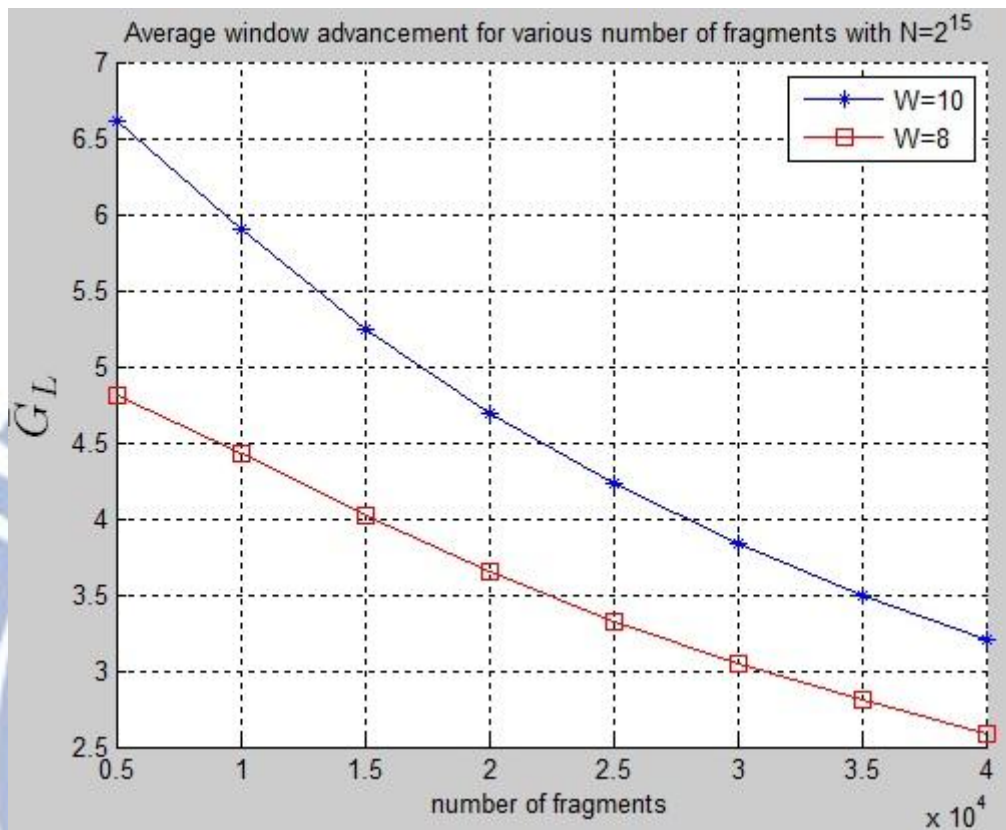


Fig. 14 Average window advancement with various number of fragments in hash table with  $N=2^{15}$ .



# Chapter 6.

## Conclusion

---

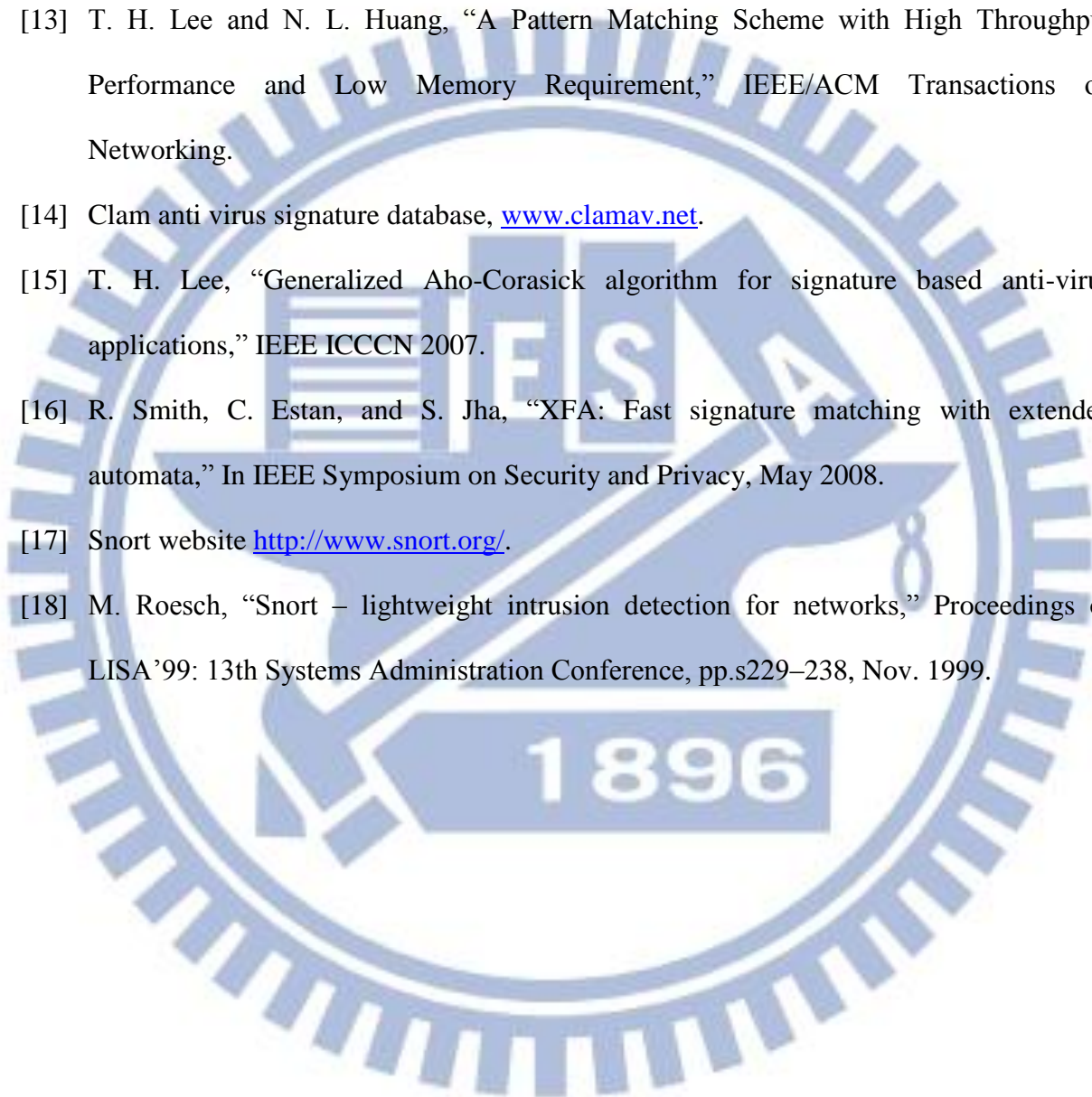
In this thesis, we have proposed a pattern matching system that consists of a dynamic pre-filter module and a verification module for both plain strings and simple REs. The purpose of dynamic pre-filter module is to quickly find the starting positions of suspicious substrings which may result in match of some signatures. It is implemented by adding substrings information step by step to accelerate the speed of search. Moreover, we can still use window with large size if length of the next fragment is larger than or equal to  $W_1$ , so the window can shift more due to the large window size.

We have discussed static and dynamic methods in pre-filter module and have compared two methods in simulation result in time complexity and memory complexity. Even with less memory requirement, dynamic pre-filter can perform better. The advantage of dynamic pre-filter is to decrease the effect of the additional information. As a result, the dynamic pre-filter achieves high performance in pre-filter module. It is interesting to detect the patterns in different expressions.

# References

---

- [1] S. E. Schechter, J. Jung, and A. W. Berger, "Fast detection of scanning worm infections," 7<sup>th</sup> International Symposium on Recent Advances in Intrusion Detection (RAID), French Riviera, September 2004.
- [2] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," TR CS-74-440, Stanford University, Stanford, California, 1974.
- [3] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, Vol. 20, October 1977, pp. 762-772.
- [4] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," Communications of the ACM, Vol. 18, June 1975, pp. 333-340.
- [5] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," TR-94-17, 1994.
- [6] B Bloom, "Space/time trade-offs in hash coding with allowable errors," ACM, 13(7): 422-426, May 1970.
- [7] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: a survey," Internet Mathematics, vol. 1, no. 4, pp. 485-509.
- [8] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel Bloom filters," IEEE Micro, vol. 24, no. 1, pp. 52-61, Jan./Feb. 2004.
- [9] M. Attig, S. Dharmapurikar, and J. Lockwood, "Implementation results of Bloom filters for string matching," Field-Programmable Custom Computing Machines, pp. 322-323, Apr. 2004.
- [10] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," IEEE/ACM Transactions on Networking, vol. 14, pp. 397-409, Apr. 2006.

- 
- [11] N. S. Artan and H. J. Chao, "Multi-packet signature detection using prefix Bloom filters," IEEE GLOBECOM, vol. 3, pp. 1811–1816, 2005.
- [12] N. S. Artan, K. Sinkar, J. Patel, and H. J. Chao, "Aggregated Bloom filters for intrusion detection and prevention hardware," IEEE GLOBECOM, pp. 349–354, Nov. 2007.
- [13] T. H. Lee and N. L. Huang, "A Pattern Matching Scheme with High Throughput Performance and Low Memory Requirement," IEEE/ACM Transactions on Networking.
- [14] Clam anti virus signature database, [www.clamav.net](http://www.clamav.net).
- [15] T. H. Lee, "Generalized Aho-Corasick algorithm for signature based anti-virus applications," IEEE ICCCN 2007.
- [16] R. Smith, C. Estan, and S. Jha, "XFA: Fast signature matching with extended automata," In IEEE Symposium on Security and Privacy, May 2008.
- [17] Snort website <http://www.snort.org/>.
- [18] M. Roesch, "Snort – lightweight intrusion detection for networks," Proceedings of LISA '99: 13th Systems Administration Conference, pp.s229–238, Nov. 1999.



101

碩士論文

動態過濾器應用在偵測病毒特徵碼防毒軟體

交通大學

電機學院  
電信工程研究所碩士班

王廣煜

