

國立交通大學

電信工程學系碩士班 碩士論文

針對簡單正規表示式之字串比對演算法

An Algorithm for simple regular expression
matching



研究生：林建碩

指導教授：李程輝 教授

中華民國九十九年六月

針對簡單正規表示式之字串比對演算法

學生：林建碩

指導教授：李程輝教授

國立交通大學
電信工程學系碩士班

摘要

字串比對的技術，由於能準確地偵測出關鍵字，是現今在防毒/防蟲技術上的重要應用。在眾多有名的字串比對演算法中，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。然而 AC 演算法是根據純粹字串的比對來設計的，如此對於以正規表示式來表示的病毒/蠕蟲卻無法直接應用。

在本篇論文中，我們使用有系統的演算法，來建構一個字串比對系統，並針對有限長度且可用簡單正規表示式之字串。所提出之系統包含預先過濾器及驗證模組。經由預先過濾器，系統可快速的略過明顯不含字串的文件範圍，且在掃描到可疑字串之起始位置時回報給驗證模組；驗證模組為 AC 演算法的延伸，其中包含了多階層的狀態轉移圖，以及和階層的狀態轉移圖相關的分岔函數。在掃描的過程，可同步處理不同階層的狀態轉移圖。

實驗結果顯示，我們所提出的演算法跟 ClamAV、及加強之 ClamAV、延伸有限自動狀態機(XFA)比較，我們的系統具有較佳的處理效能並且擁有滿意之記憶體占用大小。

An Algorithm for simple regular expression matching

Student: Chien-Shuo Lin

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. Our proposed signature matching system consists of a pre-filter and a verification module. The purpose of pre-filter is to quickly exclude the parts of input string which obviously do not contain signatures and find the starting positions of suspicious sub-strings which may result in match of some signatures. The verification module is an extension of the AC algorithm that consists of multiple levels of goto graphs. Goto graphs in the same level are connected by a novel fork function. Those in different levels could be traversed concurrently. Experimental results show that, compared with ClamAV implementation and its enhancement and the extended finite automaton (XFA), our proposed system yields better throughput performance with acceptable memory requirement.

誌 謝

感謝交通大學電信工程學系 NTL 實驗室的各位，郁文學長、景融學長、迺倫學姐、曉薇、韋儒、奕璉、永昌、永祥，學弟妹國書、煜傑、順閔、謙和、建男、運良、晴嬋、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、韋儒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

中文摘要		i
英文摘要		ii
誌謝		iii
目錄		iv
圖目錄		v
一、	Introduction	1
二、	Problem Definition	4
三、	Related Works	5
四、	The Proposed Signature Matching System	10
四.A	Verification Module	10
四.A.1	The goto function	11
四.A.2	The failure function	13
四.A.3	The output function	13
四.A.4	The fork function	13
四.B	Pre-filter	14
四.C	The signature matching machine	15
五、	Compression of Goto Graph	21
六、	Experimental Result	23
七、	Conclusion	26
附錄	The Aho-Corasick Algorithm	27
參考文獻		31

圖目錄

Figure 1	Data structures of ClamAV implementation.	6
Figure 2	XFA recognizing (a) RE_1 , (b) RE_2 , (c) RE_3 , and (d) RE_1 and RE_2 . Some less important transitions are not shown.	9
Figure 3	The goto graphs for $RE_1 = abc$, $RE_2 = ab*cd*e$, $RE_3 = bc*ad*e$, $RE_4 = pqr*vs$, and $RE_5 = pq\{2,4\}qrqs\{3,5\}tu*vw*x\{2,6\}y$.	13
Figure 4	The stateful pre-filter architecture for $m = 6$ and $k = 3$.	17
Figure 5	Performance comparison of ClamAV implementation and our proposed signature matching system for clean files of various sizes.	24
Figure 6	Performance comparison of ClamAV implementation and our proposed signature matching system with a string $S_1abcdeS'_1$ in various place of file.	25
Figure A.1.	(a) goto function, (b) failure function, and (c) output function for $Y = \{he,$ $she, his, hers\}$.	27

Chapter 1.

Introduction

Because of the rapid advances of computer and network technologies, modern computer viruses/worms can spread at a speed much faster than human-mediated responses. Various viruses/worms such as Code Red [3], Nimda [4], and Slammer [5] that were detected in recent years infected hundreds of thousands of computers on the Internet in a very short period of time and caused huge economic loss to our society. 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.

Behavior anomaly and signature matching are two major techniques for virus/worm detection. Behavior anomaly can be used to detect and prevent the outbreak of an attack because an infected host is likely to behave differently from a normal host. For example, a host infected by some virus/worm may try to infect other vulnerable hosts on the Internet with port/address scanning. Therefore, one can detect an infected host with the observation of high new connection attempt rate or high failure ratio of new connection attempts [6]. Behavior anomaly can be used to detect the so-called “zero-day” attacks. However, it tends to create false positives if the normal behavior cannot be precisely specified. The idea of signature matching is to look for specific patterns in the payload of a packet or across packets. The patterns are derived from the strings of malicious codes contained in viruses/worms. Although it is limited to known viruses/worms with identified patterns, the signature matching technique is quite valuable because of its accuracy. Fortunately, the signature of a new virus/worm can often be quickly derived nowadays once it occurs.

There are some well-known pattern matching algorithms such as Knuth-Morris-Pratt (KMP) [7], Boyer-Moore (BM) [8], and Aho-Corasick (AC) [9]. The KMP and BM algorithms are efficient for single pattern matching but are not scalable for multiple patterns. The AC algorithm pre-processes the patterns and builds a finite automaton which can match multiple patterns simultaneously. Another advantage of the AC algorithm is that it guarantees deterministic

performance under all circumstances. As a consequence, the AC algorithm is widely adopted in various systems. In fact, both ClamAV [1], an open source for virus/worm detection, and Snort [2], another open source for intrusion detection, adopt the AC algorithm for string matching.

As security attacks become sophisticated, regular expressions which are much more expressive than plain strings were used to specify their signatures. It is well known that a regular expression can be recognized with a non-deterministic finite automaton (NFA), which is equivalent to a deterministic finite automaton (DFA). There are some famous algorithms [11], [12] to construct an NFA recognizing a given regular expression. However, NFA-based solutions are often inefficient on a processor with limited parallelism. Hardware accelerators were proposed to achieve high performance [13]-[22]. As an example, a high-performance space-efficient FPGA-based implementation of NFA was presented in [14]. In this design, the NFA is directly converted into logic gates and registers. Using powerful Graphics Processing Units (GPUs) is another alternative to achieve high performance [25]. GPUs are specialized for computationally-intensive and highly parallel operations. DFA-based implementations result in fast signature matching but may require a huge amount of memory space. In [26], a Delayed Input DFA (D²FA) which uses default transitions, an idea similar to the failure transition of the AC algorithm, was proposed to reduce the number of state transitions and hence the space requirement of a DFA. A reduction of state transitions for more than 95% was achieved with different sets of regular expressions used in real products. Although the idea works for selected sets of regular expressions, it still has the risk of resulting in a huge number of states. Two signature rewrite rules were suggested in [23] to reduce the number of states in a DFA. A grouping algorithm was also provided to reduce the number of DFAs for a given set of regular expressions.

Fortunately, the regular expressions used to specify virus/worm signatures are often simple ones. For example, the signatures defined in ClamAV 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 [24]. Unfortunately, the memory space requirement grows exponentially in the number of * operators, which makes the generalized AC algorithm infeasible for virus/worm scanning. The ClamAV implementation requires a small memory space. However, its throughput performance is unacceptable for a large pattern set. Besides, it may result in false negatives. A variation of the ClamAV implementation, called variable

height trie, was proposed to improve throughput performance [16]. It only increases the speed of string matching and does not remove the possibility of false negatives. The extended finite automata (XFA) proposed in [28] is a possible solution for matching simple regular expressions. The XFA augment finite state automata with finite scratch memory and instructions to manipulate this memory. The ClamAV implementation and its variation and the XFA are related to our work and, therefore, will be reviewed and compared with our design.

The purpose of this paper is to present a high-performance, reasonable memory requirement signature matching system for plain strings and simple regular expressions that can be efficiently implemented on general-purpose processors. It can be directly applied to anti-virus/worm applications for matching exploit signatures or used as a matcher primitive for matching vulnerability signatures [29]. The proposed signature matching system consists of a pre-filter and a verification module. It has space complexity comparable to NFA-based solutions. Compared with the ClamAV implementation, the proposed signature matching system can significantly improve system throughput performance. Compared with the variable height trie and the XFA, our proposed system yields better throughput performance with much less memory space requirement.

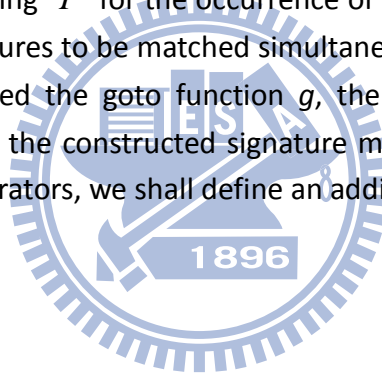
The rest of this paper is organized as follows. Section II contains problem definition. Related works are reviewed in Section III. Our proposed signature matching system is presented in Section IV. Section V contains an efficient compression scheme to reduce memory space requirement. Experimental results are provided and discussed in Section VI. Finally, we draw conclusion in Section VII. For completeness, the AC algorithm is briefly described in the Appendix.

Chapter 2.

Problem Definition

We address in this paper the problem of detecting occurrence of a group of plain strings and simple regular expressions in a given input string. 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 T 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 need the goto function g , the failure function f , and the output function $output$ for the constructed signature matching system. Moreover, to handle $\{min,max\}$ operators, we shall define an additional fork function F .



Chapter 3.

Related Works

The ClamAV implementation (and its enhancement) and the XFA are related to our work and are reviewed separately below. For ease of description, we consider three regular expressions $RE_1 = ab^*cd$, $RE_2 = ef^*gh$, and $RE_3 = pq\{2,4\}rs$ as examples for this section.

A. ClamAV Implementation and Its Enhancement

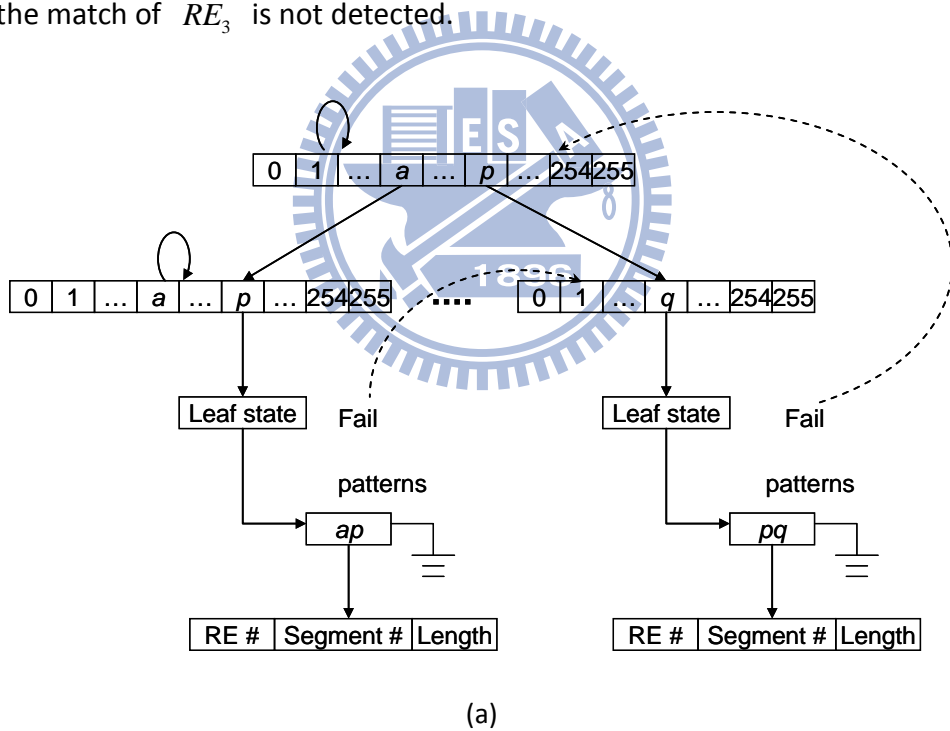
In ClamAV implementation, a regular expression is segmented into strings by the three $*$, $?$, and $\{min, max\}$ operators. An AC automaton is constructed for the first two bytes of all strings. As a result, strings are grouped based on their first two bytes. When the first two bytes of some group are matched, a sequential search is performed for all strings in that group. For RE_1 , RE_2 , and RE_3 , there are six strings ab , cd , ef , gh , pq , and rs , each of them forms a group. Figure 1(a) shows the corresponding AC automaton. Note that the next move function (see Appendix) is used for non-leaf states while the failure function (shown as dashed lines) is required for leaf states. The failure function is consulted when no match is found after searching sequentially the strings attached to a leaf state. The information stored under a string includes its length, which regular expression it belongs to, and the segment number of the string in the regular expression. Figure 1(b) illustrates the data structure used to represent regular expressions. For each regular expression, we need to store the number of operators and the type of each operator.

During scanning, a data structure is maintained to indicate up to which segment a regular expression had been matched and the position in the text of the last matched segment. Consider a regular expression which consists of k segments. Assume that the first e segments had been matched and the e^{th} segment ends at the i^{th} position of the text. Assume further that another segment is matched at the j^{th} position. This newly matched segment is discarded if it is not the $(e+1)^{th}$ segment or i and j do not satisfy the condition specified by the operator which separates the e^{th} and the $(e+1)^{th}$ segments. Consider RE_3 as an example.

Assume that the first segment pq was matched for the first time after the i^{th} symbol is processed. If a segment is further matched at the j^{th} position, then the new match is discarded if it is not the second segment. Assume that it is the second segment. A match of RE_3 is found if $4 \leq j - i \leq 6$. Otherwise, the new match is discarded. Figure 1(c) shows the data structure used during scanning for our example.

Obviously, the memory space requirement of the ClamAV implementation is small because the depth of the trie is only two. However, since all strings attached to a leaf state are searched sequentially when the state is visited, the throughput performance of ClamAV implementation degrades significantly when there are a large number of signatures. Moreover, it is possible to generate false negatives.

For example, if the first segment of RE_3 is matched for the second time at the j^{th} position and the second segment is matched at the k^{th} position such that $k - j = 6$, then the match of RE_3 is not detected.



regular expressions	number of operators	types of operators
RE_1	1	*
RE_2	1	*
RE_3	1	{2, 4}

(b)

regular expressions	matched segment	position
RE_1	i_1	p_1
RE_2	i_2	p_2
RE_3	i_3	p_3

(c)

Figure 1. Data structures of ClamAV implementation.

A variable height trie was proposed to improve throughput performance of ClamAV implementation [16]. The basic idea of the variable height trie is to build the trie as deep as possible, subject to a maximum height constraint. By doing so, the leaf states will be visited fewer times than the original ClamAV implementation. Moreover, the number of strings under a leaf state can be significantly reduced. Therefore, the time spent on sequential search is largely reduced. The tradeoff is larger memory space requirement. It was found that a maximum height of 3 yields good throughput performance with acceptable memory requirement.

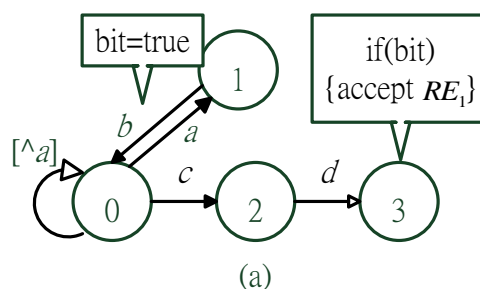
B. XFA

The idea of XFA is to use a finite scratch memory to remember various types of information relevant to the progress of signature matching. One bit is augmented for a * operator and a counter is added for a $\{min, max\}$ operator. Figures 2(a)-2(c) show the XFA recognizing RE_1 , RE_2 , and RE_3 , respectively. As in [29], for simplicity, some less important transitions are not shown in the XFA. For example, the transition from state 2 to state 1 labeled with symbol a is omitted in Figure 2(a). Note that a bit is augmented for (the * operator of) RE_1 because one has to know whether or not string ab was found before to determine if there is a match when string cd is found. As shown in Figure 2(a), bit b_1 augmented for RE_1 is set if state 0 is entered from state 1, meaning that ab occurs in input string. A match of RE_1 is found if the XFA enters state 3 with bit $b_1 = 1$. Similarly, the bit b_2 augmented for RE_2 is set if state 0 in Figure 2(b) is entered from state 1 and a match of RE_2 is found if the XFA enters state 3 with bit $b_2 = 1$. In Figure 2(c), the transition from state 1 to state 2 activates a counter (augmented for RE_3) with initial value zero. When the XFA is in state 2, the counter is incremented by one for each processed input symbol. The counter stays at five if more than four input

symbols are processed. A match of RE_3 is found if state 4 is visited and the counter value is greater than one and smaller than five. Figure 2(d) shows the combined XFA recognizing RE_1 and RE_2 . For convenience, a state is called a final state if a match of some signature is found when it is entered, as long as all conditions for the match are satisfied. For example, state 3 of Fig. 2(a) is a final state because RE_1 is matched if it is entered and the condition $b_1 = 1$ is true.

XFA tries to combine the advantages of deterministic and non-deterministic matching. The number of states for the combined XFA is roughly equal to the total number of symbols in all signatures. This is an advantage of XFA. The tradeoff is higher complexity during scanning.

A potential problem of XFA is that it may result in false positives if there is no mechanism to remember which sub-string sets an augmented bit. As an example, if $RE = ab^*bc$, then a false match will be detected for input string abc because the first two bytes set the augmented bit and the last two bytes make XFA enter a final state with the augmented bit set. The reason for such a problem is that a pattern can occur starting from any position of the input string and, therefore, the start state of XFA has to be always an active state. A second potential problem of XFA is that it may have to maintain multiple counters for a $\{min, max\}$ operator in order to avoid false negatives. For example, if $RE = ab\{4, 6\}cd$, then up to three counters have to be maintained. If only one counter is maintained for the first occurrence of ab , then $ababababecd$ will not be detected. One can similarly show that false negative is possible if up to two counters are maintained. Obviously, the situation becomes worse if max is a large value.



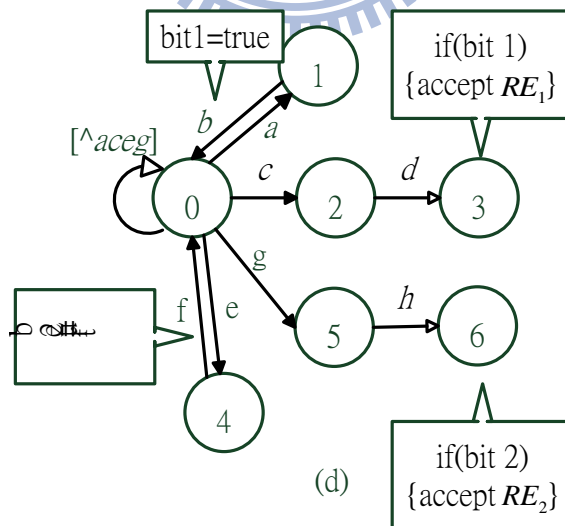
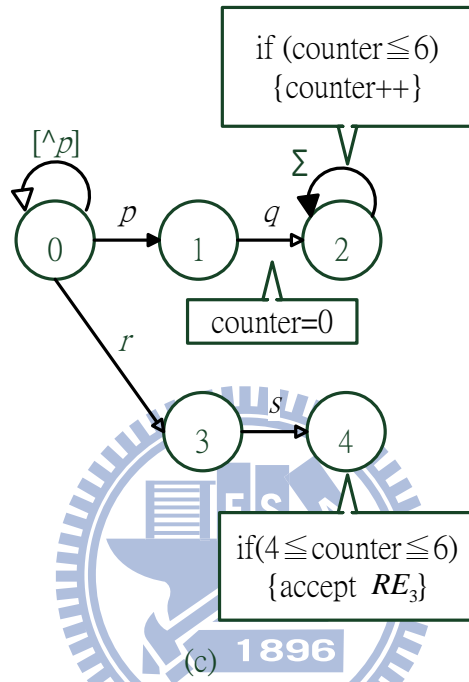
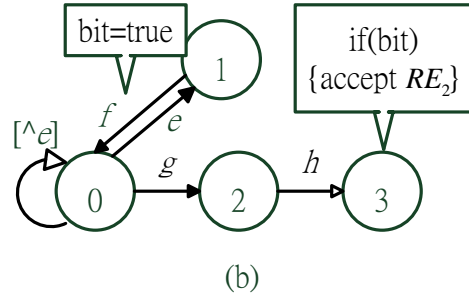


Figure 2. XFA recognizing (a) RE_1 , (b) RE_2 , (c) RE_3 , and (d) RE_1 and RE_2 .
Some less important transitions are not shown.

Chapter 4.

The Proposed Signature Matching System

Our proposed signature matching system is an extension of the generalized AC algorithm presented in a paper previously published by one of the authors [26]. The idea of the generalized AC algorithm is similar to that of XFA. A counter is augmented for a $\{min, max\}$ operator. However, no bit is augmented for any $*$ operator. Instead, multiple goto graphs are constructed so that information relevant to the progress of signature matching is implicitly remembered by traversing different goto graphs.

The 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, as compared with the generalized AC algorithm. For better comprehension, we shall first describe verification module, then pre-filter, followed by signature matching machine.

A. Verification Module

The verification module is an enhancement of the AC algorithm. Before describing the construction procedures for the four functions, i.e., goto, failure, output, and fork, of the verification module, we define some terms which will be used in this section.

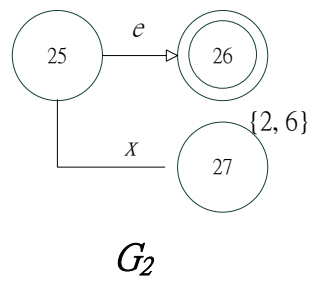
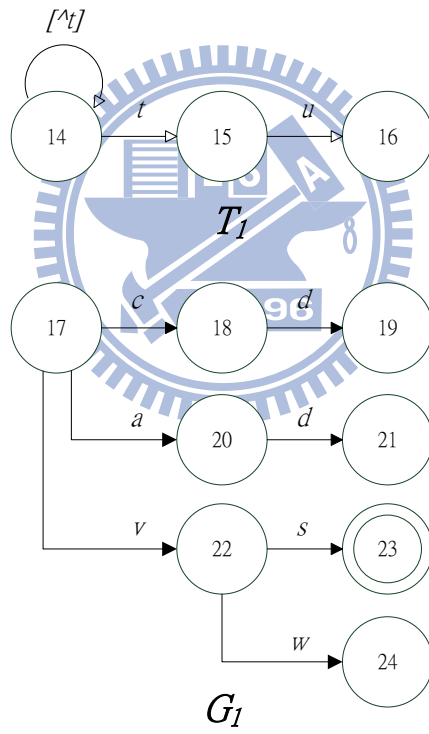
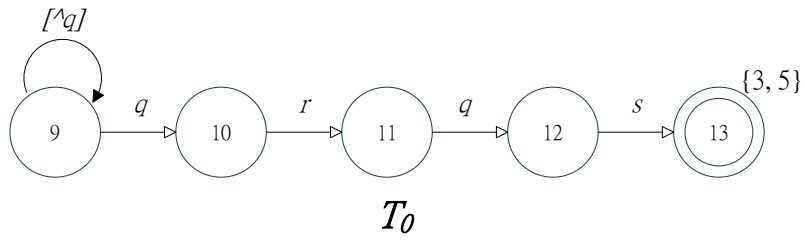
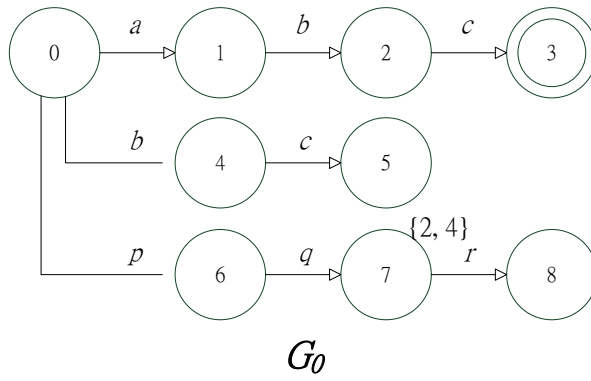
A regular expression is fragmented by $*$ operators. For example, $RE = S_1\{min_1, max_1\}S_2\{min_2, max_2\}S_3 * S_4 * S_5\{min_3, max_3\}S_6$ contains three fragments, i.e., the first fragment $S_1\{min_1, max_1\}S_2\{min_2, max_2\}S_3$, the second fragment S_4 , and the third (or the last) fragment $S_5\{min_3, max_3\}S_6$. A plain string is considered to contain exactly one fragment. Let M denote the maximum number of $*$ operators in any regular expression. As a result, there are at most $M + 1$ fragments for each regular expression. Let $Y_i, 0 \leq i \leq M$, be the set that contains the i^{th} fragments of all regular expressions. All plain strings are included in Y_0 . We need to construct the three functions for every Y_i . We shall only describe the

construction procedure for Y_0 since it can be applied to other Y_i , $1 \leq i \leq M$. Without loss of generality, assume that Y_0 contains n plain strings R_1, R_2, \dots , and R_n and m regular expressions R_{n+1}, R_{n+2}, \dots , and R_{n+m} which have only $\{min, max\}$ operators. Moreover, among the n plain strings, the first n_1 are complete signatures and the last $n_2 = n - n_1$ are the first fragments of signatures specified with regular expressions. Similarly, R_{n+1}, R_{n+2}, \dots , and R_{n+m_1} are complete signatures and $R_{n+m_1+1}, R_{n+m_1+2}, \dots$, and R_{n+m} are simply the first fragments of multi-fragment signatures.

A.1 The goto function

Let $Z_0 = \{R_1, R_2, \dots, R_n, r_{n+1}, r_{n+2}, \dots, r_{n+l}\}$ where r_{n+k} is the first string of R_{n+k} , $1 \leq k \leq l$. As an example, if $R_{n+k} = S_1\{min_1, max_1\}S_2\{min_2, max_2\}S_3$, then we have $r_{n+k} = S_1$. A goto graph, denoted by G_0 , is constructed with algorithm AC1 (see Appendix) for Z_0 . Note that the self-loop at the start state, if exists, is removed. More goto graphs are constructed for the remaining parts of R_{n+k} , $1 \leq k \leq l$. Let $R_{n+k} - r_{n+k}$ be the remaining part of R_{n+k} . For example, if $R_{n+k} = S_1\{min_1, max_1\}S_2\{min_2, max_2\}S_3$, then we have $R_{n+k} - r_{n+k} = S_2\{min_2, max_2\}S_3$. The same procedure is performed recursively to construct the goto graphs for $R_{n+k} - r_{n+k}$ assuming that $Y_0 = \{R_{n+k} - r_{n+k}\}$. The only difference is that the self-loop at the start states of goto graphs constructed for the remaining part of R_{n+k} remain intact. For the previous example where $R_{n+k} - r_{n+k} = S_2\{min_2, max_2\}S_3$, two more goto graphs are constructed for $\{S_2\}$ and $\{S_3\}$. For differentiation, a goto graph constructed for the remaining part of some R_{n+k} , $1 \leq k \leq l$, is called a **T** graph. The construction of goto graphs for Y_0 is completed after all the remaining parts of R_{n+k} , $1 \leq k \leq l$, are handled.

Obviously, there are a total of $M + 1$ **G** graphs, one for each Y_i . The **G** graph constructed for Y_i is called the Level i **G** graph and denoted by G_i . Similarly, the **T** graphs constructed for Y_i are called the Level i **T** graphs. The number of Level i **T** graphs is equal to the number of $\{min, max\}$ operators contained in Y_i . Figure 3 shows the goto graphs for $RE_1 = abc$, $RE_2 = ab * cd * e$, $RE_3 = bc * ad * e$, $RE_4 = pqr * vs$, and $RE_5 = pq\{2, 4\}qrqs\{3, 5\}tu * vw * x\{2, 6\}y$.



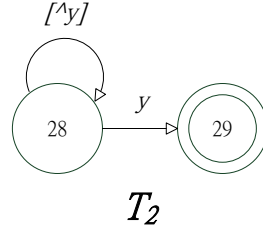


Figure 3. The goto graphs for $RE_1 = abc$, $RE_2 = ab^*cd^*e$, $RE_3 = bc^*ad^*e$, $RE_4 = pqr^*vs$, and $RE_5 = pq\{2,4\}qrqs\{3,5\}tu^*vw^*x\{2,6\}y$.

A.2 The failure function

As the procedure described above, there are two kinds of goto graphs: G graph and T graph. However, not all goto graphs need failure function. For every Level i graph G_i , since it is constructed by the first substring of the i^{th} fragment of all signatures. If any failure occurs, it means that no substring is matched. Thus, there is no failure function for G_i .

Besides, no failure function is needed for the case of T graphs with the value of the min is equal to the max , take $RE = ab\{4,4\}cd$ for example. Since RE is matched only if expected number of symbols, that is, four symbols are apart from substring ab with cd . If any failure occurs, RE is failed to match.

However, for the case of T graphs with the value of the min is not equal to the max , we construct goto function and failure function according to AC algorithm with the substring followed by $\{min, max\}$ operator.

A.3 The output function

Consider the goto graph G_0 . For every state P on graph G_0 , let $output(P) = \emptyset$. If state P is represented by some R_i , $1 \leq i \leq n_1$, then modify $output(P)$ as $output(P) = output(P) \cup \{i\}$. For every state P on a T graph constructed for $R_{n+k} - r_{n+k}$, we assign $output(P) = \emptyset$. Let r'_{n+k} be the last string of $R_{n+k} - r_{n+k}$. If goto graph T is constructed with $\{r'_{n+k}\}$ for some k , $1 \leq k \leq m_1$, then $output(P)$ is modified as $output(P) = output(P) \cup \{n+k\}$ if state P on graph T is represented by r'_{n+k} . In this paper, a state with non-empty output will be referred to as a final state.

A.4 The fork function

The fork function of state P , denoted by $F(P)$, is either empty or gives a

value min , another value max , and $forked_state$, the start state of some goto graph \mathbf{T} . Again, consider the goto graph \mathbf{G}_0 . For every state P on graph \mathbf{G}_0 , we set the fork function $F(P) = \emptyset$. If state P is represented by the first string of r_{n+k} , then $F(P)$ is changed to $min = \min$, $max = \mathbf{max}$, and $forked_state =$ the start state of the goto graph constructed with the second string of r_{n+k} . Here, \min and \mathbf{max} are, respectively, the minimum and maximum values of the $\{min, max\}$ operator which separates the first and the second strings of r_{n+k} . As an example, assume that $r_{n+k} = S_1\{min_1, max_1\}S_2\{min_2, max_2\}S_3$ and state P is represented by string S_1 . In this case, $F(P)$ gives $min = min_1$, $max = max_1$ and $forked_state =$ the start state of the goto graph constructed with $\{S_2\}$. For a goto graph \mathbf{T} constructed for $R_{n+k} - r_{n+k}$, we set $F(P) = \emptyset$ for every state P on graph \mathbf{T} . Assume that $R_{n+k} - r_{n+k}$ contains i $\{min, max\}$ operators. Consequently, there are $i+1$ \mathbf{T} graphs, called $\mathbf{T}_1, \mathbf{T}_2, \dots$, and \mathbf{T}_{i+1} , constructed for $R_{n+k} - r_{n+k}$. Let \mathbf{T}_j be constructed with $\{S_{j+1}\}$ and the minimum and maximum values of the $\{min, max\}$ operator which separates S_j and S_{j+1} are min_j and max_j , respectively. We change $F(P)$ to give $min = min_{j+1}$, $max = max_{j+1}$, and $forked_state =$ the start state of goto graph \mathbf{T}_{j+1} if state P is on graph \mathbf{T}_j for some $j \leq i$ and is represented by S_j . For example, if $R_{n+k} - r_{n+k} = S_2\{min_2, max_2\}S_3$, then there are two \mathbf{T} graphs \mathbf{T}_1 and \mathbf{T}_2 such that \mathbf{T}_1 is constructed with $\{S_2\}$ and \mathbf{T}_2 is constructed with $\{S_3\}$. The fork function of state P on graph \mathbf{T}_1 which is represented by S_2 gives $min = min_2$, $max = max_2$, and $forked_state =$ the start state of goto graph \mathbf{T}_2 . For convenience, a state with non-empty fork function is called a fork state.

B. Pre-filter

The pre-filter adopted in this paper is an extension of the stateful design proposed previously by the authors [Lee and Huang]. Some strings are extracted from signatures to build two pre-filters, called Pre-Filter 1 and Pre-Filter 2. We call a string that is used to build Pre-Filter i a Pre-Filter i pattern. A string is a Pre-Filter 1 pattern iff it is the first string of some element in Y_0 . A string that is the first string of any element in Y_i for some i , $1 \leq i \leq M$, is a Pre-Filter 2 pattern. We describe the construction of Pre-Filter 1 because Pre-Filter 2 can be constructed similarly.

The m -byte prefix of every Pre-Filter 1 pattern is used to construct Pre-Filter 1. A parameter k ($< m$), called block size, is selected. Given block size k , $m-k+1$ membership query modules, denoted by MQ_0, MQ_1, \dots , and MQ_{m-k} , are built for Pre-Filter 1. Let $a_i^0 a_i^1 \dots a_i^{m-1}$ be the m -byte prefix of Pre-Filter 1 pattern A_i . The sub-string $a_i^j a_i^{j+1} \dots a_i^{j+k-1}$ is a member of MQ_j , $0 \leq j \leq m-k$. Each MQ_j is implemented with a bitmap. A hash function $HASH$ is used to build the membership query modules. The h^{th} bit of MQ_j is set to 1 iff there exists pattern A_i such that $h = HASH(a_i^j a_i^{j+1} \dots a_i^{j+k-1})$. Consequently, a membership query module reports a 1 if the query result is positive or 0 otherwise.

Note that, depending on pre-filter patterns, the lengths of prefix and block sizes used to construct Pre-Filter 1 and Pre-Filter 2 can be different. We use m_i and k_i to represent, respectively, the length of prefix and block size adopted for Pre-Filter i . The membership query modules built for Pre-Filter i are denoted by MQ_0^i, MQ_1^i, \dots , and $MQ_{w_i}^i$, where $w_i = m_i - k_i$.

C. The signature matching machine

During scanning, a set of goto graphs called *Active_Graphs* is maintained. Only **G** graphs can be contained in *Active_Graphs*. Initially, we have $Active_Graphs = \{G_0\}$. Depending on the content of *Active_Graphs*, the operation of pre-filters has two modes, i.e., Mode 1 and Mode 2. It is operated in Mode 1 if $Active_Graphs = \{G_0\}$ or Mode 2 otherwise. Initially, we have $Active_Graphs = \{G_0\}$ and, therefore, the operation is in Mode 1. A master bitmap MB_1 of size $w_1 + 1$ is used in Mode 1 operation. In Mode 2 operation, an additional master bitmap MB_2 of size $w_2 + 1$ is required. The purpose of using master bitmaps is to accumulate results obtained from previous queries to improve throughput performance. To simplify the operation, we choose the same block size for both pre-filters, i.e., $k_1 = k_2 = k$. We describe the pre-filter operation for $m_1 > m_2$. This is the case in our experiments. The operation for the case $m_2 > m_1$ is similar. There is only Mode 2 operation if $m_1 = m_2$.

In Mode 1 operation, only Pre-Filter 1 is needed. The initial content of MB_1 is set as 1^{w_1+1} , where b^x means bit b repeats x times. A search window W of length m_1 is used to scan the input text string. Fig. 5 shows the architecture of the Pre-Filter 1 for $m = 6$ and $k = 3$. Initially, the symbols contained in the search window is $t_1 t_2 \dots t_{m_1}$, where t_i is the i^{th} symbol of input text string. Let $t_i t_{i+1} \dots t_{i+m_1}$ be the symbols of input text string contained in search window W . The sub-string $t_{i+m_1-k+1} t_{i+m_1-k+2} \dots t_{i+m_1}$ is used to query MQ_0^1 , MQ_1^1 , ..., and $MQ_{w_1}^1$. Let qb_i be the report of MQ_i^1 and $QB = qb_0 qb_1 \dots qb_{w_1}$. Further, let $MB_1 = mb_0^1 mb_1^1 \dots mb_{w_1}^1$. After the query result QB is obtained, we perform $MB_1 = MB_1 \otimes QB$, where \otimes is the bitwise AND operation. A suspicious sub-string is found and the verification module is invoked if $mb_{w_1}^1 = 1$. The search window W is advanced by $w_1 + 1$ positions if $mb_i^1 = 0$ for all i , $0 \leq i \leq w_1 - 1$, or $w_1 - r$ positions if $mb_r = 1$ and $mb_i = 0$ for all i , $r < i \leq w_1$. In other words, the window advancement is determined by the rightmost 1 of mb_i , $0 \leq i \leq w_1 - 1$, if at least one of them is a 1. If W is advanced by g positions, MB_1 is updated as $MB_1 = 1^g | MB_1 \gg g$, where “|” represents concatenation and $MB_1 \gg x$ means master bitmap MB_1 is right-shifted by x bits. Assume that $mb_{w_1}^1 = 1$ and the verification module is invoked. If no match is found, then the window advancement g is equal to $w_1 + 1$ if $mb_i^1 = 0$ for all i , $0 \leq i \leq w_1 - 1$, or $w_1 - r$ if $mb_r = 1$ and $mb_i = 0$ for all i , $r < i \leq w_1 - 1$. Assume that a match is found. If a complete signature is matched, then the scanning process ends. Otherwise, the match is only the first fragment of a multi-fragment signature. In this case, goto graph G_1 is added to *Active_Graphs*, the information of the matched fragment (including the signature it belongs to and the ending position in input text string) are recorded, search window is advanced as if no match is found, and the scanning process continues according to Mode 2 operation. Let t^* be the symbol of input text string which is the first symbol contained in search window, after advancement. For simplicity, the above rule of window advancement will be referred to as window advancement according to the content of (updated) MB_1 .

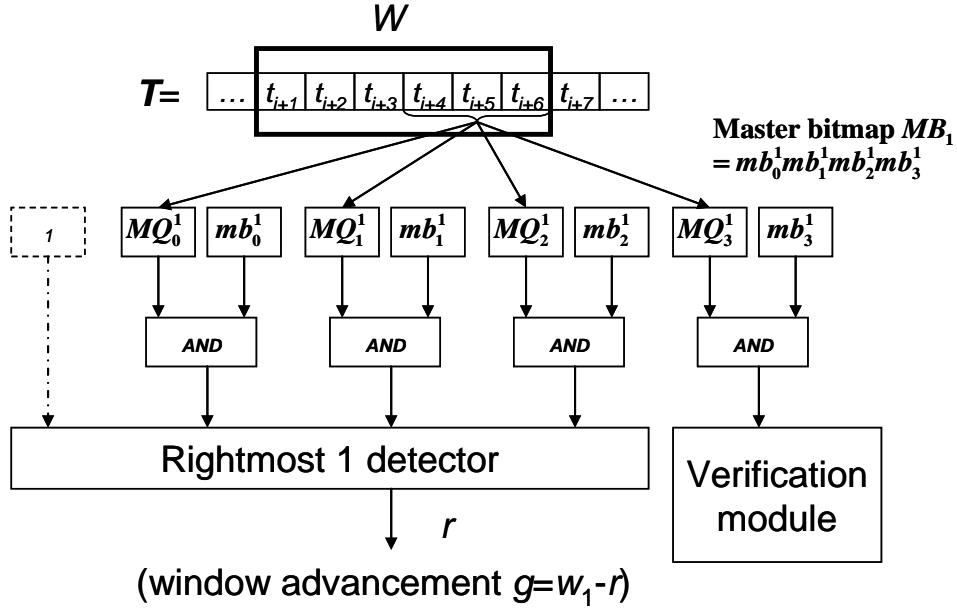


Figure. 4. The stateful pre-filter architecture for $m = 6$ and $k = 3$.

In Mode 2 operation, both Pre-Filter 1 and Pre-Filter 2 are used. The content of the second master bitmap MB_2 is set as 1^{w_2+1} initially. Another search window W' of length m_2 is adopted for scanning input text string, starting from symbol t^* . Let $t_i t_{i+1} \dots t_{i+m_2}$ be the symbols of input text string contained in search window W' . The sub-string $t_{i+m_2-k+1} t_{i+m_2-k+2} \dots t_{i+m_2}$ is used to query MQ_i^1 , $0 \leq i \leq w_1$, and MQ_i^1 , $0 \leq i \leq w_2$. Let QB_1 and QB_2 be the query results reported by Pre-Filter 1 and Pre-Filter 2, respectively. Note that, since the length of W' is m_2 , only the results reported by MQ_i^1 , $0 \leq i \leq w_2$, can be utilized to check if a suspicious sub-string which potentially matches the first fragment of some signature is found. In other words, Pre-Filter 1 has to be used as if its membership query modules were built with m_2 -byte prefixes of all Pre-Filter 1 patterns. Therefore, after QB_1 and QB_2 are obtained, we perform $MB_1 = MB_1 \otimes QB_1$, $MB_1 = 1^{w_1-w_2} | MB_1 \gg w_1 - w_2$, and $MB_2 = MB_2 \otimes QB_2$. For convenience, we shall use $R = MB_1 \oplus MB_2$ to represent a bitmap of length $w_2 + 1$ which is obtained by bitwise ORing the rightmost $w_2 + 1$ bits of MB_1 with MB_2 . There are four possible cases.

Case 1. $mb_{w_1}^1 = 0$ and $mb_{w_2}^2 = 0$.

If $mb_{w_1}^1 = 0$ and $mb_{w_2}^2 = 0$, then no suspicious sub-string is found. The window advancement g is determined according to the content of R . The scanning process continues after performing $MB_1 = 1^g | MB_1 \gg g$ and $MB_2 = 1^g | MB_2 \gg g$.

Case 2. $mb_{w_1}^1 = 1$ and $mb_{w_2}^2 = 0$.

The situation is the same as Mode 1 operation if $mb_{w_1}^1 = 1$ and $mb_{w_2}^2 = 0$. The difference is that window advancement g is determined according to the content of R . If no signature is matched, then the scanning process continues after performing $MB_1 = 1^g | MB_1 \gg g$ and $MB_2 = 1^g | MB_2 \gg g$.

Case 3. $mb_{w_1}^1 = 0$ and $mb_{w_2}^2 = 1$.

If $mb_{w_1}^1 = 0$ and $mb_{w_2}^2 = 1$, then all the goto graphs contained in *Active_Graphs*, except G_0 , are traversed concurrently. Assume that goto graph G_i ($i \geq 1$) is contained in *Active_Graphs* and a match is found in traversing G_i . The scanning process ends if a complete signature is matched. Otherwise, only the i^{th} fragment of some signature, say R_{n+k} , is matched. In this case, we check if the $(i-1)^{th}$ fragment of R_{n+k} was matched. If it is true, then the matched fragment is recorded and the traversal on graph G_i ends. If no complete signature is matched for all concurrent traversals, then the scanning process continues after performing $MB_1 = 1^g | MB_1 \gg g$ and $MB_2 = 1^g | MB_2 \gg g$, where g is the window advancement determined according to the content of R .

Case 4. $mb_{w_1}^1 = 1$ and $mb_{w_2}^2 = 1$.

The operation for the case $mb_{w_1}^1 = 1$ and $mb_{w_2}^2 = 1$ is the same as that for Case 3, except that goto graph G_0 is included in concurrent traversals. The scanning process ends if a complete signature is matched. Assume that only a fragment of

some signature is matched. In this case, we check if the preceding fragment of the same signature was matched. If it is true, then the newly matched fragment is recorded. In case no complete signature is matched, the window advancement g is determined according to the content of R and the scanning process continues after performing $MB_1 = 1^g | MB_1 \gg g$ and $MB_2 = 1^g | MB_2 \gg g$.

Note that the pre-filters can work correctly without master bitmaps. However, with master bitmaps, the search window can be advanced by more positions, compared with the implementation without it. Consider for example Pre-Filter 1 with $m_1 = 8$ and $k_1 = 3$. Assume that the results of the first and the second queries (both in Mode 1) are 101010 and 001010, respectively. Without master bitmap MB_1 , W is advanced by one position after each query. On the other hand, with MB_1 , it is advanced by one position after the first query and six positions after the second query. Initially, the master bitmap $MB_1 = 111111$. It becomes 110101 after the first query. Since the results of the second query is 001010, the content of MB_1 becomes 000000, after the bitwise AND operation. Therefore, the search window W is advanced by six positions and the content of master bitmap is updated as 111111. It was proved that the implementation with master bitmap is optimal in the sense that it is equivalent to using all previous query results.

Now we describe the operation of verification module. Assume that the pre-filters find a suspicious sub-string and the verification module is invoked. Consider Mode 1 operation or Case 2 of Mode 2 operation. For these two cases, only goto graph G_0 is traversed. The traversal on graph G_0 ends if a complete signature is matched or the failure function is consulted. If a fork state is visited, the fork function will give the start state of some T graph. At this moment, a process is forked to concurrently traverse the T graph, from its start state. (New Example Is Needed. As an example, consider the goto graphs shown in figure. 4. A process is forked to traverse graph T_0 if state 9 is visited. As another example, a process is forked to traverse graph T_1 if state 4 of T_0 is visited.) If a state whose representing string matches the first fragment of R_k for some k , $n_1 + 1 \leq k \leq n$, is visited, then goto graph G_1 is put in *Active_Graphs* so that it will be traversed if succeeding suspicious sub-strings which falls in Case 3 or Case 4 of Mode 2 operation is found by the pre-filters. In this case, the information of the matched fragment, including the signature it belongs to and the ending position in input text string, are recorded.

Traversal on a goto graph \mathbf{T} is as follows. Let \mathbf{min} and \mathbf{max} be, respectively, the minimum and the maximum values given by the fork function of the state visited that causes a forked process to traverse graph \mathbf{T} . A counter ctr is maintained when traversing graph \mathbf{T} . The content of ctr is initialized to \mathbf{min} and the next \mathbf{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 . Let S^P denote the string representing state P and $|S|$ be the length of string S . The content of ctr is updated as $ctr = ctr + increment$, where $increment = |S^P| - |S^{f(P)}|$. The traversal ends if a match of signature is found or $ctr > \mathbf{max}$. If a fork state is visited, then one more process is forked to traverse another \mathbf{T} graph, from its start state given by the fork function. Assume that graph \mathbf{T} is constructed with $\{S\}$, where S is the last string of the first fragment of R_{n+k} for some k , $l_1 + 1 \leq k \leq l$. If the state represented by string S is visited, meaning that the first fragment of R_{n+k} is matched, the information of the matched fragment is recorded.

It is possible that no complete signature is matched when traversals end. In this case, the pre-filters resume their execution according to Mode 2 operation.

The operations for Case 3 and Case 4 of Mode 2 are similar. Traversal on goto graph \mathbf{G}_0 is exactly the same as that described above. Traversal on Level i ($i \geq 1$) graphs are similar to those on Level 0 graphs. The difference is that we need to check whether or not it is a true match when a match of the i^{th} fragment of some signature is found. Assume that the i^{th} fragment of R_{n+k} , $l_1 + 1 \leq k \leq l$, is matched when traversing some Level i graph. It is a true match only if the $(i-1)^{th}$ fragment of R_{n+k} was matched previously and the starting position of the newly matched i^{th} fragment is greater than the ending position of the previously matched $(i-1)^{th}$ fragment. If it is a true match, then the information of the i^{th} fragment of R_{n+k} is recorded and goto graph \mathbf{G}_{i+1} is added to *Active_Graphs* if R_{n+k} has more than i fragments.

Chapter 5.

Compression of Goto Graph

The G graphs, especially G_0 , are likely to have a large number of states for a large signature set. Therefore, a straightforward implementation using a two-dimensional table requires a huge amount of memory space. In this section, we present a compression scheme which can significantly reduce the memory space requirement.

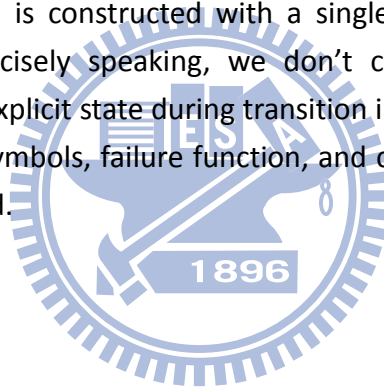
Consider graph G_0 . (The other G graphs can be compressed similarly.) In our proposed compression scheme, states are classified according to the number of child states. State P is said to be a branch state, a single-child state, or a leaf state, if it has at least two child states, exactly one child state, or no child state, respectively. A single-child state P is a first single-child state if its parent state is a branch state. Finally, state P is said to be an explicit state if it is the start state, a branch state, a first single-child state, a final state, a fork state, or a fragment-end state, i.e., a state represented by R_i for some i , $n_1 + 1 \leq i \leq n$. We store all strings in Z_0 and some data structures for the explicit states. Note that every final state, fork state, and fragment-end state has to be a branch state, a single-child state, or a leaf state.

The strings in Z_0 are stored contiguously in a *compacted_ G_0 _strings* file. For example, if $Z_0 = \{he, she, his, hers\}$, then the *compacted_ G_0 _strings* file is simply *heshehishers*. Similar to the *AC-bnfa* scheme adopted by Snort, branch states are further classified into Branch_2, Branch_3, Branch_4, Branch_5, and Branch_256 states. State P is a Branch_ i ($2 \leq i \leq 5$) state if it has exactly i children states. For such a state, we store i pairs of (*symbol*, *next state*). If state P has more than five children states, it is classified as a branch_256 state and we store sequentially 256 next states corresponding to 256 possible input symbols. Note that the next state could be the *END* state for some input symbols. Our experimental results show that there are only a small number of branch_256 states and the number of children states is much larger than five for most branch-256 states. By storing all the 256 next states, we waste a little memory space but achieve high-speed look-up for state transition. Assume that state P is a single-child state with representing string S^P . Let A_i be the first string in Z_0 which contains

S^P as a prefix. For state P , we store $(position, distance)$, where $position$ is the position of the $|S^P|^{th}$ byte of A_i in the *compacted_G0_strings* file and $distance$ is the number of bytes from state P to its nearest descendent explicit state, i.e., the explicit state whose representing string is the shortest one which contains S^P as a proper prefix.

Finally, for each leaf state, we basically store nothing but an identifier to indicate that it is a leaf state. Of course, every explicit state needs flags to indicate whether or not it is a final state, a fork state, and/or a fragment-end state. For a final state, we need to store the identification of the matched signature(s). For a fork state, the minimum and the maximum values as well as the starting state of some \mathbf{T} graph to be traversed are stored. Note that the number of states on the compressed \mathbf{G}_0 graph is equal to the number of explicit states, which normally is much smaller than the number of states in the original \mathbf{G}_0 graph constructed with algorithm AC1. As a result, the memory requirement is significantly reduced.

Since every \mathbf{T} graph is constructed with a single string, the memory space requirement is small. Precisely speaking, we don't create states for \mathbf{T} graphs actually, since there is no explicit state during transition in \mathbf{T} graphs. All we need to store is an array of input symbols, failure function, and counter increment when the failure function is consulted.



Chapter 6.

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 $g(P, \sigma) \neq \text{fail}$ 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 $(e+1)^{\text{th}}$ fragment or i and j do not satisfy the condition specified by the operator which separates the e^{th} and the $(e+1)^{\text{th}}$ 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 $(i + |sre_2| + 1)^{\text{th}}$ 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 $(i + |sre_2| + 1)^{\text{th}}$ 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 $2 \leq j - i - |sre_2| - |sre_3| - 1 \leq 4$. Otherwise, the newly matched fragment is discarded and the data structure remains intact.

Note that, strictly speaking, the ClamAV implementation may result in false negatives. For example, consider the same regular expression $RE = sre_1 ? sre_2 \{2,4\} sre_3 \{3,5\} sre_4$ and assume that the input text is $sre_1sre_1asre_2abc{sre_3}abcd{sre_4}$. There is obviously a match starting at the $(sre_1 + 1)^{th}$ position. However, the Clam AV implementation does not detect the match because the second sre_1 will be discarded when it is found.

The performance of ClamAV implementation can be improved by using variable height trie [Avfs]. The variable height trie requires more memory space for larger maximum heights. It was found that a trie with maximum height three is a good tradeoff between throughput performance and space requirement. Therefore, we shall compare our proposed system with tries of maximum height two and three.

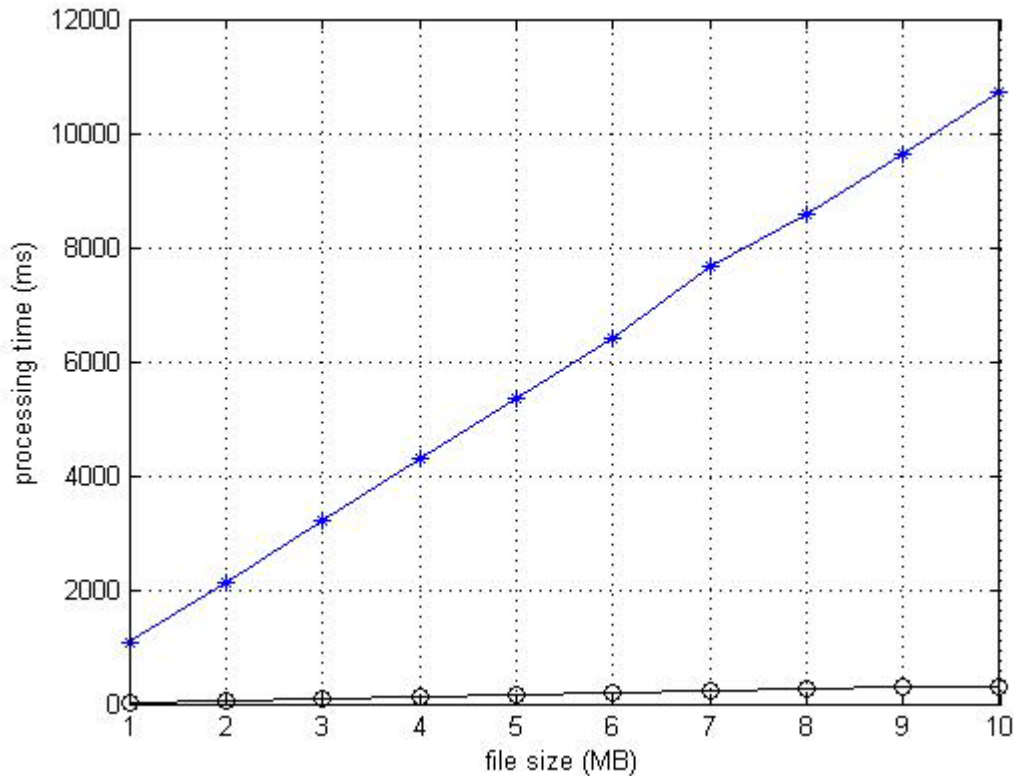


Figure 5. Performance comparison of ClamAV implementation and our proposed signature matching system for clean files of various sizes.

Figure 5 shows the comparison of CPU execution time for randomly generated files of various sizes without any signature occurrence. It can be seen that the CPU execution time is proportional to file size. The CPU time required by the ClamAV implementation is about 20 times of that required by our proposed system. Figure

6 illustrates similar comparison with a string $S_1abcdeS'_1$ inserted into a randomly generated file of size slightly larger than 2M bytes to match a signature (W32.Gop) of the form $RE=S_1*S'_1$. Again, the CPU execution time required by the ClamAV implementation is about 20 times of that required by our proposed system. We also conducted simulations with a string $S_1abcdeS'_1$ inserted at various positions to match a signature (DOS.Bg-2) of the form $RE=S_1\{1,6\}S'_1$. The results are similar. 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.

As for memory requirement, ClamAV implementation uses 362K bytes and our proposed system uses about 1.94M bytes. The pre-filter requires 128K bytes and the verification module needs 1.8M bytes. There are 2,486 final states and, therefore, the output function takes about 5K bytes. (In our implementation, we use two bytes for signature ID.) We believe the amount of memory required by our proposed signature matching system is acceptable for practical systems.

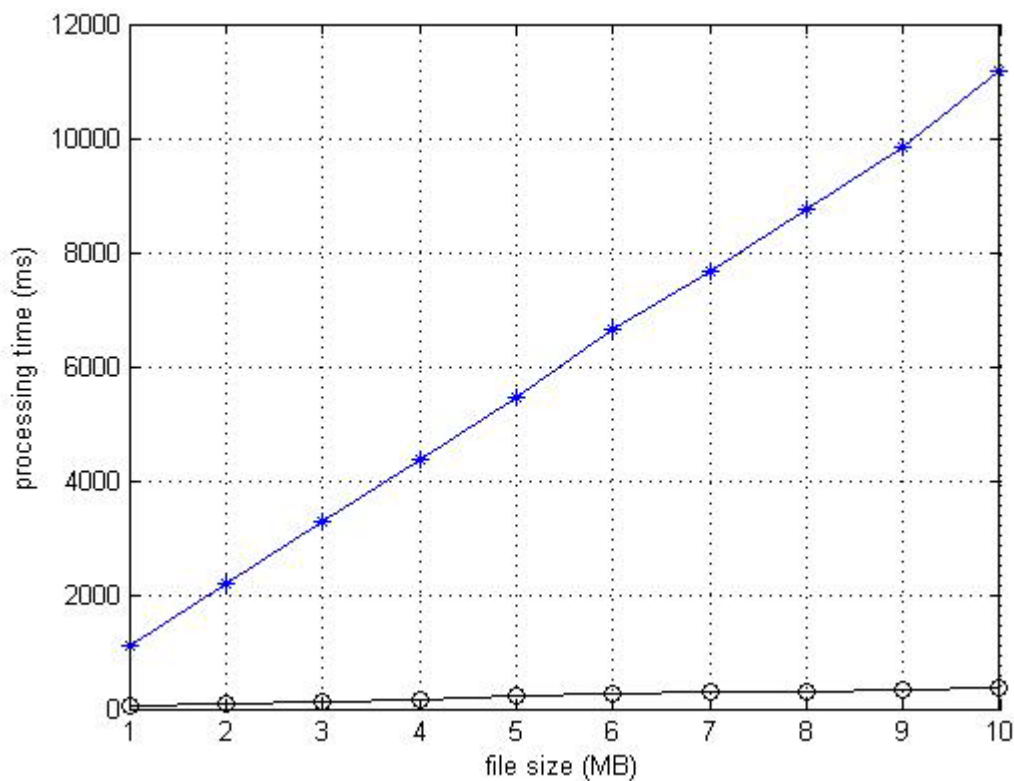


Figure 6. Performance comparison of ClamAV implementation and our proposed signature matching system with a string $S_1abcdeS'_1$ in various place of file.

Chapter 7.

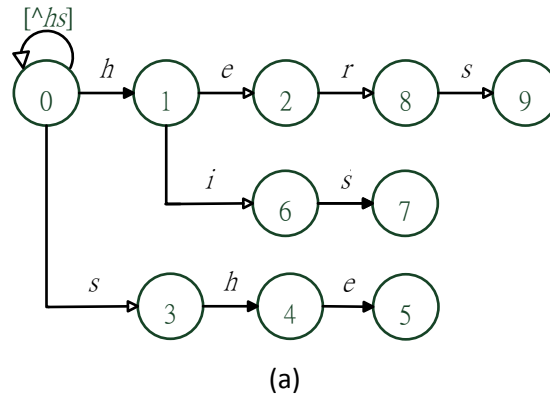
Conclusion

We have presented in this paper a systematic approach to construct a signature matching system for simple regular expressions which are used to define virus/worm signatures in ClamAV. Like the Aho-Corasick algorithm, the verification module of our proposed system is dictated by three functions, namely, the goto, failure, and output functions. Experimental results using ClamAV signatures show that, compared with the ClamAV implementation and its enhancement, our proposed system achieves much better throughput performance while requiring an acceptable amount of memory.

Our work presented in this paper provides some guidelines for writing signatures. For example, the non-overlapping condition is very important in reducing the space complexity. In case the non-overlapping condition is to be violated, one should minimize the number of * operators in those overlapped signatures. As another example, the throughput performance can be largely improved for long pre-filter patterns. Extension of our work to other types of signatures is an interesting and useful further research topic.

Appendix: The Aho-Corasick Algorithm

The Aho-Corasick (AC) algorithm is dictated by three functions: a goto function g , a failure function f , and an output function $output$. Fig. A.1 shows the three functions for the pattern set $Y = \{he, she, his, hers\}$ [9].



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. A.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. One state, numbered 0, is designated as the start state. String S^P is said to represent state P on a goto graph if the shortest path from the start state to state P spells out S^P . For example, string her represents state 8 in Fig. 1. The start state is represented by the empty string ε . The length of string S is represented by $|S|$.

Note that there might be a self-loop at the start state of a goto graph. However, it

becomes a tree after removing the self-loop, if exists. In the following definitions, we ignore the self-loop. We call state P_1 the parent of state P_2 and state P_2 the child of state P_1 if there exists a symbol σ such that $g(P_1, \sigma) = P_2$. State P_2 is said to be a descendent of state P_1 and state P_1 an ancestor of state P_2 if S^{P_1} is a proper prefix S^{P_2} . The tree which consists of state P and all its descendant states is called the sub-tree of P .

The goto function g maps a pair (state, input symbol) into a state or the message *fail*. For the example shown in Fig. A.1, we have $g(0, h) = 1$ and $g(1, \sigma) = \text{fail}$ 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 $f(P_1) = P_2$ if and only if (iff) S^{P_2} is the longest proper suffix of S^{P_1} that is also a prefix of some pattern. The output function maps a state into a set (could be empty) of patterns. The set $output(P)$ contains a pattern if the pattern is a suffix of S^P .

Let P_1 be the current state and σ the current input symbol. Also, let T denote the input string. Initially, the start state is assigned as the current state and the first symbol of T is the current input symbol. An operation cycle of the AC algorithm is defined as follows.

1. If $g(P_1, \sigma) = P_2$, the algorithm makes a state transition such that state P_2 becomes the current state and the next symbol in T becomes the current input symbol. If $output(P_2) \neq \emptyset$, the algorithm emits the set $output(P_2)$. The operation cycle is complete.
2. If $g(P_1, \sigma) = \text{fail}$, the algorithm makes a failure transition by consulting the failure function f . Assume that $f(P_1) = P_2$. The algorithm repeats the cycle with P_2 as the current state and σ as the current input symbol.

It can be shown that the maximum number of state transitions is $2n-1$ for scanning if $|T|=n$. This number can be reduced to n if the next move function δ is adopted. The next move function is defined as $\delta(P, \sigma) = g(P, \sigma)$ if $g(P, \sigma) \neq \text{fail}$ or $\delta(P, \sigma) = \delta(f(P), \sigma)$ otherwise.

The procedures to construct the goto, failure, and output functions are described in

Algorithms AC1 and AC2 below [9]. The goto function and the failure function are constructed, respectively, in Algorithms AC1 and AC2. The output function is partially constructed in Algorithm AC1 and completed in Algorithm AC2.

Algorithm AC1. Construction of the goto function.

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

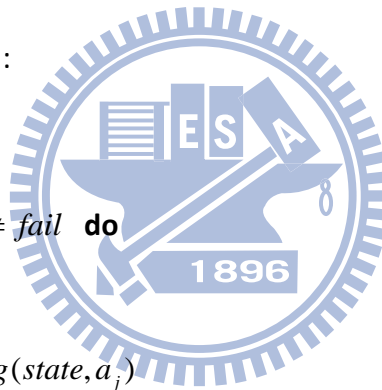
Output. Goto function g and a partially computed output function $output$.

Method. We assume $output(P) = \emptyset$ 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  $\sigma$  such that  $g(0, \sigma) = fail$  do  $g(0, \sigma) \leftarrow 0$ 
end
procedure  $enter(a_1 a_2 \dots a_m)$ :
  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)  $\leftarrow$   $\{a_1 a_2 \dots a_m\}$ 
  end
end

```



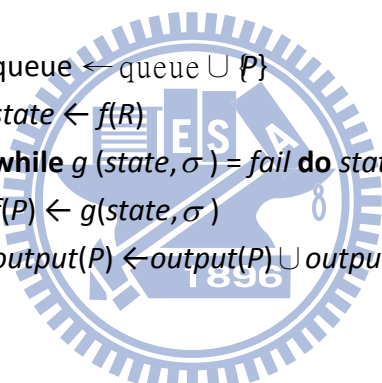
Algorithm AC2. Construction of the failure function.

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  $\sigma$  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  $\sigma$  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
  end
```



References

- [1] Clam anti virus signature database, www.clamav.net.
- [2] SNORT system, www.snort.org.
- [3] D. Moore, C. Shannon, and J. Brown, "Code-Red: a case study on the spread and victims of an Internet worm," in Proc. ACM/USENIX Internet Measurement Workshop, France, Nov. 2002.
- [4] CAIDA. Dynamic graphs of the Nimda worm. <http://www.caida.org/dynamic/analysis/security/nimda>.
- [5] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the Slammer worm," IEEE Security and Privacy, 1(4): 33-39, July 2003.
- [6] S. E. Schechter, J. Jung, and A. W. Berger, "Fast detection of scanning worm infections," 7th International Symposium on Recent Advances in Intrusion Detection (RAID), French Riviera, September 2004.
- [7] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," TR CS-74-440, Stanford University, Stanford, California, 1974.
- [8] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, Vol. 20, October 1977, pp. 762-772.
- [9] 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.
- [10] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Technical Report, May 1994.
- [11] K. Thompson, "Programming techniques: Regular expression search algorithm," Commun. ACM, 11(6):419-422, 1968.
- [12] V. M. Glushkov, "The abstract theory of automata," Russian Mathematical Surveys, 16:1-53, 1961.
- [13] R. W. Floyd and J. D. Ullman, "The compilation of regular expression into integrated circuits," Journal of ACM, vol. 29, no. 3, pp. 603-622, July 1982.
- [14] R. Sidhu and V. Prasanna, "Fast regular expression matching using FPGAs," in Field-Programmable Custom Computing Machines (FCCM), April 2001.
- [15] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," Proceedings of 13th International Conference on Field Programmable Logic and Applications, 2003.
- [16] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok, "Avfs: An on-access anti-virus file system," USENIX Security Symposium, 2004.
- [17] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," IEEE Symposium on Field Programmable Custom Computing Machines, Napa, CA, 2004.
- [18] P. Sutton, "Partial character decoding for improved regular expression matching

- in FPGAs,” in IEEE International Conference on Field Programmable Technology (FPT), Dec. 2004.
- [19] S. Yusuf and W. Luk, “Bitwise optimized CAM for network intrusion detection systems,” Proceedings of 15th International Conference on Field Programmable Logic and Applications, 2005.
- [20] B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A scalable architecture for high-throughput regular-expression pattern matching,” ISCA, 2006.
- [21] C. H. Lin, C. T. Huang, and S. C. Chang, “Optimization of regular expression pattern matching circuits on FPGA,” in Proc. Of Conference on Design, Automation and Test in Europe, 2006.
- [22] J. Moscola, Y. H. Cho, and J. W. Lockwood, “A scalable hybrid regular expression pattern matcher,” in Field-Programmable Custom Computing Machines (FCCM), 2006.
- [23] J. C. Bispo, I. Sourdis, J. M. Cardoso, and S. Vassiliadis, “Regular expression matching for reconfigurable packet inspection,” in IEEE International Conference on Field Programmable Technology (FPT), Dec. 2006.
- [24] 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.
- [25] M. Alicherry, M. Muthuprasanna, and V. Kumar, “High speed pattern matching for network IDS/IPS,” IEEE International Conference on Network Protocols, 2006.
- [26] T. H. Lee, “Generalized Aho-Corasick algorithm for signature based anti-virus applications,” IEEE ICCCN 2007.
- [27] 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.
- [28] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, “Algorithms to accelerate multiple regular expressions matching for deep packet inspection,” ACM SIGCOMM 2006.
- [29] J. Rejeb and M. Srinivasan, “Extension of Aho-Corasick algorithm to detect injection attacks,” SCSS (1) 2007.
- [30] R. Smith, C. Estan, and S. Jha, “XFA: Fast signature matching with extended automata,” In IEEE Symposium on Security and Privacy, May 2008.
- [31] T. H. Lee and N. L. Huang, 2008, “An efficient and scalable pattern matching scheme for network security applications,” IEEE ICCCN Workshop, 2008.
- [32] N. Schear, D. Albrecht, and N. Borisov, “High-speed matching of vulnerability

signatures,” In Recent Advances in Intrusion Detection (RAID), 2008.

- [33] M. Norton, “Optimizing pattern matching for intrusion detection,” <http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf>, July 2004.
- [34] J. E. Hopcroft and J. D. Ullman, “Introduction to automata theory, languages, and computation,” 2nd edition, Addison-Wesley, 2001.

