目錄

1. INTRODUCTION 1
2. THE AHO-CORASICK ALGORITHM 3
3. RELATED WORKS 4
   3.1. Bitmap Data Structure 4
   3.2. Banded-row Format 4
   3.3. AC-bnfa 5
   3.4. HEXA 5
   3.5. ClamAV Implementation 6
   3.6. Wu-Manber Algorithm 6
4. THE PROPOSED COMPRESSION SCHEME 8
5. THE PROPOSED ARCHITECTURE 11
   5.1. Pre-filter Design 11
   5.2. Verification Engine Design 11
   5.3. Pattern Matching Machine 13
6. COMPARISON WITH RELATED WORKS 17
7. ANALYTICAL COMPARISON WITH WU-MANBER ALGORITHM 21
   7.1. Pre-filter Performance 21
      7.1.1. The Wu-Manber Algorithm 21
      7.1.2. The Stateless Architecture 22
      7.1.3. The Stateful Architecture 22
   7.2. Overall System Performance 23
   7.3. Numerical Results 23
8. CONCLUSION 25
REFERENCES 26

摘要

近年來，字樣比對技術已被應用於處理網路資訊安全議題，諸如入侵偵測、病毒掃描及反垃圾郵件等。知名的 Aho-Corasick(AC)演算法能同時比對多個字樣，並具有最差情況下的效能保證，因此被廣泛採用。然而，隨著傳輸技術的發展，在高速網路環境中，採用 AC 演算法已跟不上資料傳輸的速度，再者，在字樣總長度大的情況下，AC 演算法將需要極大的記憶體空間以存放比對所需的二維狀態轉換表。在本成果報告中，我們首先提出一項新的設計，可有效壓縮 AC 演算法的資料結構。施以多種先前提出的壓縮技術，狀態轉換表所占空間皆仍與字樣總長度成正比關係，相反地，我們提出的設計則能將狀態轉換表壓縮成僅與字樣個數成正比（不被字樣總長度影響）。除此之外，我們進而提出一包含狀態性預先過濾器及以 AC 演算法為基礎的驗證模組的架構。其中我們引入了狀態性預先過濾的概念，所謂狀態性係參考過去的過濾結果(而非只參考當下的狀態)來決定當下的過濾結果，並以簡單的位元串實現之。此實現方法等效於參考所有過去的過濾結果，故為最佳的實現方法。相較於多項先前技術，我們提出的架構在比對效能及記憶體使用量這兩方面都有顯著的進步。

關鍵詞：Aho-Corasick 演算法、布隆過濾器、深度封包檢測、字樣比對

Abstract

Pattern matching technique has recently been applied to network security applications such as intrusion detection, anti-virus, and anti-spam. The Aho-Corasick (AC) algorithm can match multiple patterns simultaneously with worst-case performance guarantee and thus is widely adopted. However, as the transmission technology advances, the AC algorithm cannot keep up with wire speed in a high-speed network. Moreover, it may require a huge amount of space to store a two-dimensional state transition table when the total length of patterns is large. In this paper, we first propose a novel scheme to compress the data structure of the AC algorithm. The size of the resulting two-dimensional state transition table is proportional to the number of patterns, whereas that in several previous designs is proportional to the total length of patterns. Then an architecture consisting of a stateful pre-filter and an AC-based verification engine is presented. We introduce the stateful pre-filter concept, which makes use of previous query results, and propose a simple bitmap-based implementation to realize it. The proposed implementation is optimal in the sense that it is equivalent to utilizing all previous query results. Compared with various previous designs, our proposed architecture achieves significant improvement in both throughput performance and memory usage.

Keywords: Aho-Corasick algorithm, Bloom filter, deep packet inspection, Pattern matching.

1. INTRODUCTION

Pattern matching has been an important technique in information retrieval and text editing for many years. Because of its accuracy, it has recently been adopted in network security appliances for signature matching to detect malicious attacks. In the wide sense, pattern matching is a problem of searching for occurrences of plain strings and/or regular expressions in an input text string. In this paper, we only consider matching of plain strings.

There are some well-known pattern matching algorithms such as Knuth-Morris-Pratt (KMP) [1], Boyer-Moore (BM) [2], and Aho-Corasick (AC) [3]. The KMP and BM algorithms are efficient for single pattern matching but are not suitable 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 linear-time deterministic performance under all circumstances. As a consequence, the AC algorithm is widely adopted in various systems. However, even with the linear-time worst-case performance guarantee, the throughput of the AC algorithm cannot keep up with wire speed in a high-speed network because of the rapid advances in transmission and switching technologies. To achieve high throughput performance, some hardware accelerators of the AC algorithm were proposed previously [4]−[6]. These designs take advantage of hardware capability for massive parallel processing. As an example, the bit-split design proposed in [5] adopts multiple processing engines, each handles a few bits of a symbol, to execute the AC algorithm. A consequence of using multiple processing engines is that the number of transitions and hence memory requirement is reduced. High throughput performance can be achieved if multiple processing engines work concurrently. These processing engines can of course be serialized for software implementation. However, serialization impacts throughput performance significantly. It is possible to take advantage of multi-thread, multi-core features of processors for parallel processing. How to efficiently implement pattern matching algorithms on such a processor, which is not the focus of this paper, is left as a further research topic.

Another problem of the AC algorithm is its potential huge memory space requirement. A straightforward implementation of the AC algorithm is to construct a two-dimensional state transition table for the finite automaton. Obviously, such an implementation requires prohibitively huge amount of memory space when the total length of patterns is large. Several schemes had been proposed to reduce the memory requirement. Some of them are adopted by Snort [7], [8], an open source for intrusion detection/prevention, and ClamAV [9], another open source for anti-virus/worm applications. The bitmap architecture [10] and the History-based Encoding, eXecution and Addressing (HEXA) [12] compress the data structure significantly. These compression schemes are related to our work and will be reviewed in Section 3.

The purpose of this paper is to present an architecture for pattern matching with high throughput performance and low memory requirement. We first present a novel compression scheme that significantly reduces the memory requirement of the AC algorithm. Then we propose an architecture that can largely improve throughput performance and further reduce memory requirement. In comparison with the bitmap architecture, HEXA, and the schemes adopted by ClamAV and Snort, our proposed architecture results in improvement in throughput performance and requires either less or slightly more memory space. For example, for 5,000 randomly selected ClamAV signature patterns, our proposed architecture achieves, respectively, 88.4% and 94.0% reduction in memory requirement as compared with the bitmap architecture and a reduced HEXA implementation (see Section 6). In average, the improvement of processing time for scanning various types of files is 53.5%, 45.7%, 34.4% and 66.5% as compared with the bitmap architecture, reduced HEXA, and the schemes used in Snort and ClamAV, respectively.

Similar to the Wu-Manber (WM) algorithm [13] and the Hash-AV+ClamAV scheme [30], our proposed

architecture consists of a pre-filter and a verification engine. The pre-filter in the WM algorithm was implemented as a shift table and the verification engine checks all candidate patterns sequentially when a potential starting position is identified. In the Hash-AV+ClamAV scheme, the pre-filter is a Bloom filter [14], [15] and the verification engine is a simplified version of the ClamAV implementation without the failure function. Our designs are quite different from theirs. In our design, the pre-filter accumulates query results with bitwise AND operations and uses the accumulation result for filtering. Therefore, the pre-filter is stateful in our design while it is stateless in the WM algorithm and the Hash-AV+ClamAV scheme. It is proved in [33] that the proposed implementation for the stateful concept is optimal in the sense that it is equivalent to utilizing all previous query results. Our verification engine, which is a modification of the AC automaton, checks all candidate patterns simultaneously rather than sequentially. Numerical results show that our design outperforms both the WM algorithm and the Hash-AV+ClamAV scheme. There are several other pre-filter designs proposed previously [16]−[28]. Similar to hardware accelerators for the AC algorithm, these designs were aimed to utilize parallel processing to achieve high throughput performance and is out of the scope of this paper.

The rest of this paper is organized as follows. In Sections 2 and 3, we review the AC algorithm and some related works, respectively. Our proposed compression scheme and architecture are presented in Sections 4 and 5, respectively. In Sections 6 and 7, we show analysis and numerical results. Finally, we draw conclusion in Section 8.

2. THE AHO-CORASICK ALGORITHM

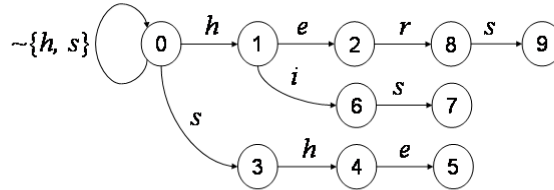
In this section, we briefly review the AC algorithm of constructing a finite state pattern matching machine for a given set of patterns $Y = \{ P_1, P_2, ..., P_y \}$. Basically, the AC pattern matching machine is dictated by three functions: a goto function g, a failure function f, and an output function *output*. Fig. 1 shows the AC pattern matching machine for $Y = \{he, she, his, hers\}$ [3], an example we shall use in this section.

One state, numbered 0, is designated as the start state. The goto function g maps a pair (state, input symbol) into a state or the message *fail*. For the considered example, we have $g(0, h) = 1$ and $g(1,\sigma) =$ *fail* if $\sigma \neq e$ or i. State 0 is a special state which never results in the *fail* message, i.e., $g(0,\sigma) \neq$ *fail* for all $\sigma \in \Sigma$, the alphabet. With this property, one input symbol is processed by the pattern matching machine 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. String u is said to represent state S if the shortest path on the goto graph from state 0 to state S spells out u. Let u and v be the strings that represent states S and Q, respectively. We have $f(S) = Q$ if and only if (iff) v is the longest proper suffix of u that is also a prefix of some pattern. It is not difficult to verify that $f(5) = 2$ for our example. The output function maps a state into a set (could be empty) of patterns. The set $output(S)$ contains pattern P iff P is a suffix of the string representing state S. As an example, we have $output(5) = \{he, she\}$.

The operation of the AC pattern matching machine is as follows. Let S be the current state and a the current input symbol. Also, let T denote the input text string. An operation cycle is defined as follows.

1) If $g(S, a) = Q$, the machine makes a state transition such that state Q becomes the current state and the next symbol of T becomes the current input symbol. If $output(Q) \neq \varnothing$ (empty set), the machine emits the set $output(Q)$. The operation cycle is complete.
2) If $g(S, a) =$ *fail*, the machine makes a failure transition by consulting the failure function f. Assume that $f(S) = R$. The pattern matching machine repeats the cycle with R as the current state and a as the current input symbol.

Initially, the start state is assigned as the current state and the first symbol of T is the current input symbol. It was proved that the pattern matching machine makes at most $2n-1$ state transitions in processing an input text string of length n. Note that failure transitions can be eliminated if the goto function is replaced with the next move function so that the pattern matching machine becomes a deterministic finite automaton. The next move function δ is defined as $\delta(S, a) = g(S, a)$ if $g(S, a) \neq$ *fail* or $\delta(S, a) = \delta(f(S), a)$ otherwise. In this case, the number of state transitions is exactly n when an input text string of length n is scanned.



(a) The goto function.

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

(b) The failure function.

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

(c) The output function.

Fig. 1. The AC pattern matching machine for $Y = \{he, she, his, hers\}$ [3].

3. RELATED WORKS

In this section, we review previous compression designs for the AC pattern matching machine and the WM algorithm. We assume that there are y patterns.

3.1. Bitmap Data Structure

Fig. 2(a) shows the bitmap data structure of the compression scheme proposed in [10]. For simplicity, it is drawn assuming that each symbol consists of three bits. Every state requires one such data structure. The failure pointer of state S points to the data structure associated with state R if $f(S) = R$ and the rule pointer points to the first element of a linked list for the matched patterns in state S. The ith bit of the bitmap is a 1 iff $g(S, i) \neq fail$. Assume that there are exactly K symbols $\sigma_1, \ldots, \sigma_K$ such that $g(S, \sigma_j) = R_j \neq fail$, $1 \le j \le K$. In this case, the addresses of the data structures associated with states $R_1, \ldots, R_K$ are stored in an array. The next pointer points to the first element of the array. In order to find $g(S, i)$, we need to count the total number of 1's in the bitmap up to the ith position if $g(S, i) \neq fail$. To reduce the processing time, one can maintain running sums of every 32 bits in the bitmap. We assume in this paper that running sums are maintained for higher throughput performance. For convenience, this compression scheme is referred to as the bitmapped AC.

It was found that, on the AC automaton constructed for patterns of Snort database, it is highly likely that there is only one symbol σ which satisfies $g(S, \sigma) \neq fail$ if state S is away from the root state. Therefore, path compression was introduced to squeeze as many such states as possible into a single state. On a 32-bit pointer machine, a maximum compression ratio of 4:1 can be obtained with the path compression technique since a path compressed state can contain data equivalent to 4 bitmap compressed states. It was reported that in practice a 2.54:1 compression ratio is achieved.

In [29], the Chinese Remainder Theorem (CRT) was applied to replace the process of counting the number of 1's in a bitmap up to a certain position with a modulo operation. Let $M = \prod_{j=1}^{K} m_j$, where m_j, $1 \le j \le K$, are a group of integers that are relatively prime. Further, let x_j, $1 \le j \le K$, be an integer which satisfies $0 \le x_j \le m_j - 1$. According to CRT, there exists a magic integer $X \in \{0, 1, ..., M-1\}$ such that $X \equiv x_j \pmod{m_j}$, $1 \le j \le K$.

Consider state S and assume that $g(S, \sigma_j) = R_j \neq fail$ for all j, $1 \le j \le K$. Let $T(\sigma_j) = m_j$, where $T(\cdot)$ is a translation function from the symbol set to a numerical set. The magic number X which satisfies $X \equiv j \pmod{m_j}$, $1 \le j \le K$, is stored associated with state S. As a result, the counting process can be replaced with a modulo operation. This scheme will be referred to as the bitmapped-CRT.

A shortcoming of the bitmapped-CRT is that the magic number could be huge if a state has a large number of child states. In this case, the storage requirement will be huge and the modulo operation may require many CPU cycles.

3.2. Banded-row Format

In the banded-row format [11], which is used in Snort, the row elements are stored from the first non-zero value (or *non-fail* value in the goto transition table) to the last non-zero value, known as band values. For example, the banded-row format of the sparse vector (0 0 0 2 4 0 0 0 6 0 7 0 0 0 0 0) is (8 3 2 4 0 0 0 6 0 7), where the first element indicates the number of vector elements stored, named bandwidth, and the second element represents the index (numbered from 0) of the first vector element stored followed by band values. Since the next move table is normally not as sparse as the goto table, we choose to compress the goto table in this paper. The corresponding pattern matching machine is referred to as the

banded-row format AC.

It is likely that some band values are the *fail* message. To reduce processing time, one can replace all band values with the results of the next move function so that no failure transition is necessary if the input symbol falls in the band. We name such a modification enhanced banded-row format AC.

3.3. AC-bnfa

AC-bnfa is another alternative adopted by Snort for pattern matching. As shown in Fig. 2(b), AC-bnfa stores for each state a transition list, which contains at least two words. The first word (in current implementation, only the least significant 24 bits) stores the state number. The second word, called control word, stores a control byte and the failure state which takes 24 bits. The control byte contains one bit to indicate whether or not some patterns are matched in the state and another bit to show if the number of its child states, denoted by C, is greater than or equal to 64. If $C<64$, then the succeeding C words are used to store the input symbols (1 byte) and the corresponding next states (3 bytes). In case $C \geq 64$, a full array of 256 words is used to store all the possible input symbols and the corresponding next states. The (input symbol, corresponding next state) pairs are searched sequentially if $C \leq 5$ or with binary search if $5<C<64$. A simple table lookup is sufficient if $C \geq 64$.

3.4. HEXA

HEXA is a compact data structure for representation of a directed graph with edges labeled by symbols in a finite alphabet [12]. In HEXA, a state is uniquely identified by the string of labeling symbols on the path leading to it, from the root state. Consider a d-state directed graph. A minimal perfect hash function η is required to generate a unique number in [0, $d-1$] for each identifier. Let d_i be the identifier of state i. The data structure for state i, stored in memory location $\eta(d_i)$, consists of two bits to indicate whether or not there are left and right child states and perhaps some auxiliary information, depending on applications. Finding a minimal perfect hash function is complicated for large values of d. The remedy suggested in [12] is to use perfect hash function which generates a unique number in [0, $\bar{d}-1$] for each identifier for some $\bar{d} \geq d$. Additional bits, called discriminators, can be added as part of the identifiers to facilitate finding a perfect hash function.

The directed graphs for pattern matching applications normally are cyclic, which implies the HEXA identifiers may become unbounded. Fortunately, the AC automaton possesses the property that, for every state S, all paths of length j leading to S are labeled by the same string of symbols if j is smaller than or equal to the length of the string representing state S. This property enables the AC automaton to be represented with a variant of HEXA, called bounded HEXA (bHEXA). Let u be the string that represents state S. Any suffix of u can be used as the identifier of S. As an example, consider the AC automaton in Fig. 1. The possible identifiers for state 9 are - (empty string), *s*, *rs*, *ers*, and *hers*.

Different from HEXA, not all history input symbols are used to identify a bHEXA state. The number of history input symbols needed is state-dependent. Therefore, bHEXA has to indicate the state identifier lengths. This increases memory space requirement. However, the fact that a state has multiple choices for its identifier increases the probability of finding a perfect hash function using short discriminators or even without discriminators.

Similar to the straightforward implementation of the AC algorithm, bHEXA stores the AC automaton, with the failure transitions eliminated, in a full two-dimensional table. The major differences are 1) the size of the table is $\bar{d}$ by $|\Sigma|$, rather than d by $|\Sigma|$, and 2) each element stored in the table is a state identifier length (and the additional discriminator, if used), rather than a state pointer of $\lceil \log_2 d \rceil$ bits. According to the experimental results provided in [12], the memory space requirement of bHEXA is about 20-30% of that required by the straightforward implementation for several real world and randomly generated pattern sets.

3.5. ClamAV Implementation

ClamAV [9] implements a variation of the AC algorithm. It does not store the complete trie. Instead, it limits the depth of the trie and partitions the patterns into groups so that two patterns are in the same group iff they have the same prefix of length equal to the depth of the trie. All patterns in the same group are saved as a linked list. In its current implementation, the depth is equal to two. Fig. 2(c) shows an example for $Y = \{he, she, his, hers\}$. Note that the next move function is used for non-leaf states 0, 1, and 3 while the failure function (shown as dashed lines) is required for leaf states 2, 4, and 5. When a leaf state is visited, all patterns on its linked list are checked sequentially.

In [30], the Bloom filter was adopted to improve throughput performance of ClamAV implementation for situations where the majority of data do not contain any pattern. This scheme, called Hash-AV+ClamAV, uses α hash functions to build the Bloom filter. The input of the hash functions is a string of β bytes. The authors analyzed the length distribution of ClamAV signatures and suggested choosing $\beta = 9$ and $\alpha = 4$. The four hash functions selected are "mask" [30], "xor+shift" [30], fast hash from hashlib.c [31], and sdbm [32].

In Hash-AV+ClamAV, a sliding window of β bytes is used to move down the input text string during scanning. The α hash functions are applied sequentially to the β bytes contained in the window. The first hash function is always applied while other hash functions are applied only if all the previous queries report hits. The ClamAV implementation of pattern matching is invoked for verification iff all four query results are hits.

Obviously, inserting all β-byte sub-strings starting at the first γ offsets of all signatures into the Bloom filter allows the sliding window to be moved γ bytes at a time. However, the false positive probability will be increased. Hash-AV+ClamAV uses this strategy and chooses $\gamma = 4$.

3.6. Wu-Manber Algorithm

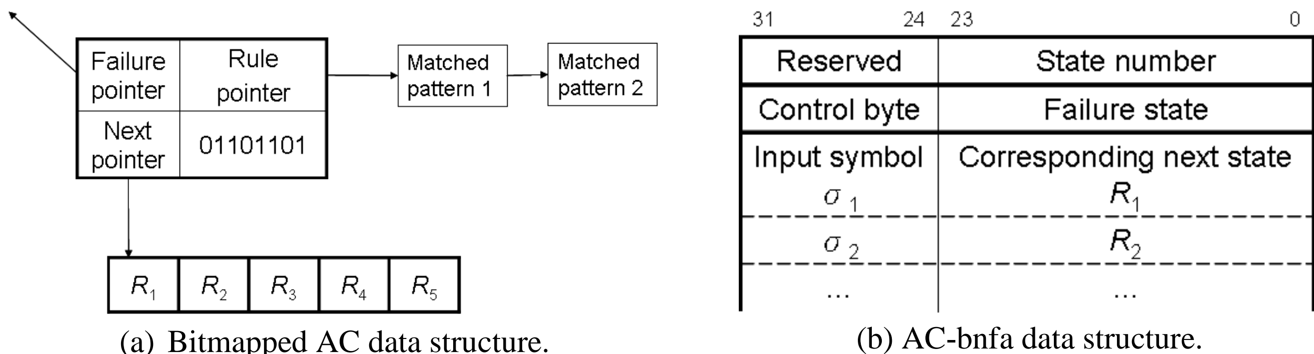
The WM algorithm [13] builds its own pre-filter and verification engine. In the pre-filter, only the first *m* symbols of each pattern are considered, where *m* is the length of the shortest pattern. Let $p_i^1 p_i^2 \dots p_i^m$, $1 \le i \le y$, represent the first *m* symbols of the ith pattern and $T = t_1 t_2 \dots t_n$ be the input text string. A *SHIFT* table, a *HASH* table, a *PAT_POINT* list, and a *PREFIX* table are required in the WM algorithm. The *SHIFT* table is used to determine how many symbols in the text can be shifted from the current position. The *HASH* and *PREFIX* tables and the *PAT_POINT* list are used when verification is needed. The *SHIFT* table is constructed with a hash function, denoted by *hash*, as shown below. In the construction procedure, N represents the number of entries in the *SHIFT* table and variable k, called block size, is a design parameter.

Construction of the *SHIFT* table

for $h \leftarrow 1$ **until** N **do** $SHIFT[h] \leftarrow m - k + 1$;
for $j \leftarrow 1$ **until** $m - k + 1$ **do**
  begin
    for $i \leftarrow 1$ **until** y **do**
      begin
        $h \leftarrow hash(p_i^j p_i^{j+1} \dots p_i^{j+k-1})$;
        $SHIFT[h] \leftarrow m - k + 1 - j$;
      end
  end

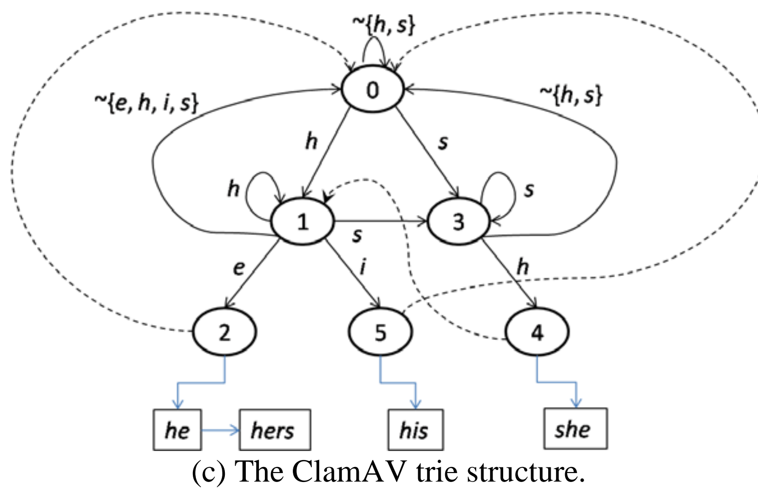
The *PAT_POINT* list is a list of pointers that point to patterns. Let P_a and P_b be, respectively, the patterns pointed by *PAT_POINT*[*i*] and *PAT_POINT*[*i*+1]. Further, let $p_a^1 p_a^2 ... p_a^m$ and $p_b^1 p_b^2 ... p_b^m$ be the first *m* symbols of P_a and P_b, respectively. It holds that $hash(p_a^{m-k+1} ... p_a^m) \leq hash(p_b^{m-k+1} ... p_b^m)$. If *SHIFT*[*h*] = 0, then the hth entry of the *HASH* table, *HASH*[*h*], contains a pointer which points to the first element of *PAT_POINT* list, say, *PAT_POINT*[*j*], such that the pattern P, with *m*-symbol prefix $p^1 p^2 ... p^m$, that is pointed by *PAT_POINT*[*j*] satisfies $hash(p^{m-k+1} ... p^m) = h$. In case *SHIFT*[*h*]$\neq$0, then *HASH*[*h*] = *HASH*[*h*+1]. Finally, the jth entry of the *PREFIX* table, *PREFIX*[*j*], contains the hash value of the *B*-symbol prefix of the pattern pointed by *PAT_POINT*[*j*].

A search window of length *m* is used during scanning. Initially, the search window is aligned with the input text string, i.e., the sub-string within the search window is $t_1 t_2 ... t_m$. The last *k* symbols of the text string within the search window are hashed. Let *h* be the hash value. If *SHIFT*[*h*]$\neq$0, then the search window is advanced by *SHIFT*[*h*] positions. In case *SHIFT*[*h*]=0, the patterns pointed by *PAT_POINT*[*j*], *HASH*[*h*]$\leq j <$ *HASH*[*h*+1], are candidate patterns and are verified sequentially. The verification starts with hashing the first *B* symbols of the text string within the search window. Let *prefix* denote the hash value. The pattern pointed by *PAT_POINT*[*j*], *HASH*[*h*] $\leq j <$ *HASH*[*h*+1], is checked against the text string directly if *PREFIX*[*j*] = *prefix*. After verification, the search window is advanced by one position.



(a) Bitmapped AC data structure.

(b) AC-bnfa data structure.



(c) The ClamAV trie structure.

Fig. 2. Data structure of previous compression designs.

4. THE PROPOSED COMPRESSION SCHEME

In our proposed compression scheme, we classify states according to the number of child states and whether or not patterns are matched. Note that there might be a self-loop at the start state. However, the goto graph becomes a tree after removing the self-loop, if exists. In the following definitions, we ignore the self-loop and consider the goto graph as a tree.

State R is said to be a child state of state S, and state S the parent state of state R, if there exists a symbol σ such that $g(S,\sigma) = R$. State S is said to be a branch state, a single-child state, or a leaf state, if it has at least two child states, exactly one child state, or no child state, respectively. Moreover, state S is said to be a final state if *output*(S) is not empty.

The data structures for branch, single-child, and leaf states are different. For a branch state S, we store $f(S)$ and $g(S,\sigma)$ for all σ. As a result, we still have a two-dimensional state transition table. However, the number of rows is only equal to the number of branch states which is at most $y-1$ for y patterns. The enhanced banded-row format AC is adopted to compress the two-dimensional table. The resulting state transition table is named the Branch State Transition (BST) table.

Assume that state S is a single-child state and $g(S,\sigma) = R \neq$ *fail*. According to the AC algorithm, it holds that the state number of R is that of S plus one. Therefore, we need only store symbol σ, denoted by S.symbol, and $f(S)$ for state S.

For a leaf state S, we simply store $f(S)$. Of course, in addition to a type identifier, every state needs a flag to indicate whether or not it is a final state and, if it is, another data structure is required to emit the matched patterns. The data structure used in this paper is linked list.

As an example, assume that alphabet $\Sigma = \{a, b, c, d, e, f, g, h, i, j\}$ and pattern set $Y = \{$*cabf, cabfdeghij, cabfgcbe, fgc, fgccabf, dabc*$\}$. The corresponding goto graph of the original AC algorithm is shown in Fig. 3. There are only two branch states, namely, states 0 and 4. Assume that the symbols in Σ are sequentially encoded from 0 to 9. The vector representing goto transitions for state 0 is (0 0 1 22 0 15 0 0 0 0) and is stored as (10 0 0 0 1 22 0 15 0 0 0 0) in our scheme. Similarly, the goto transition vector for state 4 is (*fail fail fail* 5 *fail fail* 11 *fail fail fail*) and is stored as (4 3 5 0 15 11). Note that both $g(4, e)$ and $g(4, f)$ result in the *fail* message and are replaced with $\delta(4, e) = 0$ and $\delta(4, f) = 15$, respectively.

Consider state 15 which is a single-child state. For this state, we need only store S.symbol = g and $f(15)$ = 0. Fig. 4 shows the data structures of our proposed compression scheme for this example.

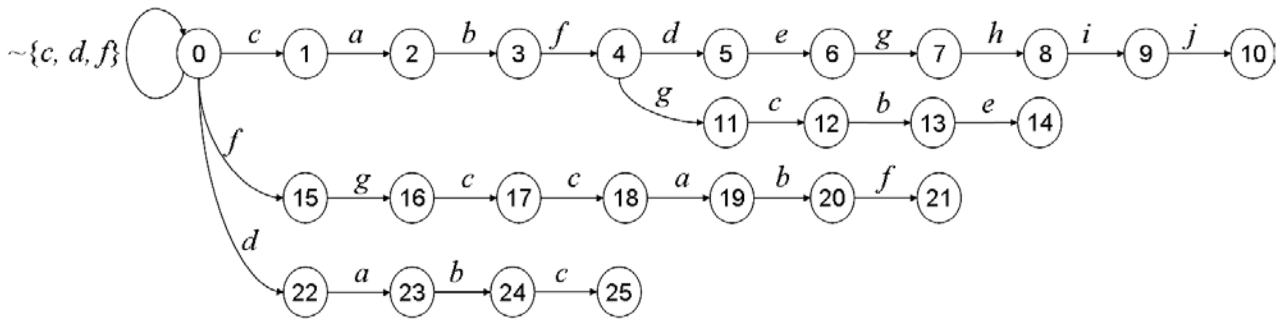


Fig. 3. The goto function for $\Sigma = \{a, b, c, d, e, f, g, h, i, j\}$ and $Y = \{$*cabf, cabfdeghij, cabfgcbe, fgc, fgccabf, dabc*$\}$.

Branch state	Bandwidth	Start index	Band values									
0	10	0	0	0	1	22	0	15	0	0	0	0
4	4	3	5	0	15	11						

(a) The Branch State Transition (BST) table.

S	1	2	3	5	6	7	8	9	11	12	13	15
***S*.symbol**	*a*	*b*	*f*	*e*	*g*	*h*	*i*	*j*	*c*	*b*	*e*	*g*

S	16	17	18	19	20	22	23	24
***S*.symbol**	*c*	*c*	*a*	*b*	*f*	*a*	*b*	*c*

(b) Data structure for single-child states.

S	1	2	3	4	5	6	7	8	9	10	11	12	13
***f*(*S*)**	0	0	0	0	22	0	0	0	0	0	16	17	0

S	14	15	16	17	18	19	20	21	22	23	24	25
***f*(*S*)**	0	0	0	1	1	2	3	4	0	0	0	1

(c) The failure function.

S	***output*(*S*)**
4	{0}
10	{1}
12	{3}
14	{2}
17	{3}
21	{0, 4}
25	{5}
others	∅

(d) The output function.

Fig. 4. Data structures of the proposed compression scheme for Σ = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*} and *Y* = {*cabf*, *cabfdeghij*, *cabfgcbe*, *fgc*, *fgccabf*, *dabc*}.

The pattern matching machine for our proposed compression scheme is described below.

Algorithm 1. Pattern matching machine for the proposed compression scheme.

begin
  $S \leftarrow 0$; $i \leftarrow 1$;
  while $i \leq n$ **do**
    begin
      if *S* is a branch state **then**
        begin
          if $\text{BST}[S][1] \leq t_i \leq \text{BST}[S][1]+\text{BST}[S][0]-1$ **then**
            begin
              $S \leftarrow \text{BST}[S][2+t_i-\text{BST}[S][1]]$;
              if $output(S) \neq \varnothing$ **then** emit *output*(*S*);
              $i \leftarrow i+1$;
            end
          else $S \leftarrow f(S)$;
        end
      else if *S* is a single-child state **then**
        begin

```
            if  t_i = S.symbol then
              begin
                S ← S+1;
                if output(S) ≠ ∅  then emit output(S);
                i ← i+1;
              end
            else S ← f(S);
           end
        else // S is a leaf state
             S ← f(S);
     end
end
```

5. THE PROPOSED ARCHITECTURE

In this section, we present an architecture to largely improve the throughput performance for situations where the majority of data do not contain any pattern. This design philosophy should be acceptable because, in normal conditions, signature is a small fraction of malicious program, which in turn is a small fraction of total traffic. An alternative design philosophy is to achieve high throughput performance in the worst case. Such a design philosophy, however, could be costly.

Our proposed pattern matching architecture consists of a pre-filter and a verification engine. The pre-filter is designed based on Bloom filters and the verification engine is modified from the AC algorithm.

5.1. Pre-filter Design

As in the WM algorithm, only the first *m* symbols of each pattern are considered in the pre-filter. Given block size *k*, there are *m*–*k*+1 membership query modules in our pre-filter design. Every membership query module uses its own memory space. Recall that $p_i^1 p_i^2 ... p_i^m$ are the first *m* symbols of pattern P_i. The sub-string $p_i^j p_i^{j+1} ... p_i^{j+k-1}$, $1 \le j \le m-k+1$, is a member stored in the jth membership query module. For convenience, these membership query modules are denoted by MQ_1, MQ_2, …, and MQ_{m-k+1}. Different from the WM algorithm, our pre-filter design uses an array of bitmaps, rather than the *SHIFT* table. Similar to the WM algorithm, we use a hash function to construct our membership query modules. The hth bit of MQ_j is set to 1 iff there exists pattern P_i such that $h = hash(p_i^j p_i^{j+1} ... p_i^{j+k-1})$.

Again, a search window *W* of length *m* is used during scanning. Initially, *W* is aligned with *T* so that the first symbol of *T*, i.e., t_1, is at the first position of *W*. The last *k* symbols in *W*, i.e., $t_{m-k+1} t_{m-k+2} ... t_m$ at this moment, are used to query MQ_1, MQ_2, …, and MQ_{m-k+1}. Let qb_i be the report of MQ_i and QB = $qb_1 qb_2 ... qb_{m-k+1}$ denote the bitmap of current query result. We observe that not only current query result but also previous ones are useful for filtering. Therefore, we introduce the stateful concept in pre-filter design. In our implementation, we use a master bitmap of size *m*−*k*+1 bits to accumulate results obtained from previous queries. Let MB = $mb_1 mb_2 \dots mb_{m-k+1}$ represent the master bitmap. Initially, we have mb_i = 1 for all *i*, $1 \le i \le m-k+1$. After a query result is fetched, we perform $MB = MB \oplus QB$, where $\oplus$ is the bitwise AND operation. A suspicious sub-string is found and the verification engine is consulted if mb_{m-k+1} = 1. The advancement of *W* is *m*−*k*+1 positions if mb_i = 0 for all *i*, $1 \le i \le m-k$, or *m*−*k*+1−*r* positions if mb_r = 1 and mb_i = 0 for all *i*, $r < i \le m-k$. If *W* is decided to be advanced by *g* positions, *MB* is right-shifted by *g* bits and filled with 1's for the holes left by the shift. Fig. 5 shows the pre-filter architecture for *m* = 6 and *k* = 3. A virtual membership query module MQ_0, which always reports a 1, is added to make the rightmost 1 detector function correctly.

Note that the pre-filter can work correctly without the master bitmap (stateless). In this case, only current query result is used to determine window advancement. It is not hard to see that, with the master bitmap, *W* can be advanced by more positions. It is worth to be pointed out that using the master bitmap is optimal in the sense that it is equivalent to utilizing all previous query results. Proof of this important property, which can be found in [33], is omitted owing to space limitation.

In general, performing multiple queries in each round can help to increase window advancement and reduce false positive probability. However, it requires more processing time than performing only one query. Assume that in each round *L* queries are performed with *L* different hash functions and *L* independent sets of membership query modules. Similar to the single-query case, the last *k* symbols within *W* are used for multiple queries during scanning. Let QB_i, $1 \le i \le L$, represent the bitmap reported from the ith query and $QB = QB_1 \oplus QB_2 \oplus ... \oplus QB_L$. The master bitmap is updated as

$MB = MB \oplus QB$. The window advancement and whether or not a suspicious sub-string is found are decided according to the value of *MB* in the same way as the single-query case. The optimal value of L that maximizes throughput performance will be analyzed in Section 7.

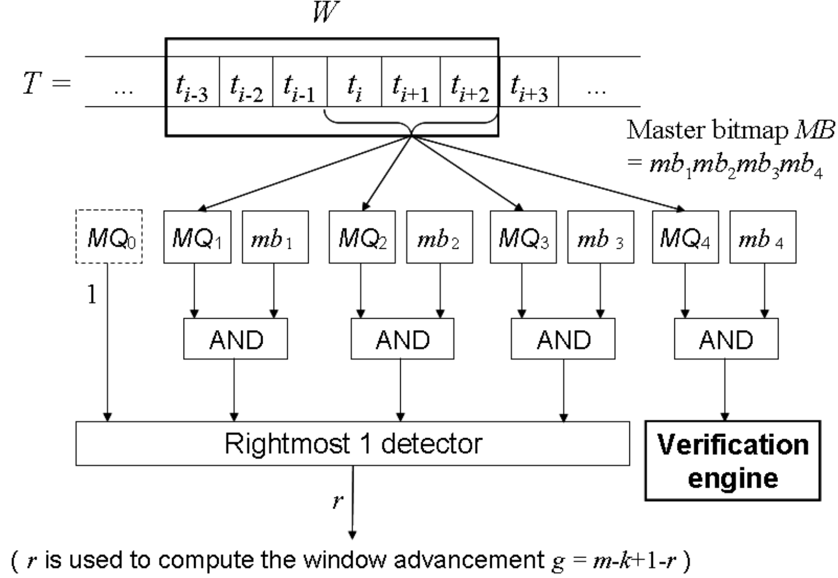


Fig. 5. The stateful pre-filter architecture for m = 6 and k = 3.

5.2. Verification Engine Design

To verify all candidate patterns simultaneously, we design the verification engine based on the AC algorithm. Because of the adoption of pre-filter, the AC algorithm has to be modified. The first modification, which concerns the goto function, is to delete the self-loop, if exists, at the start state. The reason is that the task of consuming a symbol at the start state is taken over by the pre-filter. The second modification is to omit the failure function because once the goto function returns the *fail* message, we know that the suspicious sub-string found by the pre-filter is a false positive. The third modification is regarding the output function. Assume that patterns P_i and P_j satisfy $P_i = uP_jv$, where u is a non-empty string. Let *S* be the state represented by string uP_j. In the original AC algorithm, *output*(*S*) includes pattern P_j. In our proposed architecture, pattern P_j is removed from *output*(*S*). The reason is that if pattern P_j occurs in *T*, the pre-filter will notify the verification engine when the starting position of pattern P_j is aligned with the search window. If *output*(*S*) includes pattern P_j, then P_j will be detected for multiple times if uP_j is a sub-string of *T*.

Since the failure function is not necessary, all we need to store are the goto function and the output function. The output function is simplified because at most one pattern is matched in each state. Another consequence of using pre-filter is that the space requirement of the goto function can be further reduced.

We call single-child state *R* a first single-child state if its parent state is a branch state. State *S* is said to be an explicit state if it is the start state, a branch state, a first single-child state, or a final state. We store all patterns and some data structures for the explicit states. The patterns are stored contiguously in the *Compacted_Patterns* file. For example, if *Y* = {*he, she, his, hers*}, then the *Compacted_Patterns* file is *heshehishers*.

For branch states, we store a transition table similar to the BST table presented in Section 4. The only

difference is that the banded-row format, instead of the enhanced version, is used here. This is because the *fail* messages should be kept so that the verification procedure ends at the right moment when a *fail* message is encountered.

Assume that state S is a single-child state and is represented by string u. State R is said to be a descendent state of state S if it is represented by uv (concatenation of u and v), where v is a non-empty string. Furthermore, state R is said to be a descendent explicit state of state S if, in addition to being a descendent state of state S, R is an explicit state. State R is said to be the nearest descendent explicit state (NDES) of state S if state R is a descendent explicit state of state S and there is no other explicit state on the path from state S to state R.

Suppose that state S is a first single-child state and state R is its NDES. Let $P_l = uvr$ be the first pattern in pattern set which contains uv as a prefix. The data structure for state S includes S.position and S.distance, where S.position and S.distance represent, respectively, the position of the $(|u|+1)^{th}$ byte of P_l in the *Compacted_Patterns* file and $|v|$, where $|x|$ denotes the length of string x . If the start state or a final state is a single-child state, its data structure is the same as that for state S.

Finally, for each leaf state, we store nothing but an identifier to indicate that all input symbols result in the *fail* message. Of course, every explicit state needs a flag to indicate whether or not it is a final state and, if it is, the identification of the matched pattern is stored. Note that the resulting goto graph after compression contains only explicit states. As a result, the memory requirement is further reduced in comparison with the compression scheme presented in Section 4.

Consider the same example studied in Section 4. Fig. 6 shows our compressed goto graph. Compared with the figure shown in Fig. 3, one can see that the number of states is reduced from 26 to 12. Note that the states on the compressed goto graph are renumbered such that S.NDES for explicit single-child state S is numbered S+1. As an example, let S be state 7, a first single-child state. Since state 8, represented by *fgc*, is the NDES of S with distance 2 and *fgc* is the first pattern which contains *fgc* as a prefix , we store S.position = 24 and S.distance = 2. As another example, if S is state 8, we store S.position = 29 and S.distance = 4.

States 0 and 2 are branch states while states 4, 6, 9, and 11 are leaf states. The remaining states are either first single-child states or single-child final states. Fig. 7 shows the data structures of our verification engine for this example.

5.3. Pattern Matching Machine

Let c_x be the xth symbol in the *Compacted_Patterns* file. The pattern matching machine for our proposed stateful architecture is described below. In the description, b^a represents a binary string with bit b repeating a times, and $1^g/(MB>>g)$ means the master bitmap is right-shifted by g bits and filled with 1's for the holes left by the shift. Moreover, $qb_1^z qb_2^z \ldots qb_{m-k+1}^z$ are the bitmap of the query result reported from the zth set of membership query modules when $t_i t_{i+1} \ldots t_{i+k-1}$ is used as the input for query.

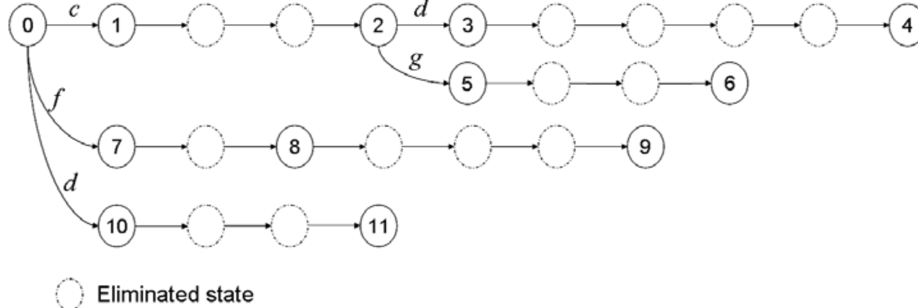


Fig. 6. The compressed goto graph for Σ = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*} and *Y* = {*cabf, cabfdeghij, cabfgcbe, fgc, fgccabf, dabc*}.

Branch state	Bandwidth	Start index	Band values			
0	4	2	1	10	*fail*	7
2	4	3	3	*fail*	*fail*	5

(a) The BST table.

S	**1**	**3**	**5**	**7**	**8**	**10**
***S*.position**	2	10	20	24	29	34
***S*.distance**	3	5	3	2	4	3

(b) Data structure for first single-child states and single-child final states.

S	***output*(*S*)**
2	{0}
4	{1}
6	{2}
8	{3}
9	{4}
11	{5}
0, 1, 3, 5, 7, 10	∅

(c) The output function.

cabfcabfdeghijcabfgcbefgcfgccabfdabc

(d) *Compacted_Patterns*.

Fig. 7. Data structures of the verification engine for Σ = {*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*} and *Y* = {*cabf, cabfdeghij, cabfgcbe, fgc, fgccabf, dabc*}.

Algorithm 2. Pattern matching machine for the proposed stateful architecture.

begin
  $i \leftarrow m-k+1$; $MB \leftarrow 1^{m-k+1}$;
  while $i \leq n-k+1$ **do**
    begin
      /* Pre-filter */
      $QB \leftarrow qb_1^1 qb_2^1 \ldots qb_{m-k+1}^1$;
      for $z \leftarrow 2$ **until** L **do** $QB \leftarrow QB \oplus qb_1^z qb_2^z \ldots qb_{m-k+1}^z$;
      $MB \leftarrow MB \oplus QB$; // $MB = mb_1 mb_2 \ldots mb_{m-k+1}$
      if $mb_{m-k+1} = 1$ **then**
          begin
        $S \leftarrow 0$; $j \leftarrow i-m+k$; **call** Verification_Engine(S, j);
        end
      if $mb_1 mb_2 \ldots mb_{m-k} = 0^{m-k}$ **then** g $\leftarrow m-k+1$;
      else if $mb_1 mb_2 \ldots mb_{m-k} = mb_1 \ldots mb_{r-1} 1\, 0^{m-k-r}$
      then g $\leftarrow m-k+1-r$;
      $i \leftarrow i+g$; $MB \leftarrow 1^g \,/\, (MB >> g)$;
    end
end

```
procedure Verification_Engine(S, j)
begin
   while j ≤ n do
      begin
         if output(S) ≠ ∅  then emit output(S);
         if S is a branch state then
            begin
               if BST[S][1] ≤ t_j ≤ BST[S][1]+ BST[S][0]−1 then
                  begin
                     if BST[S][2+t_j −BST[S][1]] ≠ fail then
                        begin
                           S ← BST[S][2+t_j −BST[S][1]];
                           j ← j+1;
                        end
                     else // BST[S][2+t_j −BST[S][1]] = fail
                        break;
                  end
               else //  t_j < BST[S][1] or  t_j > BST[S][1]+ BST[S][0]−1
                  break;
            end
         else if S is an explicit single-child state then
            begin
               x ← S.position;    w ← S.distance;
               if  t_j t_{j+1}...t_{j+w−1} = c_x c_{x+1}...c_{x+w−1}  then
                  begin
                     S ← S+1; // S+1 is S.NDES
                     j ← j+w;
                  end
               else //  t_j t_{j+1}...t_{j+w−1} ≠ c_x c_{x+1}...c_{x+w−1}
                  break;
            end
         else // S is a leaf state
            break;
      end
end
```

The number of memory accesses required by each type of state is analyzed below. Assume that four bytes are fetched in a memory access. For a branch state, to process one byte, we need one memory access to obtain bandwidth (two bytes), the index of the first vector element stored (one byte), state type (two bits), and the final state indication (one bit). Another memory access is required if the input symbol is within the band. Of course, we need one more memory access for the identification of the matched pattern (two bytes) if the state is a final state. Therefore, a branch state requires one to three memory accesses. For an explicit single-child state *S*, *S*.distance bytes are processed with at most $\lceil S.\text{distance}/4 \rceil + 3$ memory accesses, where $\lceil \bullet \rceil$ is the ceiling function. More specifically, we need one to obtain state type, *S*.distance (two bytes), and the final state indication (one bit), another one to get

S.position (four bytes), up to $\lceil S.\text{distance}/4 \rceil$ to reach S.NEDS, and one for the matched pattern if state S is a final state. For a leaf state, we need one memory access to obtain state type and the identification of the matched pattern. Since the operations are quite simple, the proposed architecture is also suitable for hardware implementation.

6. Comparison with Related Works

In this section, we compare the performances of the investigated pattern matching algorithms. All algorithms are implemented in C++ and the experiments are conducted on a PC with an Intel Pentium 4 CPU operated at 2.80GHz with 512MB of RAM. Patterns are selected from ClamAV database. The AC algorithm without compression cannot handle the full set of ClamAV signatures because of the explosion of its memory space requirement. Therefore, we conduct the first simulation with 5,000 randomly selected signatures. The minimum, maximum, and average lengths of the selected signatures are 10, 163, and 34.26 bytes, respectively. The total number of states d generated by the AC algorithm is 158,743. Since the shortest pattern is 10 bytes, we set search window length m = 10. Selection of block size k impacts false positive rate and average window advancement. The false positive rate tends to be large for a small value of k and the average window advancement is small for a large value of k. Our experimental results show that $k = 4$ is a good choice. The size (in bits) of a membership query module *N*, or equivalently, the number of entries in the *SHIFT* table, is 2^{16} which results in a false positive probability of approximately 0.0734. For a fair comparison, we set the pre-filter size of the Hash-AV+ClamAV scheme to be 2^{19} bits since it uses only one membership query module. For β=9, the Hash-AV+ClamAV scheme requires a "two-scan" approach. That is, Hash-AV+ClamAV is first performed for signatures longer than or equal to β+3 bytes and then ClamAV is run for the rest of the signatures. To avoid the "two-scan" mode, we remove signatures shorter than β+3 bytes for the Hash-AV+ClamAV scheme. We compare the basic version of our proposed architecture, i.e., one query per round, with various related pattern matching algorithms. A simple hash function is adopted for our proposed pre-filter. Let $t_i t_{i+1} t_{i+2} t_{i+3}$ be the data block to be hashed. The hash function generates $t_i t_{i+3}$ as the hash result. The effect of multiple queries per round will be studied in the next section.

We implemented a reduced version of bHEXA to avoid using a complicated perfect hash function to speed up the matching procedure. The order to generate the identifiers of states is described below. The identifier of state *S* is generated before that of state *R* if the representing string of *S* is shorter than that of state *R*. If the lengths of the representing strings of states *S* and *R* are identical, then the identifier of state *S* is generated before that of state *R* if *S* is created earlier than *R*, according to the AC algorithm. Let *u* be the representing string of state *S*. We choose *v* as the state identifier of *S* iff *v* is the shortest suffix of *u* that is different from the identifiers already generated. Note that the identifier of the start state is the empty string. As for memory requirement, we let $\bar{d} = d =$ 158,743. The number of bits required to represent state identifier lengths increases as the maximum length of state identifiers increases. In our implementation, patterns which generate states with identifier lengths greater than 7 are deleted so that 3 bits are sufficient to represent the identifier lengths. The number of patterns deleted is 1,431. No discriminators are used. To speed up the matching procedure, we use simple bitwise operations as the hash function η, rather than a perfect hash function, which often requires complicated computations. To avoid memory collision, we only store the data structures of the states generated by 20 representative patterns out of the remaining 3,569 patterns. Note that the memory requirement is kept at $\bar{d} = d$ entires. Obviously, the memory requirement and the processing time of such a reduced implementation should be less than those of the original bHEXA scheme.

The performance of the AC algorithm using goto and failure functions is close to that of the one using the next move function. Therefore, we only show the performance of the AC algorithm using the next move function. Similarly, we omit the performance of the banded-row format AC and the bitmapped-CRT because the enhanced banded-row format AC performs slightly better than the former and the latter does not provide significant improvement over the bitmapped AC. Using path compression technique tends to increase processing time of bitmapped AC and thus its performance is also omitted. We name our proposed compression scheme presented in Section 4 the compressed AC and the stateful architecture in

Section 5 the Pre-filter+AC scheme. Table I summarizes the processing time for each algorithm when scanning various types of files that contain no patterns. The processing time for scanning a file with pattern occurrence at position n is roughly the same as that for scanning a clean n-byte file if an algorithm halts when a match is found.

As shown in Table I, AC-bnfa and bimapped AC require relatively high processing time. This is because the former needs linear or binary search to perform state transition and the latter needs to compute the population count in a 32-bit bitmap.

TABLE I
PROCESSING TIME FOR SCANNING VARIOUS TYPES OF FILES AND MEMORY REQUIREMENT

Processing time (ms)		Schemes									
		AC-bnfa	ClamAV	Bitmapped AC	reduced bHEXA	Enhanced banded-row format AC	Compressed AC	Hash-AV + ClamAV	AC	WM	Pre-filter + AC
scanned files	AC.cpp (4KB)	1.40	1.08	1.09	0.77	0.77	0.77	0.64	0.63	0.62	0.31
	list.txt (10KB)	2.66	2.18	1.72	1.57	1.39	1.26	1.44	1.26	1.24	1.16
	index.htm (78KB)	16.26	18.27	9.84	9.19	6.41	5.64	5.76	5.32	5.75	5.15
	bootcfg.exe (186KB)	32.98	36.54	22.68	20.75	15.94	13.76	15.98	12.02	14.22	11.10
	wmvcore.dll (2.2MB)	431.10	419.38	321.25	238.27	209.85	182.06	167.84	153.91	133.28	113.59
Memory requirement (M bytes)		2.46	0.32	7.83	15.24	3.47	1.82	0.39	162.92	0.39	0.91

The ClamAV scheme has to check all patterns on the associated linked list using sequential string comparisons, whenever a leaf state is visited. The checking procedure is quite time-consuming when the linked list contains a large number of patterns. Therefore, the ClamAV scheme also requires relatively high processing time. The Hash-AV+ClamAV scheme improves the performance of the ClamAV scheme.

The processing time of bHEXA is larger than that of the AC algorithm. This is because bHEXA has to extract history input symbols and perform a hash function to generate the memory location which contains information of the next state. For the enhanced banded-row format AC, every success transition requires memory access and computation to obtain the updated current state. By contrast, for the compressed AC, the current state for a success transition can be easily updated by increasing the current state number by one when a single-child state is visited. That is the reason why the compressed AC requires less processing time than the enhanced banded-row format AC.

The AC, the WM, and the Pre-filter+AC schemes yield the best performance on processing time among all investigated schemes. The main reason for the AC algorithm is that it uses the non-compressed state transition table and, therefore, the updated current state can be obtained without any decompression operations.

To evaluate the performance of our proposed Pre-filter+AC scheme under various fractions of malicious traffic, we replaced some content (at random locations) of a Windows' executable of size 2.2MB with 10, 100, or 1000 signatures. If the average size of a malicious program is 1K bytes, then the fraction of malicious traffic is about 0.45%, 4.5%, or 45% for 10, 100, or 1000 signatures, respectively. The experimental results are shown in Fig. 8. The processing times of the AC algorithm are also shown for

comparison. The figure shows that the processing time of the Pre-filter+AC scheme increases as the number of signatures in the scanned file increases. However, it is still faster than the AC algorithm.

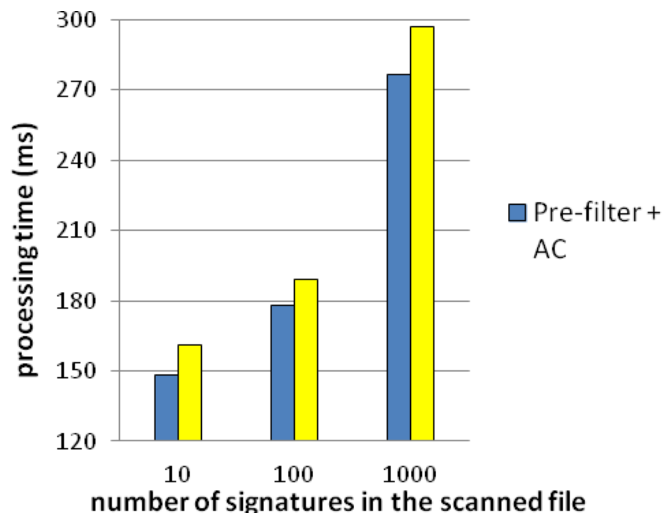


Fig. 8. Processing time for scanning files with different numbers of signatures.

Memory requirements of the investigated schemes are provided in the last row of Table I. All schemes with compressed data structures require acceptable memory space. Our proposed Pre-filter+AC scheme requires less than 1M bytes, including 64K bytes for pre-filter, 172K bytes for patterns, and 674K bytes for data structures of the modified AC automaton. Since the proposed Pre-filter+AC scheme and the WM algorithm yield the best throughput performance with acceptable memory requirement, in the next experiment, we compare these two schemes with full rule set.

As of June 2010, there are 29,179 string signatures contained in ClamAV database. The minimum, maximum, and average lengths of the signatures are 10, 210, and 66.43 bytes, respectively. Since the shortest pattern is still 10 bytes, we use $m = 10$ again. For the full rule set, the time required to construct the data structure of the proposed Pre-filter+AC scheme is 1812 ms. The construction is needed only in the beginning or when the rule set is changed.

Table II shows the memory requirement comparison of the Pre-filter+AC scheme and the WM algorithm for different values of N. As one can see, when N is small, the WM algorithm requires less memory space than the Pre-filter+AC scheme. However, the memory requirement of the WM algorithm grows rapidly as N increases. By contrast, the growth for the Pre-filter+AC scheme is relatively slow. When $N = 2^{22}$, the memory requirement for the Pre-filter+AC scheme is 59.5% of that for the WM algorithm. The percentage decreases as N increases. This is because the memory requirement for the verification engine in the Pre-filter+AC scheme is not influenced by the value of N, while that in the WM algorithm is.

TABLE II
MEMORY REQUIREMENT COMPARISON FOR DIFFERENT VALUES OF N

Memory requirement (M bytes)		N			
		2^{16}	2^{18}	2^{20}	2^{22}
Schemes	Pre-filter + AC	4.58	4.78	5.56	8.71
	WM	2.25	2.84	5.20	14.64

Both schemes need to store the pattern set. The Pre-filter+AC scheme uses *Compacted_Patterns* file, which requires 1.94M bytes. The WM algorithm needs slightly larger memory space because an ending symbol is required for each pattern. The pre-filters of both schemes take 0.06M, 0.26M, 1.04M, and

4.19M bytes of memory for $N = 2^{16}$, 2^{18}, 2^{20}, and 2^{22}, respectively. The verification engine of the Pre-filter+AC scheme requires 2.58M bytes, independent of N. For the WM algorithm, the verification engines takes 0.22M, 0.61M, 2.19M, and 8.48M bytes of memory for $N = 2^{16}$, 2^{18}, 2^{20}, and 2^{22}, respectively.

Fig. 9 shows the processing time comparison of the Pre-filter+AC scheme and the WM algorithm for different values of N. The scanned file is a Windows' executable of size 2.2MB. As one can see, the Pre-filter+AC scheme needs less processing time than the WM algorithm. The reasons are: 1) the stateful concept allows the search window to be advanced more in comparison with the stateless design, and 2) the verification engine in the Pre-filter+AC scheme checks all candidate patterns simultaneously, while that in the WM algorithm needs to check them one by one.

7. Analytical Comparison with Wu-Manber Algorithm

In this section, we analyze and compare the performances of the WM algorithm, the proposed stateful architecture, and the stateless version. The average window advancement per unit time, which determines achievable throughput, is selected as performance metric. For simplicity of analysis, we assume that symbols in patterns and input text string are independent, uniformly distributed over alphabet. Recall that the query result $QB = QB_1 \oplus QB_2 \oplus ... \oplus QB_L = qb_1 qb_2 \ ... \ qb_{m-k+1}$. Let ρ represent the probability of $qb_i = 1$ for any i, $1 \le i \le m-k+1$. We have $\rho = [1-(1-1/N)^y]^L$. Let G_L denote the random variable of window advancement for a round of L queries. The average window advancement, denoted by $\overline{G}_L$, can be evaluated by

$$\overline{G}_L = \sum_{g=1}^{m-k+1} gP(G_L = g) \tag{1}$$

where $P(G_L = g)$ is different for different algorithms.

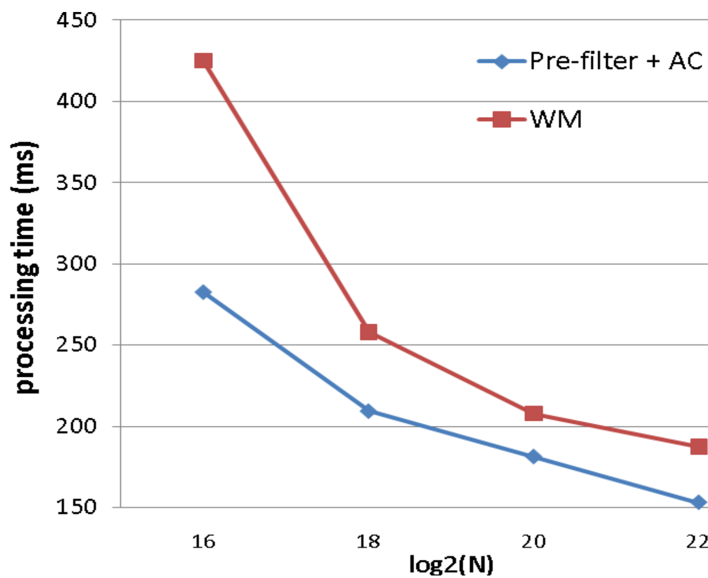


Fig. 9. Processing time comparison for different values of *N*.

7.1. Pre-filter Performance

We use $\overline{T}_h$ to represent the average time consumed in one query. As a consequence, the average time spent in a round of L queries is $L\overline{T}_h$ and $\overline{G}_L/(L\overline{T}_h)$ determines pre-filter throughput. It is reasonable to assume that $\overline{T}_h$ is the same for algorithms that use the same set of L hash functions. Assuming that all investigated algorithms use the same set of hash functions, we can conclude that pre-filter throughput is proportional to $\overline{G}_L / L$. The optimal value of L which maximizes throughput satisfies

$$L = \arg\max_H \{\overline{G}_H / H\}. \tag{2}$$

We derive $\overline{G}_L$ for the WM algorithm, the proposed stateful architecture, and the stateless version below.

7.1.1. The Wu-Manber Algorithm

Conceptually, the window advancement decided by the *SHIFT* table in the WM algorithm is equivalent

to that decided by the stateless version of our proposed architecture, except that the window is advanced by only one position if $qb_{m-k+1}=1$. Therefore, we have

$$P(G_L=g)=\begin{cases} P(qb_{m-k+1}=1)+P(qb_{m-k+1}=0)\,P(qb_{m-k}=1) \\ =\rho+(1-\rho)\rho, & \text{if } g=1 \\ P(qb_{m-k+1-g}=1)\prod_{i=1}^{g}P(qb_{m-k+2-i}=0) \\ =\rho(1-\rho)^{g}\ , & \text{if } 1<g<m-k+1 \\ \prod_{i=1}^{m-k+1}P(qb_i=0)=(1-\rho)^{m-k+1} \\ =(1-\rho)^{g}\ , & \text{if } g=m-k+1 \end{cases} \tag{3}$$

and $\bar{G}_L$ can be obtained from (1) and (3).

7.1.2. The Stateless Architecture

For the stateless version of our proposed architecture, the bit qb_{m-k+1} is not involved in determining window advancement. Consequently, we have

$$P(G_L=g)=\begin{cases} P(qb_{m-k+1-g}=1)\prod_{i=1}^{g-1}P(qb_{m-k+1-i}=0) \\ =\rho(1-\rho)^{g-1}, & \text{if } 1\le g<m-k+1\,. \\ \prod_{i=1}^{m-k}P(qb_i=0)=(1-\rho)^{m-k} \\ =(1-\rho)^{g-1}, & \text{if } g=m-k+1 \end{cases} \tag{4}$$

Similarly, $\bar{G}_L$ can be obtained from (1) and (4). Note that the average window advancement of this architecture is greater than that of the WM algorithm. The reason is that when $qb_{m-k+1}=1$, the window advancement of the WM algorithm is always one and it can be greater than one for our algorithm.

7.1.3. The Stateful Architecture

For the proposed stateful architecture, we use Markov chain to analyze the average window advancement. Again, the bit mb_{m-k+1} is not involved in determining window advancement. The states of the Markov chain correspond to the values of $mb_1mb_2...mb_{m-k}$, the left $m-k$ bits of MB after bitwise ANDing with QB (but before the right shift). As a result, there are 2^{m-k} states on the Markov chain. Since the symbols in patterns and input text string are independent, uniformly distributed over alphabet, the Markov chain is homogeneous.

Let X_l be the state after the lth round of queries. Further, let $p_{i,j}=P(X_{l+1}=j\mid X_l=i)$, $0\le i,\ j\le 2^{m-k}-1$, denote state transition probabilities and $\Pi=(\pi_0,\ \pi_1,\ ...,\ \pi_{2^{m-k}-1})$ represent the stationary probability distribution. Given $p_{i,j}$, one can compute Π and then $\bar{G}_L$ can be obtained by

$$\bar{G}_L=\sum_{i=0}^{2^{m-k}-1}\pi_i g_i\,, \tag{5}$$

where g_i is the window advancement under the condition that current state is state i.

Instead of using complicated expressions for general cases, we explain the derivation of $p_{i,j}$ with an example which assumes m = 10 and k = 4. For this example, there are 64 states on the Markov chain.

Let states 0, 1, …, and 63 correspond to $mb_1mb_2...mb_6$ = 000000, 000001, …, and 111111, respectively. We demonstrate the calculation of $p_{i,j}$ for i = 1 and 0 ≤ j ≤ 63 here. State 1 corresponds to $mb_1mb_2...mb_6$ = 000001. As a result, the window will be advanced by 1 position and the content of MB becomes 1000001. After bitwise ANDing with $QB = qb_1qb_2...qb_7$, MB becomes $qb_1 00000 qb_7$ and the updated $mb_1mb_2...mb_6$ is $qb_1 00000$. The next state j is state 0 if qb_1 = 0 or state 32 otherwise. Therefore, we have

$$p_{1,j} = \begin{cases} 1-\rho, & \text{if } j = 0 \\ \rho, & \text{if } j = 32 \\ 0, & \text{if } j = 1, 2, ..., 31, 33, ..., \text{or } 63 \end{cases}. \quad (6)$$

7.2. Overall System Performance

Let $\overline{C}_L$ be the average time spend on L queries and verification, if needed. $\overline{G}_L / \overline{C}_L$ determines achievable system throughput and the optimal value of L which maximizes throughput is given by

$$L = \arg\max_H \{\overline{G}_H / \overline{C}_H\}. \quad (7)$$

Let $\overline{T}_v$ represent the average time consumed in verification. In the WM algorithm, verification is required when the window advancement decided by the *SHIFT* table is 0. In the proposed stateless architecture, verification is required when $qb_{m-k+1} = 1$. Note that the two conditions are equivalent. For the proposed stateful architecture, verification is required if mb_{m-k+1}=1 after bitwise ANDing with *QB*. Since mb_{m-k+1} is always 1 before bitwise ANDing with *QB*, the probability of mb_{m-k+1}=1 after bitwise ANDing with *QB* is equal to $P(qb_{m-k+1}=1)$. Therefore, we have

$$\overline{C}_L = L\overline{T}_h + P(qb_{m-k+1} = 1)\overline{T}_v = L\overline{T}_h + \rho\overline{T}_v \quad (8)$$

for the WM algorithm and the proposed stateless and stateful architectures. Note that the value of $\overline{T}_v$ depends on number of patterns and the verification algorithm. We shall study numerically the optimal value of L in the next sub-section.

7.3. Numerical Results

The throughput performance depends on the values of m, k, N, y, and L. To find the optimal value of L that maximizes throughput, we fix the other parameters. For m = 10, k = 4, $N = 2^{13}$, and y = 10,000, L = 1 satisfies (2) that maximizes pre-filter throughput for the WM algorithm and the stateless/stateful architectures. Therefore, we choose L = 1 to compare pre-filter throughput, which, as mentioned before, is proportional to $\overline{G}_L / L$. Fig. 10 shows the results for various pattern numbers.

As one can see in Fig. 10, the proposed pre-filter with master bitmap (stateful) outperforms both the proposed pre-filter without master bitmap and the pre-filter of the WM algorithm (stateless). This is because previous query results, in addition to the current one, are used to determine window advancement in the stateful design. Besides, the proposed pre-filter without *MB* outperforms the pre-filter of the WM algorithm. The reason has been mentioned in Section 7.1.2.

The experiment shows that $\overline{T}_h = 4.90 \times 10^{-5}$ ms and $\overline{T}_v = 2.55 \times 10^{-4}$ ms for our proposed algorithms with the same values of m, k, N, and y above. From (7), the optimal L that maximizes system performance of the stateful architecture is 4. To demonstrate the effect of multiple queries in each round, we conduct experiments to compare the processing times for the stateful architecture with L = 1 and L = 4. Fig. 11 shows the results for various file sizes. One can note that the implementation with L = 1 requires about 1.7 times the processing time of that with L = 4. The processing time of the WM algorithm is also

provided in the figure for comparison. The WM algorithm requires about 1.3 and 2.2 times the processing time of our proposed stateful architecture with $L = 1$ and $L = 4$, respectively.

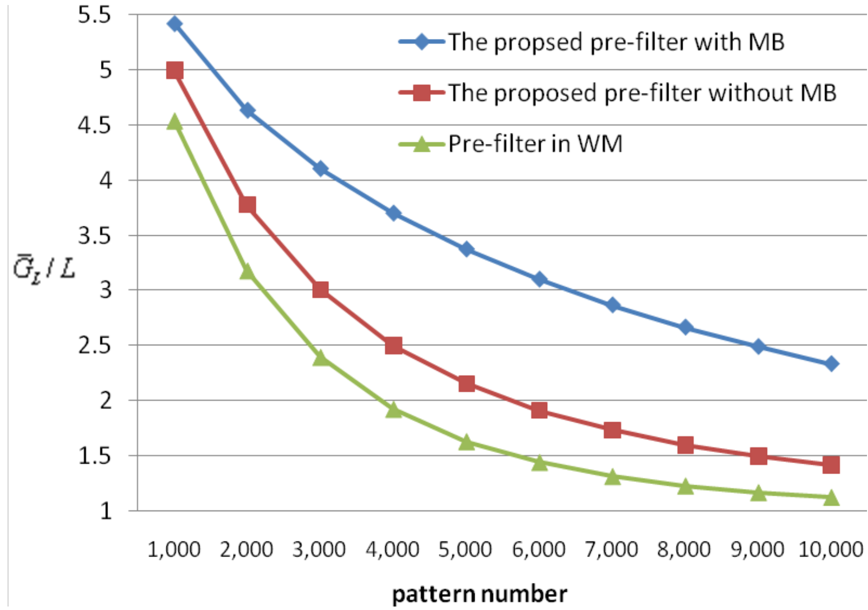


Fig. 10. Pre-filter performance comparison for various pattern numbers.

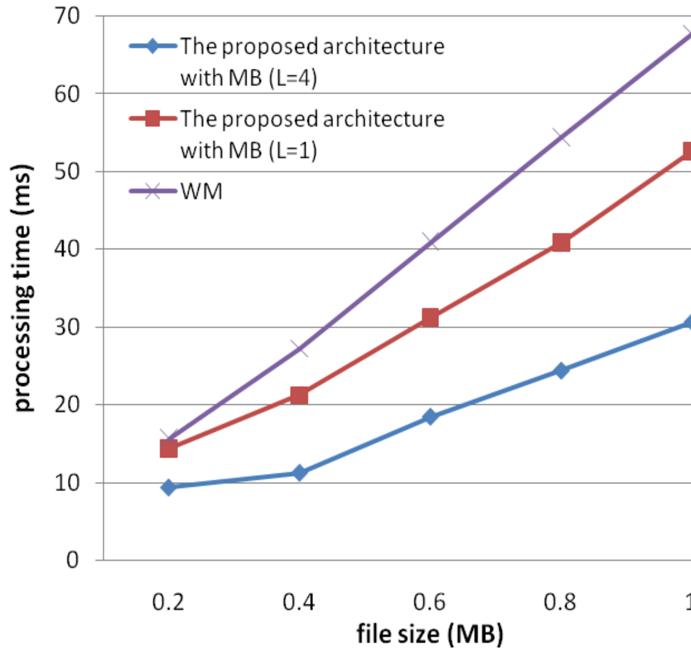


Fig. 11. Processing time comparison for scanning random texts with various sizes.

8. CONCLUSION

In this paper, we first present a novel implementation of the Aho-Corasick pattern matching algorithm that significantly compresses the required data structure. In the implementation, we reduce the two-dimensional state transition table so that its memory requirement is proportional to the number of patterns instead of the total length of patterns. Then, we propose an architecture that further reduces the memory requirement and achieves high throughput performance. In the architecture, we introduce the stateful pre-filter concept and present an AC-based verification engine which can check all candidate patterns simultaneously. Master bitmap with simple shift and bitwise AND operations is used to efficiently implement the stateful pre-filter concept. Such a simple implementation is optimal because it is equivalent to utilizing all previous query results.

The performances of different pre-filter designs are evaluated both mathematically and numerically. The effect of multiple queries in each round is also studied. Results show that the proposed pre-filter with master bitmap (stateful) outperforms both the proposed pre-filter without master bitmap and the pre-filter of the widely used Wu-Manber algorithm (stateless). In addition to the Wu-Manber algorithm, various related worksare compared with our proposed schemes. The stateful architecture performs the best in terms of both memory requirement and processing time among the schemes that yield satisfactory performance for both metrics. Therefore, for applications that require high throughput performance with memory space constraint, e.g., an embedded security appliance in a high-speed network environment, our proposed stateful architecture is the preferable choice.

Clearly, a larger value of search window length gives a better throughput performance. However, the length of search window is upper bounded by the length of the shortest pattern. Consequently, a virus expert, if possible, should avoid short patterns in deriving signatures to improve performance and reduce false alarms. An interesting further research topic is to implement and compare the performances of various pattern matching algorithms on multi-thread, multi-core processors.

REFERENCES

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

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

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

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

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

[6] T. H. Lee and C. C. Liang, “A high-performance memory-efficient pattern matching algorithm and its implementation,” IEEE TENCON, 2006.

[7] Snort website http://www.snort.org/.

[8] M. Roesch, “Snort – lightweight intrusion detection for networks,” Proceedings of LISA’99: 13th Systems Administration Conference, pp.s 229–238, Nov. 1999.

[9] Clam AntiVirus (ClamAV) website http://www.clamav.net/.

[10] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” IEEE INFOCOM, 2004.

[11] M. Norton, “Optimizing pattern matching for intrusion detection,” Sourcefire Inc., Sep. 2004.

[12] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, “HEXA: compact data structures for faster packet processing,” IEEE ICNP, pp. 246-255, 2007.

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

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

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

[16] 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.

[17] 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.

[18] S. Dharmapurikar and J. Lockwood, “Fast and scalable pattern matching for content filtering,” Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems, pp. 183–192, 2005.

[19] S. Dharmapurikar and J. Lockwood, “Fast and scalable pattern matching for network intrusion detection systems,” Selected Areas in Communications, IEEE Journal on, vol. 24, pp. 1781–1792, Oct. 2006.

[20] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, “Longest prefix matching using Bloom filters,” Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 201–212 , 2003.

[21] 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.

[22] N. S. Artan and H. J. Chao, “Multi-packet signature detection using prefix Bloom filters,” IEEE GLOBECOM, vol. 3, pp. 1811–1816, 2005.

[23] N. S. Artan and H. J. Chao, “TriBiCa - trie bitmap content analyzer for high-speed network intrusion detection,” IEEE INFOCOM, pp. 125–133, May 2007.

[24] 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.

[25] N. S. Artan, R. Ghosh, G. Yanchuan, and H. J. Chao, “A 10-Gbps high-speed single-chip network intrusion detection and prevention System,” IEEE GLOBECOM, pp. 343–348, Nov. 2007.
[26] N. S. Artan, M. Bando, and H. J. Chao, “Boundary hash for memory-efficient deep packet inspection,” IEEE ICC, pp. 1732–1737, May 2008.
[27] M. Bando, N. S. Artan, and H. J. Chao, “Highly memory-efficient loglog hash for deep packet inspection,” IEEE GLOBECOM, Nov./Dec. 2008.
[28] N. S. Artan, Y. Haowei, and H. J. Chao, “A dynamic load-balanced hashing scheme for networking applications,” IEEE GLOBECOM, Nov./Dec. 2008.
[29] T. F. Sheu, N. F. Huang, and H. P. Lee, “A time- and memory- efficient string matching algorithm for intrusion detection systems,” IEEE GLOBECOM, 2006.
[30] O. Erdogan and P. Cao, “Hash-AV: fast virus signature scanning by cache-resident filters,” International Journal of Security and Networks, vol. 2, pp. 50-59, Mar. 2007.
[31] GNU. hashlib.c – functions to manage and access hash tables for bash. In http://www.opensource.apple.com/darwinsource/10.3/bash-29/bash/hashlib.c, 1991.
[32] O. Yigit. sdbm - substitute dbm. In http://search.cpan.org/src/NWCLARK/perl-5.8.4/ext/SDBM_File/sdbm, 1990.
[33] T. H. Lee and N. L. Huang, “Design and evaluation of an efficient pattern matching architecture,” TR-EE010-NCTU-2010.