A Pattern-Matching Scheme With High Throughput Performance and Low Memory Requirement

Tsern-Huei Lee, *Senior Member, IEEE*, and Nai-Lun Huang

***Abstract*—Pattern-matching techniques have recently been applied to network security applications such as intrusion detection, virus protection, and spam filters. The widely used Aho–Corasick (AC) algorithm can simultaneously match multiple patterns while providing a worst-case performance guarantee. However, as transmission technologies improve, the AC algorithm cannot keep up with transmission speeds in high-speed networks. 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 present a pattern-matching architecture consisting of a stateful pre-filter and an AC-based verification engine. The stateful pre-filter is optimal in the sense that it is equivalent to utilizing all previous query results. In addition, the filter can be easily realized with bitmaps and simple bitwise-AND and shift operations. The size of the two-dimensional state transition table in our proposed architecture is proportional to the number of patterns, as opposed to the total length of patterns in previous designs. Our proposed architecture achieves a significant improvement in both throughput performance and memory usage.**

***Index Terms*—Aho–Corasick (AC) algorithm, Bloom filter, deep packet inspection, pattern matching.**

I. Introduction

PATTERN matching has been an important technique in information retrieval and text editing for many years and has recently been applied to signature matching to help detect malicious attacks against networks. In a wider sense, pattern matching searches for occurrences of plain strings and/or regular expressions in an input text string. This paper only considers the matching of plain strings.

Well-known pattern-matching algorithms include Knuth–Morris–Pratt (KMP) [1], Boyer–Moore (BM) [2], Wu–Manber (WM) [13], and Aho–Corasick (AC) [3]. The KMP and BM algorithms are efficient for single-pattern matching, but are not suitable for matching multiple patterns. The WM algorithm is an adaptation of the BM algorithm to multiple patterns. The AC algorithm preprocesses the patterns and builds a finite automaton that can match multiple patterns simultaneously, but may require a huge amount of memory space to do so. A straightforward implementation of the AC algorithm is to construct a two-dimensional state transition table for the finite automaton. Such an implementation requires a prohibitively huge amount of memory space when the total pattern length is large. Several schemes had been proposed to reduce the memory requirement. Some are incorporated in Snort [7], [8], an open-source intrusion detection/prevention application, and ClamAV [9], another open-source anti-virus/worm application. These compression schemes are related to our work and will be reviewed in Section II.

The AC algorithm guarantees linear-time deterministic performance under all circumstances and has thus been 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 transmission speeds in high-speed networks. Previous studies have exploited hardware capacity for massive parallel processing to proposed hardware accelerators for pattern-matching engines [4]–[6], [32], [33], but this lies outside the scope of this paper.

This paper presents a pattern-matching architecture with high throughput performance and low memory requirements. Similar to the WM algorithm and the Hash-AV+ClamAV scheme [30], our proposed architecture consists of a pre-filter and a verification engine. The function of the pre-filter is to query data structures built from patterns to find the starting positions of potential pattern occurrences. Once a suspicious starting position is found, the verification engine confirms true pattern occurrence. In the WM algorithm, the pre-filter 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. The proposed design, however, uses a bit vector, called master bitmap, with simple bitwise-AND and shift operations to accumulate query results. Consequently, our proposed pre-filter is stateful, as opposed to the statelessness in the WM algorithm and the Hash-AV+ClamAV scheme. We prove in this paper that the proposed stateful pre-filter 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. Several other pre-filter designs have been previously proposed [16]–[28]. Similar to the hardware accelerators of pattern-matching engines, these designs used parallel processing to achieve high throughput performance and thus lie outside the scope of this paper.

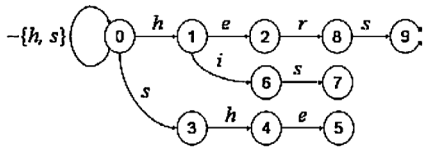
It should be noted that our proposed stateful pre-filter is suitable for patterns of moderate or large lengths. For short patterns, its throughput performance degrades. This is a common

Manuscript received September 24, 2009; revised August 05, 2010; July 25, 2011; February 23, 2012; and July 12, 2012; accepted August 27, 2012; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor I. Keslassy.

The authors are with the Institute of Communication Engineering, National Chiao Tung University, Hsinchu 300, Taiwan (e-mail: tlee@banyan.cm.nctu.edu.tw; nellen.cm93g@nctu.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2012.2224881



(a)

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

(b)

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

(c)

Fig. 1. AC pattern-matching machine for $Y = \{he, she, his, hers\}$ [3]. (a) Goto function. (b) Failure function. (c) Output function.

disadvantage of schemes using pre-filters. The impact of short patterns on our proposed architecture is studied in Section V. Nevertheless, the stateful concept improves throughput with little cost of master bitmap and simple bitwise-AND and shift operations.

The rest of this paper is organized as follows. Section II reviews some related work, including the original AC algorithm. Our proposed pattern-matching architecture is presented in Section III. In Section IV, we prove that our proposed architecture functions correctly and is an optimal design. Experimental results are provided in Section V, and conclusions are drawn in Section VI.

II. Related Work

In this section, we review the AC algorithm, previous compression designs for the AC algorithm, and the WM algorithm. Throughout this paper, we shall use $Y = \{P_1, P_2, \ldots, P_y\}$ to represent pattern set and $T = t_1 t_2 \ldots t_n$ to denote the input text string to be scanned.

A. Aho–Corasick Algorithm

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].

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 example, in Fig. 1, we have $g(0, h) = 1$ and $g(1, \sigma) = fail$ if $\sigma \neq e$ or i. State 0 is a special state that never results in the $fail$ message, i.e., $g(0, \sigma) \neq fail$ for all $\sigma \in \Sigma$, the alphabet. 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 of patterns (which could be empty). The set $output(S)$ contains pattern P iff P is a suffix of the string representing state S. As an example, we have $output(5) = \{he, she\}$.

Note that there might be a self-loop on the start state of the goto graph. In the following definitions, we ignore the self-loop so that the goto graph can be considered 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 operation of the AC pattern-matching machine is as follows. Let S be the current state and a be the current input symbol. 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 $output(Q) \neq \emptyset$ (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. The property $g(0, \sigma) \neq fail$ for all $\sigma \in \Sigma$ guarantees that one input symbol is processed by the pattern-matching machine in every operation cycle.

A straightforward implementation of the goto function uses a two-dimensional table to look up the next state. However, such an implementation requires a huge amount of memory space when the total length of patterns is large. Some compression schemes are reviewed in the following sections.

B. Bitmap Data Structure

In the bitmap data structure [10], each state S has a $|\Sigma|$-bit bitmap and a state array, where $|\Sigma|$ is the size of Σ. The ith bit of the bitmap is a 1 iff $g(S, i) \neq fail$. The state array stores each *non-fail* $g(S, i)$, sorted by the value of i. As a result, to find $g(S, i)$, one needs to count the total number of 1's in the bitmap of state S 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, and 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.

For certain applications such as anti-virus programs, it is highly likely that the state away from the start state is a single-child state. Therefore, path compression was introduced in [10] to squeeze four single-child states into one state. Memory requirements can be further reduced if the length of the compressed paths is not restricted. In the scheme proposed in [12], a state is eliminated if: 1) it is a single-child state; and 2) there is no incoming failure transition to it. Another effective compression technique proposed in [12], called leaf compression, eliminates leaf states with two modifications: 1) pushing the indication of the match to the penultimate state, and 2) copying the failure transitions of leaf states to the corresponding penultimate states as their new goto transitions.

The first modification is realized by adding one bit for each outgoing goto transition in a state, indicating whether or not it leads to a final state. The second modification reduces the number of transitions taken during the automaton traverse. If both path compression and leaf compression are adopted, then leaf compression adds a match indication bit to each symbol of the corresponding compressed path.

C. Banded-Row Format

In the banded-row format [11], which is used in Snort, the row elements are stored from the first nonzero value (or *non-fail* value in the goto transition table) to the last nonzero 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, referred to as bandwidth, and the second element represents the index (numbered from 0) of the first vector element stored, followed by the band values. The AC pattern-matching machine whose goto transition table is compressed with the banded-row format is referred to as the banded-row format AC.

D. AC-Bnfa

AC-bnfa is another alternative adopted by Snort for pattern matching. For each state, it stores a transition list that contains at least two words. The first word (in the current implementation, only the least significant 24 bits) stores the state number. The second word, called the 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 B) and the corresponding next states (3 B). 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 $C \geq 64$.

E. ClamAV Implementation

ClamAV [9] implementation limits the depth of the goto graph to two and partitions the patterns into groups so that two patterns are in the same group iff they have the same prefix of length two. All patterns in the same group are saved as a linked list associated with a leaf state. Whenever a leaf state is visited, all patterns on its linked list are checked sequentially.

The Hash-AV+ClamAV scheme [30] adds a pre-filter to the ClamAV implementation. The pre-filter, called Hash-AV, is a Bloom filter built with α hash functions. 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 [29]. 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 ClamAV implementation is invoked for verification iff all α query results are positive. Obviously, inserting all β-byte substrings 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 $\gamma = 4$.

F. Wu–Manber Algorithm

The WM algorithm [13] consists of a pre-filter and a verification engine. To construct the pre-filter, only the first m symbols of each pattern are considered, where m is the length of the shortest pattern. Let $p_i^1 p_i^2 \dots p_i^m, 1 \leq i \leq y$, represent the m-symbol prefix of pattern P_i. A *SHIFT* table, a *HASH* table, a *PAT_POINT* list, and a *PREFIX* table are required. The *SHIFT* table is used to determine how many symbols in the text can be safely skipped during scanning. The *HASH* table, *PREFIX* table, and the *PAT_POINT* list are used when verification is needed. The *SHIFT* table is related to our pre-filter design and is described as follows.

Assume that the *SHIFT* table has N entries. A hash function, denoted by $hash$, is required for the construction of the *SHIFT* table. The input of $hash$ is a block of size k symbols, and its output falls in $[0, N-1]$. Initially, set $SHIFT[h] = m - k + 1$ for all $h, 0 \leq h \leq N-1$. Then, change $SHIFT[h]$ to $m - k + 1 - j$ if there exists $i, 1 \leq i \leq y$, such that $hash(p_i^j p_i^{j+1} \dots p_i^{j+k-1}) = h, 1 \leq j \leq m - k + 1$, and $hash\ (p_\varepsilon^{j+l} p_\varepsilon^{j+l+1} \dots p_\varepsilon^{j+l+k-1}) \neq h$ for all $\varepsilon, 1 \leq \varepsilon \leq y$, and all $l, 1 \leq l \leq m - k + 1 - j$.

A search window of length m is used during scanning. Initially, the search window is aligned with the input text string, i.e., the substring contained in the search window is $t_1 t_2 \dots t_m$. During scanning, the last k symbols of the text string contained in the search window are hashed. Let h be the hash result. If $SHIFT[h] \neq 0$, then the search window is advanced by $SHIFT[h]$ positions. In case $SHIFT[h] = 0$, the verification engine is invoked and the candidate patterns are verified sequentially. After verification, the search window is advanced by one position.

III. Proposed Pattern-Matching Architecture

As mentioned before, 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.

A. Pre-Filter Design

As in the WM algorithm, only the first m symbols of each pattern are considered in constructing the pre-filter, where m has to be smaller than or equal to the length of the shortest pattern. To achieve a high degree of system performance, m is normally chosen to be the length of the shortest pattern. Given a block size k, our pre-filter design includes $m-k+1$ membership query modules denoted as $\mathrm{MQ}_1, \mathrm{MQ}_2, \dots,$ and MQ_{m-k+1}. Every membership query module has N bits. Recall that $p_i^1 p_i^2 \dots p_i^m$ is the m-symbol prefix of pattern P_i. The hth bit of $\mathrm{MQ}_j, 1 \leq j \leq m - k + 1$, is set to 1 iff there exists a pattern P_i such that $h = hash(p_i^j p_i^{j+1} \dots p_i^{j+k-1})$. Unlike the WM algorithm, our pre-filter design uses membership query modules rather than the *SHIFT* table. As will be seen later, such a design allows previous query results to be easily accumulated to improve system performance.

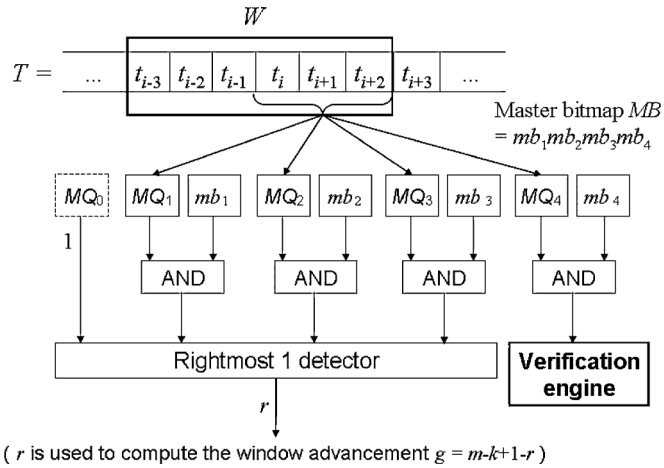


Fig. 2. Stateful pre-filter architecture for $m = 6$ and$k = 3$.

As an example, assume that alphabet $\Sigma = \{0, 1, 2, \ldots, 9\}$ and pattern set $Y = \{P_1, P_2, P_3\}$, where $P_1 = 04648, P_2 = 30692$, and $P_3 = 614621$. Since the length of the shortest pattern is 5, we can choose $m = 5$. Assume that $k = 2$, $N = 100$, and the hash function is the identity mapping. For such a setting, there are $m - k + 1 = 4$ membership query modules. Let $\mathrm{MQ}_i[j], 1 \leq i \leq 4, 0 \leq j \leq 99$, be the content of the jth bit of the ith membership query module. We have $\mathrm{MQ}_1[j] = 1$ iff $j = 04, 30$, or 61; $\mathrm{MQ}_2[j] = 1$ iff $j = 06, 14$, or 46; $\mathrm{MQ}_3[j] = 1$ iff $j = 46, 64$, or 69; and $\mathrm{MQ}_4[j] = 1$ iff $j = 48, 62$, or 92.

Again, a search window W of length m is used during scanning. Initially, W is aligned with the input text string T so that the substring of T contained in W is $t_1 t_2 \ldots t_m$. In general, assume that the substring of T contained in W is $t_{\varepsilon+1} t_{\varepsilon+2} \ldots t_{\varepsilon+m}$. The k-symbol suffix, i.e., $t_{\varepsilon+m-k+1} t_{\varepsilon+m-k+2} \ldots t_{\varepsilon+m}$, is used to query the membership query modules. Let $qb_i, 1 \leq i \leq m - k + 1$, be the report of MQ_i and $QB = qb_1 qb_2 \ldots qb_{m-k+1}$ denote the bitmap of the current query result. Note that $qb_i = 1$ iff $\mathrm{MQ}_i[h]$ was set to 1, where $h = hash\,(t_{\varepsilon+m-k+1} t_{\varepsilon+m-k+2} \ldots t_{\varepsilon+m})$. We use a master bitmap of size $m - k + 1$ bits to accumulate the previous query results and act as the state of the pre-filter. Let $MB = mb_1 mb_2 \ldots mb_{m-k+1}$ represent the master bitmap. Initially, we set $MB = 1^{m-k+1}$, i.e., $mb_i = 1$ for all $i, 1 \leq i \leq m - k + 1$. After fetching a query result, we perform $MB = MB\&QB$, where & is the bitwise-AND operation. A suspicious substring starting from the first symbol contained in W is found, and the verification engine is invoked if $mb_{m-k+1} = 1$. The search window W is advanced by $m - k + 1$ positions if $mb_i = 0$ for all $i, 1 \leq i \leq m - k$, or $m - k + 1 - r$ positions if $mb_r = 1$ and $mb_i = 0$ for all $i, r < i \leq m - k$. If W is determined to be advanced by g positions, *MB* is right-shifted by g bits and filled with 1's for the holes left by the shift. Fig. 2 shows the pre-filter architecture for $m = 6$ and $k = 3$. A virtual membership query module MQ_0, which always reports a 1, is added to ensure the rightmost 1 detector functions correctly. The correctness of the stateful pre-filter is proved in Section IV.

The pre-filter is stateless without the master bitmap. In this case, only the 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. We provide analytical comparison of stateful and stateless designs in the Appendix. In fact, as shown in Section IV, the proposed implementation using master bitmap and simple bitwise-AND and shift operations is optimal in the sense that it is equivalent to utilizing all previous query results.

We use an example to explain the operation of the stateful pre-filter. Consider the membership query modules constructed above and assume that $T = 23764614621$. Initially, W contains 23764 and MB is 1111. The substring 64 is used for the first query, and the query result $QB = 0010$. Since $MB\&QB = 0010$, W is advanced by one position, and MB becomes 1001. In the second iteration, the substring contained in W is 37646, and the query result is $QB = 0110$. Since $MB\&QB = 0000$, the search window W is advanced by four positions, and MB is updated as 1111. In the third iteration, the substring 62 is used as the query, and the result is $QB = 0001$. Since $MB\&QB = 0001$, a suspicious substring starting from the first symbol contained in W is found. Therefore, the verification engine is invoked, and the pattern $P_3 = 614621$ is detected. After the verification, the search window is advanced by four positions. Since the length of the remaining input text string, i.e., 21, is smaller than m, the scanning process ends.

Note that, for the above example, the query result in the first iteration, i.e., $QB = 0010$, indicates that it is impossible to find a pattern occurrence starting from the third or the fourth symbol contained in W because $qb_1 = qb_2 = 0$. In other words, after W is advanced by one position, neither the second nor the third symbol contained in W can be the starting symbol of a suspicious substring. Therefore, although the query result has $qb_2 = qb_3 = 1$ in the second iteration, it is safe to advance W by four positions. We use the master bitmap MB to accumulate previous query results and carry them from previous iterations to the current one. Without MB (which becomes stateless), the search window can only advance by one position in the second iteration.

In general, performing multiple queries in each iteration can reduce false positive probability and increase window advancement. However, this requires more processing time than is needed to perform a single query. Assume that in each iteration 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 $QB_i, 1 \leq i \leq L$ represent the bitmap reported from the ith query and $QB = QB_1 \& QB_2 \& \ldots \& QB_L$. The master bitmap is updated as $MB = MB\&QB$. The window advancement and the presence of a suspicious substring are determined according to the value of MB in the same way as in the single-query case. The optimal value of L that maximizes throughput performance will be analyzed in the Appendix.

B. Verification Engine Design

The verification engine is designed based on the AC algorithm so that all candidate patterns can be verified simultaneously. The use of the pre-filter requires modification to the AC algorithm. The first modification, which concerns the goto function, is to delete the self-loop, if exists, at the start state

because 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 substring 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 nonempty 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 multiple times if uP_j is a substring of T.

Since the failure function is not necessary, only the goto function and the output function need to be stored. The output function is simplified because at most one pattern is matched in each state. Similar to the scheme proposed in [12], we adopt the concept of variable-length path compression to reduce the memory requirements of the goto function. However, since there is no restriction caused by the failure function, our proposed architecture can yield better compression results than the scheme proposed in [12].

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 design data structures for 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 a branch state, we use the banded-row format, which allows fast random access without imposing a large memory requirement, to store its goto transition vector. As a result, we still have a two-dimensional state transition table. The resulting state transition table is named the Branch State Transition (BST) table. Note that the number of rows in the BST table is only equal to the number of branch states, which is at most $y - 1$ for y patterns.

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 (the concatenation of u and v), where v is a nonempty 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 the 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 respectively represent 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. Note that the data structure of state S does not contain S.NDES because one can always set the state number of S.NDES as one plus that of S.

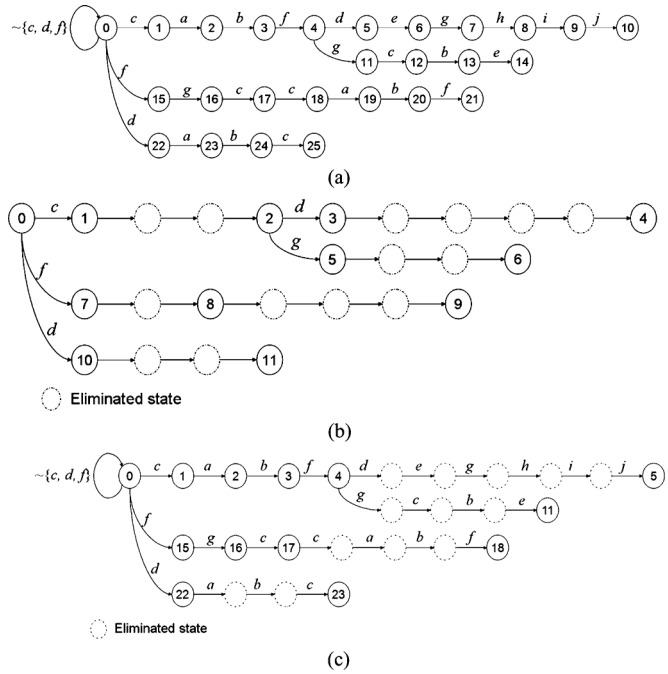


Fig. 3. Goto graph for $\Sigma = \{a, b, c, d, e, f, g, h, i, j\}$ and $Y = \{cabf, cabfdeghij, cabfgcbe, fgc, fgccabf, dabc\}$. (a) Goto graph of the original AC algorithm. (b) Compressed goto graph in our proposed architecture. (c) Path-compressed goto graph of the scheme proposed in [12].

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. Similar to leaf compression [12], our design significantly reduces memory requirements for leaf states. Our design needs two bits for every explicit state to indicate its type (branch, single-child, or leaf) and another bit to indicate whether or not a match is found in the state. Except for the matched pattern (which is needed for all schemes), these three bits are the only information required for a leaf state. As for leaf compression, one bit is added to every goto transition or to every symbol of the corresponding compressed path if path compression is adopted. For a large pattern set, we expect the number of explicit states to be much smaller than the number of symbols of all patterns.

As an example, assume that alphabet $\Sigma = \{a, b, c, d, e, f, g, h, i, j\}$ and pattern set $Y = \{cabf, cabfdeghij, cabfgcbe, fgc, fgccabf, dabcs\}$. The corresponding goto graph of the original AC algorithm is shown in Fig. 3(a). Fig. 3(b) shows our compressed goto graph. Note that the states on the compressed goto graph are numbered so that the state number of S.NDES for explicit single-child state S is $S + 1$. Compared to Fig. 3(a), the number of states is reduced from 26 to 12. The single-child states 1, 3, 5, 7, and 10 are first single-child states and thus remain on the goto graph. Note that it is possible to eliminate those first single-child states if the label on each outgoing goto transition of a branch state is allowed to be a string. However, by doing so, the state transitions of a branch state become complicated, and system performance is degraded. Therefore, we do not adopt this strategy. Fig. 3(c) shows the path-compressed goto graph of the scheme proposed in [12]. Because of the

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

(a)

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

(b)

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

(c)

cabfcabfdeghijcabfgcbefgcfgccabfdabc

(d)

Fig. 4. Verification engine data structures for $\Sigma = \{a, b, c, d, e, f, g, h, i, j\}$ and $Y = \{cabf, cabfdeghij, cabfgcbe, fgc, fgccabf, dabc\}$. (a) BST table. (b) Data structure for first single-child states and single-child final states. (c) Output function. (d) *Compacted_Patterns*.

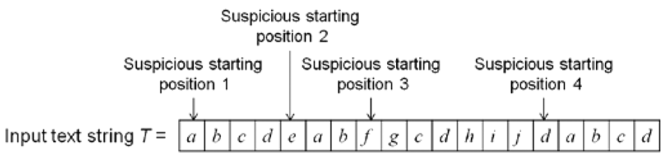


Fig. 5. Input text string and suspicious starting positions.

incoming failure transitions, the single-child states 1, 2, 3, 15, 16, 17, and 22 in Fig. 3(c) cannot be removed.

In Fig. 3(b), 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. 4 shows the data structures of our verification engine for this example. Assume the symbols in Σ are sequentially encoded from 0 to 9. The vector representing goto transitions for state 0 is ($fail$ $fail$ 1 10 $fail$ 7 $fail$ $fail$ $fail$ $fail$) and is stored as (4 2 1 10 $fail$ 7). Similarly, the goto transition vector for state 2 is ($fail$ $fail$ $fail$ 3 $fail$ $fail$ 5 $fail$ $fail$ $fail$) and is stored as (4 3 3 $fail$ $fail$ 5). 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 that contains fgc as a prefix, we store S.position $= 24$ and S.distance $= 2$. If S is state 8, we store S.position $= 29$ and S.distance $= 4$.

To explain the operation of the proposed verification engine, let us consider the above example with $\Sigma = \{a, b, c, d, e, f, g, h, i, j\}$ and $Y = \{cabf, cabfdeghij, cabfgcbe, fgc, fgccabf, dabc\}$. The data structures are shown in Fig. 4. The input text string T and the suspicious starting positions identified by the pre-filter are illustrated in Fig. 5. When the verification engine is invoked, the verification procedure starts to traverse the compressed goto graph from state 0. The verification procedures stops once the goto transition fails or the symbols of T are exhausted.

Consider the suspicious starting position 1. Since state 0 is a branch state, the verification engine consults the BST table and knows that the first symbol in the suspicious substring, i.e., a, is outside the band of state 0, which implies $g(0, a) = fail$. Therefore, the suspicious substring is a false positive and the verification procedure stops. Next, consider the suspicious starting position 2. Again, the verification procedure starts by consulting the BST table. This time, the engine finds that the first symbol in the suspicious substring, i.e., e, is inside the band of state 0. However, the band values indicate that $g(0, e) = fail$. Thus, this suspicious substring is also a false positive, and the verification procedure does not need to continue. Now, let us consider the suspicious starting position 3. The BST table indicates that the first symbol in the suspicious substring, i.e., f, is inside the band of state 0 and $g(0, f) = 7$. Consequently, the engine moves the current state from state 0 to state 7. Since state 7 is a first single-child state and its S.position and S.distance are respectively 24 and 2, the engine checks if the following substring in T of length two is the same as the two-symbol substring starting from the 24th position of *Compacted_Patterns*. The checked result is true, therefore the engine updates the current state by increasing the state number by one to state 8. Since $output(8) = \{3\}$, Pattern 3, i.e., fgc , is detected. The verification procedure does not stop here. State 8 is a single-child final state, and its S.position and S.distance are respectively 29 and 4. Thus, the engine compares the substring $dhij$ in T, and the substring $cabf$ in *Compacted_Patterns*. The substrings are different, which implies the goto transition fails, therefore the verification procedure stops. Finally, consider the suspicious starting position 4. As in the previous case, the current state is moved from state 0 to state 10, and then from state 10 to state 11. State 11 is a leaf state, which implies: 1) some pattern is detected here; and 2) the goto transition will fail, and therefore the verification procedure stops. Since we have $output(11) = \{5\}$, we know that Pattern 5, i.e., $dabc$, is detected.

C. Time Complexity

The number of memory accesses required by each type of state is analyzed as follows. Assume that A bytes are fetched in a memory access. For a branch state, to process one byte, we need $\lceil 4/A \rceil$ memory access to obtain bandwidth (2 B), the index of the first vector element stored (1 B), state type (2 bits), and the final state indication (1 bit). In addition, $\lceil 3/A \rceil$ memory access is required to get a band value (3 B) if the input symbol is within the band. If the state is a final state, another $\lceil 2/A \rceil$ memory access is performed for the identification of the matched pattern (2 B). Therefore, a branch state requires at least $\lceil 4/A \rceil$ and at most $\lceil 4/A \rceil + \lceil 3/A \rceil + \lceil 2/A \rceil$ memory accesses. For an explicit single-child state S, S.distance bytes are processed with at most $\lceil 5/A \rceil + \lceil S.\text{distance}/A \rceil + \lceil 2/A \rceil$ memory accesses. More specifically, we need $\lceil 5/A \rceil$ to obtain S.position (3 B), S.distance (1 B), state type (2 bits), and the final state indication (1 bit), up to $\lceil S.\text{distance}/A \rceil$ to reach S.NEDS, and $\lceil 2/A \rceil$ for the matched pattern if state S is a final state. For a leaf state, we need $\lceil 3/A \rceil$ memory access to obtain the state type and the identification of the matched pattern. Since the operations are quite simple, the proposed architecture is also suitable for hardware implementation.

IV. Correctness and Optimality Proofs of the Stateful Pre-Filter

In this section, we prove the correctness and optimality of the proposed stateful pre-filter.

A. Proof of Correctness

Consider the Kth iteration. Let $W = t_{i+1}t_{i+2}\dots t_{i+m}$ and $MB = mb_1mb_2\dots mb_{m-k+1}$ respectively be the contents of the search window and the master bitmap in the beginning of the considered iteration. Further let $QB = qb_1qb_2\dots qb_{m-k+1}$ be the query result and $MB' = MB\&QB = mb'_1mb'_2\dots mb'_{m-k+1}$ be the updated master bitmap. We prove that the proposed stateful pre-filter does not miss any pattern occurrence by showing the following claim. Note that given an $\varepsilon, 1 \le \varepsilon \le m-k+1$, and any $s, 0 \le s \le \varepsilon - 1$, if $hash(t_{i+m-k-s+1}\dots t_{i+m-s}) \neq hash(p_j^{\varepsilon-s}\dots p_j^{k+\varepsilon-s-1})$ for all patterns P_j, then there is no pattern occurrence starting from $t_{i+m-k-\varepsilon+2}$. Therefore, it is safe to advance the search window by $g = m-k+1-r$ positions if $mb'_\varepsilon = 0$ for all $\varepsilon, r+1 \le \varepsilon \le m-k+1$ (without verification) or $mb'_\varepsilon = 0$ for all $\varepsilon, r+1 \le \varepsilon \le m-k$ and $mb'_{m-k+1} = 1$ (after verification).

Claim: If $mb'_\varepsilon = 0, 1 \le \varepsilon \le m-k+1$, then it holds that there exists some $s, 0 \le s \le \varepsilon - 1$, such that $hash(t_{i+m-k-s+1}\dots t_{i+m-s}) \neq hash(p_j^{\varepsilon-s}\dots p_j^{k+\varepsilon-s-1})$ for all patterns P_j.

According to the operation of the proposed stateful pre-filter, $mb_{m-k+1} = 1$ is always true, thus $mb'_{m-k+1} = 0$ implies $qb_{m-k+1} = 0$. Therefore, the claim is true for $\varepsilon = m-k+1$ because $s = 0$ meets the requirement. We prove the case $1 \le \varepsilon < m-k+1$ by mathematical induction.

For $K = 1$, i.e., the first iteration, it is true that $mb'_\varepsilon = 1$ iff $qb_\varepsilon = 1$. In other words, we have $mb'_\varepsilon = 0$ iff $hash(t_{i+m-k+1}\dots t_{i+m}) \neq hash(p_j^{\varepsilon}\dots p_j^{k+\varepsilon-1})$ for all patterns P_j. Therefore, for the first iteration, the claim is true by choosing $s = 0$. Assume that it is true for $K = J$ and consider the $(J+1)$th iteration.

The notation needs to be clarified since two iterations are considered simultaneously. Let $W = t_{i+1}t_{i+2}\dots t_{i+m}, MB = mb_1mb_2\dots mb_{m-k+1}, QB = qb_1qb_2\dots qb_{m-k+1}$, and $MB' = MB\&QB = mb'_1mb'_2\dots mb'_{m-k+1}$ respectively be the search window content, the master bitmap, the query results, and the updated master bitmap of the Jth iteration. Similarly, let $\bar{W}, \overline{MB}, \overline{QB}$, and $MB'' = \overline{MB}\&\overline{QB}$ be those of the $(J+1)$th iteration. Assume that the search window is advanced by g positions in the Jth iteration. If $g = m-k+1$, we have $\overline{MB} = 1^{m-k+1}$, and the claim is true for the $(J+1)$th iteration by choosing $s = 0$. Consider the case $g < m-k+1$. After advancing the search window and updating the master bitmap, we have $\bar{W} = t_{i+g+1}t_{i+g+2}\dots t_{i+g+m}$ and $\overline{MB} = 1^g mb'_1\dots mb'_{m-k-g}1$.

Let $MB'' = \overline{MB}\&\overline{QB} = mb''_1mb''_2\dots mb''_{m-k+1}$ and assume that $mb''_\varepsilon = 0$ for some $\varepsilon, 1 \le \varepsilon < m-k+1$. If $1 \le \varepsilon \le g$, then it must hold that $\overline{qb_\varepsilon} = 0$, and therefore the claim is true by choosing $s = 0$. Assume that $g < \varepsilon < m-k+1$. Since $mb''_\varepsilon = \overline{mb_\varepsilon}\&\overline{qb_\varepsilon}$, we have $\overline{qb_\varepsilon} = 0$ or $\overline{mb_\varepsilon} = 0$. The claim is true for the $(J+1)$th iteration by choosing $s = 0$ if $\overline{qb_\varepsilon} = 0$. Assume that $\overline{qb_\varepsilon} = 1$, which implies $\overline{mb_\varepsilon} = 0$. Since $\overline{mb_\varepsilon} = mb'_{\varepsilon-g}$, we know by hypothesis that there exists an $s, 0 \le s \le \varepsilon - g - 1$, such that $hash(t_{i+m-k-s+1}\dots t_{i+m-s}) \neq hash\ (p_j^{\varepsilon-g-s}\dots p_j^{k+\varepsilon-g-s-1})$ for all patterns P_j. Let $i' = i+g$ and $s' = s+g$. We conclude that there exists an $s', g \le s' \le \varepsilon - 1$, such that $hash(t_{i'+m-k-s'+1}\dots t_{i'+m-s'}) \neq hash\ (p_j^{\varepsilon-s'}\dots p_j^{k+\varepsilon-s'-1})$ for all patterns P_j. Therefore, the claim is also true for the $(J+1)$th iteration for $g < m-k+1$. This completes the correctness proof of the proposed pre-filter.

B. Proof of Optimality

We now prove that the implementation with the master bitmap is optimal in the sense that it is equivalent to using all previous query results. For convenience, we call the scheme implemented with master bitmap Scheme A and the one using all previous query results Scheme B. We assume that Scheme B, as Scheme A, advances the search window for as many positions as possible in each iteration. The two schemes are equivalent if, in each iteration: 1) they advance the search window by the same number of positions; and 2) if one invokes the verification engine, the other as well. We prove the equivalence of Schemes A and B by mathematical induction on iteration number.

Consider the Kth iteration. Let $W = t_{i+1}t_{i+2}\dots t_{i+m}$, $MB = mb_1mb_2\dots mb_{m-k+1}, QB = qb_1qb_2\dots qb_{m-k+1}$, and $MB' = MB\&QB = mb'_1mb'_2\dots mb'_{m-k+1}$respectively be the search window content, the master bitmap, the query result, and the updated master bitmap of Scheme A. The symbol $t_{i+\delta}, 1 \le \delta \le m$, is said to be a possible starting symbol of pattern occurrence for Scheme B in the beginning of the Kth iteration iff it cannot be excluded based on the results of the first $(K-1)$ queries. We shall show that $mb_\varepsilon = 1$ for Scheme A iff $t_{i+m-k-\varepsilon+2}$ is a possible starting symbol of pattern occurrence of Scheme B in the beginning of the Kth iteration (Condition 1) and the condition for both schemes to invoke the verification engine is $qb_{m-k+1} = 1$ (Condition 2). Note that $mb'_\varepsilon = 1$ (for Scheme A) iff $mb_\varepsilon = 1$ (for Scheme A) and $qb_\varepsilon = 1$, and $qb_\varepsilon = 1$ iff $t_{i+m-k-\varepsilon+2}$ is considered a possible starting symbol of pattern occurrence based solely on the Kth query result (for Scheme B). Therefore, Condition 1 implies both schemes advance the search window by the same number of positions (after verification, if needed) in the Kth iteration.

For $K = 1$, we have $W = t_1t_2\dots t_m$ and $MB = 1^{m-k+1}$. Since no query was performed prior to the first iteration, every symbol contained in the search window is a possible starting symbol of pattern occurrence of Scheme B in the beginning of the first iteration. Therefore, Condition 1 is true. Moreover, Scheme A invokes the verification engine iff $mb'_{m-k+1} = 1$. Since $mb_{m-k+1} = 1$, the condition is identical to $qb_{m-k+1} = 1$, which implies $hash(t_{m-k+1}\dots t_m) = hash\ (p_j^{m-k+1}\dots p_j^m)$ for some pattern P_j, which in turn implies Scheme B has to invoke the verification engine to check for potential pattern occurrence starting from t_1. Therefore, Condition 2 is also true and the two schemes are equivalent for the first iteration.

Assume that the two schemes are equivalent for the Jth iteration and consider the $(J+1)$th iteration. Let $W = t_{i+1}t_{i+2}\dots t_{i+m}, MB = mb_1mb_2\dots mb_{m-k+1}$, $QB = qb_1qb_2\dots qb_{m-k+1}$, and $MB' = MB\&QB =$

$mb'_1 mb'_2 \ldots mb'_{m-k+1}$ be the parameters of the Jth iteration and $\bar{W} = \overline{t_{i+1}t_{i+2}} \ldots \overline{t_{i+m}}, \overline{MB} = \overline{mb_1 mb_2 \ldots mb_{m-k+1}}, \overline{QB} = \overline{qb_1 qb_2 \ldots qb_{m-k+1}}$, and $MB'' = \overline{MB} \& \overline{QB} = mb''_1 mb''_2 \ldots mb''_{m-k+1}$ be those of the $(J+1)$th iteration for Scheme A. Assume that the search window is advanced by g positions in the Jth iteration. If $g = m-k+1$, we have $\bar{W} = t_{i+m-k+2} t_{i+m-k+3} \ldots t_{i+2m-k+1}$. The last $k-1$ symbols contained in W, i.e., $t_{i+m-k+2}, t_{i+m-k+3}, \ldots$, and t_{i+m}, can never be excluded as possible starting symbols of pattern occurrences based on the first J queries, and the symbols $t_{i+m+1}, t_{i+m+2}, \ldots, t_{i+2m-k+1}$ are newly contained by the search window and cannot be excluded based on the first J queries either. Thus, we conclude that all symbols contained in $\bar{W}$ are possible starting symbols of pattern occurrences for Scheme B in the beginning of the $(J+1)$th iteration. Note that we also have $\overline{MB} = 1^{m-k+1}$ for Scheme A. Therefore, the $(J+1)$th iteration is just like the first iteration. Consequently, the two schemes are equivalent for the $(J+1)$th iteration. Assume that $1 \le g < m-k+1$. In this case, we have $\bar{W} = \overline{t_{i+1}t_{i+2}} \ldots \overline{t_{i+m}} = t_{i+g+1} t_{i+g+2} \ldots t_{i+g+m}$ and $\overline{MB} = 1^g mb'_1 \ldots mb'_{m-k-g} 1$. Again, the last $k-1$ symbols contained in W, i.e., $t_{i+m-k+2}, t_{i+m-k+3}, \ldots$, and t_{i+m}, can never be excluded as possible starting symbols of pattern occurrences for Scheme B based on the first J queries, and the symbols $t_{i+m+1}, t_{i+m+2}, \ldots, t_{i+m+g}$ are newly contained by the search window and cannot be excluded based on the first J queries either. Therefore, the symbols $t_{i+m-k+2}, t_{i+m-k+3}, \ldots$, and $t_{i+m-k+1+g}$ cannot be excluded in the beginning of the $(J+1)$th iteration. In other words, $\overline{t_{i+m-k-\varepsilon+2}}, 1 \le \varepsilon \le g$, is a possible starting symbol for pattern occurrence of Scheme B in the beginning of the $(J+1)$th iteration. This corresponds to $\overline{mb_\varepsilon} = 1$ for all $\varepsilon, 1 \le \varepsilon \le g$. Consider the bit $\overline{mb_\varepsilon}, g+1 \le \varepsilon \le m-k+1$. According to the proof of the claim in Section IV-A, if $\overline{mb_\varepsilon} = \overline{mb'_{\varepsilon-g}} = 0$, then the symbol $t_{i+m-k-(\varepsilon-g)+2} = \overline{t_{i+m-k-\varepsilon+2}}$ can be excluded as a possible starting symbol of pattern occurrence for Scheme B. Assume that $\overline{mb_\varepsilon} = 1$. Since $\overline{mb_\varepsilon} = \overline{mb'_{\varepsilon-g}}$, we have $mb_{\varepsilon-g} = qb_{\varepsilon-g} = 1$. By hypothesis, $mb_{\varepsilon-g} = 1$ implies $t_{i+m-k-(\varepsilon-g)+2}$ is a possible starting symbol of pattern occurrence of Scheme B in the beginning of the Jth iteration. This fact, together with $qb_{\varepsilon-g} = 1$, implies that $t_{i+m-k-(\varepsilon-g)+2} = \overline{t_{i+m-k-\varepsilon+2}}$ remains a possible starting symbol for pattern occurrence of Scheme B in the beginning of the $(J+1)$th iteration. Therefore, Condition 1 is true for the $(J+1)$th iteration.

Finally, assume that $mb''_{m-k+1} = 1$, which is the condition for Scheme A to invoke the verification engine. In this case, it holds that $\overline{qb_{m-k+1}} = 1$, which implies that Scheme B cannot exclude $\overline{t_{i+1}}$ as a potential starting symbol of pattern occurrence. Hence, it will invoke the verification engine to check pattern occurrence starting from $\overline{t_{i+1}}$. Consequently, Condition 2 is also true for the $(J+1)$th iteration. This completes the proof of equivalence for Schemes A and B.

V. Experimental Results

In this section, we compare the performances of the investigated pattern-matching schemes. All schemes are implemented in C++. For convenience, we name our proposed scheme Pre-filter+AC. To study the impact of short patterns to Pre-filter+AC, we conduct simulations for small values of m.

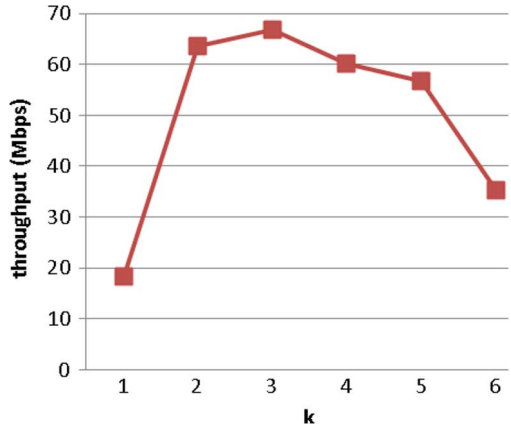


Fig. 6. Throughput performance of the Pre-filter+AC scheme for different values of block size k.

A. Simulation Settings

The experiments are conducted on a PC with an Intel Pentium 4 CPU operating at 2.80 GHz with 512 MB of RAM, 8 kB L1 data cache, and 512 kB L2 cache. The entire ClamAV pattern set is used, containing 29 179 string signatures. The minimum, maximum, average, and total lengths of the signatures are 10, 210, 66.43, and 1 938 433 B, respectively. The total number of states generated by the AC algorithm is 1 844 895. Since the shortest pattern is 10 B, we set the search window length $m = 10$. We concatenated executable files and script files to form the input text since malware can appear in the form of an executable or script.

B. Numerical Results

For the entire ClamAV pattern 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 pattern set is changed. Fig. 6 shows the throughput performance of Pre-filter+AC for different values of block size k. Note that for $k = 1$, the false positive probability of pre-filter is large, implying the verification engine is frequently invoked, reducing system throughput. The average window advancement tends to decrease for large values of k. According to our experimental results $k = 2, 3, 4$, and 5 are good choices to achieve high system throughput, with $k = 3$ being optimal. Thus, $k = 3$ is used in the following experiments.

The size N (in bits) of a membership query module in the proposed Pre-filter+AC scheme (which is also the number of entries in the *SHIFT* table of the WM algorithm) is 2^{16}. The resulting false positive probability is approximately 0.359. To ensure a fair comparison, we set the pre-filter size of the Hash-AV+ClamAV scheme at 2^{19} bits since it uses only one membership query module. Recall that to advance the search window by four positions each time to improve system performance, the Hash-AV+ClamAV scheme inserts all β-byte substrings starting at the first four offsets of all signatures into its membership query module. As a result, signatures

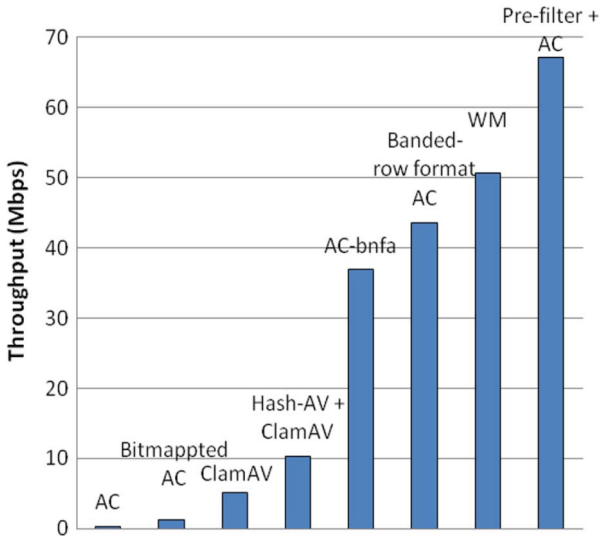


Fig. 7. Throughput comparison.

TABLE I
MEMORY REQUIREMENT COMPARISON

Schemes	AC	Bitmapped AC	ClamAV	Hash-AV + ClamAV
Memory requirement (MB)	1428.24	88.61	2.12	2.18
Schemes	AC-bnfa	Banded-row format AC	WM	Pre-filter +AC
Memory requirement (MB)	22.17	24.41	2.25	4.58

shorter than $\beta + 3$ bytes appearing in the input text may not be detected. For $\beta = 9$ (the value suggested by the authors of [30]), the scheme requires a "two-scan" approach. That is, Hash-AV+ClamAV is first performed for signatures longer than or equal to 12 B, and ClamAV is then executed for the rest of the signatures. We omit the second scan by removing signatures shorter than $\beta + 3$ bytes for the Hash-AV+ClamAV scheme in performance comparison.

We compare the basic version of the proposed Pre-filter+AC, i.e., one query per iteration, with various pattern-matching schemes from the literature. A simple hash function is adopted for our proposed pre-filter. Let $t_i t_{i+1} t_{i+2}$ be the data block to be hashed. The hash function generates $t_i t_{i+2}$ as the hash result. Fig. 7 and Table I show the comparisons of throughput and memory requirements.

Our proposed Pre-filter+AC scheme requires less than 4.6 MB, including 0.06 MB for pre-filter, 1.94 MB for patterns, and 2.58 MB for the data structures of the modified AC automaton. In addition to low memory requirements, the proposed Pre-filter+AC scheme yields higher throughput than all the other schemes. The comparisons are discussed in the following.

C. Comparison to AC-Based Implementations

The AC-based implementations include the original AC algorithm, bimapped AC, AC-bnfa, and banded-row format AC. As expected, the data structure of the original AC algorithm uses an extremely huge amount of memory space for the entire ClamAV pattern set. This results in low throughput because of cache misses. Among the other schemes, the bitmapped AC requires the most memory space because every state needs a bitmap of 256 bits or 32 B. It was reported that the path compression technique can achieve a 2.54:1 compression ratio [10], but even with path compression, its memory requirements are still larger than those of the other schemes. The bitmapped AC also yields lower throughput performance than the other schemes because it needs to compute the population count in a 32-bit bitmap.

The banded-row format AC and the AC-bnfa have relatively high throughput. Banded-row format AC actually outperforms AC-bnfa because it allows fast random access, whereas AC-bnfa requires linear or binary search to perform state transition. The properties of fast random access and efficient memory reduction are the major reasons for us to adopt the banded-row format in our design.

Note that our proposed Pre-filter+AC scheme requires much less memory space than the banded-row format AC because the size of the two-dimensional state transition table in Pre-filter+AC is proportional to the number of patterns rather than the total length of patterns as in banded-row format AC.

D. Comparison to ClamAV-Based Implementations

The ClamAV implementation stores a partial AC trie of depth two. As a result, it requires the least amount of memory space among the investigated schemes. However, it has to sequentially check all patterns associated with a leaf state whenever it is visited, which could be time-consuming. In addition, no symbols in the input text are consumed during the checking procedure if the checking fails. Thus, the throughput performance of ClamAV is unsatisfactory. With the use of a pre-filter, the Hash-AV+ClamAV scheme improves throughput performance of the original ClamAV implementation.

E. Comparison to WM Algorithm

Experiments on the WM algorithm found the optimal block size is $k = 4$, which was used for the WM algorithm in performance comparison. As shown in Fig. 7 and Table I, the proposed Pre-filter+AC scheme and the WM algorithm provide the best throughput performance, and the memory requirements of both are acceptable.

To evaluate performance under various fractions of malicious traffic, we replaced some content of a 2.2-MB Windows executable with 10, 100, or 1000 signatures. If the average size of a malicious program is 1 kB, then the respective fraction of malicious traffic for 10, 100, or 1000 signatures is about 0.45%, 4.5%, or 45%. Fig. 8 shows the experimental results. The throughput of the Pre-filter+AC scheme is seen to decrease as the number of signatures in the input text increases. However, it still outperforms the WM algorithm.

Table II compares the memory requirements for different values of N. Recall that N is the size (in bits) of a membership query module in the Pre-filter+AC scheme and is also the number of entries in the *SHIFT* table of the WM algorithm. 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 of the Pre-filter+AC scheme is 59.5% of that of the WM algorithm. The percentage decreases as N increases. This

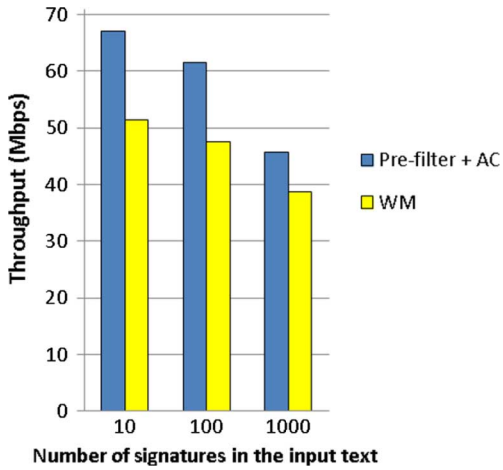


Fig. 8. Throughput for scanning files with different numbers of signatures.

TABLE II
MEMORY REQUIREMENT COMPARISON FOR DIFFERENT VALUES OF N. IN THIS TABLE, N IS THE SIZE (IN BITS) OF A MEMBERSHIP QUERY MODULE IN THE PRE-FILTER+AC SCHEME AND IS ALSO THE NUMBER OF ENTRIES IN THE *SHIFT* TABLE OF THE WM ALGORITHM

Memory requirement (MB)		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

is because the memory requirement of the verification engine in the Pre-filter+AC scheme is not influenced by the value of N, while that in the WM algorithm is. The size of the *HASH* table used in the verification engine of the WM algorithm increases as N increases. Both schemes need to store the pattern set. The Pre-filter+AC scheme uses the *Compacted_Patterns* file, which requires 1.94 MB. The WM algorithm needs slightly larger memory space because an ending symbol is required for each pattern. The pre-filters of both schemes take 0.06, 0.26, 1.04, and 4.19 MB of memory for $N = 2^{16}, 2^{18}, 2^{20}$, and 2^{22}, respectively. The verification engine of the Pre-filter+AC scheme requires 2.58 MB, independent of N. For the WM algorithm, the verification engines respectively takes 0.22, 0.61, 2.19, and 8.48 MB of memory for $N = 2^{16}, 2^{18}, 2^{20}$, and 2^{22}.

Fig. 9 shows the throughput comparison of the Pre-filter+AC scheme and the WM algorithm for different values of N. The input text is a 2.2-MB Windows executable. The Pre-filter+AC scheme has higher throughput than the WM algorithm because: 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.

F. Impact of Short Patterns

To study the impact of short patterns on Pre-filter+AC, we conduct simulations for small values of m. The value of k is chosen to be the optimal one for each m. The input text is still a 2.2-MB Windows executable. As shown in Fig. 10, the throughput performance reduces as the value of m decreases. In other words, pre-filtering does not help (and possibly even hurts)

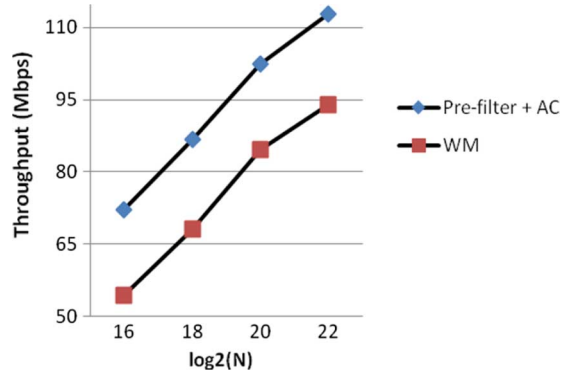


Fig. 9. Throughput comparison for different values of N.

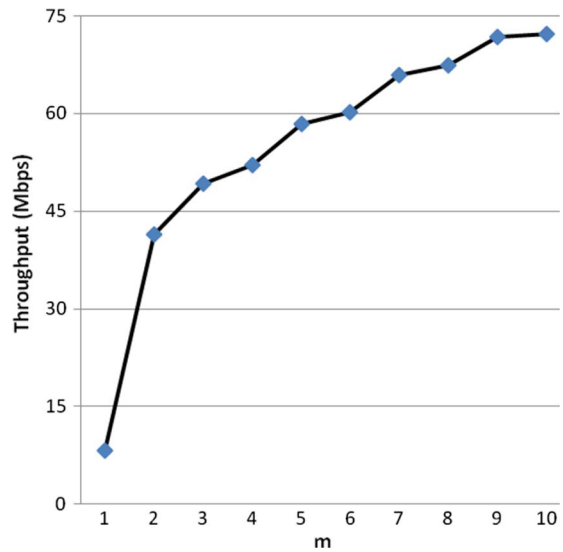


Fig. 10. Impact of m on the throughput performance of the Pre-filter+AC scheme.

system performance if there are short patterns. This is a common drawback of schemes using pre-filters. One possible remedy is to utilize the verification result to help advance the pre-filter. How to efficiently combine the operations of the pre-filter and verification engines remains to be studied.

VI. CONCLUSION

In this paper, we propose a pattern-matching architecture that achieves high throughput with low memory requirements. In the architecture, we introduce the stateful pre-filter concept and present an AC-based verification engine that can check all candidate patterns simultaneously. A master bitmap with simple bitwise-AND and shift operations is used to efficiently accumulate previous query results. 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 iteration is also studied. Results show that the proposed pre-filter with the master bitmap (stateful) outperforms both the proposed pre-filter without the master bitmap and the pre-filter of the widely used Wu-Manber algorithm (stateless). Moreover, our proposed schemes are compared to various related works. The stateful architecture performs the best in

terms of both memory requirement and throughput among the schemes that yield satisfactory performance for both metrics. Therefore, for applications that require high throughput performance with memory space constraints, e.g., an embedded security appliance in a high-speed network environment, our proposed stateful architecture is the preferred solution.

Clearly, a larger search window provides better throughput performance. However, the length of the search window is upper-bounded by the length of the shortest pattern. Consequently, to improve performance and reduce false positives, a virus expert should try to avoid short patterns in deriving signatures. Two interesting future research topics would be the implementation and performance comparison of various pattern-matching algorithms on multithread, multicore processors and the analysis of pre-filter performance for general cases.

APPENDIX

In this Appendix, we analytically compare the performances of stateful and stateless pre-filter designs. The stateless designs include the pre-filter presented in the WM algorithm and our proposed pre-filter without the master bitmap. The average window advancement per unit time, which determines achievable throughput, is selected as the performance metric. For simplicity of analysis, we assume that symbols in patterns and input text string are independent and uniformly distributed over the alphabet. Good hash functions, together with (random) window advancement, can make this assumption acceptable to certain degree. Recall that the query result $QB = QB_1 \& QB_2 \& \ldots \& QB_L = qb_1 qb_2 \ldots qb_{m-k+1}$. 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 L queries. The average window advancement, denoted by $\bar{G}_L$, can be evaluated by

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

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

A. 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 L queries is $L\overline{T}_h$, and $\bar{G}_L/(L\overline{T}_h)$ determines the 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 $\bar{G}_L/L$. The optimal value of L that maximizes throughput satisfies

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

In the following, we separately derive $\bar{G}_L$ for the pre-filter in the WM algorithm, the stateless version of the proposed pre-filter, and the stateful pre-filter.

1) Wu–Manber Pre-Filter: 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 pre-filter, except that the window is advanced by only one position if $qb_{m-k+1} = 1$. Therefore, we have (3), shown at the bottom of the page, and $\bar{G}_L$ can be obtained from (1) and (3).

2) Stateless Version of the Proposed Pre-Filter: For the stateless version of our proposed pre-filter, the bit qb_{m-k+1} is not involved in determining window advancement. Consequently, we have (4), shown at the bottom of the page. Similarly, $\bar{G}_L$ can be obtained from (1) and (4). Note that the average window advancement of the stateless version of our pre-filter is greater than that of the WM algorithm. This is because when $qb_{m-k+1} = 1$, the window advancement in the WM algorithm is always one, and it can be greater than one for our proposed pre-filter.

3) Stateful Pre-Filter: For the proposed stateful pre-filter, we use a 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 $mb_1 mb_2 \ldots mb_{m-k}$, the leftmost $m-k$ 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 the patterns and input text string

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

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

are independent and uniformly distributed over the alphabet, the Markov chain is homogeneous.

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

The derivation of $p_{i,j}$ is explained as follows. Let $i = i_{m-k-1} i_{m-k-2} \ldots i_0$ (binary representation), $j = j_{m-k-1} j_{m-k-2} \ldots j_0$, and assume that, in state i, the search window is to be advanced by g positions. Furthermore, let $i' = i'_{m-k-1} i'_{m-k-2} \ldots i'_0$ be the leftmost $m-k$ bits of the right-shifted master bitmap in state i. If $i = 0^{m-k}$, then $g = m - k + 1$ and $i' = 1^{m-k}$. In this case, we have $p_{i,j} = \rho^x (1-\rho)^{m-k-x}$, where x is the number of 1's in $j_{m-k-1} j_{m-k-2} \ldots j_0$. Assume that $g < m - k + 1$. We have $i = i_{m-k-1} \ldots i_g 10^{g-1}$ and $i' = 1^g i_{m-k-1} \ldots i_g$. The state transition probability $p_{i,j} = 0$ if there exists r, $0 \leq r \leq m-k-1$, such that $i'_r = 0$ and $j_r = 1$. Otherwise, we have $p_{i,j} = \rho^{x_1} (1-\rho)^{m-k-x_1-x_2}$, where x_1 is the number of 1's in $j_{m-k-1} j_{m-k-2} \ldots j_0$ and x_2 is the number of 0's in $1^g i_{m-k-1} \ldots i_g$.

B. Overall System Performance

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

$$L = \arg\max_H \{\bar{G}_H / \bar{C}_H\}. \tag{6}$$

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 stateless version of Pre-filter+AC scheme, verification is required when $qb_{m-k+1} = 1$. Note that the two conditions are equivalent. For the proposed stateful Pre-filter+AC scheme, 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

$$\bar{C}_L = L\overline{T}_h + P(qb_{m-k+1} = 1)\overline{T}_v = L\overline{T}_h + \rho\overline{T}_v \tag{7}$$

for the WM algorithm and the stateless and stateful versions of the proposed Pre-filter+AC scheme. Note that the value of $\overline{T}_v$ depends on number of patterns and the verification algorithm. We numerically study the optimal value of L next.

C. 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, the other parameters are fixed as follows:

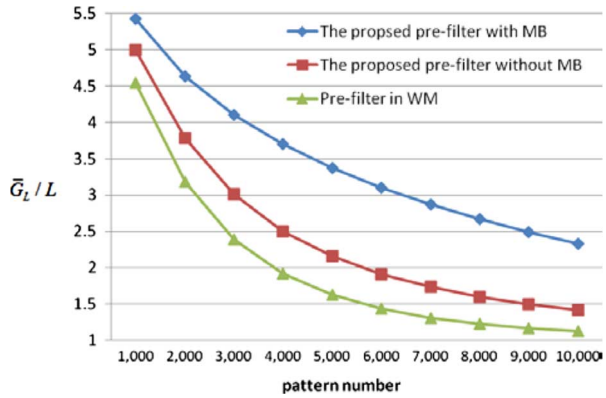


Fig. 11. Pre-filter performance comparison for various pattern numbers.

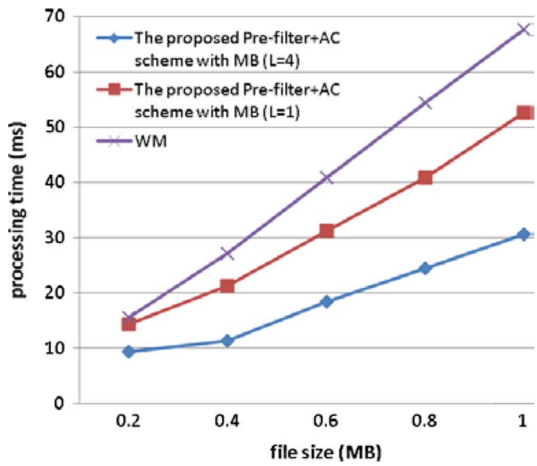


Fig. 12. Processing time comparison for scanning random texts of various sizes.

$m = 10, k = 4, N = 2^{13}$, and $y = 10\,000$. For this scenario, $L = 1$ satisfies (2) that maximizes throughput for the WM pre-filter and the stateless and stateful versions of the proposed pre-filter. Therefore, we choose $L = 1$ as a basis for comparing pre-filter throughput, which, as mentioned before, is proportional to $\bar{G}_L / L$. Fig. 11 shows the results for various pattern numbers.

As noted in Appendix-A.2, the stateless version of the proposed pre-filter is seen to outperform the pre-filter of the WM algorithm, and the proposed stateful pre-filter provides the best performance because all previous query results, in addition to the current one, are used to determine window advancement in the stateful design.

The experiment shows that, given the abovementioned values for m, k, N, and y, $\overline{T}_h = 4.90 \times 10^{-5}$ ms and $\overline{T}_v = 2.55 \times 10^{-4}$ ms for our proposed Pre-filter+AC scheme. From (6), $L = 4$ optimally maximizes system performance of the proposed stateful Pre-filter+AC scheme. To demonstrate the effect of multiple queries in each iteration, we conduct experiments to compare the processing times for the scheme with $L = 1$ and $L = 4$. Fig. 12 shows the results for various file sizes. 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, requiring about 1.3 and 2.2 times the processing time of our proposed stateful Pre-filter+AC scheme with $L = 1$ and $L = 4$, respectively.

REFERENCES

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

[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, pp. 762–772, Oct. 1977.

[3] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, pp. 333–340, Jun. 1975.

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

[5] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proc. 32nd Annu. Int. Symp. Comput. Archit.*, 2005, pp. 112–122.

[6] T. H. Lee and C. C. Liang, "A high-performance memory-efficient pattern matching algorithm and its implementation," in *Proc. IEEE TENCON*, 2006, pp. 1–4.

[7] "Snort," [Online]. Available: http://www.snort.org/

[8] M. Roesch, "Snort—Lightweight intrusion detection for networks," in *Proc. 13th LISA*, Nov. 1999, pp. s229–238.

[9] "Clam AntiVirus (ClamAV)," [Online]. Available: http://www.clamav.net/

[10] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. IEEE INFOCOM*, 2004, vol. 4, pp. 2628–2639.

[11] M. Norton, "Optimizing pattern matching for intrusion detection," Sourcefire, Inc., Columbia, MD, Sep. 2004.

[12] A. Bremler-Barr, Y. Harchol, and D. Hay, "Space-time tradeoffs in software-based deep packet inspection," in *Proc. IEEE HPSR*, 2011, pp. 1–8.

[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," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, May 1970.

[15] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2004.

[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," in *Proc. Field-Program. Custom Comput. Mach.*, Apr. 2004, pp. 322–323.

[18] S. Dharmapurikar and J. Lockwood, "Fast and scalable pattern matching for content filtering," in *Proc. ACM Symp. Archit. Netw. Commun. Syst.*, 2005, pp. 183–192.

[19] S. Dharmapurikar and J. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 10, pp. 1781–1792, Oct. 2006.

[20] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2003, pp. 201–212.

[21] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Trans. Netw.*, vol. 14, no. 2, pp. 397–409, Apr. 2006.

[22] N. S. Artan and H. J. Chao, "Multi-packet signature detection using prefix bloom filters," in *Proc. IEEE GLOBECOM*, 2005, vol. 3, pp. 1811–1816.

[23] N. S. Artan and H. J. Chao, "TriBiCa—Trie bitmap content analyzer for high-speed network intrusion detection," in *Proc. IEEE INFOCOM*, May 2007, pp. 125–133.

[24] N. S. Artan, K. Sinkar, J. Patel, and H. J. Chao, "Aggregated bloom filters for intrusion detection and prevention hardware," in *Proc. IEEE GLOBECOM*, Nov. 2007, pp. 349–354.

[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," in *Proc. IEEE GLOBECOM*, Nov. 2007, pp. 343–348.

[26] N. S. Artan, M. Bando, and H. J. Chao, "Boundary hash for memory-efficient deep packet inspection," in *Proc. IEEE ICC*, May 2008, pp. 1732–1737.

[27] M. Bando, N. S. Artan, and H. J. Chao, "Highly memory-efficient loglog hash for deep packet inspection," in *Proc. IEEE GLOBECOM*, 2008, pp. 1–6.

[28] N. S. Artan, Y. Haowei, and H. J. Chao, "A dynamic load-balanced hashing scheme for networking applications," in *Proc. IEEE GLOBECOM*, 2008, pp. 1–6.

[29] O. Yigit, "sdbm—substitute dbm," 1990 [Online]. Available: http://search.cpan.org/src/NWCLARK/perl-5.8.4/ext/SDBM_File/sdbm

[30] O. Erdogan and P. Cao, "Hash-AV: Fast virus signature scanning by cache-resident filters," in *Proc. IEEE GLOBECOM*, 2005, vol. 3, pp. 1767–1772.

[31] GNU, "hashlib.c—Functions to manage and access hash tables for bash," 1991 [Online]. Available: http://www.opensource.apple.com/darwinsource/10.3/bash-29/bash/hashlib.c

[32] Y. E. Yang, H. Le, and V. K. Prasanna, "High performance dictionary-based string matching for deep packet inspection," in *Proc. IEEE INFOCOM*, 2010, pp. 1–5.

[33] J. Jiang, Y. Tang, B. Liu, X. Wang, and Y. Xu, "SPC-FA: Synergic parallel compact finite automaton to accelerate multi-string matching with low memory," in *Proc. ANCS*, 2009, pp. 163–164.

Tsern-Huei Lee (S'86–M'87–SM'98) received the B.S. degree from National Taiwan University, Taipei, Taiwan, in 1981, the M.S. degree from the University of California, Santa Barbara, in 1983, and the Ph.D. degree from the University of Southern California, Los Angeles, in 1987, all in electrical engineering.

Since 1987, he has been a member of the faculty with National Chiao Tung University, Hsinchu, Taiwan, where he is a Professor with the Department of Communication Engineering. During the past years, he served as a consultant of various companies to develop large-scale QoS-enabled frame-based switches/routers, integrated access devices, and unified threat management Internet appliances. His current research interests are in communication protocols, broadband switching systems, traffic management, wireless communications, and network security.

Prof. Lee received an Outstanding Paper Award from the Institute of Chinese Engineers in 1991.

Nai-Lun Huang received the B.S. and M.S. degrees in communication engineering from National Chiao Tung University (NCTU), Hsinchu, Taiwan, in 2004 and 2006, respectively, and is currently pursuing the Ph.D. degree in communication engineering at NCTU.

Her current research interests include pattern matching, pre-filtering technique, and network security.

Ms. Huang is a recipient of the third prize in the contest of Embedded Software Design in SW/HW Codesign track held by the Ministry of Education of Taiwan Government in 2007. She received Academic Achievement Awards from NCTU in 2001 and 2005.