An Efficient and Scalable Pattern Matching Scheme for Network Security Applications

Tsern-Huei Lee and Nai-Lun Huang
Department of Communication Engineering
National Chiao Tung University
Taiwan
Email: {tlee@banyan.cm.nctu.edu.tw, nellen.cm93g@nctu.edu.tw}

***Abstract*—Because of its accuracy, pattern matching technique has recently been applied to Internet security applications such as intrusion detection/prevention, anti-virus, and anti-malware. Among various famous pattern matching algorithms, the Aho-Corasick (AC) can match multiple pattern strings simultaneously with worst-case performance guarantee and is adopted in both Clam AntiVirus (ClamAV) and Snort intrusion detection open sources. The AC algorithm is based on finite automaton which can be implemented straightforwardly with a two-dimensional state transition table. However, the memory requirement prohibits such an implementation when the total length of the pattern strings is large. The ClamAV implementation limits the depth of the finite automaton and combines with linked lists to reduce memory requirement. The banded-row format is adopted to compress the state transition table and used as an alternative pattern matching machine in Snort. In this paper we present a novel implementation which requires small memory space and achieves high throughput performance. Compared with the banded-row format, our proposed scheme achieves 39.7% reduction in memory requirement for 5,000 patterns randomly selected from ClamAV signatures. Besides, the processing time of our proposed scheme is, on the average, 83.9% of that of the banded-row format for scanning various types of files. Compared with the ClamAV implementation with the same 5,000 patterns and files, our proposed scheme requires slightly more memory space but achieves 80.6% reduction in processing time on the average.**

I. INTRODUCTION

Pattern matching has been an important technique in information retrieval and text editing for many years. Recently, it has been applied to Internet security for signature matching to detect virus, worms, intrusion, etc., because of its accuracy.

There are some well-known pattern matching algorithms such as Knuth-Morris-Pratt (KMP) [1], Boyer-Moore (BM) [2], and Aho-Corasick (AC) [3]. The KMP and BM algorithms are efficient for single pattern matching but are not scalable for multiple patterns. The AC algorithm pre-processes the patterns and builds a finite automaton which can match multiple patterns simultaneously. Another advantage of the AC algorithm is that it guarantees deterministic performance under all circumstances. It can be shown that the number of state transitions is at most $2n$-1 for an input text string of length n. In fact, this number can be reduced to n if the next move function is adopted. As a consequence, the AC algorithm is widely adopted in various systems, especially when worst-case performance is an important design factor.

A straightforward implementation of the AC algorithm is to construct a two-dimensional state transition table for the finite automaton. However, the huge amount of memory space required makes such an implementation infeasible. As an example, assume that a pattern set results in 1M states and each state is represented with four bytes. If every symbol is a byte, meaning that the number of possible inputs is 2^8, then the total memory requirement is about 1G bytes which is obviously not acceptable for an embedded system. Several schemes had been proposed to reduce the memory requirement. The bitmap architecture presented in [4] can significantly compress the data structure. However, it requires to compute the population count in a 256-bit bitmap and thus may seriously degrade the throughput performance unless hardware acceleration is adopted. The banded-row format [5] proposed by Marc Norton, the Snort IDS Team lead at Sourcefire Inc., can compress the state transition table significantly. For convenience, the banded-row format based implementation of the AC algorithm will be referred to as the banded-row format AC. In Clam AntiVirus (ClamAV) [6] implementation, two data structures, i.e., AC automaton and linked lists, are used to reduce memory requirement. The AC automaton is constructed only for the first two bytes of all pattern strings. Pattern strings which have the same first two bytes form a linked list associated with some leaf state of the AC automaton. As will be seen later in Section VI, such an implementation largely reduces memory requirement but sacrifices throughput performance. Both the banded-row format and the ClamAV implementation will be reviewed in Section III.

In this paper, we first present an idea to improve the throughput performance of the banded-row format AC and then propose another scheme which can further improve throughput performance and reduce memory requirement. Compared with the banded-row format AC, our proposed scheme achieves 39.7% reduction in memory requirement for 5,000 patterns randomly selected from ClamAV signatures. Besides, the processing time of our proposed scheme is, on the average, 83.9% of that of the banded-row format AC for scanning various types of files. Compared with the ClamAV implementation with the same 5,000 patterns and files, our proposed scheme requires slightly more memory space but achieves 80.6% reduction in processing time on the average.

978-1-4244-2390-3/08/$25.00 ©2008 IEEE

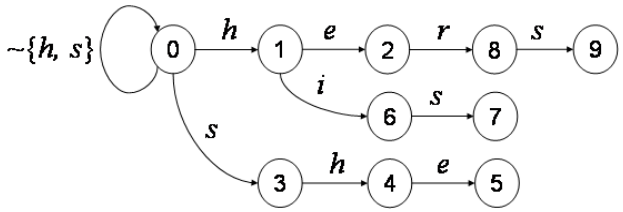
The rest of this paper is organized as follows. In Sections II and III, we review the AC algorithm and some related works, respectively. Section IV presents our idea which improves the throughput performance of the banded-row format AC. Section V contains our proposed scheme, followed by the complexity analysis and experimental results in Section VI. Finally, we draw conclusion in Section VII.

II. 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 pattern strings $Y = \{ p_1, p_2, ..., p_y \}$. 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 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 the example shown in Fig. 1, we have $g(0, h) = 1$ and $g(1, \sigma) = fail$ if σ is not *e* or *i*. State 0 is a special state which never results in the *fail* message, i.e., $g(0, \sigma) \neq fail$ for all input symbols σ. 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 in 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 string. It is not difficult to verify that $f(5) = 2$ for the example shown in Fig. 1. The output function maps a state into a set (could be empty) of pattern strings. The set *output*(*S*) contains pattern string *p* iff *p* is a suffix of the string representing state *S*. As an example, we have *output*(5) = {*he, she*} for the example shown in Fig. 1.



(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.

Figure 1. The AC pattern matching machine for $Y = \{he, she, his, hers\}$.

The operation of a 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*. This is an important property because it provides performance guarantee in the worst case. Notice 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. In this case, the number of state transitions is exactly *n* when an input text string of length *n* is processed.

III. RELATED WORKS

It is clear that, in an AC pattern matching machine, the goto function requires much more storage than the failure and the output functions. A straightforward implementation of the goto function is to use a two-dimensional state transition table. However, the memory requirement for such an implementation may become prohibitively large when the total length of the pattern strings is large. Two related compression schemes which are used in Snort and ClamAV are briefly reviewed below.

A. Banded-row format

The state transition table of the goto function in the AC pattern matching machine is often a sparse matrix because it is likely to contain only a few *nonfail* elements in each row. There are various compression schemes to reduce the memory requirement of a sparse matrix [5], [7]. The banded-row format [5] proposed by Marc Norton, the Snort IDS Team lead at Sourcefire Inc., is an effective compression scheme which allows fast random access to the data.

For the banded-row format, the row elements are stored from the first nonzero value (or *nonfail* value in the goto transition table of AC pattern matching machine) 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 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. Both the goto table and the next move table of AC pattern matching machine can be compressed with the banded-row format. However, the next move table is not as sparse as the goto table, so we choose the goto table. The corresponding pattern matching scheme is referred to as the banded-row format AC, as mentioned before.

Obviously, the banded-row format can be generalized to multiple bands. As an example, the two-band banded-row format of the above sparse vector is (2 3 2 4)(3 8 6 0 7), where (2 3 2 4) and (3 8 6 0 7) denote the first and the second bands, respectively. The elements of the two bands have similar meanings as those in the original banded-row format. Our experiments show that one band is a better choice than multiple bands because there is no significant difference in terms of the reduction of memory requirement and multiple bands yield worse throughput performance than one band because it needs to distinguish more cases.

B. *ClamAV*

ClamAV [6] is an open source anti-virus toolkit for UNIX. The main purpose of it is e-mail scanning on mail gateways. It is the most widely used open source anti-virus scanner available.

ClamAV uses a variation of the AC algorithm. To look up each input symbol quickly, ClamAV constructs a trie structure with a 256-element lookup array for each 8-bit symbol. The memory requirement of ClamAV depends on how deep the trie is. Since the AC algorithm constructs an automaton of depth equal to the longest pattern length, the memory requirement of ClamAV's structure would be unacceptably large because some patterns are of length more than 2,000 bytes. Therefore, ClamAV modifies the AC algorithm so that the trie is constructed only to some maximum depth, and all patterns with the same prefix are stored in a linked list under the appropriate leaf state. The maximum depth is dictated by the shortest pattern length, which is currently two bytes. Fig. 2 shows the ClamAV trie structure.

ClamAV follows the trie transition to process each input symbol. When a leaf state is visited, all patterns on its linked list are checked using sequential string comparisons. As a result, the throughput performance of ClamAV may severely degrade if a leaf state with a linked list containing a large number of patterns is visited.

IV. IMPROVING THE BANDED-ROW FORMAT AC

As described in Section III, the transition table of the banded-row format AC is a compressed version of the goto table. Thus, some band values may be fail. To speed up matching procedure, we replace all band values with the

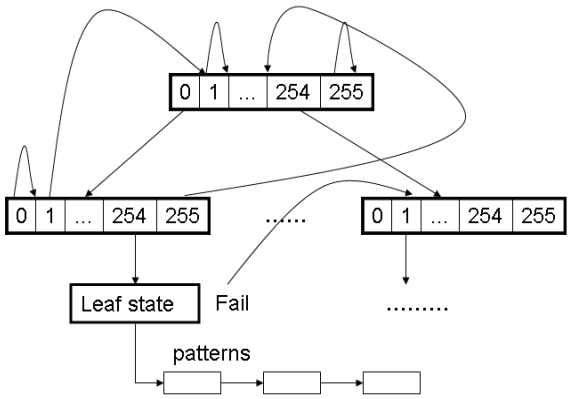


Figure 2. The ClamAV trie structure.

results of the next move function, so that no failure transition is necessary if the input symbol falls in a band. As an example, assume that alphabet $\Sigma = \{a, b, c, d, e, f, g, h\}$ and $Y = \{abcda, abba, cda, cabc\}$. Assume further that the symbols in Σ are sequentially encoded as 0, 1, 2, 3, 4, 5, 6 and 7. For clearness, the original goto graph is shown in Fig. 3. The goto transition vector for state 8 is (11 *fail fail* 9 *fail fail fail fail*) and, therefore, its corresponding banded-row format is given by (4 0 11 *fail fail* 9). Since the two symbols b and c which result in *fail* fall in the band, their transitions are replaced with the results of the next move function. The goto transitions for the two symbols a and d, which also fall in the band, are the same as their next move transitions. As a consequence, the transition vector stored for state 8 is (4 0 11 0 8 9). With this replacement, the number of failure transitions during text scanning can be reduced and, thus, the throughput performance improves.

V. OUR PROPOSED SCHEME

In our proposed compression scheme, we classify states according to the number of child states and whether or not pattern strings 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* if there exists a

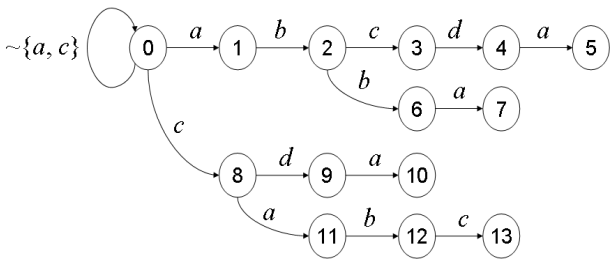


Figure 3. The goto function for $\Sigma = \{a, b, c, d, e, f, g, h\}$ and $Y = \{abcda, abba, cda, cabc\}$.

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. It is clear that a leaf state is always a final state but not the converse. Finally, state S is said to be an explicit state if it is a branch state or a final state.

We store all pattern strings and some data structures for the states on the goto graph. The data structures for branch, single-child, and leaf states are different. Assume that state S is a branch state. In this case, we store $f(S)$ and $g(S, \sigma)$ for all possible input symbols σ. As a result, we also have a two-dimensional state transition table. However, the number of rows is only equal to the number of branch states. It is not hard to see that the number of branch states is at most y-1 if there are y pattern strings. To save space, the banded-row format is adopted to compress the two-dimensional table. To speed up matching procedure, we adopt the idea proposed in Section IV. For convenience, the resulting state transition table is named the Branch State Transition (BST) table.

Assume that state S is a single-child state. We say state R is a descendent state of state S if state R is a child state of state S or a descendent state of some child state of state S. In other words, state R is a descendent state of state S if there exist strings u and v such that u represents state S and uv (concatenation of u and v) represents state R for some nonempty string v. Furthermore, state R is said to be a descendent explicit state of state S if R is an explicit state and a descendent state of state S. Note that, based on our definition, state R is different from state S if state R is a descendent explicit state of state S. 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 descendent explicit state of state S which is represented by string uw where string w is a proper prefix of string v. Suppose that state R is the NDES of state S. It is true that there exists at least one pattern string p_k such that $p_k = uvr$ for some string r. The data structure for the single-child state S includes S.pattern, S.position, S.distance, and $f(S)$, where S.pattern, S.position, and S.distance store, respectively, the identification of pattern string p_k, $|u|$ (length of string u), and $|v|$. Note that, if states are numbered sequentially from a single-child state to its NDES, then the state number of state S.NDES is that of state S plus S.distance. In our realization, we use such a numbering scheme as in the original AC algorithm.

Finally, assume that state S is a leaf state. In this case, we simply store $f(S)$. Of course, every state needs a flag to indicate whether or not it is a final state and, if it is, another data structure is necessary to emit the matched pattern strings.

Consider the example in Section IV again. There are only three branch states, namely states 0, 2, and 8. The vector representing goto transitions for state 0 is (1 0 8 0 0 0 0 0) and it is stored as (8 0 1 0 8 0 0 0 0 0) in our scheme. Similarly, the goto transitions for state 2 is (*fail* 6 3 *fail fail fail fail fail*) and is stored as (2 1 6 3). Finally, the goto transitions for state 8 is (11 *fail fail* 9 *fail fail fail fail*) and we store it as (4 0 11 0 8 9).

Assume that the pattern strings *abcda*, *abba*, *cda*, and *cabc* are identified by numbers, 0, 1, 2, and 3, respectively. State 1 is an example of single-child state. Its NDES is state 2 with distance 1. Let S be state 1. There are two pattern strings, i.e., *abcda* and *abba*, which can be used as S.pattern. In our example, we picked pattern string *abcda* with identification 0. The data structures for the other single-child states can be obtained similarly. Fig. 4 shows the data structures of our proposed scheme for this example.

Our proposed pattern matching machine is described below. For convenience, we use $p_k[m]$ to represent the m^{th} symbol of pattern string p_k and assume input text string $T = t_1 t_2 ... t_n$. Note that, since the states from a single-child state to its NDES are numbered sequentially, the updated current state after each success transition from a single-child state to its NDES can be easily obtained by increasing the current state number by one.

```
Pattern Matching Machine
S ← 0;   i ← 1;   // initialization
While (i ≤ n)
{
   If (S is a branch state)
   {
      If (BST[S][1]≤t_i ≤ BST[S][1]+ BST[S][0]-1)
      {
            S ← BST[S][2+t_i - BST[S][1]];
            If (output(S) ≠ ∅)
                  emit output(S);
            i ← i+1;
      }
      Else
            S ← f(S);
   }
   Else if (S is a single-child state)
   {
      p_k ← S.pattern;   m ← S.position+1;   St ← S;
      While (m≤ St.position+ St.distance)
      {
            If (t_i = p_k[m]) {S ← S+1;   i ← i+1;
                              m ← m+1;}
            Else {S ← f(S);   break;}
      }
      If (m= St.position+ St.distance+1)
            If (output(S) ≠ ∅)
                  emit output(S);
   }
   Else   // S is a leaf state
      S ← f(S);
}
```

Branch state	Band-width	Start index	Band values							
0	8	0	1	0	8	0	0	0	0	0
2	2	1	6	3						
8	4	0	11	0	8	9				

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

S	***S*.pattern**	***S*.position**	***S*.distance**
1:	0	1	1
3:	0	3	2
4:	0	4	1
6:	1	3	1
9:	2	2	1
11:	3	2	2
12:	3	3	1

(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	8	9	10	0	1	0	0	1	1	2	3

(c) The failure function.

S	***output*(*S*)**
5	{0, 2}
7	{1}
10	{2}
13	{3}
others	∅

(d) The output function.

Figure 4. Data structures of our proposed scheme for $\Sigma = \{a, b, c, d, e, f, g, h\}$ and $Y = \{abcda, abba, cda, cabc\}$.

VI. Analysis and Experimental Results

In this section, we compare the memory requirement and processing time of the AC algorithm, the ClamAV scheme, the banded-row format AC, our modified version, and our proposed scheme.

Firstly, let us consider the AC algorithm. We name the AC pattern matching machine dictated by a goto function, a failure function, and an output function AC 1. As mentioned in Section II, we can eliminate the failure function by replacing the goto function with the next move function, and we name this version AC 2. Both the goto function and the next move function can be realized with two-dimensional tables of $O(L|\Sigma|)$ elements straightforwardly, where L represents the total length of all pattern strings, which is the upper bound of the state number. To realize the failure and the output functions, we need some data structures which both take space $O(L)$. Therefore, the space complexity of AC 1 and AC 2 is $O(L|\Sigma|)$. With the next move function, AC 2 makes exactly n state transitions in processing an input text string of length n. On the other hand, AC 1 needs at least n and at most $2n$-1 transitions. Therefore, the time complexity of both AC 1 and AC 2 is $O(n)$, but actually AC 2 is normally faster than AC 1.

Secondly, consider the ClamAV scheme. ClamAV uses a trie structure of depth two as shown in Fig. 2 to perform pattern matching. On the first level of the trie, there is a $|\Sigma|$-element lookup array. Each element on the array may point to a second-level lookup array, which also contains $|\Sigma|$ elements. All pattern strings should be grouped according to their 2-byte prefix and stored under the appropriate leaf state. Therefore, the space complexity of ClamAV is $O(|\Sigma|^2+L)$.

Thirdly, consider the banded-row format AC and our modification described in Section IV. In both schemes, the goto tables are compressed with the banded-row format. The only difference is that all band values are replaced with the results of the next move function in our modification. Therefore, the space complexity of both schemes is $O(LB)$, where B denotes the average bandwidth.

Finally, consider our proposed scheme presented in Section V. Since the number of branch states is at most y-1, the BST table takes space $O(yB)$. As mentioned above, L is the upper bound of the number of all states, so the number of single-child states is not greater than L. Therefore, the data structure for single-child states takes space $O(L)$ as all pattern strings and the data structures for the failure function and the output function take. Consequently, the space complexity of our proposed scheme is $O(yB+ L)$.

In addition to theoretical analysis, we conduct practical experiments to compare all of these schemes. All schemes 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. The pattern strings are 5,000 randomly selected ClamAV signatures. Fig. 5 shows the results of the experiments for memory requirement. Table I shows the processing time for each scheme applied to various types of files that contain no pattern strings. To test the processing time for scanning a file with pattern occurrences, we duplicated the file wmvcore.dll several times and inserted a pattern string in each copy at various positions. The resulting files were processed by the program of each pattern matching scheme. All the programs halt when a match is found. The experimental results are shown in Fig. 6.

As shown in Fig. 5, the ClamAV scheme, the banded-row format AC, our modified version, and our proposed scheme require much less storage than the AC algorithm does. The memory requirements of the banded-row format AC, our modified version, our proposed scheme, and the ClamAV scheme are about 1.92%, 1.92%, 1.16%, and 0.08% of that of AC 2, respectively. Note that the ClamAV scheme has the least memory requirement. This is because the data structure of ClamAV is a trie with only two levels. However, with such a trie, every time a leaf state is visited, the ClamAV scheme has to check all pattern strings on the associated linked list using sequential string comparisons. The checking procedure is quite time-consuming when the linked list contains a large number of pattern strings. If the checking fails, the current state transits from the leaf state to its failure state. In other words, the checking procedure does not consume any input symbol, although it takes time. Therefore, the ClamAV scheme requires much processing time, as can be seen in Table I and Fig. 6. Compared with the ClamAV scheme, our proposed scheme requires slightly

TABLE I. PROCESSING TIME COMPARISON FOR SCANNING VARIOUS TYPES OF FILES WITH NO PATTERN OCCURRENCES

Processing time (ms)		Schemes					
		AC 1	AC 2	ClamAV	Banded-row format AC	Our modified banded-row format AC	Our proposed scheme
Scanned files	AC.cpp (4KB)	0.75	0.63	35	1.11	0.77	0.78
	list.txt (10KB)	1.35	1.26	41	1.72	1.39	1.39
	dosx.exe (54KB)	4.68	4.23	53	5.14	4.96	4.71
	index.htm (78KB)	5.78	5.32	58	7.03	6.41	5.79
	bootcfg.exe (186KB)	13.13	12.02	62	16.23	15.94	14.38
	wmvcore.dll (2.2MB)	172.66	153.91	285	218.43	209.85	195.62

more memory space (1.88M bytes vs. 0.13M bytes) but achieves 80.6% reduction in processing time on the average for scanning various types of files as listed in Table I.

As shown in the experimental results, the banded-row format AC, our modified version, and our proposed scheme have satisfactory performance on both memory requirement and processing time. Among them, our proposed scheme is the best for both performance metrics. In comparison with the other two, our proposed scheme achieves 39.7% reduction in memory requirement. Note that, for the banded-row format AC and our modified version, every success transition requires memory access and computation to extract the updated current state from the banded-row format. However, for our proposed scheme, the success transition can be easily done by increasing the current state number by one when a single-child state is visited. Therefore, our proposed scheme requires less processing time than the banded-row format AC and our modified version. According to our experimental results, the processing time of our proposed scheme is, on the average, 83.9% of that of the banded-row format AC for scanning the files listed in Table I.

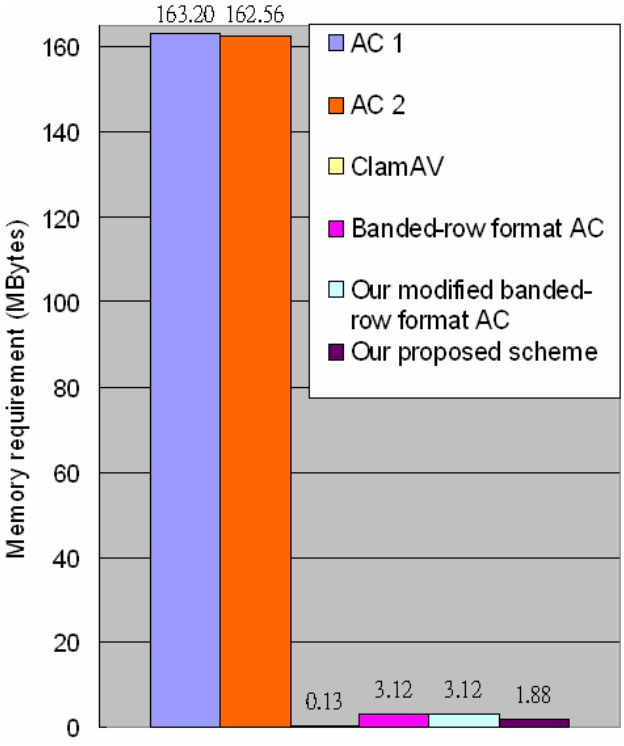


Figure 5. Memory requirement for 5,000 pattern strings randomly selected from ClamAV signatures.

VII. CONCLUSION

In this paper, we first present an idea to improve the throughput performance of the banded-row format AC and then propose a scalable implementation of the Aho-Corasick pattern matching algorithm. The performance of our proposed implementation is compared with those of other related works both theoretically and experimentally. Compared with the banded-row format AC, our proposed implementation achieves 39.7% reduction in memory requirement for 5,000 pattern strings randomly selected from ClamAV signatures and 16.1% reduction in processing time on the average for scanning various types of files. Compared with the ClamAV implementation, our proposed implementation requires slightly more memory space but

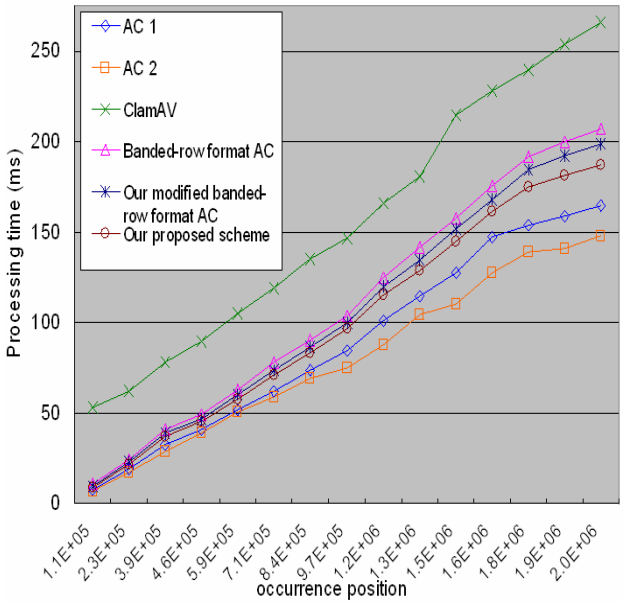


Figure 6. Processing time comparison for scanning a file with a pattern occurrence.

achieves 80.6% reduction in processing time. Based on the analysis and experimental results, we believe that our proposed scheme is more preferable than the banded-row format AC and the ClamAV implementation. An interesting further research topic is to design an efficient pattern matching algorithm to handle regular expressions.

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, October 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, June 1975.

[4] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," *IEEE Infocom*, 2004.

[5] Marc Norton (the Snort IDS Team), "Optimizing pattern matching for intrusion detection," Sourcefire Inc., September 2004.

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

[7] R.E. Tarjan and A.C.-C. Yao, "Storing a sparse table," *Communications of the ACM*, vol. 22, pp. 606-611, November 1979.

[8] Y. Miretskiy, A. Das, C.P. Wright, and E. Zadok, "Avfs: an on-access anti-virus file system," *Proceedings of the 13the USENIX Security Symposium*, pp. 73–88 ,2004.

[9] 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, September 2004.

[10] 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.

[11] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," *Symposium on High-Performance Interconnect (HotI), Stanford, CA*, pp. 44-51, August 2003.

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