# A Time- and Memory- Efficient String Matching Algorithm for Intrusion Detection Systems

Tzu-Fang Sheu
Institute of Communication Engineering
National Tsing-Hua University
Hsinchu, Taiwan
sunnie@ieee.org

Nen-Fu Huang
Department of Computer Science
National Tsing-Hua University
Hsinchu, Taiwan
nfhuang@cs.nthu.edu.tw

Hsiao-Ping Lee
Dept. of Information Management Science
Chung Shan Medical University
Taichung, Taiwan

***Abstract*—Intrusion Detection Systems (IDSs) are known as useful tools for identifying malicious attempts over the network. The most essential part to an IDS is the searching engine that inspects every packet through the network. To strictly defend the protectorate, an IDS must be able to inspect packets at line rate and also provide guaranteed performance even under heavy attacks. Therefore, in this paper we propose an efficient string matching algorithm (named ACM) with compact memory as well as high worst-case performance. Using a *magic number heuristic* based on the Chinese Remainder Theorem, the proposed ACM significantly reduces the memory requirement without bringing complex processes. Furthermore, the latency of off-chip memory references is drastically reduced. The proposed ACM can be easily implemented in hardware and software. As a result, ACM enables cost-effective and efficient IDSs.**

***Keywords- Content Inspection, Intrusion Detection, Network Security, String Matching.***

## I. INTRODUCTION

Network services become very popular today and many companies have provided their services over the Internet. Once a system is broken into or suffered denial of service attacks, this will cause serious damage to a company. Therefore, people demand more secure networks and systems. Intrusion Detection Systems (IDSs) are one of the most useful tools to identifying malicious attempts over the network and protecting the systems without modifying the end-user software. Different from firewalls that only check specified fields of the packet *headers*, IDSs detect the malicious information in the *payloads*. An IDS typically contains a database that describes the fingerprints (patterns) of malicious behavior. The number of patterns is generally a few thousands and still increasing. The patterns may appear *anywhere* in any packet *payload*. Therefore, IDSs must be capable of in-depth packet inspection even when suffering serious attacks; otherwise the protectorate will not be defended strictly. Without doubt, the most essential technology to an IDS is a powerful multiple-pattern matching algorithm.

Many multiple-pattern matching algorithms have been proposed. Boyer-Moore [1] is a well-known single-pattern matching algorithm. The Boyer-Moore-based algorithms [1]-[3] use the *bad character* heuristic to improve the performance of pattern matching. Utilizing the bad group-character heuristic, a variant of the *bad character* heuristic, Wu-Manber [4] builds tables as pre-filters to improve the matching performance. However, there are two drawbacks in these algorithms [1]-[4]: (1) The smaller length of the minimum patterns will result in worst performance. Unfortunately, the minimum pattern in the IDSs is very small. (2) Although these algorithms provide better average-case performance, they do not perform well in the worst case. Dharmapurikar et al [5] and Song et al [6] use the Bloom Filters as pre-filters in their designs of hardware-based IDSs. The Bloom Filter could be optimized by using parallel architecture in the hardware, but the throughput of Bloom Filters decreases seriously in the software implementations [8]. A hierarchical multiple-pattern matching algorithm proposed in [7] builds small tables as pre-filters and improves the best-case and average-case performance by an occurrence-frequency heuristic. However, in these filter-based algorithms [4]-[7], if there is a match in the pre-filters, the exact string matching in the second stage, usually using *sequential search*, is also required. Furthermore, the performance of algorithms [1]-[7] decreases while the number of patterns increases. Consequently, these algorithms have bad worst-case performance.

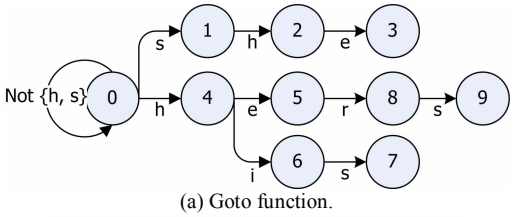
Guaranteed performance is very important especially for the equipment in the core and edge network. The Aho-Corasick algorithm (AC) had the best worst-case computational time complexity [9]. However, as for realistic implementations, the performance of an algorithm is not only affected by the computation time, but also strongly affected by the number of required memory references. The off-chip memory reference costs about 80 ~ 200 clock cycles and the gap may keep increasing [8]. Because of requiring large memory space, the AC needs frequent off-chip memory references and then results in poor performance. Tuck et al modified the AC with a compressed data structure, which reduced the memory size, but also increased the processing time [10]. Therefore, in this paper, we will propose a practical multiple-pattern matching algorithm that has better worst-case performance as well as smaller required memory. The proposed novel scheme is based on the property of Chinese Remainder Theorem and contributes modifications to the AC.

## II. RELATED WORK

A multiple-pattern matching algorithm is used to search the input string $T$ for *all* occurrences of *any* pattern $p_i \in P$, or to corroborate that no pattern belonging to the set $P$ is in $T$. In the IDS, $T$ is the packet stream and $P$ is a set of signatures of the malicious contents predefined in IDSs. In this section, we

This work was supported by the MediaTek Fellowship, MOE Program for Promoting Academic Excellent of Universities (II) under the grant number NSC-94-2752-E-007-002-PAE, and NSC project under the grant number NSC-94-2213-E007-021.

1-4244-0357-X/06/$20.00 ©2006 IEEE




(a) Goto function.

| st | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Fail | 0 | 4 | 5 | 0 | 0 | 0 | 1 | 0 | 1 |

| st | Output |
|---|---|
| 3 | she, he |
| 5 | he |
| 7 | his |
| 9 | hers |

(b) Fail and Output Function

Figure 1.The Aho-Corasick algorithm.

```
struct ACO{                          struct ACB {
struct ACO *next_state[ | Λ | ];     bitmap next_flag[| Λ |];
struct Result *pattern_list;         struct ACB *fail_ptr;
};                                   struct ACB *next_start;
                                     struct Result *pattern_list;
                                     };
```

(a) The basic data structure. (b) The data structure of ACB.

Figure 2. The data structure for one state.

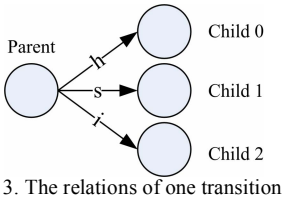



Figure 3. The relations of one transition pair.

will describe the multiple-pattern matching algorithms, AC algorithm [9] and its variant [10], which theoretically provide the best worst-case performance.

### A. The Aho-Corasick Algorithm

AC is an automaton-based algorithm. There are three functions in the AC: Goto(*st*, *code*), Fail(*st*) and Output(*st*), where *st* is the state identification and *code* is a scanned symbol. The Goto function is a state transition function, which is constructed by a set of patterns: *P*. The Goto(*st*, *code*) returns the next state or a *fail* message for the current scanned *code* of the input string. The construction of the Goto function is based on the principle that every prefix of the patterns is only represented by one state. The Fail function points to a state that is the longest suffix of the current state. The Output function stores the matched patterns corresponding to the current state. These three functions are constructed off-line and will be used in the in-line matching stage.[1]

Figure 1 shows an example of the Goto, Fail, and Output functions, where a pre-defined pattern set *P*={she, he, his, hers}. In the matching stage, given an input *T* = 'sihe' for example, the matching procedure scans one symbol at a time and starts from the rooted state of the automaton (State 0). Since, Goto(0, 's') = 1, the machine goes to State 1 and reads the next symbol 'i'. Because 'i' is not an expected symbol for the State 1 (Goto(1, 'i') = *fail*), the Fail function is called and get Fail(1) = 0. Then the machine goes to the State 0, and reads the next symbol 'h'. As Goto(0, 'h') = 4 and Goto(4, 'e') = 5, the machine goes to State 5 and has a valid output: Output(5) = {he}. As a result, we can know that the input *T* contains one pattern 'he'. This example illustrates how the AC works.

### B. The Basic Implementation of the Aho-Corasick Algorithm

To efficiently implement the AC algorithm, a proper data structure for the state machine is required. The paper [9] mentioned that the Goto and Fail functions could be combined into one function: $\delta(st, code)$. According to [9], the basic original data structure ACO is shown in Figure 2(a), where $\Lambda$ is the alphabet set of patterns and $|\Lambda|$ is the size of the alphabet set. The ACO structure represents one state of the AC algorithm. The pointer *next_state* indicates the address of the next state and *pattern_list* points to the corresponding Output function. We can see that all addresses of next states can be obtained directly and only one state transition is required per symbol. However, in a system of 32-bit pointers and $|\Lambda| = 256$, the ACO structure requires 1028 bytes per state. For an IDS, ACO needs about 10 MB to construct the state machine.

### C. The Aho-Corasick Algorithm with Bitmap

Tuck et al proposed a bitmap structure for the AC algorithm (ACB) to compress the state machine (Figure 2(b)) [10]. The state machine is still based on the Goto and Fail functions of the AC algorithm. The bitmap *next_flag*[*a*] indicates whether there is a valid forwarding path for the input symbol *a* (Goto(*st*, *a*)≠*fail*). The ACB structure can reduce the memory to only 44 Bytes for each state. However, to obtain the address of the next state for any input symbol *a*, ACB matching has to calculate the *offset* to the starting pointer *next_start* by scanning *every* bits prior to *a* in the bitmap array and *accumulating* the number of flagged bits. This routine is called *popcount*. For example, if the input symbol is 's', as the ASCII code of 's' is 0x76, ACB matching has to read 118 bits and accumulate these bits to obtain the offset for 's'. We can see that the accumulation routine is very time-consuming. In the worst case, it costs $|\Lambda|$ bit-access, $|\Lambda|+1$ adds and one multiply for each input symbol to obtain the address. Tuck et al admitted that the *popcount* is very expensive for software implementations [10]. Although in the hardware implementations the *popcount* may have the opportunity to be optimized, the complexity and the cost are still high.

## III. A Cost-Effective String Matching Algorithm

The automaton-based string matching algorithm has the advantage that each input symbol will be read only once and no reverse scan is required, independent of input strings and the number of patterns. Since AC is known as the best theoretical worst-case algorithm, we will focus on modifying the AC algorithm to be practical for implementations. In this section, we will also propose a novel data structure that requires a small amount of memory and does not increase the processing time.

In an automaton, since the next state only depends on the current state and the current input, simply consider the parent and children states as shown in Figure 3. Assume that we can find a simple function, say $\Re$, so that the input symbols {h, s,

[1]For the detailed construction procedures, please refer to [9].

1-4244-0357-X/06/$20.00 ©2006 IEEE
*This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE GLOBECOM 2006 proceedings.*

```
struct ACM{
bitmap next_flag[| Λ |];
struct ACM *fail_ptr;
struct ACM *next_start;
struct Result *pattern_list;
long_int MagicNum;
};
```

Figure 4. The data structure with a magic number.

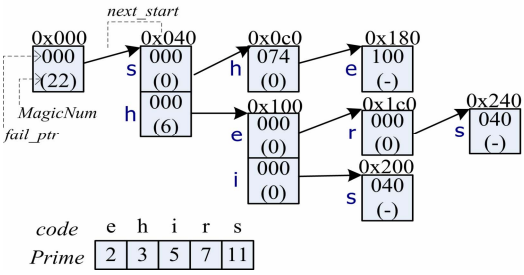



Figure 5. The architecture of ACM state machine, where the number in the parentheses is the magic number.

i} can be mapped to the corresponding child nodes {0, 1, 2} in order. That is, in a general formulation:

$$\{a_1, a_2, ..., a_k\} \xrightarrow{\Re} \{0, 1, ..., k-1\}, \qquad (1)$$

where $a_i$ are the input symbols and $k$ is the number of children. Assume there is a *magic number* $\chi$ and we define $\Re$ as

$$\Re_i : \chi \% f(a_i) = i - 1, \; i = 1, 2, ..., k, \qquad (2)$$

where $f$ is a function that translates the symbol set into a numerical set, and $n$ % $m$ returns the remainder when $n$ is divided by $m$. In other words, $\Re$ acts as a path decoder that returns the corresponding path for each input. Thus, if we can prove the magic number $\chi$ exists, $\Re$ will be obtained. Since $\Re$ has only one simple modulo operation, the state transition process will be fast.

### A. The Magic Number

At first, we have to prove that the magic number exists. The ***Chinese Remainder Theorem*** **(CRT)** can help us to prove this [16]. The theorem is as follows:

**Chinese Remainder Theorem (CRT).** Let $M = \prod_{i=1}^{k} m_i$, where $m_i$ are integers and relatively prime; that is, $gcd(m_i, m_j) = 1$ for $1 \le i, j \le k$, and $i \ne j$.[2] Let $x_1, x_2, ..., x_k$ be integers. Consider the system of congruences:

$$\begin{aligned} X &\equiv x_1 \pmod{m_1} \\ X &\equiv x_2 \pmod{m_2} \\ &... \\ X &\equiv x_k \pmod{m_k}, \end{aligned} \qquad (3)$$

where $X$ and $x_i$ are said to be congruent modulo $m_i$, $1 \le i \le k$. Then there exists exactly one $X$ and $X \in \{0, 1, ..., M-1\}$. ■

Therefore, if let the function $f$ number the symbols by

[2]The $gcd(a, b)$ means the greatest common divisor of $a$ and $b$.

```
Procedure ACM_Matching
Input: A string: T, the starting pointer of the ACM state machine:
rootState, and an array with prime numbers: Prime.
Output: The matched pattern set T: P^M.
1  Initialize: P^M ← ∅ . State ← rootState;
2  For each input symbol: code ← T[i] do
3   While State->next_flag[code] is not set or State ≠ rootState do
4    State ← State->fail_ptr;
5   End
6   If State->next_flag[code] is set then
7    If State->MagicNum = 0 then
8     State ← State->next_start;
9    Else
10    State ← State->next_start +
                ((State->MagicNum)%Prime[code])*Sizeof_ACM;
11   P^M ← P^M ∪ Out(State->pattern_list);
12  End
13  Return;
```

Figure 6. The matching procedure using the ACM structure.

prime numbers, that means $\{a_1, a_2, ..., a_k\} \xrightarrow{f} \{m_1, m_2, ..., m_k\}$, then by CRT we know the magic number $\chi$ exists. $\chi$ is now the $X$ in CRT. Since $f$ is one-to-one mapping, we can use a table, *Prime*, to store the prime number for each possible input symbol. The *Prime* table has at most $|\Lambda|$ entries, and so that it is very small and can be kept in the on-chip cache. Thus the prime number of an input symbol can be obtained by a fast lookup. To obtain the magic number $\chi$, we can use the following algorithm.

**Chinese Remainder Theorem Algorithm.** Let $z_i = M/m_i$ and $y_i = z_i^{-1} (\mathrm{mod}\, m_i)$ for each $i$ = 1, 2,..., $k$, where $z_i^{-1}$ means the multiplicative inverse of $z_i$. (Note that $z_i^{-1}$ exists if gcd($z_i$, $m_i$) = 1.) Then the solution to the congruence system of the Chinese Remainder Theorem is

$$X = (\sum_{i=1}^{k} x_i y_i z_i) \bmod M. \qquad (4) \blacksquare$$

For example, assume we have three valid symbols {h, s, i} that have paths to the child state as shown in Figure 3. Assign three prime numbers {2, 3, 5} for {h, s, i} respectively. According to Figure 3, we want to find a magic number $\chi$ that satisfies $\chi$ % 2 = 0, $\chi$ % 3 = 1, and $\chi$ % 5 = 2. Using the CRT algorithm, we get $z_1$=15, $z_2$= 10, $z_3$= 6 and $y_1$=1, $y_2$= 1, $y_3$= 1. By Equ. (4), $\chi$ = 22. Using 's' for test: as the prime number of 's' is 3 and $\chi$ = 22, we know that the next state of the input symbol 's' is the node Child 1 ( $\chi$ % 3 = 1).

### B. The Multiple-Pattern Matching with a Magic Number

We propose a data structure with a magic number, named ACM, as shown in Figure 4. In the structure ACM, a bitmap *next_flag* is used for fast checking whether there is a valid child. To reduce the size, only one pointer *next_start* pointing to the first valid child state is stored in the data structure. The *MagicNum* stores the magic number $\chi$. The ACM state machine is organized based on the Goto and Fail functions of the AC algorithm. Figure 5. illustrates the memory organization of the ACM state machine. Note that when there is no valid child for the leaf nodes, the magic numbers of the leaf nodes are labeled NULL.

1-4244-0357-X/06/$20.00 ©2006 IEEE
*This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE GLOBECOM 2006 proceedings.*

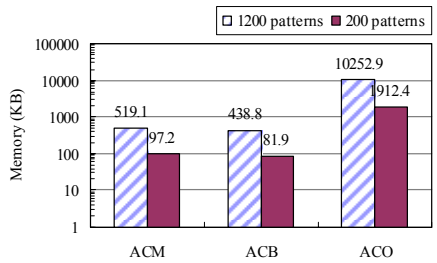



Figure 7. The total memory requirement for the ACM, ACB and ACO structure in the case of 1200 and 200 patterns respectively.

The matching procedure using the ACM structure is shown in Figure 6. In the ACM matching, given an input symbol *a* for example, if *next_flag*[*a*] is not flagged, then the machine traverses the pointer *fail_ptr* until a state has a flagged *next_flag*[*a*] or the machine returns to the root state. If the machine traverses to the root state and the *next_flag*[*a*] is not flagged, the machine will stop in the root state and read the next symbol. Otherwise, while *next_flag*[*a*] is flagged, using the simple function $\Re$ as shown in Equ. (2), the pointer to the next state is

$$\begin{aligned} nextState &= next\_start + \Re \times Sizeof\_ACM \qquad (5) \\ &= next\_start + (MagicNum \,\%\, Prime[a]) \times Sizeof\_ACM, \end{aligned}$$

where *Sizeof_ACM* is the structure size and $\Re$ is the offset to the first valid next state (*next_start*). Obviously, only three reads (*next_start*, *MagicNum* and *Prime*[*a*]) and three operations are required for indicating the next state. According to the lines 7-10 of the ACM matching procedure, the cost is the same in the worst case. As the number of fail transitions is never more than the depth of a state, the number of state transitions for each input symbol will be at most 2 [9]. Consequently, the cost of fail transition is small.

Due to the definition of $\Re$ and CRT, ACM matching has a special property: if there is only one child, $\chi$ will be zero. Observing the ACM state machine, we can find that approaching the leaves, more and more states have only one child state. Therefore, this heuristic can be used in the ACM matching to reduce the computation. That is, if the *next_flag*[*a*] is set and the *MagicNum* in the current state is zero, we know that there is only one child state and the pointer to the next state for the symbol *a* is *next_start*. The forwarding path can be obtained directly without computing the function $\Re$.

The ACM structure is only 52 bytes for each state when the size of magic number is 64 bits, which is much smaller than the ACO structure of 1028 bytes, and so that it successfully reduces the memory requirement. Additionally, the state transition time will be fast because of the simple path decoder $\Re$ and the magic number heuristic.

We illustrate the ACM matching with Figure 5., and assume the input string is 'ish'. Scan the string from left to right, and start from the root state at 0x000. As the bitmap *next_flag*['i'] is not flagged, the machine stays in the state at 0x000. Reading the next symbol 's', we find that it is flagged, and then get *MagicNum* = 22 in the state 0x000. As *Prime*['s'] = 11 and the *next_start* of the state 0x000 is 0x040, the address of the next state for 's' can be calculated by 0x040+(22%11)×0x34 = 0x040, where the size of ACM is 52 bytes (0x34). Then the machine goes to the state at 0x040 and checks the bitmap for the next symbol 'h'. Since the *next_flag*['h'] is flagged and the *MagicNum* is zero, ACM matching knows that it is the only child and the address of the next state is 0x0c0, which is read directly from the *next_start* of the state 0x040.

### C. Implementation Issues

According to the definition of the magic number and the CRT theorem, we notice that if there are too many child states and the alphabet set is large, the magic number will be a large number. In the hardware implementations, this will not be a problem. Many papers proposed optimized hardware designs for high performance modular arithmetic with long operands, which can archive one operation per clock cycle [11]. Therefore, the ACM matching algorithm can be easily implemented in the hardware and gain high performance. However, in the software implementations, there is a limitation on the length of an operand. To overcome this, if the magic number is too large for the software implementations, the *running sum scheme* will be used instead of $\Re$ in the ACM matching. A *union structure* is used here and then eight running sums and the 64-bits magic number share the same memory space. Fortunately, in the case of importing 1200 distinct patterns from the Snort database [12], only 0.078% states of the state machine has to use the running sum scheme. Another issue of implementing ACM on some general processors is sometimes the expensive cost of modulo operations. We will show the simulation results later and illustrate that ACM outperforms ACB even when running on a general processor without optimized modulo instruction.

## IV. RESULTS

In the simulations, with detachment we use the free and real pattern set released by Snort [12]. Since the patterns of Snort are written in mixed plain text and hex formatted bytecodes, we assume that the alphabet size ($|\Lambda|$) is 256 in the simulations. To evaluate the performance of algorithms in a real intense attack, we use a trace from the Capture-the-Flag contest held at the Defcon9 as the input streams of the programs. The Defcon Capture-the-Flag contest is the largest security hacking game, which tries to break into the servers of others while protecting your own server hiding several security holes [13]. In the simulations, we evaluate the performance by calculating the number of instructions used in the algorithms and then multiplying the cost of each instruction. The costs of the instructions refer to a real AMD processor [14], where the number of instructions per clock for add, mov, mul, cmp, bt, and mod is 3, 3, 1, 3, 3, 1/71 respectively, and the operation cost of mod is high.

Let $C_M$ represent the total memory requirement and $C_T$ be the average execution time. Figure 7 and Figure 8 show $C_M$ and $C_T$ for the ACM, ACB and ACO matching respectively in the case of 200 patterns and 1200 patterns. Note that the $C_M$ of ACM includes the memory requirement of the *Prime* table. We can see that the total memory requirement of ACM is 519.2 KB in the case with a big pattern set $|P| = 1200$, which is

1-4244-0357-X/06/$20.00 ©2006 IEEE
*This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE GLOBECOM 2006 proceedings.*

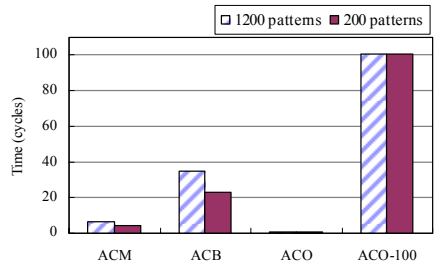



Figure 8. The average execution time per symbol of ACM, ACB, and ACO matching in the case of 1200 and 200 patterns respectively.

TABLE 1. THE NORMALIZED COST OF ACM, ACB AND ACO IN THE CASE OF 200 AND 1200 PATTERNS.

| | $C/C^{ACM}$ (Num. of Patterns = 200) | $C/C^{ACM}$ (Num. of Patterns = 1200) |
|---|---|---|
| **ACM** | 1 | 1 |
| **ACB** | 5.124 | 5.923 |
| **ACO** | 2.851 | 2.168 |
| **ACO-100** | 145.42 | 110.61 |

only 5.1% of the basic AC and a little (18%) more than that of ACB. Furthermore, the memory size of ACM is still in the scale of the on-chip cache that general chipsets support. Therefore, we can say that the ACM can be easily implemented in the hardware and software, and can gain high performance due to no off-chip memory access.

To date, the largest size of on-chip memory supported by the FPGAs is about 1 MB and the size of L1 cache and L2 cache of general processors is only 128 KB ~ 2 MB [15]. Therefore, according to the memory requirement shown in Figure 7, the full state machine of ACO can not be stored in the on-chip memory. The external memory references are required in the ACO matching. Figure 8 draws the average execution time per byte of ACM, ACB and ACO matching respectively in the case of importing 200 and 1200 patterns. There are two cases for ACO matching: the result labeling ACO is not assessed any latency penalty for the external memory references, and the other one labeling ACO-100 needs 100 cycles for each external fetch. Figure 8 shows that ACM performs about 5.67 times better than ACB in the case of 1200 patterns, and 5.34 times over ACB in the case of 200 patterns. Comparing ACM with ACO and ACO-100, we can see that ACM outperforms ACO-100 and the external memory references drastically affect the performance of ACO matching. Note that the cost of modulo operation in the simulations is extremely higher than others. Even assessed the penalty of high operation cost, ACM still outperforms ACB and is moderately slower than ACO. If implemented in embedded systems or FPGAs, ACM will be more efficient.

As the required time and memory are usually trade-off, to compare the overall costs of these three algorithms, we define an evaluation function $C$: $C = C_M \times C_T$. The higher $C$ means the more cost is required in the implementations. The total cost for ACM, ACB and ACO is labeled $C^{ACM}$, $C^{ACB}$, and $C^{ACO}$ respectively. For easy comparison, we show the normalized cost ($=C/C^{ACM}$) of each algorithm in Table 1. Table 1 demonstrates that the cost of ACM is smaller than others. Even requiring a little more memory than ACB, ACM has better overall efficiency, which is about 5.1 ~ 5.9 times better than ACB. Although the theoretic execution time of ACO is shorter than that of ACM, the overall cost of ACM is about 2.1 ~ 2.8 times smaller. For realistic implementations, we can see that the overall cost of ACM is about 110 ~ 145 times better than that of ACO-100. Therefore, we can say that ACM is a time- and memory-efficient algorithm for string matching.

## V. CONCLUSIONS

In this paper, an efficient multiple-pattern matching algorithm, ACM, for intrusion detection has been proposed. Combining a novel scheme using a magic number derived from the Chinese Remainder Theorem with the modified Aho-Corasick algorithm, the ACM algorithm requires only a small amount of memory while providing high worst-case performance. ACM is a practical algorithm and especially suitable for the heavy-loaded IDSs. Simulations show that the overall efficiency of ACM is about 2 ~ 145 times better than the state-of-the-art algorithms. Consequently, ACM enables cost-effective IDSs that can survive in heavy attacks.

## REFERENCES

[1] R.S. Boyer and J.S. Moor. A Fast String Searching Algorithm. *Communications of the ACM*, Vol. 20, No. 10, pp. 762-772, October 1977.
[2] R. Nigel Horspool. Practical Fast Searching in Strings. *Sofetware Practice and Experience*, Col. 10, No. 6, pp. 501-506, 1980.
[3] C. Jason Coit, Stuart Staniford, and Joseph McAlerney. Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort. *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition*, 2001.
[4] Sun Wu and Udi Manber. A Fast Algorithm for Multi-Pattern Searching. *Tech. Rep. TR94-17, Department of Computer Science, University of Arizona*, May 1994.
[5] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep Packet Inspection Using Parallel Bloom Filters. *IEEE Micro*, Vol. 24, No. 1, Jan 2004, pp. 52-61.
[6] Haoyu Song and John W. Lockwood. Multi-pattern Signature Matching for Hardware Network Intrusion Detection Systems. *Proceedings of IEEE Globecom 2005*. St. Louis, MO, Nov. 28, 2005.
[7] Tzu-Fang Sheu, Nen-Fu Huang and Hsiao-Ping Lee. A Novel Hierarchical Matching Algorithm for Intrusion Detection Systems. *Proceedings of IEEE Globecom 2005*. Vol. 3, pp. 1691-1695, St. Louis, MO, Nov. 28, 2005.
[8] Ozgun Erdogan and Pei Cao. Hash-AV: Fast Virus Signature Scanning by Cache-Resident Filters. *Proceedings of IEEE Globecom 2005*. St. Louis, MO, Nov. 28, 2005.
[9] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, Vol. 18, Np. 6, pp. 330-340, June 1975.
[10] Nathan Tuck, Timothy Sherwood, Brad Calder, George Varghese. Deterministic Memory –Efficient String Matching Algorithms for Intrusion Detection. *Proceedings of the IEEE Infocom Conference*, Hong Kong, March 2004.
[11] A. J. Elbirt and C. Paar. Towards and FPGA Architecture Optimized for Publick-Key Algorithms. *Proceedings of the SPIE's Symposium on Voice, Viedo, and Data Communications*. Sept 1999.
[12] Snort. http://www.snort.org.
[13] Crispin Cowan. Defcon Capture the Flag: Defending Vulnerable Code from Intense Attack. *DARPA DISCEX III Conference, Washington DC*, April 2003.
[14] Torbjorn Granlund. Instruction Latencies and Throughput for AMD and Intel x86 processors. http://swox.com/doc/x86-timing.pdf. Sep. 2005.
[15] Intel Corp. http://processorfinder.intel.com/scripts/default.asp.
[16] Yuke Wang. New Chinese Remainder Theorems. *Conference Record of the Thirty-Second Asilomar Conference on Signals, Systems & Computers*. Vol. 1, pp. 165-171, Nov. 1998.

1-4244-0357-X/06/$20.00 ©2006 IEEE
*This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE GLOBECOM 2006 proceedings.*