

# Memory Efficient String Matching Algorithm for Network Intrusion Management System

YU Jianming<sup>1,2</sup> (余建明), XUE Yibo<sup>2,3</sup> (薛一波), LI Jun<sup>2,3</sup> (李 军)

1 Department of Automation, Tsinghua University, Beijing, China

2 Research Institute of Information Technology, Tsinghua University, Beijing, China

3 Tsinghua National Lab for Information Science and Technology, Beijing, China

## Abstract

As the core algorithm and the most time consuming part of almost every modern Network Intrusion Management System (NIMS), string matching is essential for the inspection of network flows at line speed. This paper presents a memory and time efficient string matching algorithm specifically designed for NIMS on commodity processors. Modifications of the Aho-Corasick (AC) algorithm based on the distribution characteristics of NIMS patterns drastically reduce the memory usage without sacrificing speed in software implementations. In tests on the Snort pattern set and traces that represent typical NIMS workloads, the Snort performance was enhanced 1.48%~20% compared to other well known alternatives with an automaton size reduction of 4.86~6.11 compared to the standard AC implementation. The results show that special characteristics of the NIMS can be used into a very effective method to optimize the algorithm design.

Received: 2005-12-23

Supported by Juniper Research Grant and Intel IXA University Program.

To whom correspondence should be addressed.

E-mail: junl@tsinghua.edu.cn; Tel: +86-10-62796400

**Key words: string matching; network intrusion management system (NIMS)**

## **Introduction**

NIMS are fundamental security applications that are growing in popularity in various network environments. The heart of almost every modern NIMS has a string matching algorithm. The NIMS uses string matching to compare the payload of the network packet and/or flow against the pattern entries of intrusion detection rules.

String matching requires significant memory and time costs. For example, the string matching routines in Snort account for up to 70% of the total execution time and 80% of the instructions executed on real traces [3]. The size of the string matching data structure is more than 150 MB when using the AC algorithm [4] and the Snort rule set distributed on July 27, 2005. Moreover, as the number of potential threats and their associated patterns continues to grow, the memory and time costs of string matching are likely to increase as well.

These challenges motivate research on the design of string matching algorithms specific to NIMS applications [5-12]. However, most previous algorithms have not utilized the specific characteristics of NIMS patterns to improve the string matching performance. The  $E^2xB$  [8] algorithm utilized the characteristics of NIMS input based on the observation that the input size is relatively small (on the order of packet size) and the expected matching probability is also small (which is common in NIDS environments).

Hardware applications have also been proposed including FPGA and ASIC [13-20]. The hardware methods can certainly achieve higher string matching performance, but the rule set can not be easily updated, especially with the ASIC method. Software algorithms are less expensive and more flexible. With special-purpose, programmable chips tailored to network devices such as network processors (NP), software algorithms can also achieve high performance and can combine the low cost and flexibility of commodity processors with the speed and scalability of custom silicon (ASIC chips).

In this work the characteristics of NIMS patterns are used to design a faster string matching algorithm that takes less memory. An improved AC algorithm, the Character Indexed AC (CIAC), was developed to dramatically reduce the memory requirement.

## **1. Snort and String Matching**

### **1.1. A Introduction to Snort**

Snort is the most popular open source network intrusion detection system and its detection model is used for reference by many commercial products.

Snort captures packets from a network interface which are preprocessed before sending to the detection engine. The preprocessing includes layer three IP fragment reassembly, layer four TCP session reconstruction, and so forth. The detection engine checks packet payloads against the intrusion detection rules. If one or more rules

match, an attack is detected and the corresponding response functions are launched.

The detection rules form a rule set with all the pattern entries of the rules form into a pattern set. For newer versions above Snort version 2.0, the detection rules are divided into many groups, referred to as sub-rule sets in this paper. The pattern entries of each sub-rule set form a sub-pattern set. For example, the TCP and UDP rules are divided into sub-rule sets by the source and destination port numbers. When a TCP or UDP packet arrives, its destination and source port number are used to find the appropriate sub-rule set to be checked. Then a string matching algorithm such as AC is used to compare the packet payload with the corresponding sub-pattern set. If there are matching patterns, the rules that contain the matching patterns are checked to confirm whether an attack is occurring.

## **1.2. String Matching Algorithm**

String matching consists of finding one, or more generally, all the occurrences of a *search string* in an *input string*. In NIMS applications, the pattern is the search string, while the payload is the input string. If more than one search string simultaneously matches against the input string, this is called *multiple pattern matching*. Otherwise, it is called *single pattern matching*.

### **1.2.1. BM Algorithm**

The BM algorithm [21] is the most well-known single pattern matching algorithm.

The BM algorithm utilizes two heuristics, *bad character* and *good suffix*, to reduce the

number of comparisons. Both heuristics are triggered on a mismatch. The BM algorithm takes the far most shift caused by the two heuristics.

Horspool proposed a variation of The BM algorithm, the BMH algorithm [22], which utilizes only an improved bad character heuristic. BMH is simpler to implement than BM, which preserving the average performance of BM.

### **1.2.2. MWM Algorithm**

The MWM algorithm [23] uses the bad character heuristic like the BM algorithm but with a two-byte shift table. The MWM algorithm also performs a hash on the two-byte prefix of the current substring of the input string to index into a group of search strings. The MWM algorithm can efficiently deal with large amounts of search strings. However, its performance depends on the length of the shortest search string and the characteristic of the input string.

### **1.2.3. AC\_BM and SBMH Algorithms**

The set-wise Boyer-Moore-Horspool (SBMH) algorithm [5] is regarded as the first NIDS-specific string matching algorithm. This algorithm adapts heuristics like BM to simultaneously search for multiple search strings. Coit et al. [6] independently proposed a similar algorithm called AC\_BM.

### **1.2.4. E<sup>2</sup>xB Algorithm**

The E<sup>2</sup>xB algorithm [8] is an exclusion-based algorithm specific to NIDS applications. This algorithm is based on the observation that if there is at least one character of the

search string that is not contained in the input string, then the search string is not a substring of the input string. E<sup>2</sup>xB first checks the input string for missing fixed size substrings of the search string. If all the substrings of the search string can be found, a standard string matching algorithm, such as BM, is launched to determine whether actual matching occurs.

### **1.2.5. FNP Algorithm**

The FNP algorithm [10] is a multiple pattern matching algorithm implemented over the network processor. This algorithm utilizes the NP hardware-accelerated hashing engine to identify matching patterns via a link list in the event of hash collision to save the processor power.

## **2. AC and Its Variations**

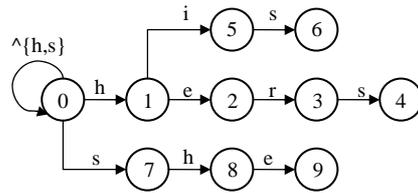
The AC algorithm is one of the most popular multiple pattern matching algorithms. This algorithm accepts all search strings simultaneously to make up of a finite state automaton (FSA) so that every prefix is represented by only one state, even if the prefix belongs to multiple search strings. The AC algorithm deals with the input string characters one by one and has proven linear performance to the length of the input string, regardless of the number and length of the search strings.

Considering that an attacker could intentionally provide input that will knowingly cause the worst case performance of an algorithm, the automaton-based algorithms such as AC are preferred robust algorithms for NIMS. The CIAC

algorithm presented in this paper is such an algorithm.

## 2.1. Implementation of AC

The AC automaton could be a non-deterministic finite automaton (NFA) or a deterministic finite automaton (DFA) which is converted from the NFA. The implementation of AC can be divided into preprocessing and searching stages. The AC preprocessing stage constructs the NFA or DFA. The NFA behavior is dictated by the goto function, failure function and output function. Suppose that the pattern set is  $P = \{hers, she, his, he\}$  and the alphabet is  $\Sigma$ . The NFA for pattern set  $P$  is shown in Figure 1. The symbol ' $\wedge\{h,s\}$ ' means all the characters of  $\Sigma$  except 'h' and 's'.



(a) Goto function

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

(b) Failure function

i	2	9	6	4
output(i)	he	she, he	his	hers

(c) Output function

Fig. 1. NFA of pattern set  $P$

The NFA goto function  $g(\ )$  works as:  $next\_state = g(current\_state, input\_character)$ . For example, from Figure 1(a)  $g(1, 'e')=2$ . The input string is processed character by character with a  $next\_state$  value calculated for each character.

When there is no valid *next\_state* value for a *current\_state* and *input\_character* pair, the output of *g* is marked as *FAIL*, for example  $g(1, 's') = FAIL$ . With  $g(\text{current\_state}, \text{input\_character}) = FAIL$ , the failure function  $f(\text{current\_state})$  is used recursively to calculate the new *current\_state* until there is a valid *next\_state* for  $g(f(\text{current\_state}), \text{input\_character})$ . For example, because  $g(4, 's') = FAIL$ , the NFA state transition procedure is:  $g(4, 's') = FAIL \Rightarrow g(f(4), 's') = g(7, 's') = FAIL \Rightarrow g(f(7), 's') = g(0, 's') = 7$ .

The output function  $output(\text{current\_state})$  determines if there are matching patterns at the current state. For example,  $output(2) = \text{"he"}$ .

The goto and failure function can be merged by using the failure function to pre-compute the next state for every character from every state in NFA. The output function and the optimized goto function construct a DFA.

The NFA and DFA data structures for each state, referred to as nodes in this paper are shown in Figure 2 where  $\sigma$  is the alphabet size. The  $next\_state[\sigma]$  matrix contains  $\sigma$  entries indicating the values of the goto function for all possible input characters.

<pre> struct NFA_Node {     struct ac_state * next_state[ <math>\sigma</math> ];     struct ac_state * fail;     struct detection_pattern * matchlist; } </pre>	<pre> struct DFA_Node {     struct ac_state * next_state[ <math>\sigma</math> ];     struct detection_pattern * matchlist; } </pre>
---	---

Fig. 2. DFA and NFA state data structure

Because DFA requires only one memory reference for each input character to calculate *next\_state*, it can owe better performance. The implementation of the AC algorithm in Snort uses DFA. The AC implementation for the pattern set  $P = \{\text{hers, she, his, he}\}$  shows in Figure 3. Each DFA state node contains 256 *next\_state* values:  $NS_{\lambda}, 0 \leq \lambda \leq 255$ . For example, if the current state is 0 and the current input character is 'h', then  $NS_{104}$  of state 0 is '1' where 104 is the ASCII value for the character 'h'. The index table contains 10 pointers to the 10 state nodes.

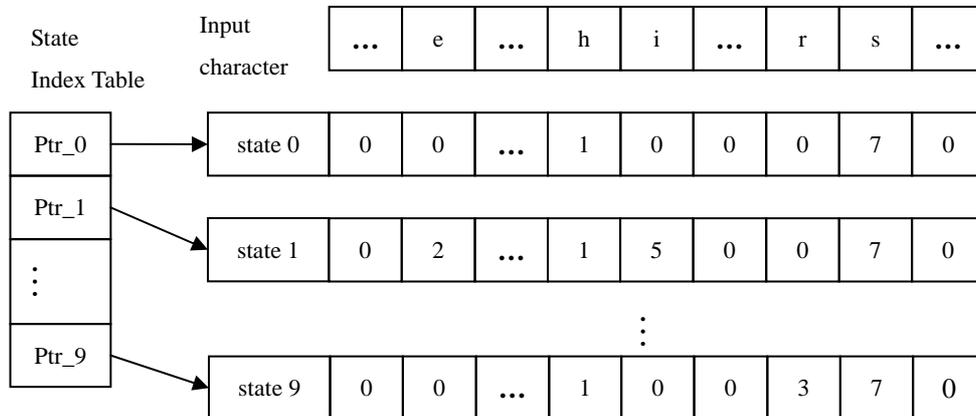


Fig. 3. DFA implementation in Snort

The standard AC DFA search procedure is shown in Figure 4.

```

current_state = 0; // searching starts from state 0
i = 0;
while (i < m) // m is the length of input text
{
    //Get the node address according the value of current_state
    Step 1: State_Table = Index_Table [current_state];
    //Determinate the next_state according to input character T[i];
    //T represents the input text
    Step 2: next_state = State_Table [T[i]];
    Step 3: current_state = next_state;
}

```

Fig. 4. Standard AC DFA search procedure

## 2.2. Variations of AC

The AC automaton requires a huge amount of memory. Many papers have been published analyzing the space complexity of automata. Marc Norton [7] proposed a “banded-row format” to store data efficiently in Snort. The *state 0* node in Figure 3 will be used as an example, as shown in Figure 5(a). The *banded-row format* node is converted as shown in Figure 5(b) which stores elements from the first non-zero value to the last non-zero value, ‘100000000007’. The first entry ‘12’ is the number of *next\_state* values stored. The second entry ‘104’ is the position of the first non-zero *next\_state* in the original standard DFA node, i.e. the ASCII value of character ‘h’.

...	h	i	j	k	l	m	n	o	p	q	r	s	...
state 0	...	1	0	0	0	0	0	0	0	0	0	7	0

(a) Original automaton node

state 0	12	104	1	0	0	0	0	0	0	0	0	0	7
---------	----	-----	---	---	---	---	---	---	---	---	---	---	---

(b) Banded-row format automaton node

Fig. 5. DFA node data structure of banded-row format

Although the memory usage can be reduced, the banded-row format automaton node cannot be randomly accessed. This incurs additional computational costs.

## 3. CIAC: Character Indexed AC Algorithm

Because the DFA search speed is higher than that of NFA, only the DFA is used in this work. But the optimization method can also be applied to NFA.

### 3.1. Central Idea

In the CIAC algorithm, the alphabet of all possible characters is  $\Sigma$  and the characters which appear in the patterns form  $\Sigma'$ ; therefore  $\Sigma'' = \Sigma - \Sigma'$ . In most cases,  $\Sigma'$  is a small subset of  $\Sigma$ . Defining the number of characters in  $\Sigma$  as  $\sigma$ , then for single-byte coding schemes for western scripts  $\sigma = 2^8 = 256$ . The number of characters in  $\Sigma'$  is  $\theta$ , then  $\theta < \sigma$ .

The number of characters appeared in each Snort sub-pattern set is counted with the results shown in Table 1, where  $NC$  represents the number of characters in a sub-pattern set and  $NR$  represents the number of sub-sets which have  $NC$  different characters. Again, the Snort rule set used is that distributed on Jul. 27, 2005.

Table 1. Number distribution of characters in Snort sub-pattern sets

NC	9	11	12	16	22	30	47	60	79
NR	1	1	8	1	4	1	1	1	17
NC	80	81	82	83	84	85	90	92	107
NR	8	4	6	3	2	3	1	1	77
NC	108	109	110	111	112	113	114	115	116
NR	22	8	7	1	1	2	1	2	1
NC	120	121	126	133	134	136	144	152	total
NR	2	2	1	1	1	1	1	1	195

The statistical data show that the largest number of characters in a sub-set is 152, so in every Snort sub-pattern set  $\theta < \sigma$ , where  $\sigma$  in this case is 256. Actually, Table 1 shows that most of the sub-sets have less than half the number of possible characters.

When using DFA, all the next state entries corresponding to the characters belong to  $\Sigma''$  are 0 in all state nodes. While using NFA, all the next state entries

corresponding to the characters belong to  $\Sigma''$  are *FAIL* in all state nodes. The data structure of each state node can be regarded as a one-dimensional matrix. All the state nodes together can be regarded as a virtual two-dimensional matrix as shown in Figure 3. The columns corresponding to the characters belonging to  $\Sigma''$  all have the same value, so these columns can be merged into one.

Suppose there are  $n$  state nodes with  $\theta$  characters, the memory required for the standard DFA is  $O(256 \cdot n)$ . If all the  $(256 - \theta)$  columns corresponding to the characters belonging to  $\Sigma''$  all combined into one column, the required memory decreases to  $O((\theta + 1) \cdot n)$ . As shown by the data in Table 1, in real systems  $\theta \ll 256$ , so the space complexity can be dramatically reduced.

### 3.2. Preprocessing

The preprocessing stage of CIAC based on DFA is:

Step 1: Construct a standard AC DFA as shown in Figure 3.

Step 2: Scan the standard DFA to find the columns which equal a zero matrix.

Step 3: Transpose the matrix to the standard AC DFA which translates the state node in Figure 3 into the character node in Figure 6. Each character node contains  $n$  next state entries corresponding to all the states.

Step 4: The columns equal to zero are converted to rows. Refer to the results in Step 2 to merge these zero rows into one row.

Step 5: A character indexed table with 256 pointers to all the character nodes is

used as shown in Figure 6. The merged row corresponding to all the characters belong to  $\Sigma'$  is represented by the node "other".

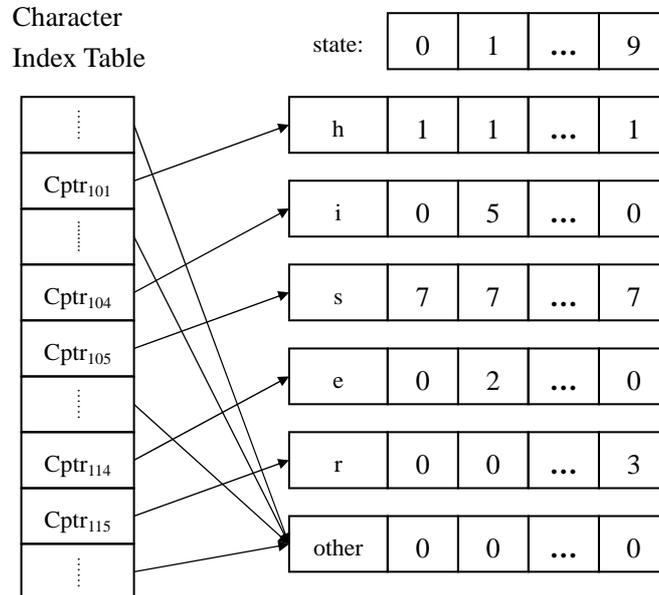


Fig. 6. CIAC automaton implementation

The output functions for the standard NFA, DFA and CIAC are all the same.

### 3.3. Searching

The CIAC search procedure of is:

Step 1: Read one input character which is used to get the pointer to the corresponding CIAC DFA character node by checking the character index table.

Step 2: Read the current state which is used to get the transfer to the next state value by checking the character node determined in step 1.

Step 3: Set the next state value determined in Step 2 as the current state.

Step 4: Check the output function  $output(current\_state)$  to determine if a matching is found.

Step 5: If there are more input characters, return to Step 1.

The implementation of the CIAC search procedure is shown in Figure 7.

```
current_state = 0; //searching starts from state 0
i = 0;
while (i < m) // m is the length of input text
{
    //Get the pointer to the appropriate character node according input
    character T[i];
    //T represent the input text
    Step 1: ptr_node = Character Index Table [T[i]];
    //Determine the next_state according the value of current_state;
    Step 2: next_state = ptr_node [current_state];
    Step 3: current_state = next_state;
}
```

Fig. 7. CIAC search procedure

CIAC allows for fast random access to the DFA data structure (step 1 and step 2 in Figure 7), in the same way as the standard AC implementation (step 1 and step 2 in Figure 4). No additional search costs are incurred, while the memory size is reduced dramatically.

## 4 Experimental Evaluation

### 4.1. Experimental Environments

The CIAC algorithm was implemented and patched into Snort to compare its performance with the other best alternatives in Snort, AC-STD (standard AC implementation), AC-FULL (standard AC optimized for speed, as showed in Figure 3), AC-BANDED (the banded-row format AC), MWM and E<sup>2</sup>xB. The E<sup>2</sup>xB source code was taken from the author's website [24]. For the other algorithms, the algorithm implementations distributed with Snort 2.4.2 were used.

The MWM algorithm in Snort is not an exact implementation, but utilizes a pattern matcher selection heuristics. If the number of patterns in the sub set of rules is less than 5, the BM algorithm is used. Otherwise, when the minimum pattern length within the rules sub set is 1, the MWM version without a BM bad character or bad word shift is used. For other situations, the version with the bad word or bad character shift is used.

The experiments used a PC with an Intel® Pentium 4A processor running at 2.4 GHz, with an L1 cache of 8 KB and an L2 cache of 512 KB, and 512 M of main memory. The host operating system was Linux (Kernel version 2.4.20-8smp, RedHat 9). The tests used Snort version 2.4.2 compiled with gcc version 3.2.2. All the experiments used the default Snort configuration.

#### **4.2. Space Performance**

The space performances of CIAC and AC-STD were compared using four different Snort rule sets: R040811 (Snort rule set distributed in Aug. 11, 2004), R050125, R050916, and R060515. The results are shown in Figure 8. The horizontal axis represents different rule sets, while the vertical axis represents the total memory usage in mega-bytes. The experimental results show that CIAC automaton size is 4.86~6.11 times smaller than that of the standard AC, for the various Snort rule sets.

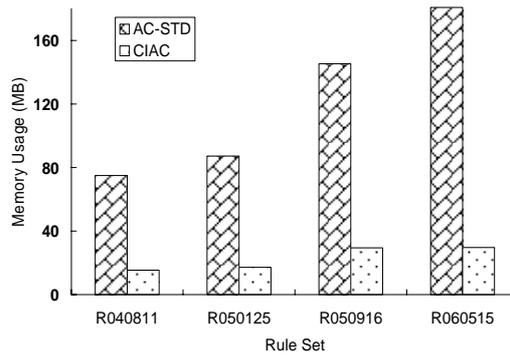


Fig. 8. Space performances of CIAC and AC-STD

The CIAC space efficiency compared to other best alternatives in Snort with the latest Snort rule set R060515 shows in Figure 9. The results show that CIAC is one of the most memory efficient algorithms with memory usage only a little larger than MWM. As mentioned earlier, CIAC has better time performance.

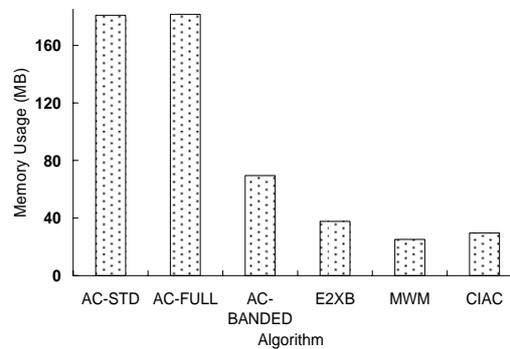


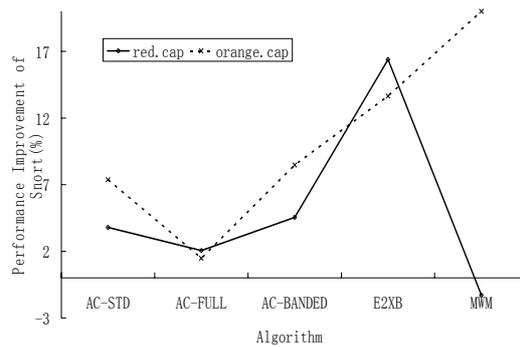
Fig. 9. Space performance of CIAC and other best alternatives in Snort

### 4.3. Time Performance

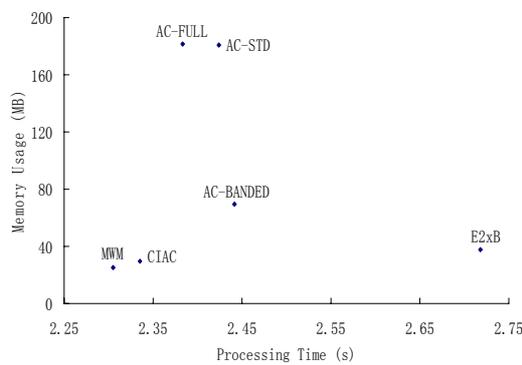
Two data package, red.ctcf.tar.gz and orange.ctcf.tar.gz, from the latest DEFCON [25] “capture the flag” data-set (DEFCON10) were used to evaluate the time performance. There are several full packet traces in each data package. In this work, all the full packet traces in each data package were merged into one file. The finally testing data red.cap (containing all the traces in data package red.ctcf.tar.gz) and orange.cap were

used in the experiments. The red.cap was 42.35 Mbytes, while the orange.cap was 365.05 Mbytes. For simplicity, the traces are read from a local file using the appropriate Snort option, which is passed to the underlying *libpcap* library. (Replaying traces from another host provided similar results.)

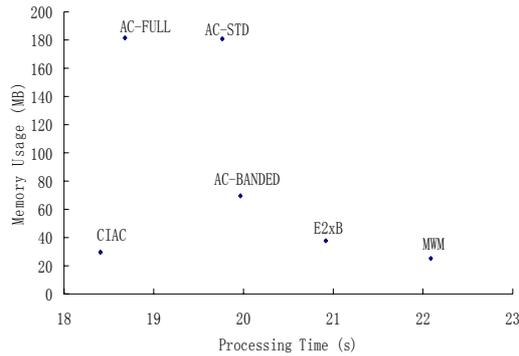
The results are shown in Figure 10. Figure 10(a) shows the Snort performance improvements. The horizontal axis represents the various algorithms, while the vertical axis represents the ratio of  $[(\text{the running time of each algorithm} / \text{the running time of CIAC}) - 1] * 100$ .



(a) Snort performance improvements



(b) Time and space performance comparison using red.cap and R060515



(c) Time and space performance comparison using orange.cap and R060515

Fig. 10. Performance comparisons of CIAC and the other best alternatives in Snort

The memory usage and processing times are shown in Figure 10 (b) and (c). The horizontal axis represents the processing time in seconds, while the vertical axis represents the memory usage in mega-bytes.

The results show that the CIAC method improves the Snort performance by 1.48%~20%, comparing to other popular algorithms currently used in Snort with only the MWM performance with the red.cap trace being 1.29% faster; because the MWM performance is more sensitive to the input trace. CIAC allows fast random access to automaton node data, so it is faster than AC\_BANDED. The automaton size reduction also improves the cache performance so that CIAC is faster than AC\_STD and AC\_FULL. The traces have many attack flows, so the performance of the exclusion-based algorithm E<sup>2</sup>xB is not good.

## 5. Conclusion and Future Work

As link speeds increase and pattern sets become larger, there is greater pressure to improve the performance of NIMS pattern matching algorithms. Software algorithms

are less expensive and more flexible than hardware methods. Efficient software algorithms combined with chips tailored to construct network devices such as network processors can also provide high performance pattern matching.

Previous string matching algorithms specific to NIMS did not fully utilize the special characteristics of NIMS patterns. None of these algorithms have studied and leveraged the character distribution of NIMS patterns.

This work describes a memory efficient string matching algorithm for NIMS based on the observation that the number of characters in a pattern set is less than the total number of characters.

Compared to other well known algorithms, the CIAC enables fast random accesses with no additional costs incurred in the search time. Test results show that the CIAC automaton size is 4.86~6.11 times smaller than for other algorithms depending on the Snort rule set. In the tests, the CIAC Snort performance was 1.48%~20% faster than the other methods based on full packet traces taken from DEFCON10. CIAC is the best algorithm considering the overall performance including both the space and time efficiencies.

Future work will include further analysis of the Snort rules to identify more intrinsic characteristics to design better algorithms more suitable to NIMS applications. Another valuable research direction is the use of these characteristics to design hybrid algorithms which adapt to different sub-set patterns. The CIAC

algorithm will be optimized on Intel's network processor to construct a higher performance NIMS detection engine.

## **Acknowledgements**

The authors thank the anonymous reviewers for providing useful comments on this paper. We also thank our colleagues at the Network Security Laboratory, Research Institute of Information Technology, Tsinghua University for their comments and enlightening discussions.

## **Reference**

- [1] Roesch M. Snort: Lightweight intrusion detection for networks. In: Proceedings of the 13<sup>th</sup> System Administration Conference and Exhibition (LISA'1999). Berkeley, CA, USA: USENIX Assoc, 1999: 229-238.
- [2] Norton M, Roelker D. The new Snort. *Computer Security Journal*, 2003, **19**(3): 37-47.
- [3] Antonatos S, Anagnostakis K G, Markatos E P. Generating realistic workloads for network intrusion detection systems. *Software Engineering Notes*, 2004, **29**(1): 207-215.
- [4] Aho A, Corasick M. Fast pattern matching: an aid to bibliographic search. *Communications of the ACM*, 1975, **18**(6): 333-340.
- [5] Fisk M, Varghese G. An analysis of fast string matching applied to content-based forwarding and intrusion detection. Technical Report CS2001-0670, San Diego:

University of California, 2002.

- [6] Coit C J, Staniford S, McAlerney J. Towards faster string matching for intrusion detection or exceeding the speed of Snort. In: Proceedings of the DARPA Information Survivability Conference and Exposition II (DISCEX'01). Los Alamitos, CA, USA: IEEE Comput. Soc, 2001, **1**: 367-373.
- [7] Norton M. Optimizing pattern matching for intrusion detection. <http://www.NIDSresearch.org>, 2004.
- [8] Anagnostakis K G, Markatos E P, Antonatos S, Polychronakis M. E<sup>2</sup>XB: a domain-specific string matching algorithm for intrusion detection. In: Proceedings of the 18<sup>th</sup> IFIP International Information Security Conference (SEC2003). 2003: 217-228.
- [9] Huang J, Tian J, Du R, Zhai J. Research of pattern matching in intrusion detection. In: Proceedings of the 2003 International Conference on Machine Learning and Cybernetics. Piscataway, NJ, USA: IEEE, 2003, **3**: 1877-1882.
- [10] Liu RT, Huang NF, Chen CH, Kao CN. A fast string-matching algorithm for network processor-based intrusion detection system. *ACM Transactions on Embedded Computing Systems*, 2004, **3**(3): 614-633.
- [11] Jiang WB, Song H, Dai YQ. Real-time intrusion detection for high speed networks. *Computers and Security*, 2005, **24**(4). 287-294.
- [12] Yu JM, Li J. A parallel NIDS pattern matching engine and its implementation on

- network processor. In: Proceedings of the 2005 International Conference on Security and Management. 2005: 375-381.
- [13] Yusuf S, Luk W. Bitwise optimized CAM for network intrusion detection systems. In: Proceedings of the 2005 International Conference on Field Programmable Logic and Applications. Tampere, Finland: IEEE, 2005: 444-449.
- [14] Baker Z K, Prasanna V K. A computationally efficient engine for flexible intrusion detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2005, **13**(10): 1179-1189.
- [15] Attig M, Lockwood J. A framework for rule processing in reconfigurable network systems. In: Proceedings of the 13<sup>th</sup> Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa, CA, USA: IEEE, 2005: 225-234.
- [16] Janardhan S, Bu L, Chandy J A. A signature match processor architecture for network intrusion detection. In: Proceedings of the 13<sup>th</sup> Annual IEEE Symposium on Field-Programmable Custom Computing Machine. Napa, CA, USA: IEEE, 2005: 235-242.
- [17] Nilsen G, Torresen J, Sorasen O. A variable word-width content addressable memory for fast string matching. In: Proceedings of the 22<sup>nd</sup> Norchip Conference. Oslo, Norway: IEEE, 2004: 214-217.
- [18] Dharmapurikar S, Krishnamurthy P, Sproull T, Lockwood J. Deep packet

- inspection using parallel Bloom filters. In: Proceedings of the 11<sup>th</sup> Symposium on High Performance Interconnects. Stanford, CA, USA: IEEE, 2003: 44-51.
- [19] Baker Z K, Prasanna V K. Time and area efficient pattern matching on FPGAs. In: Proceedings of the 12<sup>th</sup> ACM International Symposium on Field-Programmable Gate Arrays. Monterey, CA, USA: ACM, 2004: 223-232.
- [20] Tuck N, Sherwood T, Calder B, Varghese G. Deterministic memory-efficient string matching algorithms for intrusion detection. In: Proceedings of IEEE INFOCOM 2004. Piscataway, NJ, USA: IEEE, 2004, **4**: 2628-2639.
- [21] Boyer R, Moore J. A fast string searching algorithm. *Communications of the ACM*, 1977, **20**(10): 762-772.
- [22] Horspool R N. Practical fast searching in strings. *Software practice and Experience*, 1980, **10**(6): 501-506
- [23] Wu S, Manber U. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona, 1994.
- [24] E2xB algorithm patch for Snort version 2.4.2.  
<http://dcs.ics.forth.gr/Activities/Projects/snort.html>
- [25] Cowan C, Arnold S, Beattie S, Wright C, Viega J. Defcon capture the flag: Defending vulnerable code from intense attack. In: Proceedings of the DARPA DISCEX III Conference, Washington DC, 2003.