An Efficient String Matching Algorithm Using Bidirectional and Parallel Processing Structure for Intrusion Detection System

Gwo-Ching Chang[1] and Yue-Der Lin[2]
[1]Department of Information Engineering, I-Shou University
Kaohsiung, Taiwan, ROC
[e-mail: cgc@isu.edu.tw]
[2]Department of Automatic Control Engineering & Master Program of Biomedical Informatics and Biomedical Engineering, Feng-Chia University, Taichung, Taiwan, ROC
[e-mail: ydlin@fcu.edu.tw]
*Corresponding author: Gwo-Ching Chang

Received June 6, 2010; revised July 23, 2010; accepted August 13, 2010; published October 30, 2010

Abstract

Rapid growth of internet applications has increased the importance of intrusion detection system (IDS) performance. String matching is the most computation-consuming task in IDS. In this paper, a new algorithm for multiple string matching is proposed. This proposed algorithm is based on the canonical Aho-Corasick algorithm and it utilizes a bidirectional and parallel processing structure to accelerate the matching speed. The proposed string matching algorithm was implemented and patched into Snort for experimental evaluation. Comparing with the canonical Aho-Corasick algorithm, the proposed algorithm has gained much improvement on the matching speed, especially in detecting multiple keywords within a long input text string.

Keywords: String matching, Aho-Corasick algorithm, parallel processing structure, intrusion detection system, snort

This research was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2010-C1090-1001-0008)

DOI: 10.3837/tiis.2010.10.014

1. Introduction

String matching plays a very important role in bibliographic search [1], spell checking [2], network-based intrusion detection systems [3][4], and DNA sequence matching [5]. There are several well-known string matching algorithms such as Knuth-Morris-Pratt [6], Boyer-Moore (BM) [7], Wu and Manber (WM) [8] and Aho-Corasick (AC) [1]. Among these string matching algorithms, the AC algorithm performs better than the other string matching algorithms and guarantees that the worst-case matching time is linear in the length of the input text [9]. Consequently, many string matching methods are based on the original Aho-Corasick algorithm [9][10][11][12]. For instance, the bitmap AC algorithm [9] uses bitmap compression to reduce the memory usage of the AC algorithm. The bit-split AC algorithm [10] divides the length of the input text into smaller bit lengths so as to decrease the amount of memory and the number of comparisons. Coit et al. [11] combine the AC and Boyer Moore algorithm to improve the canonical AC from O(n) to the sublinear time complexity with BM algorithm. However, the main disadvantage for AC_BM is that the worst complexity is O(nm). Mishina and Kojima [12] implement AC in a vector processor and perform the string matching in parallel. However, this algorithm requires preprocessing the text, and thus is not suitable for real time matching. Hardware technologies for string matching are proposed to reduce string matching time, such as systolic array hardware [13], reconfigurable hardware [14], bloom filter [15], and content filtering coprocessor [16] and divided data parallel [17].

However, in next-generation intrusion detection applications, string matching tends to become a bottleneck as the network speed increases to tens of gigabits per second [3][4]. This is the reason why the string matching algorithm should be more efficient and must be improved for identifying packets at the line rate even if the performance and capacity of the memory of computers increase regularly.

In this study, we propose a bidirectional and parallel processing structure to enhance the performance of the original Aho-Corasick string matching algorithm. The proposed string matching algorithm was implemented and patched into Snort version 2.6.0.2 in order to extensively compare its effectiveness with that of the original AC algorithm that had been realized in Snort.

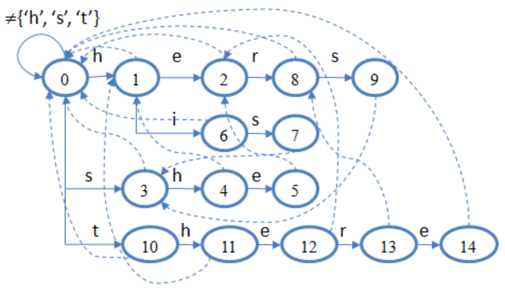
2. Methods

2.1 Aho-Corasick Algorithm

The Aho-Corasick algorithm consists of two parts. The first part involves the construction of a finite state pattern matching machine from a given set of keywords. In the first phase of the state machine construction, each keyword that has to be matched adds a state to the machine, beginning with the initial state, which is the default non-matching state, and proceeding up to the end of the keyword. Each state is represented by a number. State number 0 represents the initial state and has links to the other states. The links generated in this first part represents the goto function $g(state, `c_i')$, which returns the next state when a character 'c_i' is matched. The second part discovers if any of the keywords is present in the text string using the previously built string matching machine. In the second part, failure and output functions are found. The failure function $f(state)$ is used when the character does not match and the matching continues from the failure link state by executing the function $g(f(state), 'c_i')$. The output function returns the found keywords for each reached state.

Fig. 1 illustrates the state machine, output function, and failure function for the set of keywords {he, she, his, hers, there}. Here, ≠{'h', 's', 't'} denotes all input characters other than 'h', 's', and 't'. The solid arrows represent the forward links (function $g(state, 'c_i')$). For example, the transition labelled 'h' from state 0 to state 1 in **Fig. 1-(a)** indicates that $g(0, 'h') = 1$. The dotted arrows represent the failure links (function $f(state)$). For example, if at state number 13, the goto function $g(13, 'c_i')$ will return "fail" when the input character 'c_i' is not 'e', then the state will change from state 13 to state 8. The operation of the Aho-Corasick algorithm is summarized in Algorithm 1.

In addition, the goto function and failure function can be further merged as the next move function $N(state, 'c_i')$ in order to reduce the number of state transition and to accelerate the matching speed [1].



(a) The goto function.

State	Output
2	{he}
5	{she, he}
7	{his}
9	{hers}
14	{there}

(b) The output function.

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f(state)	0	0	0	1	2	0	3	0	3	0	1	2	8	0

(c) The failure function.

Fig. 1. The Aho-Corasick state machine for the set of keywords {he, she, his, hers, there}.

Algorithm 1: (Aho-Corasick string matching algorithm)

Input: *Given an input text string $T=c_1,c_2, \ldots, c_n$ where each c_i for $1 \le i \le n$ is an input character, The AC machine consists of three kinds of functions: goto function $g(state, 'c_i')$, failure function f(state), and output function Out(state).*

Output: *Locations at which the keywords occur in T.*

```
begin
s ← 0;
g(0, 'α') ≠ fail for all input characters α
for i = 1 to n do
    while (g(s, 'c_i') = fail) do
          s ← f(s);
  end of while-loop
    s ← g(s, 'c_i');
    if Out(s) ≠ φ then
         return (i and Out(s));
    endif
end of for-loop
end
```

2.2. Bidirectional and Parallel Processing String Matching Algorithm

In the AC algorithm described above, the characters of the input text string are scanned by the AC finite state machine from left to right. The direction from left to right is referred to as the positive direction. Similarly, the characters in the text string can also be scanned from right to left by using a reverse-directional finite state machine. Therefore, if the text string can be matched by concurrently exploiting both directional (positive- and reverse- directional) finite state machines and operating these finite state machines in parallel, the performance of the string matching will be expected to improve further.

Fig. 2-(a) shows the general structure of bidirectional string matching, which consists of positive- and reverse-directional string matching. This figure also demonstrates the starting search points ('A' and 'B') of bidirectional string matching. The starting search points are obtained on the basis of length of the input text string and the maximum length of keywords. Firstly, we determine the centre point ('O') of the text string. Consider that the longest keyword occurs exactly at centre of the text string. We then shift the centre point of text string to the right and left directions by half of the longest keyword length. The overlaying region represents the longest keyword length.

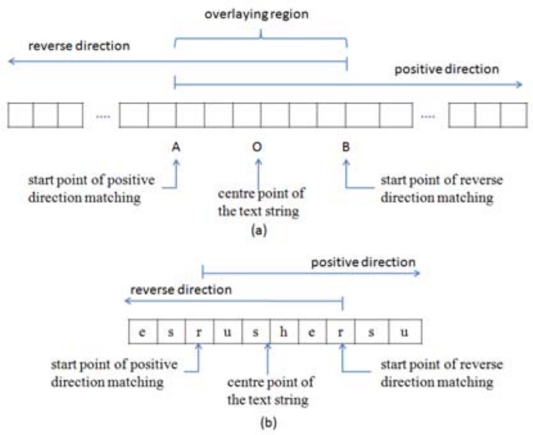
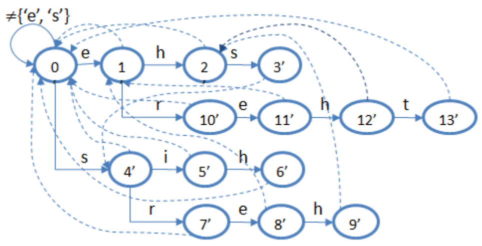

overlaying region
reverse direction
positive direction
A
O
B
start point of positive direction matching
centre point of the text string
start point of reverse direction matching
(a)
positive direction
reverse direction
e s r u s h e r s u
start point of positive direction matching
centre point of the text string
start point of reverse direction matching
(b)

Fig. 2 The structure of the bidirectional string matching. **(a)** illustrates the general form and **(b)** shows an example structure for the input text string 'esrushersu'

The construction of the reverse-directional finite state machine is similar to the positive-directional finite state machine; however, the former is generated by using reverse keywords. **Fig. 3** shows an example of goto, output, and failure functions of the reverse-directional finite state machine for the set of keywords {he, she, his, hers, there}.

Table 1 shows the merged next move functions for the bidirectional finite state machine. The operation of the bidirectional finite state pattern matching machine is stated briefly in Algorithm 2.



(a) The goto function.

State	Output
2'	{he}
3'	{she}
6'	{his}
9'	{hers, he}
13'	{there}

(b) The output function.

State	1'	2'	3'	4'	5'	6'	7'	8'	9'	10'	11'	12'	13'
f(state)	0'	0'	4'	0'	0'	0'	0'	1'	2'	0'	1'	2'	0'

(c) The failure function.

Fig. 3 Reverse-directional state machine for the set of keywords {he, she, his, hers, there}

Algorithm 2: (Bidirectional string matching algorithm)

Input: *Given an input text string* $T=c_1,c_2, \ldots, c_n$ *where each* c_i *for* $1 \le i \le n$ *is an input character, the bidirectional string matching machine consists of four kinds of functions: positive-directional next move function* N_p*(state, '*c_i*') reverse-directional next move function* N_r*(state, '*c_i*'), positive-directional output function* Out_p*(state), and reverse-directional output function* Our_r*(state).*

Output: *Locations at which the keywords occur in T.*

```
begin
$S_p \leftarrow 0$;        initial state of positive-directional state machine
$S_r \leftarrow 0$;        initial state of reverse-directional state machine
$i \leftarrow n/2 - m/2$;  starting point of positive-directional state machine
$j \leftarrow n/2 + m/2$;  starting point of reverse-directional state machine
while ($i \le n$ or $j \ge 1$) do
    $S_p \leftarrow N_p(S_p, 'c_i')$;
    if $Out_p(S_p) \ne \phi$ then
        return ($i$ and $Out_p(S_p)$);
    endif
    $S_r \leftarrow N_r(S_r, 'c_i')$;
    if $Out_r(S_r) \ne \phi$ then
        return ($j$ and $Out_r(S_r)$);
    endif
    if $i \le n$ do
        $i = i + 1$;
    endif
    if $j \ge 1$ do
        $j = j - 1$;
  endif
endof while-loop
end
```

Table 1. Next move functions of the bidirectional string matching algorithm

Positive-directional next move function $N_p(state,c_i)$			Reverse-directional next move function $N_r(state,c_i)$		
current state	input symbol	next state	current state	input symbol	next state
state 0 state 14	h	1	state 0 state 6 state 13	e	1
	s	3		s	4
	t	10		others	0
	others	0			
state 1	e	2	state 1	h	2
	i	6		r	10
	h	1		e	1
	e	3		s	4
	t	10		others	0
	others	0	state 2 state 9	s	3
state 2 state 5	r	8		e	1
	h	1		others	0
	s	3	state 3 state 4	i	5
	t	10			
	others	0		r	7
state 3 state 7 state 9	h	4		e	1
	s	3		s	4
	t	10			
	others	0		others	0
state 4	e	5	state 5	h	6
	i	6		e	1
	h	1		s	4
	s	3		others	0
	t	10			
	others	0			
state 6	s	7	state 7	e	8
	h	1		s	4
	t	10		others	0
	others	0			
state 8	s	9	state 8	h	9
	h	1		r	10
	t	10		e	1
	others	0		s	4
				others	0
state 10	h	11	state 10	e	11
	s	3		s	4
	t	10			
	others	0		others	0
state 11	e	12	state 11	h	12
	h	1		r	10
	s	3		e	1
	t	10		s	4
	others	0			
state 12	r	13		others	0
	h	1	state 12	t	13
	s	3		s	3
	t	10		e	1
	others	0		others	0
state 13	e	14			
	s	9			
	h	1			
	t	10			
	others	0			

3. Evaluation

3.1 Theoretical Evaluation

The time complexity of the AC algorithm is O(*n*) [1], where *n* is the length of the input string. The proposed bidirectional string matching algorithm is based on the AC algorithm. It splits the input string into two equal segments. The string length to be processed by the positive- and reverse-directional state machine is equal to n/2 + m/2, where m is the maximal length of the keywords. Therefore, the time complexity of the bidirectional string matching algorithm running at a two-core system is O(n/2 + m/2). The proposed algorithm can improve the performance by n/(n/2 + m/2) times while comparing with the AC algorithm. If the bidirectional string matching algorithm is used to deal with a large input text string, then *n* will be a large number that is much larger than *m*. In this case, the bidirectional string matching algorithm is nearly two times better than the AC string matching algorithm.

The bidirectional string matching algorithm increases the amount of memory required by two times because it uses positive- and reverse-directional state machines. However, it is worth consuming such memory volumes since the concept of bidirectional string matching algorithm can be extensively employed to develop the multi-directional string matching algorithm by using the same finite state machines. In other words, the memory usage of the multi-directional string matching algorithm is identical to the bidirectional string matching algorithm; however, the former algorithm will be capable of exhibiting a better performance than the latter. The structure of two bidirectional string matching algorithms as an example is illustrated in **Fig. 4**. The *k* bidirectional string matching algorithm running at a *k*-core system will promote the performance by n/(n/2*k* + m/2) times.

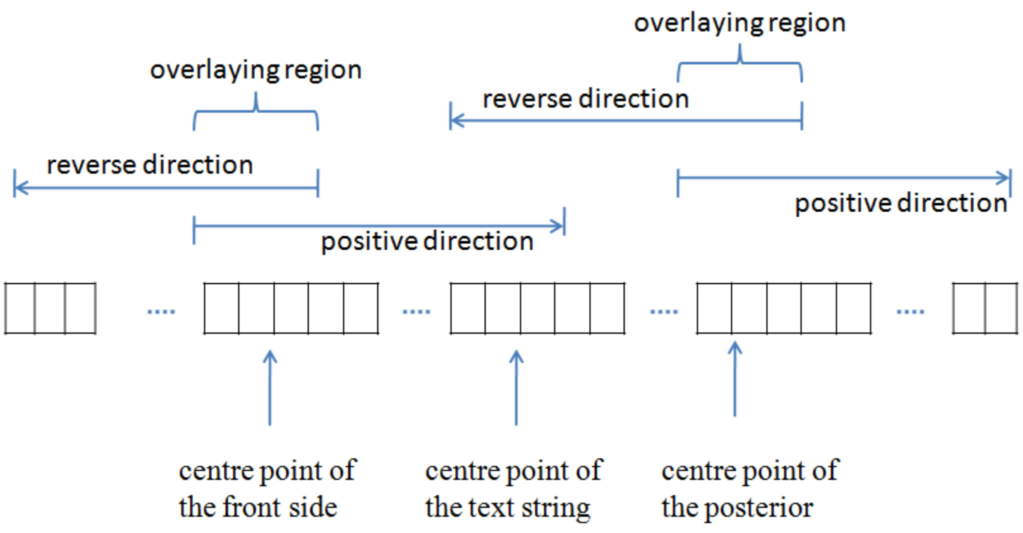


Fig. 4. The structure of four-directional string matching

3.2 Experimental Evaluation

The performances of the original AC string matching algorithm and the proposed bidirectional string matching algorithm were experimentally evaluated according to Algorithm 1 and Algorithm 2. These two algorithms were used to process the input text string ‘esrushersu’ when the set of keywords is {he, she, his, hers, there}. The original AC string matching algorithm was first assessed. Because $g(0, \text{‘e’}) = 0$, the machine remains in state 0. After going through the following processing steps—$g(0, \text{‘s’}) = 3$, $g(3, \text{‘r’}) = 0$, $g(0, \text{‘u’}) = 0$, $g(0, \text{‘s’}) = 3$, $g(3, \text{‘h’}) = 4$, and $g(4, \text{‘e’}) = 5$, the AC string matching algorithm enters state 5 and outputs $Out(5)$, indicating that it has found the keywords ‘she’ and ‘he’ at the end of position seven in the input text string. Therefore in the original AC string matching algorithm, it takes seven steps to find the first keyword.

Next, the bidirectional string matching machine is evaluated by using Algorithm 2. **Fig. 2-(b)** shows the starting points of the bidirectional string matching machine for the input text string ‘esrushersu’. In the first step, the character ‘r’ enters the positive-directional state machine and the character ‘r’ enters the reverse-directional state machine. Since $N_p(0, \text{‘r’})=0$ and $N_r(0, \text{‘r’}) = 0$, the positive- and reverse-directional state machines remain at state 0. No output is generated in this operating step. In the second step, the character ‘u’ enters the positive-directional state machine and the character ‘e’ enters the reverse-directional state machine. Since $N_p(0, \text{‘u’}) = 0$ and $N_r(0, \text{‘e’}) = 1$, the positive-directional state machine still remains at state 0 and the reverse-directional state machine enters state 1. No output is generated in the second operating step. In the third step, the character ‘s’ enters the positive-directional state machine and the character ‘h’ enters the reverse-directional state machine. Since $N_p(0, \text{‘s’}) = 3$ and $N_r(1, \text{‘h’}) = 2$, the positive-directional state machine enters state 3 and the reverse-directional state machine enters state 2 and generates $Out_r(2)$. At this point, the bidirectional string matching algorithm has found the keyword ‘he’. Therefore, in the bidirectional AC string matching algorithm, it takes only three steps to find the first keyword. According to the result of this example, the bidirectional string matching algorithm is 2.33 times more efficient than the AC string matching algorithm for the first occurrence of keywords.

Subsequently, the bidirectional string matching algorithm was implemented by modifying the Build Non-Deterministic Finite Automata (Build_NFA) function, the Build Deterministic Finite Automata from Non-Deterministic Finite Automata (Convert_NFA_DFA) function, the Add Pattern to State Machine (acsmAddPattern) function, the Compile State Machine (acsmCompile) function, and the Search Text or Binary Data for Pattern matches (acsmSearch) function in the multi-pattern search engine of Snort. The modifications of the Build_NFA, the Convert_NFA_DFA, the acsmAddPattern, and the acsmCompile functions were used to construct a reverse-directional state machine. In the acsmSearch function, the positive- and reverse-directional string searchings were alternatively executed to simulate parallel processing. Then, the proposed algorithm was patched into Snort version 2.6.0.2 to extensively compare its effectiveness with the original AC algorithm, the set-wise multi-pattern BM algorithm [18], and the WM algorithm that have already implemented in Snort. The set of keywords used in the present experiment originated from the well-known Snort rule set v2.6, which contained a total of 6718 keywords. The experiments used a host with AMD K8 Athlon64 3000 processor running Snort on the Fedora 7 Linux operating system. Another PC with Pentium IV processor was used to replay the test packet traces to host via a crossover cable. The packet traces from DEFCON10 [19] were exploited to produce the test traffic more realistically.

The results of performance evaluation for the set-wise multi-pattern BM algorithm, the WM algorithm, the original AC algorithm, and the bidirectional string matching algorithm are presented in **Fig. 5**. The throughputs are plotted against the keyword set sizes ranging from 1000 to 6718. As shown in **Fig. 5**, the performance of the bidirectional string matching algorithm is superior to those of other algorithms for different keyword set sizes. The performance of the AC algorithm and the bidirectional string matching algorithm does not vary significantly even when the keyword set sizes are changed. But, the performance of the WM algorithm deteriorates when the keyword set sizes increase from 1000 to 3000. This coincides with previous observations that the AC algorithm is theoretically independent of the keyword set size [18]. The bidirectional string matching algorithm also inherits the attributes of the AC algorithm because it is also independent of the keyword set size. The average throughput for the bidirectional string matching algorithm and the original AC algorithm is 139.4 Mbps and 89.7 Mbps, respectively. The bidirectional string matching algorithm improves the performance of the original AC algorithm by 1.55 times in this primitive experiment.

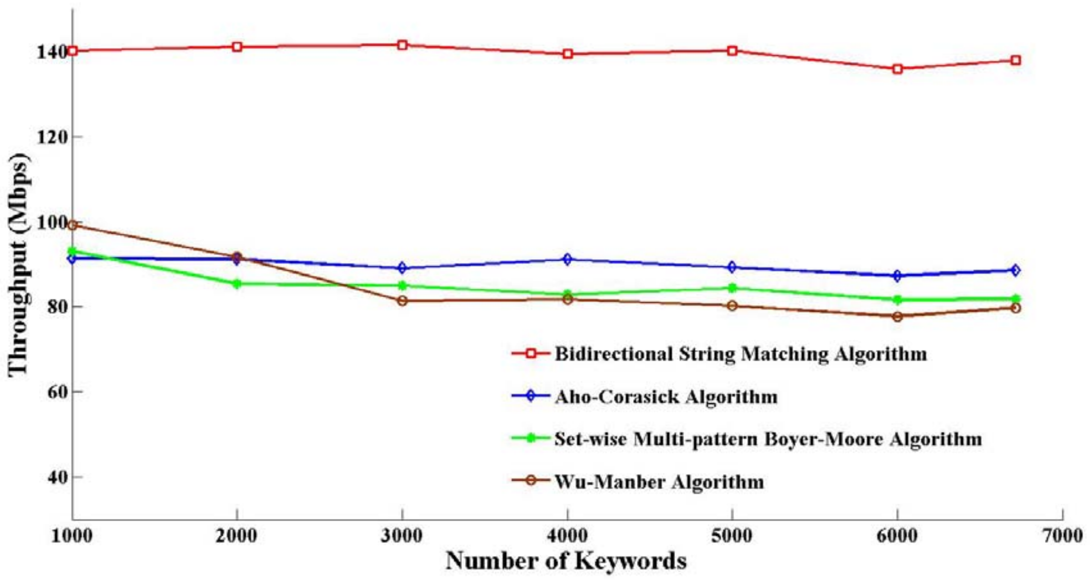


Fig. 5 The throughput of the four string matching algorithm with different keyword set sizes.

The memory comparisons for the set-wise multi-pattern BM algorithm, the WM algorithm, the AC algorithm, and the bidirectional string matching algorithm are shown in **Fig. 6**. We can observe that the amounts of memory for the AC algorithm and the bidirectional string matching algorithm linearly increase with a larger slope as the keyword set size increases. However, the changes of memory usage with the increase of the keyword set sizes for the set-wise multi-pattern Boyer-Moore algorithm and Wu-Manber algorithm are not evident. We also find that the memory usage of the bidirectional string matching algorithm is about 1.9 times that of the AC algorithm. This result fits in with our previous analysis that the bidirectional string matching algorithm consumes more memory volumes. This is an extra cost that the proposed algorithm needs to pay for enhancing the performance. Therefore, the bidirectional string matching algorithm is more suitable for sufficient memory resource available devices.

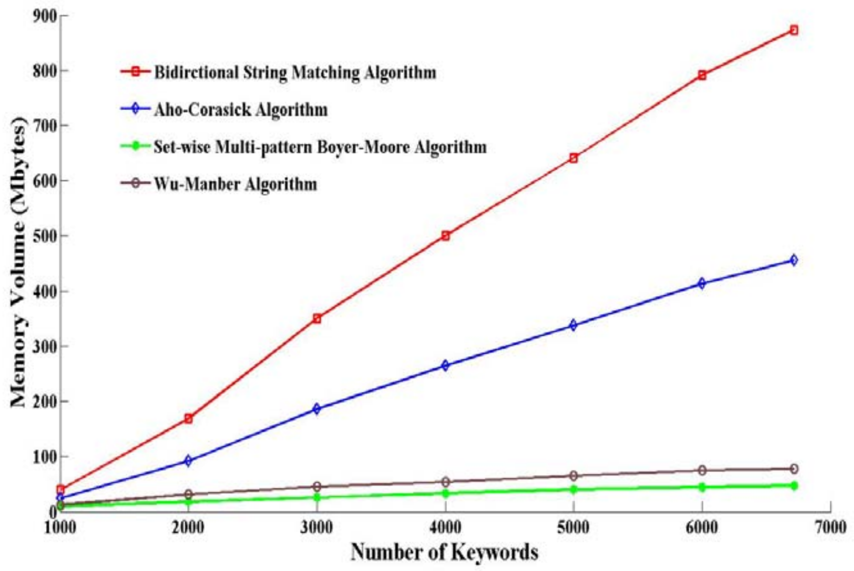


Fig. 6 The memory usage for the four string matching algorithm with different keyword set sizes.

4. Conclusions

In this study, we proposed a bidirectional and parallel processing structure to further improve the performance of the AC string matching algorithm. The proposed string matching algorithm was implemented and patched into Snort for experimental evaluation. Our results show that bidirectional and parallel string matching algorithm is more efficient than the canonical AC algorithms, especially in detecting network packets with a large data payload. In addition, a multi-directional parallel structure can be developed based on the concept of this bidirectional parallel structure, and then, it can be applied to the next-generation intrusion detection system.

References

[1] A.V. Aho and M.J. Corasick, "Efficient string matching," *Communications ACM* , vol. 18, no. 6, pp. 333-340, 1975.

[2] K. Ando, T. Kinoshita, M. Shishibori, and J. Aoe, "An improvement of the Aho-Corasick machine," *Information Science*, vol. 111, pp. 139-151, 1998.

[3] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 1, pp. 99-107, 2005.

[4] R.T. Liu, N.F. Huang, C.H. Chen, and C.N. Kao, "A fast string matching algorithm for network processor-based intrusion detection system," *ACM Trans. on Embedded Computing Systems*, vol. 3, no. 3, pp. 614-633, 2004.

[5] S. S. Sheik, S. K. Aggarwal, A. Poddar, B. Sathiyabhama, N. Balakrishnan, and K. Sekar, "Analysis of string searching algorithms on biological sequence databases," *Journal of CURR SCIENCE*, vol. 89, no. 2, pp. 368-374, 2005.

[6] D.E. Knuth, J.H. Morris Jr., and V.R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput*. vol. 6, pp. 323-350, 1977.

[7] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” *Communication ACM*, vol. 20, no. 10, pp. 762-772, 1977.
[8] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching,” *Technical Report TR-94-17, University of Arizona*, pp. 1-11, 1994.
[9] N. Tuck, T. Sherwood, B. Calder and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” in *Proc. of the IEEE Infocom conf.*, pp. 333-340, 2004.
[10] L. Tan and T. Sherwood, “A high throughput string matching architecture for intrusion detection and prevention,” in *Proc. of 32nd International Symp. on Computer Architecture*, pp. 112-122, 2005.
[11] C. Coit, S. Staniford, and J. McAlerney, “Towards faster string matching for intrusion detection,” in *Proc. of the DARPA Information Survivability Conf. and Exhibition*, pp. 367-373, 2002.
[12] Y. Mishina and K. Kojima, “String matching on IDP: A string matching algorithm for vector processors and its implementation,” in *Proc. of IEEE International Conf. on Computer Design*, pp. 394-401, 1993.
[13] H. M. Bluthgen, T. Noll, and R. Aachen, “A programmable processor for approximate string matching with high throughput rate,” in *Proc. of IEEE International Conf. on Application-Specific Systems, Architectures, and Processors*, pp. 309-316, 2000.
[14] R. Franklin, D. Carver, and B. L. Hutchings, “Assisting network intrusion detection with reconfigurable hardware,” in *Proc. of 10th IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 111-120, 2002.
[15] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, “Deep packet inspection using parallel bloom filters,” *IEEE Micro*, vol. 24, no. 1, pp.52-61, 2004.
[16] K. K. Tseng, Y. D. Lin, T. H. Lee, and Y. C. Lai, “A parallel automaton string matching with pre-hashing and root-indexing techniques for content filtering coprocessor,” in *Proc. of the 16th IEEE International Conf. on Application-Specific Systems, Architectures, and Processors*, pp. 113-118, 2005.
[17] Christopher V. Kopek, Errin W. Fulp, and Patrick S. Wheeler, “Distributed data parallel techniques for content-matching intrusion detection systems,” in *Proc. of the IEEE Military Communications Conference*, pp. 1-7, 2007.
[18] M. Fisk and G. Varghese, “Applying fast string matching to intrusion detection,” *Technical Report CS2001-0670, UCSD*, 2001.
[19] DEFCON® Hacking conference, http://www.defcon.org/

Gwo-Ching Chang received the Ph.D. degree of electrical engineering from National Taiwan University in 1997. From 1997 to 2002, he worked at Chunghwa Telecom Labs for electronic toll collection development in Taoyuan, Taiwan. Since 2002, he has joined the faculty of Department of Information Engineering, I-Shou University in Kaohsiung, Taiwan. He serves as an assistant professor now. Dr. Chang’s research interests include computer network security, embedded system design, and biomedical signal processing.

Der Lin was born in Taichung, Taiwan, in 1963. He earned the Ph.D. degree of electrical engineering from National Taiwan University in 1998. He has been a lecture of the Department of Electrical Engineering, Wufeng Institute of Technology in 1991 and 1992, and a teaching assistant in the Department of Electrical Engineering, National Taiwan University from 1996 to 1998. He joined Comtrend Corporation as a design engineer in 1998 and was responsible for the design of embedded systems in telecommunication. He joined the faculty of the School of Post Baccalaureate Chinese Medicine, China Medical College in 1999 as an assistant professor where his research efforts were focused on medical device design and biomedical signal processing. Since 2003, he became an associate professor at the Department of Automatic Control Engineering, Feng Chia University, Taichung, Taiwan. Dr. Lin is a visiting scholar of the University of Wisconsin- Madison in 2006, where he pursued the research in ECG signal processing. He is the winner of the Rotary International Scholarship in 1994-1995 and 1996-1997, and has been listed in Marquis Who's Who in the World in 2008 , 2009 and 2010 for his researches on physiological signal analysis and wearable technology. He has also been listed in IBC Top 100 Educators, 2000 Outstanding Intellectuals of the 21st Century, Foremost Educators of the World, Foremost Engineers of the World, 21st Century Award for Achievement and Leading Engineers of the World in 2008. Dr. Lin's research interests include the medical instrumentation, biomedical signal processing and development of high-speed algorithm. He is a senior member of IEEE EMB society.