A Hardware-Efficent Multi-character String Matching Architecture Using Brute-force Algorithm

Seongyong Ahn[1], Hyejong Hong[1], Hyunjin Kim[1], Jin-Ho Ahn[2], Dongmyong Baek[3] and Sungho Kang[1]
[1]Department of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea
[2]Department of Electronic Engineering, Hoseo University, Asan, Korea
[3] Next Generation Ethernet Research Team, ETRI, Deajon, Korea
{sy_ahn,hjhong,nagicman}@soc.yonsei.ac.kr, jhahn@hoseo.edu, dongmbaek@etri.re.kr, shkang@yonsei.ac.kr

Abstract— **Due to the growth of network environment complexity, the necessity of packet payload inspection at application layer is increased. String matching, which is critical to network intrusions detection systems, inspects packet payloads and detects malicious network attacks using a set of rules. Because string matching is a computationally intensive task, hardware based string matching is required. In this paper, we propose a hardware-efficient string matching architecture using the brute-force algorithm. A process element that organizes the proposed architecture is optimized by reducing the number of the comparators. The performance of the proposed architecture is nearly equal to a previous work. The experimental results show that the proposed architecture with any process width reduces the comparator requirements in comparison with the previous work.**

Keywords: network intrusion detection system, deep packet inspection, string matching, brute-force algorithm

I. INTRODUCTION

Contrary to traditional firewall, NIDS(Network Intrusion Detection System) inspects the network packet payload at application-layer and detects the malicious network packets. Because of the growth of network environment complexity and the ingenious network attacks, the efficient and effective implementation of system is important. Especially, the string matching that inspects the contents in network packet payload determines the performance of NIDS. Thus, the network wire-speed and the low cost implementation should be supported by string matching. Because the software implementation of string matching, such as the Boyer-Moore[1] and the Knuth-Morris-Pratt[2] algorithms, cannot support the network wire speed, the hardware implementation of string matching has many researches lately.

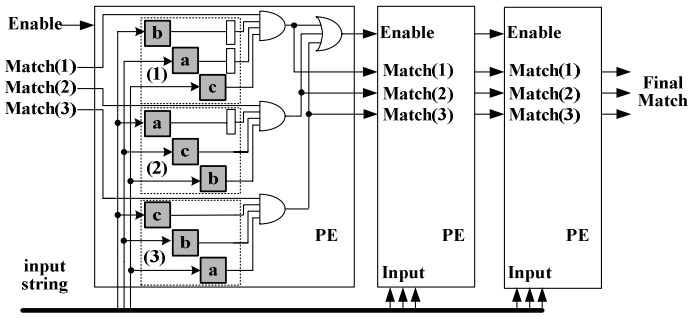
The hardware implementation of the Aho-Corasick algorithm[3] that is designed for multiple pattern matching requires a lot of memories in order to store the all possible next states for current state. The bit-split Aho-Corasick algorithm[4] optimizes the Aho-Corasick algorithm by splitting the input character. Although the bit-split Aho-Corasick algorithm reduces the possible next state, the memory requirement is still high. The bloom filter based string matching[5] has a more efficient data structure than the memory based implementation. However, to prevent the false positive error, it uses the hash tables of the two levels. Also, the bloom filter requires the hash table for each pattern length. Cho et.al.[6] uses the brute-force algorithm. The brute-force based hardware organizes the pattern using logic instead of memory. Therefore, string matching using the brute-force based hardware supports high speed of pattern matching but the hardware cost is high.

This paper proposes the hardware-efficient string matching architecture using the brute-force algorithm. The proposed string matching architecture reduces the requirement of the comparators per process element while minimizing the degradation of performance. The rest of paper is organized as follows. In section 2, we briefly present the previous work. In section 3, we describe the proposed string matching architecture and the performance analysis. In section 4, the comparison of our string matching architecture with the previous work is shown. Finally, we conclude with a summary of the proposed string matching architecture.

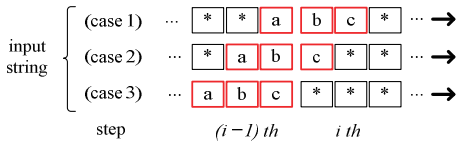
II. MULTI-CHARACTER PROCESSOR ARRAY

The brute-force algorithm has high speed string matching performance, but it is targeted for single pattern matching. The multi-character processor array method[6] using the brute-force algorithm employs parallel set of process element chains for multiple pattern matching. One process element chain handles one pattern, and the set of process element chains process multiple patterns parallel. Each process element in the process element chain processes a substring of a pattern. The substring is part of the pattern. Each pattern spilt into the multiple substrings which have the length of process width, and implemented by the process element chain which has multiple substrings. The architecture of the process element that has substring *'abc'* and the chained process elements are shown in Figure 1(a).

The process element with process width *n* in the previous work processes *n* characters per step. In the multi-character matching, all possible *n* cases should be matched by string matching architecture. In other words, patterns can be found anywhere in the input string. All possible cases for pattern *'abc'* are shown in Figure 1(b) and they should be found by the process element with substring *'abc'* in Figure 1(a). To find the all possible cases in the multi-character matching, the previous work stores the results of matching in *(i-1) th* step and composes the results of matching in *(i-1) th* step and *i th* step for matching of all the possible cases. The dotty lined box (1) in the Figure 1(a) is the set of comparators that targets the input string *(case 2)* in the Figure 1(b). In *(i-1) th* step, the matching



(a) Previous architecture of process element and chained process element



(b) Three possible cases that should be matched by PE

Figure 1. The previous architecture of process element, chained process elements and the three possible cases that should be matched by PE

result of the pattern *'*ab'* is stored at register. In *i th* step, the matching result of the pattern *'c**'* and the matching result of the previous step are ANDed. In the same way, the dotty lined box (2) in the Figure 1(a) finds the input string *(case 1)* in the Figure 1(b). Because the matching result of the substring is transferred to next process element, the next substring can be matched sequentially.

Since the previous work uses the set of comparators for each pattern position, the hardware cost is high. Therefore, a hardware-efficient architecture of multi-character processor array is proposed in this paper. In our work, the process element is optimized by reduction of the required comparators. The alignment element is proposed for the matching of all the possible input strings.

III. PROPOSED METHOD

In this section, we propose the hardware-efficient architecture of process element. In the proposed architecture, the comparators which detect partial matching of target substring, such as *(case 1)* and *(case 2)* in Figure 1(b), are eliminated from every process element. Instead, input string is aligned by the alignment element which is added only to the first process element of each process element chain. Although the alignment element requires an additional step, the performance degradation is not severe because the occurrence of malicious packets is rare. The performance analysis is described in the sub-section D.

A. Proposed Process Element

In previous work, process element uses n^2 comparators for matching n characters per step. When process width is increased linearly, the number of comparators is increased exponentially. To solve the hardware cost problem, the process element which uses n comparators with process width n is proposed in this paper. The architecture of proposed process element with process width 3 is shown in Figure 2.

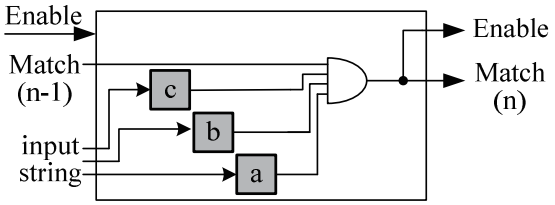


Figure 2. The proposed architecture of process element

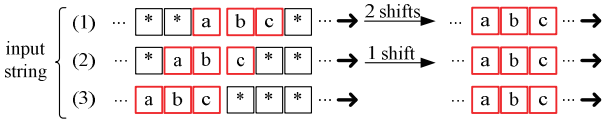


Figure 3. The example of string alignment using the proposed alignment element

An *enable* signal is turn on/off the process element. A *match* signal indicates whether input string is matched or unmatched in the process element. The results of comparators and the result of matching in prior process element are ANDed and the ANDed signal activates the *enable* and *match* signals. The *enable* and *match* signals of first process element are always active. The enable and match signals are transferred to next process element. By the transferring the enable signal, the unnecessary computation is removed because all process element is not always activated. If *match* signal is transferred to the last process element, the final matching signal is activated. Since the input string is transferred to the all process elements concurrently, the string matching can be performed continuously through the process elements.

B. Alignment element

The proposed process element removes the comparator which detects fractions of target, such as *(case 1)* and *(case 2)* in Figure 1(b). Thus the proposed process element cannot search the pattern in the all the possible input strings. The alignment element is proposed to solve the problem of the multi-character matching. If the suffix of input string is matched to the prefix of the matching target pattern, the alignment element aligns input string. Consider the example in Figure 3:

(1) In the case of input string *'**a'*, the suffix of the input string *'a'* is matched to the prefix of the substring *'a'*. The matching in current process element is determined by the matching results of next two characters of the input string. The input string *'**a'* is shifted by the alignment element. In the next step, the shifted input string *'abc'* is compared to the substring *'abc'* in the process element.

(2) In the case of input string *'*ab'*, the suffix of the input string *'ab'* is matched to the prefix of the substring *'ab'*. The matching in current process element is determined by the matching results of next one characters of input string. The input string *'*ab'* is shifted by the alignment element. In the next step, the shifted pattern *'abc'* is compared to the substring *'abc'* in the process element.

(3) In the case of input string *'abc'*, the input string is exactly matched to the substring in the process element. In the

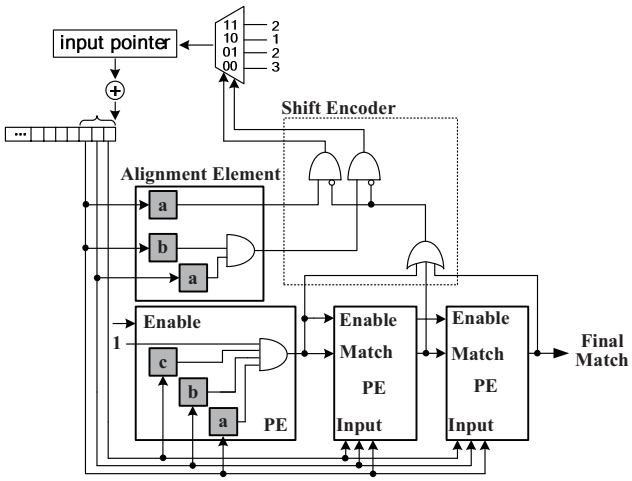


Figure 4. The proposed string matching architecture with process element and alignment element

next step, the next *n* input string is compared to the substring in the next process element.

The exact matching spends 1 step. But the partial matching spends additional step for the alignment of the input string. Therefore, the performance of string matching architecture is affected by the number of the pattern alignments. In the subsection D, we analyze the performance degradation in comparison of the previous work.

One alignment element is needed to one pattern. The alignment element handles partial matching of substring in the first process element. Therefore, the required comparators per alignment element increase as the process width increases. The requirement of the comparators per alignment element is represented by Equation (1). The function *f(n)* indicates requirement of the comparators per alignment element with the process width *n*.

$$f(n) = (n-1) + f(n-1),\ f(1) = 0 \tag{1}$$

C. Overall Architecture

The proposed string matching architecture consists of the process elements and the alignment elements. The $\left\lceil \frac{a\ matching\ target\ pattern\ length}{process\ width} \right\rceil$ process elements and one alignment element is needed for one target pattern. The process element with process width *n* has *n* comparators and is chained. When the substring in the process element is $p = \{n_1 n_2 n_3 ... n_n\}$ and the process width *n*, the alignment element consists of the sets of the comparators for $\{n_1 n_2 n_3 ... n_n\}, \{n_2 n_3 ... n_n\}, ..., \{n_n\}$. The shift value of the input pointer depends on the results of the shift encoder. The shift encoder determines the MUX control signal for input pointer using the results of the alignment element and the process elements.

The proposed string matching architecture with process width 3 is shown in Figure 4. The alignment element processes *'**a'* and *'*ab'*, prefixes of substring *'abc'*. If *'**a'* is matched, the input pointer shifts 2 characters. If *'*ab'* is matched, the input pointer shifts 1 character. If *'abc'* is matched, activated *match* and *enable* signal are transferred to next process element. If the *match* signal of the last process element is activated, the string matching architecture reports the pattern matching. The cases of the shift encoder as follow:

(1) 00: input string is not matched anywhere & input string is matched in process element.

(2) 10: 1 character of suffix of input string is matched & input string is not matched in process element.

(3) 01, 11: 2 characters of suffix of input string are matched & input string is not matched in process element.

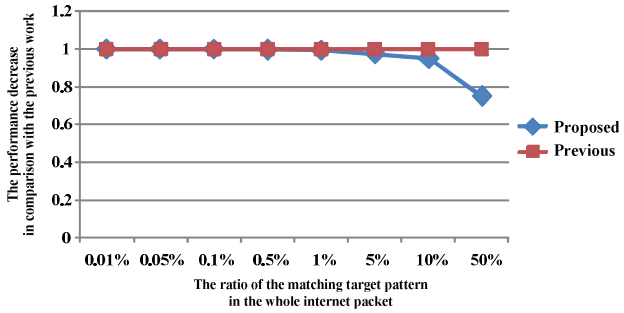


Figure 5. The performance degradation is respected to the probability of the matching target pattern occurrence

D. Performance Analysis

The previous process element processes deterministically *n* characters per step. The reduction of the required comparators per process element causes that the proposed architecture loses the deterministic performance. In other words, when the partial matching is occurred, the alignment element spends additional 1 step in order to align the pattern. In this case, the proposed architecture could process fewer than *n* characters per step. In this section, we analyze the performance of string matching architecture mathematically.

When process width is *n*, the ratio of the matching target pattern in the whole internet packet is *m* and the probabilities of the each possible position of substring in the process element is *1/n* equally, the performance of the proposed architecture is Equation (2).

$$(1-m)\times n + m\times\left(\frac{1}{n} + \frac{1}{n}\times 2 + ... + \frac{1}{n}\times n\right)$$

$$= (1-m)\times n + m\times\frac{n}{2} \tag{2}$$

If the input string is exactly matched, the process element processes *n* characters. If the suffix of the input string is matched to the prefix of substring, the process element processes *n/2* characters on average. The performance degradation is shown in Figure 5. The malicious packet is rare in the whole network packets. Therefore, the performance of proposed string matching architecture shows nearly the same performance of the previous work. Thus, within the 5% of the

TABLE I. COMPARISON OF THE PREVIOUS AND THE PROPOSED STRING MATCHING ARCHITECTURE ACCORDING TO RULES FROM SNORT 2.6

Rule Name	*# of alignment element*	***n=2***		***n=3***		***n=4***		***n=5***		***n=6***	
		# of PE	***RRC***	***# of PE***	***RRC***	***# of PE***	***RRC***	***# of PE***	***RRC***	***# of PE***	***RRC***
sql	74	576	**43.58%**	405	**60.58%**	307	**68.97%**	263	**73.37%**	219	**77.70%**
netbios	171	586	**35.41%**	455	**54.14%**	348	**62.72%**	324	**69.44%**	280	**73.15%**
oracle	337	5,474	**46.92%**	3,716	**63.64%**	2,825	**72.02%**	2,277	**77.04%**	1942	**80.44%**
backdoor	955	4,666	**39.77%**	3,278	**56.96%**	2,564	**65.69%**	2,150	**71.11%**	1875	**74.84%**
web-client	1,657	33,871	**47.55%**	22,728	**64.24%**	17,088	**72.58%**	14,493	**77.71%**	11,498	**80.93%**
All	7,784	74,008	**44.71%**	50,692	**61.55%**	38,617	**69.96%**	32,397	**75.19%**	26,974	**78.52%**

ratio, the proposed architecture can perform the string matching nearly *n* character per step with the process width n.

IV. EXPERIMENTAL RESULTS

In this section, we compute the hardware cost of overall string matching architecture using the number of comparators that are the major component of the process element and the alignment element. Experiments are done using the rule set of Snort version 2.6[7], which is a well-known open source NIDS tool. We use the C++ standard library and boost graph library[8] to order the rule sets lexicographical and estimate the proposed and previous hardware cost. The identical patterns that have different layer 2-4 information, such as the source and destination IP address, is merged to one pattern.

The experimental results are shown in Table 1. We represent the number of the alignment elements and the process elements in Table 1. Also, the reduction ratio of the number of comparators in comparison with the previous work is represented in Table 1. The number of comparators in proposed architecture includes the comparators in the process elements and the alignment elements. The reduction ration of the number of comparators(RRC) is computed by Equation (3).

$$RRC = \frac{k_{previous} - k_{proposed}}{k_{previous}} \times 100\% \quad (3)$$

$$(k_x : \#\ of\ comparators\ in\ x)$$

In Table 1, when the process width increases, the RCC increases. The reason of the increase of the RCC is that the proposed process element has *n* comparators for process width *n*, while the previous process element has n^2 comparators for process width *n*. Thus, the required comparators in the proposed process element increase linearly, but the required comparators in the previous process element increase exponentially. The experimental results show that 44.71% CRR with the process width 2 and 78.52% CRR with the process width 6. The requirement of the alignment element is equal to the number of the unique patterns in the rule set because one alignment element is needed to one unique pattern.

V. CONCLUSION

In this paper, we propose a hardware-efficient string matching architecture using the brute-force algorithm. The proposed architecture consists of the alignment elements and the process elements which have *n* comparators. Using the proposed process element with any process width, the required comparators are reduced in comparison with the previous work. Also, the performance of the proposed architecture is nearly equal to the previous work. Therefore, the proposed string matching architecture reduces the hardware cost with the same performance of the previous work.

ACKNOWLEDGMENT

This work was supported by the R&D program of MKC/KEIT. [2009-S-043-01 Development of Scalable Micro Flow Processing Technology]

REFERENCES

[1] R.S. Boyer and J.S. Moore, "A fast string searching algorithm," Communication of the ACM, vol. 20, no. 10, pp.762–772, October 1997.

[2] D.E. Knuth, J.H. Morris and V.R. Pratt, "Fast pattern matching in strings," SIAM Jounal of Computing, vol. 6, no. 2, pp. 323–350, June 1977.

[3] A. Aho and M. Corasick, "Efficient string matching: An aid to bibilographic search," Communications of the ACM, vol. 18, no. 6, pp. 333–343, June 1975.

[4] L. Tan, B. Brotherton and T. Sherwood, "Bit-split string-matching engines for intrusion detection and prevention," ACM Trans. on Architecture and Code Optimization, vol. 3, no. 1, pp. 3–34, March 2006.

[5] S. Dharmapurikar and J.W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," IEEE Journal on Selected Area in Communications, vol. 24, no.10, pp.1781–1792, October 2006.

[6] Y.K. Chang, M.L. Tsai and Y.R. Chung, "Multi-character processor array for pattern matching in network intursion detection system," Proc. of Advanced Infromation Networking and Applications, pp. 991–996, March 2008.

[7] Snort, intrusion detection system, Visist http://www.snort.org.

[8] Boost graph library, Visit http://www.boost.org.