A Hardware-Based String Matching Using State Transition Compression for Deep Packet Inspection

HyunJin Kim and Seung-Woo Lee

This letter proposes a memory-based parallel string matching engine using the compressed state transitions. In the finite-state machines of each string matcher, the pointers for representing the existence of state transitions are compressed. In addition, the bit fields for storing state transitions can be shared. Therefore, the total memory requirement can be minimized by reducing the memory size for storing state transitions.

Keywords: String matching, intrusion detection system, deep packet inspection, pattern mapping.

I. Introduction

In the deep packet inspection (DPI), a string matching engine is necessary to detect target patterns from packet payloads [1]. Due to the increase in the number of target patterns and wire speed, multiple hardware-based string matchers can be adopted, wherein the matches with target patterns are recognized in parallel. The memory-based deterministic finite automaton (DFA) provides both regularity and scalability in hardware-based string matching [2]. However, memory requirements are proportional to 2^n when the input consists of n bits, as shown in the Aho-Corasick algorithm [3]. To implement the memory-based DFA with a reasonable size, the number of state transitions should be reduced. The bit-split string matching engine [4] and its extensions [5], [6] adopt homogeneous finite-state machine (FSM) tiles in hardware-based string matchers, in which a character input symbol is split into multiple input bit position groups for each FSM tile. Therefore, the total number of state transitions for each state is reduced. However, the number of state transitions toward noninitial states is increased due to the small size of the input symbol for each FSM tile. In addition, because each FSM tile should have its own match vector table, the memory requirements for storing match vectors are great. To minimize the memory requirements for state transitions, the hardware-based bit-split string matcher in [7] does not store the state transitions toward the initial state. However, due to the increase in the memory requirements for storing the existence of state transitions, this architecture is limited to the bit-split string matching.

This letter proposes a memory-based parallel string matching engine that compresses the information of state transitions. Firstly, by adopting the proposed transition existence tables, the pointers for representing the existence of state transitions are compressed in the FSM of each string matcher. In addition, by sharing state transitions, the memory size for storing the state transition table can be reduced.

II. Proposed String Matcher Architecture

The string matching engine has multiple homogeneous string matchers to detect target patterns in parallel. Figure 1 illustrates the proposed string matcher architecture. The number in the angle bracket of the first row represents the width of entries according to the predetermined number of entries in each table. After classifying packet data, payload data can be inputted at each cycle, where the state transition for the input data can be found in a pipelining fashion. In this case, the memory blocks are accessed in order.

The most significant bit (MSB) transition table and the match vector lookup table are shown in Fig. 1(a). The state pointer indicates the present state between S states. The s-th

Manuscript received Apr. 21, 2012; revised July 9, 2012; accepted July 19, 2012.

This research was supported by the KCC (Korea Communications Commission), Korea, under the R&D program supervised by the KCA (Korea Communications Agency) (KCA-2012-12-921-05-001).

HyunJin Kim (phone: +82 31 8005 3636, hyunjin2.kim@gmail.com) is with the Department of Electronics and Electrical Engineering, Dankook University, Yongin, Rep. of Korea.

Seung-Woo Lee (beewoo@etri.re.kr) is with the Communications Internet Research Laboratory, ETRI, Daejeon, Rep. of Korea.

http://dx.doi.org/10.4218/etrij.13.0212.0165

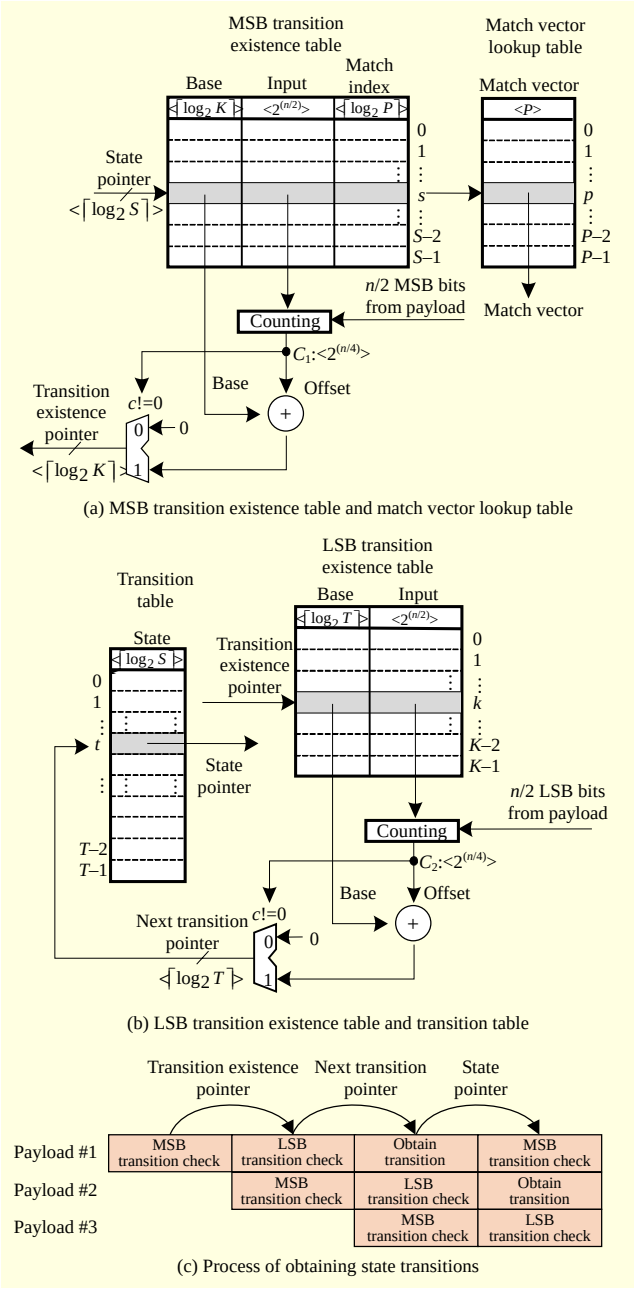

Fig. 1. Architecture of proposed string matcher.

entry in the MSB transition table contains a match index, which indicates a unique match vector in the match vector lookup table. To recognize the matches with unique patterns, the required number of entries in the lookup table is set as the number of bits in a match vector, *P*. As in [4]-[7], by only observing each bit in one match vector, the related pattern match can be recognized, which is different from the two-stage string matching in [8], in which multiple match vectors are searched to recognize each pattern match.

The *s*-th entry contains the base address in the base field for generating the transition existence pointer. The transition existence pointer indicates the *k*-th entry of the least significant bit (LSB) transition existence table. The base address means the starting address in the LSB transition table for the *k*-th entry. The *m*-th bit in the input field represents the existence of the state transitions toward a noninitial state when the value of *n*/2 MSBs in an *n*-bit input symbol is *m*–1. Assume that the state of the *s*-th entry has only one state transition toward a noninitial state when the value of the input symbol is *b*'00110011, in which the italic *b* means the binary number notation. For the state, the input field contains *b*'0000 0000 0000 1000 because the MSBs are three or *b*'0011. In Fig. 1(a), the counting value C_1 can be obtained by counting the number of "1" bits from the first low-order bit to the bit position according to the *n*/2 MSBs; the counting value is used for the offset to calculate the transition existence pointer. For example, when a state has two state transitions for the inputs with *b*'1000 and *b*'0011 MSBs, an input field can be *b*'0000 0001 0000 1000. For the two inputs, the counting value C_1 is set as *b*'0010 and *b*'0001, respectively; when the MSBs of the inputs are not *b*'0010 and *b*'0001 because the type of bit in the input field for the MSBs is "0," the counting value C_1 is set as *b*'0000. If C_1 is not zero, the transition existence pointer is calculated with the addition of the base address and offset. In this case, the state transition toward noninitial states exists for the MSBs of the input symbol. If C_1 is zero, the transition existence pointer is set as zero to represent that there are no state transitions toward noninitial states.

As shown in Fig. 1(b), the next transition pointer indicates the *t*-th entry of the transition table. An entry contains the base address in the base field for generating the next transition pointer. The base address means the starting address in the transition table for the entry. In the input field, each bit represents the existence of the state transitions toward the initial state for the *n*/2 LSBs of the *n*-bit input symbol from the payload, which is similar to the structure of the MSB transition existence table. The counting value C_2 is obtained by counting the number of "1" bits from the first low-order bit to the bit position according to the *n*/2 LSBs. In the LSB transition table, C_2 is used for the offset to calculate the next transition pointer; if C_2 is not zero, the next transition pointer is calculated with the addition of the base address and offset. In this case, the state transition toward noninitial states exists for the LSBs of the input symbol. When C_2 is zero, the next transition pointer is set as zero to represent that there are no state transitions toward noninitial states. An entry in the transition table contains the state pointer to indicate the entry of the MSB transition existence table. The process of obtaining state transitions is summarized in Fig. 1(c), in which there are three steps for obtaining the next state pointer.

Considering the parameters for the size of the memory blocks, the memory requirements of a string matcher are given as

$$S \cdot (\log_2 K + 2^{\frac{n}{2}} + \log_2 P) + P^2 + T \cdot \log_2 S + K \cdot (\log_2 T + 2^{\frac{n}{2}}), \quad (1)$$

where the other logics for calculating counting values are not considered due to their negligible size.

III. Sharing State Transitions and Pattern Mapping

The predetermined number of state pointers to be stored in the transition table of Fig. 1(b) can be reduced by sharing state transitions. By splitting the entries according to the LSBs of the input symbol, the state transitions toward noninitial states can be shared. In other words, if two states can share the same state transitions toward noninitial states for the LSBs for the input symbol from the payload, the state transition pointers in the transition table are shared. To describe the state transitions to be shared, an example of DFA for patterns {*he, she, his, hers*} is illustrated in Fig. 2, in which state s_0 is the initial state. The states represented in gray are the output states in which patterns are matched. The dashed line indicates failing pointers for noninitial states. The failing pointers for the initial state are not shown for clarity. In states s_4, s_6, s_8, and s_9, if the input is a character of "s," the next state can be s_7. In this case, the LSB of "s" is b'0011 for all four states (s_4, s_6, s_8, and s_9). Therefore, the generated next transition pointers can indicate the same entry in the transition table for the state transition; the state pointers can be shared. To share the state transitions between states in the transition table, the values in the base and input fields of one state in the LSB transition existence table should be identical with those in the other states and vice versa.

Several parameters limit the number of target patterns that are mapped onto each homogeneous string matcher. Firstly, the maximum number of states, *S*, and the number of bits in a match vector, *P*, should be considered. In addition, the number of entries in the transition table, *T*, limits the number of state transitions to be stored. Unlike the existing string matching architectures in [4]-[7], the predetermined number of entries in the LSB transition existence table, *K*, should also be considered.

Target patterns can be mapped onto a string matcher based on a predetermined order, one by one, until all target patterns are mapped, as shown in [4]. Initially, to map the first *P* target patterns among the unmapped target patterns, the mapping algorithm checks whether memory table contents can be obtained under the resource limitations by *S*, *K*, and *T*. If memory table contents can be obtained, the iteration is stopped; otherwise, the number of the adopted target patterns for the string matcher decreases by one, and the algorithm then checks whether memory table contents can be obtained. When the predetermined order is given, the time complexity can be O(*T*) to map all target patterns. A pattern length means the number of symbols in the pattern. The pattern length distribution and the number of state transitions that can be shared can be different according to characteristics of target patterns. Therefore, the optimal parameter values will be obtained by analyzing experiment results for real rule sets.

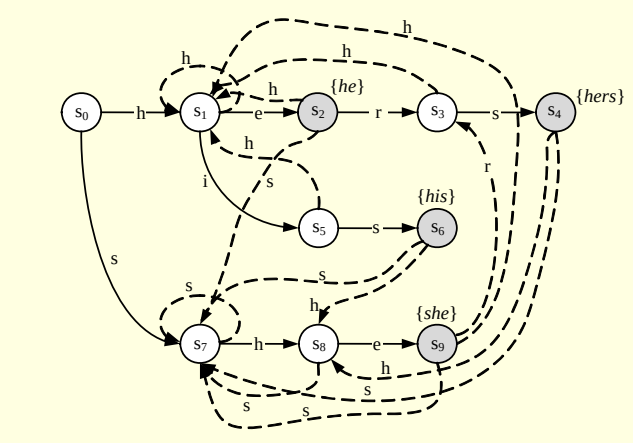

Fig. 2. Example of DFA for patterns {*he, she, his, hers*}.

Table 1. Total memory requirements by varying *S* and *P*.

S	128				256			
P	16	24	32	40	32	48	64	80
Backdoor	101	93	95	100	115	109	116	126
Deleted	87	84	87	92	93	95	101	110
Spyware-put	263	244	252	266	278	268	286	311
Web-misc	54	55	57	60	64	68	72	79

Unit of memory requirements above is kilobyte.

IV. Experiment Results

In our experiments, four large rule sets were extracted from Snort v2.8 rules [9]. Several parameters were swept to find their optimal values. The maximum number of states *S* was 128 after considering the maximum length of Snort rules. When *S* was 128, the number of bits in a PMV *P* was either 16, 24, 32, or 40; when the number of states *S* was 256, *P* was either 32, 48, 64, or 80; the maximum number of entries in the LSB transition existence table, *K*, and the maximum number of shared state transitions, *T*, were double *S*, respectively. In Table 1, the total memory requirements were shown by varying *P*. For the rule sets, the total memory requirements were minimized when *S* and *P* were 128 and 24, respectively. The memory requirements were not minimized by only increasing *P* over 24; therefore, there was a threshold point of *P* for minimizing memory requirements. In the evaluations in which *S* was 256,

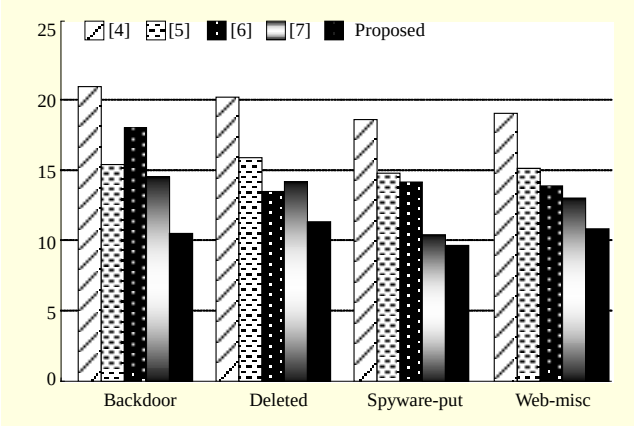

Fig. 3. Summary of comparisons in terms of normalized total memory requirements.

because the required K and P increased rapidly, the required number of string matchers was not decreased.

Then, by fixing S and P as 128 and 24, K was varied to find the optimal value; K was either 192, 256, 320, or 384 because K should be greater than S. Each state can have one or more than one state transition. In these evaluations, T was fixed as 256. As a result, when K was 256, the total memory requirements were greatly reduced for all rule sets. This was mainly due to the limited number of state transitions toward noninitial states in each string matcher. With the optimal K of 256, T was varied, that is, T was either 128, 192, 256, or 320. The obtained optimal T was 256. In these evaluations, because state pointers can be shared in the transition table, the required T was small. Considering the optimal values of K and T, it is concluded that many state pointers can be shared in the transition table in the proposed string matcher.

With the optimized parameters, Fig. 3 shows the summary of comparisons in terms of the normalized total memory requirements. The normalized total memory requirements of a rule set were obtained after dividing total memory requirements by the total sum of bytes in the unique target patterns of the rule set. In the proposed string matching architecture, the normalized memory requirements were 9.4 to 11.4 (bytes/character). Even though the memory requirements in [8] were 6.1 to 8.6 (bytes/character), the string matching architecture in the multiple match vectors should be searched to recognize each pattern. Therefore, the existing string matching architectures in [4]-[7] were evaluated because the string matching architectures provided only one match vector to recognize matched patterns like the proposed string matcher. In this comparison, the pattern mapping based on the general lexicographical sorting in [4] was adopted for all string matching architectures. For all adopted rule sets, the total memory requirements were decreased on average by 46.4%, 31.1%, 28.1%, and 18.4%, compared with those of [4]-[7], respectively. In addition, in [10], resources were reduced up to 40% compared to those in the traditional DFA-based method. Comparing the proposed string matching to [4] in terms of memory requirements, it is asserted that the reduction achieved proves that the proposed string matching can provide better performance than that shown in [10].

V. Conclusion

This letter proposed a memory-efficient parallel string matching engine in DFA-based string matching. The proposed string matcher can reduce the memory size for storing the existence of state transitions. In addition, the memory requirements can be reduced by sharing state transitions in the transition table. Considering the experiment results, it is evident that the proposed architecture is useful for reducing the storage cost of the DFA-based string matching engine.

References

[1] A. Peravi and M.J. Rahimzadeh, "A Novel Scalable and Storage-Efficient Architecture for High Speed Exact String Matching," *ETRI J.*, vol. 31, no. 5, Oct. 2009, pp. 545-553.

[2] P.-C. Lin et al., "Using String Matching for Deep Packet Inspection," *IEEE Computer*, vol. 41, no. 4, 2008, pp. 23-28.

[3] A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, vol. 18, no. 6, 1975, pp. 333-340.

[4] L. Tan, B. Brotherton, and T. Sherwood, "Bit-Split String-Matching Engines for Intrusion Detection and Prevention," *ACM Trans. Architecture Code Optimization*, vol. 3, no. 1, Mar. 2006, pp. 3-34.

[5] P. Piyachon and Y. Luo, "Compact State Machines for High Performance Pattern Matching," *Proc. IEEE Design Autom. Conf.*, 2007, pp. 493-496.

[6] C.-H. Lin, Y.-T. Tai, and S.-C. Chang, "Optimization of Pattern Matching Algorithm for Memory Based Architecture," *Proc. 3rd ACM/IEEE Symp. Architecture Netw. Commun. Syst.*, 2007, pp. 11-16.

[7] H. Kim et al., "A Memory-Efficient Pattern Matching with Hardware Based Bit-Split String Matchers for Deep Packet Inspection," *IEICE Commun. Lett.*, vol. E93-B, no. 2, Feb. 2010, pp. 396-398.

[8] H. Kim, H.-S. Kim, and S. Kang, "A Memory-Efficient Bit-Split Parallel String Matching Using Pattern Dividing for Intrusion Detection Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 11, Nov. 2011, pp. 1904-1911.

[9] Snort, Network Intrusion Detection System. http://www.snort.org

[10] A. Pandey et al., "Efficient Design and Implementation of DFA Based Pattern Matching on Hardware," *IJCSI*, vol. 9, issue 2, no. 1, Mar. 2012, pp. 286-290.