A Pattern Group Partitioning for Parallel String Matching using a Pattern Grouping Metric

HyunJin Kim and Sungho Kang, *Member,IEEE*

***Abstract*—Considering the increasing number of target patterns for the intrusion detection systems (IDS), memory requirements should be minimized for reducing hardware overhead. This paper proposes an algorithm that partitions a set of target patterns into multiple subgroups for homogeneous string matchers. Using a pattern grouping metric, the proposed pattern partitioning makes the average length of the mapped target patterns onto a string matcher approximately equal to the average length of total target patterns. Therefore, the variety of target pattern lengths can be mitigated because the number of mapped target patterns onto each string matcher is balanced.**

***Index Terms*—Computer network security, finite state machines, site security monitoring, and string matching.**

I. Introduction

As the hazardous attacks vary increasingly in the network environments, IDSs have been adopted in order to protect malicious packet payloads. Abnormal packet payload content consists of a bundle of binary codes or characters. For detecting the malicious packet payloads, target patterns should be identified in IDSs; therefore, string matching engine is a key device for the identification of target patterns in real time. As the number of target patterns increases with the variety of hazardous packet payloads, multiple string matchers can be adopted to identify the target patterns in parallel [1].

In particular, deterministic finite automaton (DFA)-based string matching could provide regularity with a memory-based string matching engine, where the throughput could be maintained regularly due to the fixed latency of state transitions. However, memory requirements can be proportional to the number of states and the number of bits in each next state pointer. Therefore, by adopting homogeneous multiple string matchers where DFAs are mapped, the memory requirements can be reduced with keeping scalability and updatability.

On the other hand, the variety of target pattern lengths can be a challenge for reducing memory requirements. Pattern length is defined as the number of characters or byte codes in a target pattern. For the homogeneous string matchers, the number of target patterns mapped onto each string matcher could be different due to the different target pattern lengths. In this case, even though the homogeneity of the string matchers increases scalability and regularity, memory usage is not efficient when adopting multiple string matchers.

Manuscript received December 3, 2009. The associate editor coordinating the review of this letter and approving it for publication was M. Ma.

H. Kim is with the Flash Solution Development Team, Memory Division, Samsung Electronics, Hwasung-City, 445-701, Korea.

S. Kang is with the Computer Systems and Reliable SoC Lab., Department of Electrical and Electronic Engineering, Yonsei University, Seoul, 120-749, Korea (e-mail: shkang@yonsei.ac.kr).

Digital Object Identifier 10.1109/LCOMM.2010.080210.092347

This paper proposes an algorithm that partitions a set of target patterns into multiple subgroups for homogeneous memory-based parallel string matchers. The proposed pattern partitioning adopts a pattern grouping metric in order to obtain the multiple subgroups. The proposed algorithm iteratively maps each target pattern with the ordered list of target patterns based on the pattern grouping metric. By applying the metric, the average length of each subgroup's target patterns is approximately equal to the average length of total target patterns. Therefore, the number of mapped target patterns onto each string matcher is balanced. In addition, the problem of the variety of target pattern lengths could be mitigated.

II. Target Architecture

The target architecture is based on a memory-based string matching with homogeneous string matchers. In a string matcher, N homogeneous finite state machine (FSM) tiles are contained. An FSM tile contains a maximum of s states and takes n bits of one character at each cycle. Target patterns are distributed and mapped onto C string matchers. Each state has 2^n pointers for the next state based on an n-bit input. The number of bits in a pointer is determined as $\lceil log_2 s \rceil$. The address of a state indicates a PMV, where the number of bits in a PMV p refers to the maximum number of mapped target patterns onto a string matcher. In a PMV, the i-th bit represents whether the i-th pattern is matched in the state. When an FSM takes one byte input, the string matcher supports the Aho-Corasick algorithm [2] with one FSM; otherwise, the bit-split string matching in [3] can be applied with multiple FSM tiles. A matched pattern is recognized with full match vectors (FMVs), which are obtained with the logical AND operation of partial match vectors (PMVs) from all FSM tiles in a string matcher.

III. Proposed Pattern Partitioning

A. Pattern Grouping Metric

The proposed pattern partitioning concentrates on balancing the average target pattern length for each string matcher; however, in [3] and [4], the shared common prefixes increased by applying lexicographical or gray-code based sorting. For example, let us assume that a string matcher has an FSM with the maximum of ten states and eight-bit input, where one state is reserved for the initial state. In addition, "add," "bolding," "caused," and "do" are assumed the lexicographically ordered target patterns. In the pattern partitioning based on the lexicographical sorting in [3], patterns "add" and "bolding" cannot be mapped onto a string matcher because a DFA for patterns "add" and "bolding" requires eleven states. Therefore, only the target pattern "add" can be mapped onto the first string

1089-7798/10$25.00 © 2010 IEEE

```
FUNCTION Get_List(Pattern Grouping Metric m, Target Patterns T)
  SORT T by descending target pattern length;
  SET ungrouped target patterns t to T;
  SET a list of target patterns L to φ;
  FOR π = 1 to T
    IF Length(L) > π · m THEN
      FIND a target pattern b with the maximized length in t
        satisfying { b | Length(L) + Length(b) ≤ (π+1) · m };
      IF b == φ THEN
        SET b to the target pattern with the minimum length in t;
      ENDIF
    ELSE // IF Length(L) ≤ π · m
      FIND a target pattern b with the minimized length in t
        satisfying { b | Length(L) + Length(b) > (π+1) · m };
      IF b == φ THEN
        SET b to the target pattern with the maximum length in t;
      ENDIF
    ENDIF
    PUSH BACK b to L;
    REMOVE b from t;
  ENDFOR
ENDFUNCTION with RETURNING the list L
```

Fig. 1. Pseudocode of obtaining the list of target patterns to be mapped.

```
FUNCTION Pattern Partitioning(Target Patterns T, String Matcher K)
  SET pattern grouping metric m to Avg(T);
  SET a list of DFAs V to φ;
  CALL Get_List(m, T) RETURNING a list L;
  WHILE (L ≠ φ)
    FOR π = p (K) to 1
      CALL Build_Tries with front π target patterns from L
        with RETURNING tries;
      IF Max(Num_states(tries)) > s (K) THEN CONTINUE;
      ELSE REMOVE π target patterns from L; BREAK;
      ENDIF
    ENDFOR
    CALL Build_DFAs(tries) RETURNING DFAs
    PUSH BACK DFAs to V;
  ENDWHILE
ENDFUNCTION with RETURNING the list of DFAs V
```

Fig. 2. Pseudocode of the proposed pattern partitioning.

matcher. In addition, patterns “bolding” and “caused” cannot be mapped onto the same string matcher. Therefore, three string matchers are required. On the other hand, the proposed pattern partitioning adopts the average length of total target patterns for the pattern grouping metric. The pattern grouping metric, therefore, is equal to the average length of the total target patterns. In this example, the pattern grouping metric is 4.75. In order to estimate worst-case number of mapped target patterns, the maximum number of states in the FSM is divided by the average target pattern length. By pairing target patterns “add” and “caused” for the first string matcher and “bolding” and “do” for the second string matcher, a balance can be achieved between string matchers in terms of the number of used states in the FSM and the average target pattern length for each string matcher.

The list of target patterns to be mapped is obtained by applying the pattern grouping metric. The pseudocode of obtaining the target pattern list is described in Fig. 1; the function *Get_List* adopts two input parameters: the pattern grouping metric **m** and the set of target patterns **T**. By repeating loops for ordering target patterns, the average length of target patterns in sequence could be set approximately equal to the value of the pattern grouping metric. The function of $Length(\mathbf{L})$ means the total sum of characters for target patterns in a list **L**. The most appropriate target pattern is determined in each turn by considering $Length(\mathbf{L})$.

B. Pattern Partitioning using Pattern Grouping Metric

With the ordered list based on the pattern grouping metric, the proposed pattern partitioning determines the target patterns mapped onto each string matcher iteratively. The pseudocode of the proposed pattern partitioning is shown in Fig. 2. First, the pattern grouping metric **m** is calculated by averaging the total target pattern lengths. Then, the function *Get_List* is called with the purpose of obtaining the list of target patterns to be mapped. In the inner loop of **FOR**, the maximum number of mapped target patterns onto each string matcher, $p(\mathbf{K})$, is adopted for the first iteration. For the determined front target patterns of the list **L**, a function *Build_Tries* is called in order to build tries. In this case, the failing pointer addition is not performed, so that the processing time can decrease. For a string matcher, if the maximum number of states among the obtained DFAs is greater than the maximum number of states available in an FSM tile $s(\mathbf{K})$, the target patterns could not be mapped onto the string matcher. Therefore, the number of target pattern to be mapped decreases by one, and then the inner loop is repeated. If the generated DFAs could satisfy the hardware resource limitation of $s(\mathbf{K})$, the inner loop is broken. In the function *Build_DFAs*, the Aho-Corasick algorithm is applied to the obtained tries in order to add failing pointers [3]. Then, the DFAs for a string matcher are stored in the list of DFAs **V**. In addition, the pattern partitioning for another string matcher is repeated until there are no remaining unmapped target patterns.

Like the pattern mapping in [3] and [4], the number of bits in a PMV p and the number of states available in an FSM tile s are predetermined irrespective of the number of total target patterns T. Therefore, considering the **WHILE** loop in Fig. 2, the pattern partitioning shows the linear complexity of $O(T)$. On the other hand, the time complexity of obtaining the ordered target pattern list can be $O(Tlog_2T)$, which is obtained based on the complexity of **SORT** and **FIND** functions in Fig. 1. Considering to the large constant factor of the pattern partitioning in time complexity, the complexity of obtaining the ordered list could not be dominant.

IV. Experimental Results

For the purpose of evaluating the proposed algorithm, a set of total target patterns, *total*, was extracted from Snort v2.8 rules [5]. The set *total* contained 7784 unique target patterns, where the average length was 18.6 (char/pattern). For the apples-to-apples comparisons, several existing pattern partitioning approaches introduced in [3] were implemented. Along with the lexicographical order, two additional cases with random order and original order without sorting were adopted, which were denoted as *lexical*, *random*, and *origin*, respectively. In addition, the gray-code based sorting in [4], which was denoted as *group*, was compared. In the existing approaches above, target patterns were partitioned sequentially according to their own pattern ordering methods; in the proposed pattern partitioning, with the list of target patterns based on the pattern grouping metric, target patterns were partitioned sequentially. The target architectures based on the Aho-Corasick algorithm and bit-split string matching were evaluated. For the bit-split string matching, each FSM tile took a two-bit input based on the design analysis in [3].

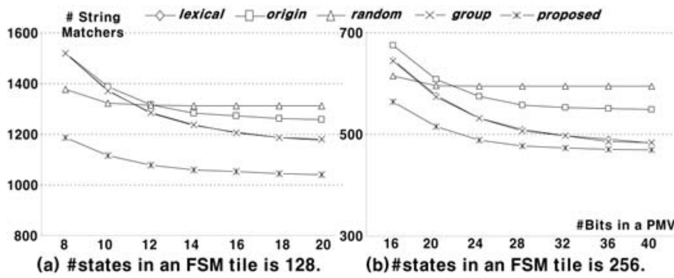

Fig. 3. Performance comparisons for the Aho-Corasick algorithm.

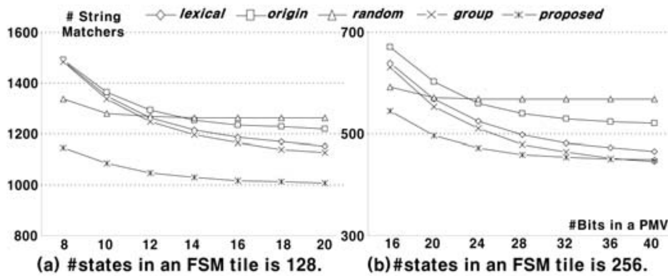

Fig. 4. Performance comparisons for the bit-split string matching.

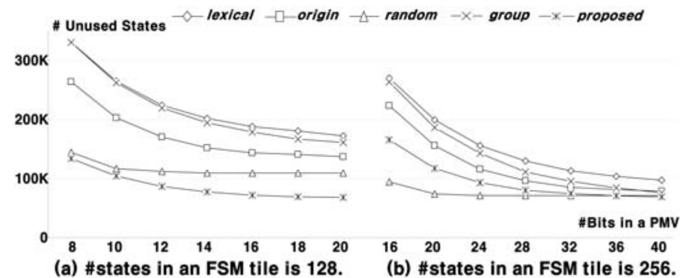

Fig. 5. Resource usage comparisons for the bit-split string matching.

Fig. 3 and Fig. 4 illustrate the performance comparisons for the Aho-Corasick algorithm and bit-split sting matching in terms of the number of adopted string matchers. Considering the average length of total target patterns, the numbers of bits in a PMV were varied from 8 to 20 and from 16 to 40 when the numbers of states in an FSM were 128 and 256, respectively. In Fig. 3, the number of adopted string matchers was reduced on average by 10.0%-17.1%, compared with other pattern partitioning approaches. In the evaluation for the bit-split string matching of Fig. 4, the number of adopted string matchers was reduced on average by 10.5%-17.2%. In particular, when the number of bits in a PMV was small and the number of states in an FSM was 128, the number of adopted string matchers was greatly reduced. This means that the proposed pattern partitioning could be more efficient when the hardware resource was tightly limited. When the number of states in an FSM was 256, *lexical* and *group* decreased the number of adopted string matchers greatly with the number of bits in a PMV. This means that the shared prefixes increased due to the increasing number of states in an FSM. However, because the number of states in an FSM was limited, the number of adopted string matchers did not decrease linearly with the increasing number of bits in a PMV.

In order to know the resource usage efficiency of the proposed algorithm, the resource usage for the bit-split string matching engine was evaluated in terms of the number of unused states in all FSM tiles in Fig. 5. The number of the unused states was reduced on average by 34.7%-49.2%, in comparison with *lexical*, *origin*, and *group*. Even though the number of unused states was small in the case of *random*, the number of adopted string matchers was greater, compared to the case of *proposed* in Fig. 3. This means that the number of mapped target patterns onto each string matcher was not balanced in *random*. In particular, the numbers of unused states for *lexical* and *group* were greater than those of other approaches, which means that the numbers of shared common prefixes in *lexical* and *group* could be greater than those of other approaches. In both cases, however, the decreasing number of adopted string matchers was not linearly proportional to the increasing shared common prefixes. This means that in *lexical* and *group*, the decrease in the number of adopted string matchers was limited mainly due to the variety of target pattern lengths and the pattern distributions for each string matcher. Even though the number of shared common prefixes was small, the proposed pattern partitioning decreased the number of unused states by increasing the number of mapped target patterns. Therefore, the proposed algorithm could mitigate the problem of the variety of target pattern lengths.

V. Conclusion

By adopting the pattern grouping metric, the proposed pattern group partitioning decreases the number of adopted string matchers by balancing the numbers of mapped target patterns between string matchers. For the tightly limited hardware resource, the number of adopted string matchers decreases significantly without any additional hardware. Considering the enhanced performance and the complexity, the proposed pattern partitioning is useful for reducing hardware cost for the DFA-based parallel string matching engine.

References

[1] P.-C. Lin, Y.-D. Lin, T.-H. Lee, and Y.-C. Lai, "Using string matching for deep packet inspection," *IEEE Computer*, vol. 41, no. 4, pp. 23-28, 2008.

[2] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333-340, 1975.

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

[4] H. Kim, H. Hong, H.-S. Kim, and S. Kang, "A memory-efficient parallel string matching for intrusion detection systems," *IEEE Commun. Lett.*, vol. 13, no. 12, pp. 1004-1006, Dec. 2009.

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