An iterative pattern mapping for parallel string matching architecture in intrusion detection systems

HyunJin Kim

The authors are associated with the Dept. of Electronics and Electrical Eng., Dankook University, Yongin-si, Gyeonggi-do, Korea

a) hyunjin2.kim@gmail.com

Abstract: This paper proposes an algorithm that maps target patterns onto parallel string matching architectures in intrusion detection systems (IDS). In the proposed iterative pattern mapping, the sets of patterns that are mapped onto string matchers are sorted in ascending order of the average pattern length in each turn. By mapping a set of patterns for a string matcher onto the other string matchers repeatedly, the required number of string matchers is reduced. Therefore, the proposed iterative pattern mapping minimizes the total memory requirement for parallel string matching architecture.

Keywords: string matching, finite-state machine, deterministic finite automaton, pattern mapping, intrusion detection system, deep packet inspection

Classification: Integrated circuits

References

1 Introduction

In IDSs, the string matching engine is an essential device that detects hazardous payload in real-time. To deal with the increase in the number of target patterns, multiple string matchers are used to recognize target patterns in parallel. Due to the transition regularity and deterministic transition time [1], homogeneous deterministic finite automata (DFA)-based string matchers are generally adopted. In the parallel string matching architecture, the memory requirements can be decreased by reducing the required number of string matchers. However, because the existing pattern mapping approaches in [2, 3, 4, 5] were based on the determined order, the required number of string matchers was not fully optimized.

This paper proposes an iterative pattern mapping for the parallel string matching architecture. With the mapping results in each turn, the sets of patterns that are mapped onto string matchers are sorted in ascending order of the average pattern length. With the sorted sets, the patterns for a string matcher is mapped onto other string matchers; the need to the string matcher, therefore, can be eliminated. By repeating these procedures, the required number of string matchers is reduced.

2 Target string matching architecture

The proposed pattern mapping algorithm can be applied to the DFA-based parallel string matching architecture; in the string matcher using the Aho-Corasick algorithm [1], memory requirements were proportional to $2^a$, when the input consists of $a$ bits. The bit-split pattern matching, which was originated from [2], reduced the total number of state transitions for each state greatly by adopting homogeneous finite-state machine (FSM) tiles in a string matcher. Therefore, the bit-split string matching is considered in the proposed pattern mapping. The general description of the bit-split DFA-based string matcher is shown as follows: each homogeneous FSM tile takes $n$ bits of one character at each cycle. Each state in an FSM tile has $2^n$ pointers for the next state. For the $s$ states of each FSM tile, pattern identifications are stored as a partial match vector (PMV), where the number of bits in a PMV, $p$, limits the number of mapped target patterns. A matched pattern is recognized by performing a bitwise AND operation of the PMVs from all of the FSM tiles. When the number of states in the DFA for an FSM tile is smaller than $s$, several states are unused, which are denoted as unused states.

3 Proposed pattern mapping

In the existing pattern mapping approaches of [2, 3, 4, 5], the target patterns are mapped based on the predetermined order; if a pattern cannot be mapped onto a string matcher, it is mapped onto another new empty string matcher. For example, let us assume that a string matcher has an FSM with eight input bits and eight states. In this case, one state in each FSM is reserved for the initial state. In addition, “abc”, “bcd”, “bcdef”, and “defg” are assumed to be a set of lexicographically sorted target patterns. After mapping the
target patterns on the pattern mapping approaches in [2, 3, 4], three string
matchers are required as follows:

\[
\{ “abc”, “bcd” \}, \{ “bcdef” \}, \{ “defg” \}, \quad (1)
\]

where the patterns between braces are mapped in a string matcher. The
number of character codes in each pattern is defined as the pattern length.
In [5], by making the average length of the mapped target patterns onto
a string matcher approximately equal to the average length of total target
patterns, the mapping result is given by:

\[
\{ “abc” \}, \{ “bcd”, “defg” \}, \{ “bcdef” \}. \quad (2)
\]

In general, when the lengths of patterns mapped onto a string matcher
are shorter, the possibility of mapping the patterns onto the other string
matchers can increase. To decide the patterns of one string matcher to be
mapped onto the other string matchers, the proposed algorithm sorts the
sets of patterns that are mapped onto string matchers in ascending order of
the average pattern lengths. For example, in the mapping result in (1), the
average pattern length of the patterns for the first string matcher is three; on
the other hand, the average pattern lengths for the second and third string
matchers are five and four, respectively. In the next iteration, if patterns
“abc” and “bcd” are mapped onto the third and second string matchers,
respectively, the target patterns can be mapped by:

\[
\{ “abc”, “defg” \}, \{ “bcd”, “bcdef” \}, \quad (3)
\]

where the subpattern “bcd” is shared in the second string matcher. As a
result, the need to the first string matcher can be eliminated.

Fig. 1 shows the pseudocode for the proposed iterative pattern mapping;
the function \texttt{Build Lexicographical DFAs} constructs initial DFAs for all
of the string matchers, \texttt{tries}, based on the lexicographical sorting in [2].
With the list of DFAs \texttt{tries} in each turn of the outer loop, the function
\texttt{Sort Pattern Lengths} returns a sorted list of DFAs \texttt{D} and sets of patterns
\texttt{L} in ascending order of the average pattern length. In the \texttt{FOR} statement
of the inner loop, the function \texttt{Build Merged Tries} is called to obtain a
list of DFAs \texttt{E} after mapping the \texttt{i}-th set of patterns in \texttt{L}, \texttt{U}, onto the string
matchers with the DFAs of \texttt{D}. If the maximum number of states in the DFAs
is larger than the maximum number of states in an FSM tile, \texttt{s}, the inner
loop is continued; otherwise, \texttt{tries} is set to a list of DFAs \texttt{E}, and then the
inner loop is broken. After exiting the inner loop of \texttt{FOR} statement, if the
number of DFAs in \texttt{tries} is not changed, the iterative mapping algorithm
finishes the outer loop of \texttt{REPEAT}; otherwise, the outer loop is repeated
to find the solution with the reduced number of DFAs.

Considering the time complexity of \(O(T)\) in [2], the iterative pattern map-
ing algorithm requires the time complexity of \(O(T^3)\). However, compared
to the number of target patterns, \(T\), the iterations in the time complexity
will not be dominant due to the limited number of iterations.
4 Experimental results

In order to evaluate the proposed algorithm, a set consisting of 7784 unique target patterns, total, was extracted from Snort v2.8 [6]. In total, the average and maximum pattern lengths were 18.6 (char/pattern) and 122 (char/pattern), respectively. For the apple-to-apple comparisons, the pattern mapping using the lexicographical pattern order [2], which was denoted as lexical, was implemented. In addition, the pattern mapping approaches with the random sorting, gray code-based sorting in [3], and greedy search [4] were evaluated, which were denoted as random, gray, and heuristic, respectively. In addition, the pattern mapping that balanced the number of mapped patterns between string matchers [5] was denoted as metric. Based on the design analysis in [2], an FSM tile was assumed to take two bits of one character as its input. Considering the maximum and average target pattern lengths in total, the number of states in an FSM was set as 128; the number of bits in a PMV was increased from eight.

Table I summarizes the performance comparisons in terms of the required number of string matchers. The number of adopted string matchers was reduced on average by 34%, 36%, 32%, 12%, and 8%, in comparison with lexical, random, gray, metric, and heuristic, respectively. Unlike random, the proposed pattern mapping decreased the required number of string matchers by increasing the number of bits in a PMV, \( p \). On the other hand, the performance enhancements over lexical, gray, metric, and heuristic were decreased by increasing \( p \). This is due to the fact that possibility of mapping patterns in one string matcher onto the other string matchers was increased by decreasing the number of bits in a PMV.

FUNCTION iterative_Mapping(Target Patterns T, String Matcher K)
    CALL Build_Lexicographical_DFAs(T, K)
    RETURNING a list of DFAs tries;
REPEAT
    CALL Sort_Pattern_Lengths(tries)
    RETURNING a sorted list of DFAs D and sets of patterns L;
    FOR \( \pi \) = one to Num_set(L)
        SET U to \( i \)-th set of patterns in L;
        CALL Build_Merged_Tries(U, D, K)
        RETURNING a list of DFAs E;
        IF Max(Num_states(E)) > s(K) THEN
            CONTINUE;
        ELSE
            SET tries to E;
            BREAK;
        ENDIF
    ENDFOR
    UNTIL Num_dfa(tries) == Num_dfa(D)
ENDIF
ENDFUNCTION RETURNING the list of DFAs tries

Fig. 1. Pseudocode of pattern mapping.
Table I. Performance comparisons in terms of the required number of string matchers.

<table>
<thead>
<tr>
<th>#bits (PMV)</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexical</td>
<td>1488</td>
<td>1348</td>
<td>1263</td>
<td>1216</td>
<td>1186</td>
<td>1170</td>
<td>1153</td>
</tr>
<tr>
<td>random</td>
<td>1338</td>
<td>1280</td>
<td>1269</td>
<td>1263</td>
<td>1263</td>
<td>1263</td>
<td>1263</td>
</tr>
<tr>
<td>gray</td>
<td>1483</td>
<td>1338</td>
<td>1248</td>
<td>1197</td>
<td>1165</td>
<td>1140</td>
<td>1128</td>
</tr>
<tr>
<td>heuristic</td>
<td>1195</td>
<td>1088</td>
<td>1017</td>
<td>975</td>
<td>956</td>
<td>942</td>
<td>935</td>
</tr>
<tr>
<td>metric</td>
<td>1149</td>
<td>1086</td>
<td>1046</td>
<td>1030</td>
<td>1017</td>
<td>1011</td>
<td>1008</td>
</tr>
<tr>
<td>proposed</td>
<td>1025</td>
<td>957</td>
<td>934</td>
<td>923</td>
<td>910</td>
<td>914</td>
<td>912</td>
</tr>
</tbody>
</table>

Fig. 2 shows a comparison of the resource usage in terms of the total number of unused states. In random, the total number of unused states was not decreased as the number of bits in a PMV was increased. In this case, because the number of shared states could not increase, the required number of string matchers did not decrease with $p$. The total number of unused states was reduced on average by 215%, 88%, 207%, 45%, and 50%, in comparison with lexical, random, gray, metric, and heuristic, respectively. Therefore, it is concluded that the resource usage was efficient by applying the proposed iterative pattern mapping.

Considering the performance enhancements and efficient resource usage, the proposed iterative pattern mapping is useful for reducing the storage cost of the DFA-based parallel string matching engine.

Acknowledgement

The present research was conducted by the research fund of Dankook University in 2012.