Acceleration of Deep Packet Inspection Using a Multi-Byte Processing Prefilter

Hyejeong HONG†, Nonmember and Sunho KANG†, Member

SUMMARY Fast string matching is essential for deep packet inspection (DPI). Traditional string matchers cannot keep up with the continuous increases in data rates due to their natural speed limits. We add a multi-byte processing prefilter to the traditional string matcher to detect target patterns on a multiple character basis. The proposed winnowing prefilter significantly reduces the number of identity blocks, thereby reducing the memory requirements.

key words: computer network security, deep packet inspection, and string matching

1. Introduction

As data rates are increasing rapidly, along with the numbers of patterns to be detected, it is increasingly difficult to perform deep packet inspection (DPI) on all of the traffic traversing a given network. Software intrusion detection systems (IDSs) such as Snort [1] perform DPI well only when data rates are low. Recently, hardware IDSs have been widely studied to overcome the limitations. The key function of hardware IDSs is fast string matching. The most popular approach to building hardware string matchers is to use the automaton based upon the Aho-Corasick algorithm [2] due to its simple construction and high scalability [3]. However, it is not fast enough for practical use because it processes only one character at a time. Some studies have modified the construction of automata so that the string matchers process multiple characters simultaneously. The naive implementation of the multi-byte string matcher is to modify the automaton that moves between states by a single character so that it moves in steps of multi-byte blocks. Setting the step size at two-character blocks doubles the processing speed, but the memory space required is up to 256 times larger, which makes it infeasible to implement this solution with a string matcher.

String matching consists of traversing an automaton built with the patterns to be searched according to the incoming character. The traversing starts from the initial state, and remains there unless one of the starting characters of the patterns comes in. Because malicious packets rarely exist on a network, the string matching remains in the initial state in most cases. That is, states close to the initial state are frequently visited in the traditional Aho-Corasick automaton, whereas those far from the initial state are rarely visited. Based on these observations, the speed of traditional string matching can be increased affordably and simply by adding hardware resources in the form of an accelerator to process frequently visited states on a multiple-character basis. The accelerator inspects multiple characters simultaneously, and if no potential match is detected, the packets bypass the time-consuming process of traditional string matching. In [4], the first several characters of the patterns are used as identity blocks for the accelerator; however, the identity blocks must be chosen carefully if the implementation is to be practical.

This paper proposes an architecture to accelerate string matching for DPI that offers affordable hardware overhead. By using the Winnowing algorithm [5], the number of identity blocks that must be stored in the block tables is reduced considerably.

2. Proposed Multi-Byte String Matcher

Basically, the proposed string matcher consists of two parts in common with the string matcher in [4]: a prefilter, which accelerates string matching, and a traditional string matcher based on the Aho-Corasick algorithm. The prefilter includes table storing identity blocks, and inspects the incoming strings to determine whether they include the blocks. If the prefilter does not detect any identity blocks from the incoming string, then it slides the inspection window by multiple characters. Otherwise, it informs the traditional string matcher so that the string matcher can start to traverse its automata from the state which the detected block points.

2.1 Limits of the Head Block Prefilters

The overhead driven by the prefilter depends on the data structure of the block tables in the prefilter. Single table and multiple table methods to construct data structures were studied in [4]. Basically, both methods use head blocks (the first \( h \) characters of the patterns) as identity blocks. The difference between them is how the head blocks are stored. The single table simply stores all head blocks. In addition, in order to make the block matcher process \( s \) characters at a time, the single table must also cover the blocks that are sequences of \( s - 1 \) characters followed by the first \( (h - s + 1) \) characters of the target patterns. Figure 1 shows an example to clarify the difference between the tables. The target pattern set \( P \) has five patterns to be de-
represent two target patterns each so the five patterns can be covered by storing only three blocks. Therefore, the number of identity blocks decreases significantly.

The process of constructing the winnowing prefilter is described as follows. First, a hash table that includes hash values for all character sequences from the target patterns is generated. As an example, when the length of the sequence (l) is two, there are nine sequences in the pattern discussion: dis, is, sc, cu, us, ss, si, io, and on. Sequences are extracted from every pattern, and then the hash value for each sequence is generated. After the hash table is completed, the patterns to be searched are segmented into blocks. The hash values are used to choose the position where a new block will start in a pattern. When the rightmost minimum value among the hash values of w sequences is the nth hash value in a pattern, the first n characters form a block, and the (n+1)th character starts a new block.

When the incoming strings come into the winnowing prefilter, they are chopped by the same method to extract the identity blocks. If a target pattern exist in the incoming strings, the second block and beyond of the pattern will be always formed. However, the first block of the pattern can either be formed or not, depending on the character just before the pattern starts. As illustrated in Fig. 1, the pattern discussion is segmented into dis, cuss, and ion. When discussion is included in the incoming string, cussion is always segmented into cuss and ion. We cannot guarantee that the first block dis is formed, because the first character d can generate various hash values combining with the character before it. Thus, the second blocks of the patterns are used as the identity blocks rather than the first blocks. By doing that, the starting point of the identity blocks that the prefilter monitors is always fixed so there is no additional comparator, unlike the head block prefilter with the multiple table.

3. Evaluation

The prefilters were implemented using the C++. The target patterns were extracted from the three large rule sets in the Snort v2.8 rules [1]: netbios, backdoor, and web-client. It is meaningless to evaluate the number of identity blocks of the head block prefilter with the single table, as it is too large to implement. Table 1 shows the numbers of identity blocks and characters to be stored of the head block prefilter with the multiple table and the winnowing prefilter according to the rule sets.

The larger the length of the head blocks (h) is, the larger memory requirements are needed. h is set as either four or eight to compare the overhead to store the blocks. When h is eight, the number of identity blocks decreases for netbios and backdoor, because some patterns are too short to be detected by the prefilter. In the case of web-client, the number of identity blocks increases with h, because the majority of the target patterns are longer than h. s means the actual processing power of the head block prefilter with the multiple table and it has to be smaller than h. For the head
the performance of the two prefilters, we measured the number of identity blocks and the number of characters to be stored for the proposed winnowing prefilter. When \( w=4 \) and \( h=2 \), the number of identity blocks is 35% to 40% lower than where \( w=8 \) and \( h=2 \). According to the evaluation results in Table 2, there is no correlation between the ratio of the target patterns and the processing width of the winnowing prefilter. The processing power is improved when \( w \) is larger, as we expected. As described in Sect. 2.2, the strings coming into the winnowing prefilter are chopped by the same method to extract the identity blocks, and one chopped substring is processed every clock. That is, the number of characters in a substring is the major factor that determines the processing width of the winnowing prefilter. Based on the observation that the processing width of the prefilter varies from 2.242 to 2.548, it is inferred that the performance of the winnowing prefilter is higher than the head block prefilter where \( s=2 \), and lower than where \( s=3 \).

We compare the number of identity blocks of the head block prefilters whose processing powers are two \((s=2)\) and the winnowing prefilters. Figure 2 illustrates the number of identity blocks and the number of characters to be stored for the winnowing prefilter relative to those of the head block prefilter. The reduction of the identity blocks varies from 55% to 82% according to the rule set used. The reduction of the identity blocks and the number of characters for the

Table 2 Comparisons of the number of the processed characters per cycle in the head block prefilter with the multiple table and the proposed winnowing prefilter for the strings with target patterns.

<table>
<thead>
<tr>
<th>Ratio of target patterns</th>
<th>1%</th>
<th>2%</th>
<th>3%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head block prefilter ( h=4 ), ( s=2 )</td>
<td>1.972</td>
<td>1.944</td>
<td>1.916</td>
</tr>
<tr>
<td>w/ multiple table ( h=8 ), ( s=2 )</td>
<td>2.938</td>
<td>2.898</td>
<td>2.880</td>
</tr>
<tr>
<td>( h=8 ), ( s=3 )</td>
<td>1.976</td>
<td>1.886</td>
<td>1.787</td>
</tr>
<tr>
<td>( h=8 ), ( s=4 )</td>
<td>2.942</td>
<td>2.926</td>
<td>2.914</td>
</tr>
</tbody>
</table>

Winnowing prefilter \( w=4 \), \( l=2 \) | 2.304 | 2.276 | 2.242 |
| \( w=8 \), \( l=2 \) | 2.548 | 2.514 | 2.484 |

Fig. 2 The number of identity blocks and the number of characters to be stored for the proposed winnowing prefilter, when \( w=4 \) and \( h=2 \) relative to the head block prefilter with the multiple table in which \( h=4 \) and \( s=2 \).
The proposed winnowing prefilter speeds up the traditional string matcher by up to 2.548 and the memory requirements are significantly reduced, and therefore the proposed winnowing prefilter is suitable for implementation using CAMs.

4. Conclusion

This paper proposes accelerated string matching for DPI by introducing a multi-byte processing prefilter. The prefilter overcomes the speed limits of traditional string matching, which process only one character at a time, at the expense of some hardware overhead. The hardware overhead is minimized using the Winnowing algorithm to extract identity blocks. The number of identity blocks that must be stored in the prefilter is decreased by 85% for all target patterns extracted from Snort v2.8 rules.

References