An Efficient Automaton Based Matching Algorithm and Its Application in Intrusion Detection System

Peifeng Wang, Yue Hu, Li Li

School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing, China,
wwppff2004@163.com, huhuyue_001@sina.com, liliustb@126.com

Abstract

String matching is a classical problem in computer science. It's essential for network content security applications, and the technique is widely applied in the pattern recognition, spelling checking, intrusion detection, worm containment, information security and so on. In this paper, a new algorithm is proposed to construct smallest suffix automaton to achieve high performance and efficient string matching. The suffix automaton construction on a set of strings is similar to the AC automaton construction. However, the difference of the smallest suffix automaton appears when going down the supply path looking for an outgoing transition. Experimental results show that the presented algorithm outperforms existing solutions in most cases.

Keywords: string matching, automaton, intrusion detection, DFA, information security

1. Introduction

The string matching problem can be described as finding all occurrences of keywords form a given pattern \( P = p_1p_2...p_m \) in a given text \( T = t_1t_2...t_n \). In recent years, many new string matching algorithms have been proposed to deal with intrusion detection, antivirus and spam prevention. The existing string matching algorithms can be classified into five categories[1]: simple matching, dynamic programming, bit parallel, backward filtering, and automaton. This paper studies the basic concepts of automaton and automaton based string matching approach.

Multiple string matching algorithms are often formulated by finite-state automata. These algorithms require handling massive amount of symbolic data in real-life applications. Aho and Corasick[2] (AC) proposed a simple, efficient algorithm to locate all occurrences of any of a finite number of keywords in a string of text. They proved that the string matching problem can be solved in a linear time proportional to the string length. AC’s automaton often has large number of states that demand high memory. Navarro and Raffinot[3] covered an idea of a complete automaton approach for biological sequence alignment and presented charts showing the best choice of search algorithms based on the alphabet size and minimum patterns sizes. Nishimuar et al[4] reported that the states for the AC machine can be rearranged to achieve CPU cache efficiency. Bit-split method by Tan, et al [5] converted large database of strings into many tiny state machines for parallel executions. Most string matching algorithms are improved from existing ones for software implementation[6-9]. Since the throughput of hardware-based solutions is much higher than the software-based ones, FPGA based [10-12] implementations are proposed for network intrusion detection recently. Other methods were suggested using Bloom filters to speed up the string matching in [13-14].

A large number of malicious attacks on the Internet are spreading today. Networks are vulnerable to the attacks. The string matching algorithm plays an important role in network intrusion detection systems, which can detect malicious attacks and protect the network systems. The deterministic finite automaton (DFA) is the typical method for the string matching architectures.

The rest of paper is organized as follows. In section 2, overview of finite automaton and Aho-Corasick algorithms (AC algorithm) are given. In section 3 the string matching problems are presented and used in intrusion detection systems. In section 4 smallest suffix-AC algorithms are proposed and analyzed. In section 5 the implementation results are presented and the comparison between the proposed scheme and original algorithm are given. Finally, section 6 concludes remarks are given.
2. Backgrounds

2.1. Finite Automaton

Many string matching algorithms build a finite automaton that scans the text string $T$ for all occurrences of the pattern $P$. The definition of a finite automaton is as follows:

A finite automaton $M$ is a 5-tuple $(Q, \Sigma, S, E, f)$, where $Q$ is a finite set of states, $S \in Q$ is the start state, $E \subseteq Q$ is a distinguished set of accepting states, $\Sigma$ is a finite input alphabet, $f$ is a function from $Q \times \Sigma$ into $Q$, called the transition function of $M$.

The finite automaton begins in state $S$ and reads the characters of its input string one at a time. If the automaton is in state $q$ and reads input character $a$, it moves from state $q$ to state $f(q, a)$. Whenever its current state $q$ is a member of $S$, the machine $M$ is to be accepted the string read so far. An input that is not accepted is to be rejected.

There is a string matching automaton for every pattern $P$; this automaton must be constructed from the pattern in a preprocessing step before it can be used to search the text string. In order to specify the string matching automaton corresponding to a given pattern $P$, we first define an auxiliary function $f$, called the suffix function corresponding to $P$.

In addition, we define the string matching automaton that corresponds to a given pattern $P[1 \ldots m]$ as follows.

(1) The state set $Q$ is $\{0, 1 \ldots m\}$. The start state $S$ is state 0, and state $E$ is the only accepting state.

(2) The transition function $f$ is defined by the following equation, for any state $q$ and character $a$.

To clarify the operation of a string matching automaton, a simple and efficient example is shown in Figure 1 below for simulating the behavior of such an automaton in finding occurrences of a pattern $P$ in an input text $T$.

![Figure 1. An automaton example](image)

$$M = (Q, \Sigma, S, E, f), \text{ where}$$

- $Q = \{q_0, q_1\}$
- $\Sigma = \{a, b\}$
- $E = \{q_1\}$
- $f(q_0, a) = q_1$, $f(q_0, b) = q_0$, $f(q_1, a) = q_1$, $f(q_1, b) = q_0$

$M$ could also say $M = (\{q_0, q_1\}, \{a, b\}, q_0, \{q_1\}, f)$

2.2. AC algorithm

One of the early multiple string matching algorithms of automaton based design approach is the AC algorithm[2]. The AC algorithm locates all occurrences of any keywords in a text string. It works in constructing a finite state string matching machine from all of the keywords, and then using the string matching machine to process the payload string in a single pass. The AC algorithm constructs a DFA for detecting all occurrences of a given set of strings by processing the input in a single pass. The input is inspected byte by byte, such that each symbol results in a state transition. Thus, the AC algorithm has deterministic performance, which does not depend on the specific input and therefore is not vulnerable to various attacks, making it very attractive to network intrusion detection system (NIDS) systems.
The AC algorithm was proposed for multiple string matching. A finite automaton that accepts all the strings in the pattern set is built in the preprocessing stage. Each character in the text is then fed sequentially to the automaton that tracks partially matched patterns through state transition, so the time complexity is $O(n)$. If one of the final states is reached, a match is claimed. Although the AC algorithm is theoretically independent of the string set size in efficiency, it will become slow for a large pattern set in practice because of the worse cache locality in accessing a large transition table. Effectively compressing the transition table to reduce the memory requirement and enhance the cache locality becomes active research in the implementation of the AC algorithm.

When the algorithm is used to search for the set of strings $P = \{\text{hers, his, she}\}$ in the text "abherstbsh". The AC automaton built on $P$ is shown in Figure 2. Failure function and its output function are as shown in Figure 3 and Figure 4 respectively.

![Figure 2](image)

![Figure 3](image)

![Figure 4](image)

### 3. Problem statements

#### 3.1. Problem of multiple string matching

Given a set of $k$ strings $\Sigma = \{T_i \mid T_i = t_{i1} t_{i2} \ldots t_{im_i}, \text{for all } i = 1, 2 \ldots k\}$ over a finite alphabet set $\Omega$, where $t_{ij} \in \Omega$ and $m_i$ is the length of string $p_i$. We want to match a given string $T_i$ in $\Sigma$ with any substring $P_t = p_{t+1} p_{t+2} p_{t+3} \ldots p_{t+m_i}$ of a given long string $P = p_1 p_2 p_3 \ldots p_k$ of arbitrary length $t$. An algorithm is developed below to find all matching substring $P_t$ embedded in $P$ such that $P_t = T_i$, where $p_j \in \Omega$ for all $1 \leq j \leq t$. $\Omega$ denotes Set of symbols used in all strings tested, $\Sigma$ denotes Set of $k$ strings $\{T_i\}$ being matched, $P_t$ denotes A substring of a string $P$ being tested; $m$ denotes Average length of substrings in set $\Sigma$.

#### 3.2. String matching in intrusion detection

String matching algorithms play an important role in intrusion detection system (IDS). The characteristics of signatures in typical open source packages are investigated, such as Snort for IDS. Through capturing and analyzing the transmitted data packets over the network, Snort compares with the known attack characteristics of string packets and detect the features of intrusion. If they match with the features in attack packets, then the early warning of intrusion detection or logging is completed. For the detection mode, Snort is misuse detection. It extracts characteristics of the intrusion detection and makes some detection rules to form a rule sets, and then the data packet matches one by one in accordance with rule sets, if the match is
considered successful intrusion, it needs to be further treatment. IDS rule base usually contains tens of thousands of characters; number of string matching requires an efficient matching algorithm. In Snort, matching the character string in the packet load is the most time-consuming process of testing the operation, and matching the first node is the next; there is a small part of the other types of matching. Therefore, the selection of string matching algorithms will directly affect the detection efficiency of IDS, if the matching speed can be improved, the overall performance of Snort will be improved accordingly.

In recent years, in order to increase the use of IDS performance, automaton based string matching to optimize and improve the matching performance, is currently the focus of much attention. Several matching methods are proposed and applied to detect attacks against network systems. The aim is to monitor and filter data traffic arriving in a network to avoid exposing host computers to security threats.

For detecting malicious attacks, many important services in current networks are based on payload inspection, in addition to headers processing. NIDS are playing an increasingly important role in the design of novel network services. Original AC automaton representations are unsuitable for high performance requirements, which require too much memory, and too much time in the matching process. Recently many works have been presented with the goal of memory reduction for DFAs, by exploiting the intrinsic redundancy in regular expression sets.

NIDS not only focus on the header fields but also have to check signatures in the data payload portion of a packet. The most widely used NIDS like Snort and Bro use DFA to describe attack signatures in their rule sets. Because of the large traffic volume and complexity of the process, signature matching can easily become the performance bottleneck in deep packet inspection.

Many NIDS are almost software-based, which are flexible but can not keep up with the traffic high-speed networks. DFA-based representations and related algorithms have been applied to address several issues in information security, and often the automaton had to be augmented with additional information. Deep packet inspection has taken an important role in networking devices and computer systems. There has been a considerable amount of recent work on implementing regular expressions for high-speed networking applications, particularly with representations based on DFA.

4. Proposed algorithm

4.1. Smallest Suffix-AC optimizations

The improved algorithm uses a smallest suffix automaton of the set of strings. The suffix automaton of \( P \) recognizes at least all the substrings of the strings in \( P \). The search algorithm slides a window of size with the length of minimum string \( l_{\text{min}} \) along the text, reading backward a suffix of the window in the suffix automaton. If a letter \( \delta \) is failed, the window past \( \delta \) can be shifted safely. If not, the beginning of the window and verify a subset of \( P \) against the text can be reached.

The smallest suffix-AC construction is an extension of the single string’s, but the resulting automaton is not necessarily minimal. The construction is linear in the size of \( P \), and it is more efficacious than the previous one.

The practical limit of previous algorithm is the construction of the automaton, such as, for large sets of strings, it is too many transition states on the search phase. Moreover, requirement of the memory of the automaton quickly increase when the string set become large. Smallest Suffix-AC uses its novel method and overcomes the bottleneck of the original automaton with a lighter and simpler data structure.

4.2. Searching algorithm

The smallest suffix automaton construction on a set of strings is similar to the AC automaton construction. The only difference between them shows that when going down the supply path, it looks for an outgoing transition labeled by \( \delta \). In the AC automaton construction, if this transition does not
exist, then just jump to the next state on the supply path. In the suffix automaton construction, a
transition labeled by $\delta$ was created from each state on the supply path to the state where the original
transition leads.

The construction of smallest suffix automaton consists of three steps which the details are as
follows, and the pseudo-code of construction is shown in Figure 5. Assume that the supply function of
all the states before state $C$ in transversal order has been computed. Consider the parent $P$ of $C$ in the
trie, leading to $C$ by $\delta$, that is, $C = \lambda(P, \delta)$. $\lambda$ is its transition function. The supply state $S(P)$ has
already been computed, and the supply function from state $S(P)$ has been gone down. Use a variable $N$
initialized to $S(P)$ and repeat the following steps:

Step 1: If $N = \emptyset$, then $S(C) \leftarrow I$. ($\emptyset$ is the supply state of the initial state.)

Step 2: If $N \neq \emptyset$ and there does not exist a transition from $N$ labeled by $\delta$, then build a transition
from state $D$ to state $C$ by $\delta$ and return to step 1 with $N \leftarrow S(N)$.

Step 3: If $N \neq \emptyset$ and there exists a transition from $N$ labeled by $\delta$ leading to a state $Im$, then set $S(C) \leftarrow Im$ and stop processing state $C$.

Build Suffix Automaton ($P = \{p_1, p_2, \ldots, p_i\}$)

\[
Trie \leftarrow Trie(P)
\]

Mark the states of entire string $pi$ as terminal

$I \leftarrow$ root of Trie

$S(I) \leftarrow \emptyset$

For $C$ in transversal order Do

$P \leftarrow$ parent in Trie of $C$

$\delta \leftarrow$ label of the transition from $P$ to $C$

$N \leftarrow S(P)$

While $N \neq \emptyset$ AND $\lambda(N, \delta) = \emptyset$ Do

$\lambda(N, \delta) \leftarrow C$

$N \leftarrow S(N)$

End of while

If $N \neq \emptyset$ Then

$S(C) \leftarrow \lambda(N, \delta)$

Else $S(C) \leftarrow I$

End of if

End of for

Figure 5. Construction of Smallest Suffix Automaton

The search is done through a window of size $\text{lmin}$, which the text has been slide along. In this
process, backwards are read the longest suffix that labels a path from the initial state. The following
two cases may occur:

1) Failing to recognize a substring, therefore shift the window so that its new starting position
   corresponds to the position next to $\delta$.

2) The algorithm reaches the beginning of the window in a state $q$ of the suffix automaton. When
   using a suffix automaton, it’s sure at this step that a prefix of one of the strings can be recognized.
   However, the suffix automaton accepts paths of size $\text{lmin}$ ending in terminal states that do not
   correspond to any prefix. Thus, make sure that read the prefix $L(q)$ and only if this is the case a
   possible occurrence by comparing each string in $F(q)$ against the text. Finally move the search window
   by 1 and start the search again.

Pseudo-code for smallest suffix-AC is given Figure 6. The construction of the smallest suffix
automaton is fast and requires little memory, which permits using this algorithm to search large sets of
strings on relatively small texts. The notation $\text{pref}(p_i)$ denotes the prefix of size $\text{lmin}$ of the string $p_i$. 
4.3. An illustrative example

We search for the set of strings $P = \{\text{acbad, dadad, adadae}\}$ in the text $\text{abadacbadadadace}$. The construction of suffix automaton for the reverse set of $P_{\text{lmin}} = \{\text{acbad, dadad, adada}\}$ is shown in Figure 7.

![Suffix automaton for $P_{\text{lmin}} = \{\text{acbad, dadad, adada}\}$, bold states are the terminal.](image)

The progress of searching with suffix automaton is as follow Table 1:

<table>
<thead>
<tr>
<th>Step</th>
<th>Character</th>
<th>State Transition</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>0</td>
<td>Read a, b, d, a in the suffix automaton, and fail on the next b.</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
<td>2</td>
<td>Shift the window after b.</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>3</td>
<td>Reach the beginning of the window in state 13.</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>5</td>
<td>Compare the string $F(13) \rightarrow$</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>6</td>
<td>Verify all the strings in $F(C)$</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>c</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>d</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>d</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>a</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>d</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>a</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. The progress of searching with suffix automaton

Figure 6. Pseudo-code for the Smallest Suffix-AC algorithm.

```
Smallest Suffix-AC ($P = \{p_1, p_2, \ldots, p_r\}, T = t_1t_2 \ldots t_n$
Construction process

$l_{\text{min}} \leftarrow$ minimal length of strings in $p_i \in P$
$S_s \leftarrow \text{Build\_Suffix\_Automaton}\left(\{\text{pref}(p_1), \text{pref}(p_2), \ldots, \text{pref}(p_r)\}\right)$

$\lambda_{S_s}$ is its transition function

For $q$ state of $S_s$ Do $F(q) \leftarrow \emptyset$

For $i = 1 \ldots r$ Do

$F(q) \leftarrow F(q) + \{i\}$ ( $q$ is the state reached by $\text{pref}(p_i)$)

End of for

Searching process

$pos \leftarrow 0$

While $pos \leq n - l_{\text{min}}$ Do

$C \leftarrow$ initial state of $S_s$

$j \leftarrow l_{\text{min}}$

While $j \geq 1$ AND $C \neq \emptyset$ Do

$C \leftarrow \lambda(C, t_{pos+j})$

$j \leftarrow j - 1$

End of while

If $C \neq \emptyset$ and $j = 0$ and $T_{pos+1 \ldots pos+l_{\text{min}}} = L(C)$ ($L(C)$ is the
prefix of a string in $F(C)$)

Then

Verify all the strings in $F(C)$

$j \leftarrow 1$

End of if

$pos \leftarrow pos + j$

End of while
```

An Efficient Automaton Based Matching Algorithm and Its Application in Intrusion Detection System
Peifeng Wang, Yue Hu, Li Li
2. abadacb adadadace
   Read b, c, a, and fail on the next d. then shift the window after d.

3. abad acbad adadace
   Read d, a, b, c, a. reach the beginning of the window in state 11. Compare the strings $F(11) \rightarrow$ acbadad. Mark an occurrence and shift the window by 1.

4. abada cbada dadace
   Read a, d, a, and fail on the next b. then shift the window after b.

5. Performance evaluation

   The test results for the smallest suffix-AC implementation includes benchmark data for the original AC previously used in Snort. We evaluates our approach under common NIDS database. The average speedup and corresponding compression rate is marked.

   Each network content security application has different pattern lengths and pattern set size. We revise the original content security packages mentioned previously by implementing the suggested algorithms accordingly and observe the acceleration below. We design and implement algorithms with Visual Studio 2010 and run Snort 2.8.3.2 on a 2.5GHz Pentium M and 4G main memories in Windows 7.

   Experiments are performed on the rule sets and the total string patterns from the Snort rule sets to be compared with. Figure 8 shows the results of our approach compared with the original AC algorithm in memory with different number of rules. Figure 9 presents the comparison of time with different number of snort rules. From the two figures, we can know that our proposed smallest suffix automaton algorithm outperforms the original ones.

   ![Figure 8](image1.png)  ![Figure 9](image2.png)

6. Summary and conclusions

   In this paper, a novel string matching scheme based on AC automaton has been proposed, and implemented in intrusion detection tools of Snort. It takes the advantage of smallest suffix automaton instead of original automaton to achieve a string matching speedup with reduced memory consumption and execution time. Such speedup is obtained with little extra price
compared with current methods. Meanwhile, an easy implemented verification module can guarantee the equivalence between original AC and Smallest Suffix-AC algorithm.

An efficient string matching algorithm has been presented which can significantly reduce the number of states and transitions by merging states while maintaining correctness of string matching. The experiments demonstrate a significant reduction in memory and execution time for different rule sets commonly used to evaluate NIDS.

In the future, some more improvement would be explored on our algorithm with FPGA or ASIC, or it can be optimized to apply in the network intrusion detection systems.

7. Acknowledgment

The authors would like to thank anonymous reviewers for their valuable comments. This work is supported by the National Science Foundation of China under Grant No.60873002

8. References