AC-Suffix-Tree: Buffer Free String Matching on Out-of-Sequence Packets

Xinming Chen and Kailin Ge
Department of Automation
Tsinghua University
Beijing, China
xinming@engin.umass.edu,
gkl05@mails.tsinghua.edu.cn

Zhen Chen and Jun Li
Research Institute of Information Technology
Tsinghua National Lab for
Information Science and Technology
Beijing, China
zhenchen@tsinghua.edu.cn,
junl@tsinghua.edu.cn

ABSTRACT
TCP and IP fragmentation can be used to evade signature detection at Intrusion Detection/Prevention System (IDS / IPS). Such fragments may arrive out-of-sequence to escape from being detected by the string matching algorithm of IDS / IPS. The common defense is buffering and reassembling packets. However, buffering of out-of-sequence packets can become impractical on high speed links due to limited fast memory capacity, especially when the concurrent flows are in large quantity, or extremely disordered in circumstances such as attacks. So such buffering strategy is vulnerable to memory exhausting denial of service (DoS).

In this paper, AC-Suffix-Tree, a buffer free scheme for string matching is proposed, which detects patterns across out-of-sequence packets without buffering and reassembly. This novel algorithm associates the classical Aho-Corasick (AC) algorithm with a pattern suffix tree to search patterns with only the state numbers of AC automaton and suffix tree stored. It demands significantly less memory than buffering the packets themselves. Therefore the IDS can resist memory exhausting DoS attack. AC-Suffix-Tree consumes 1-2 orders of magnitude less memory than buffering the entire packet, and it has the same temporal complexity as AC algorithm when there are no out-of-sequence packets.

Categories and Subject Descriptors
C.2.0 [**Computer Communication Networks**]: General—*Security and protection*

General Terms
Algorithms, Security

Keywords
String Matching, Packet Reordering, Network Security

1. INTRODUCTION
Network security devices such as IDS/IPS or content filtering devices are widely deployed. Briefly, an IDS is a device tapping packets from a link and alerts for possible intrusions. An IPS is similar to an IDS, except it is inline and drops malicious packets instead of generating alerts. Most IDS/IPS are signature based, they keep a set of signatures (strings or regular expressions) and match them against payload of incoming packets. When there is a match, the packet is identified as malicious. The most popular multi-pattern matching algorithm used in IDS/IPS is AC [2], which works at linear time. It can locate multiple patterns at the same time in network traffics.

A flow in network is a finite sequence of packets, which have the same five-tuples (source IP, destination IP, source port, destination port, protocol). If the length of a transferred string exceeds the maximum capacity of one packet, it is divided into segments and encapsulated into packets. The packets may reach network devices in an undetermined order due to multiple routes, packet retransmission or IDS evasion. Such phenomenon is called packet reordering.

In order to detect target strings (patterns) which are dispersed in different packets, an IDS/IPS has to buffer and reassemble out-of-sequence packets. What is the current situation of packet reordering in Internet? In 2005, Dharmapurikar found that packet reordering in TCP traffic only affects 2-3% of the overall traffic[6]. An older paper reports that 90% of the TCP packets were reordered in the trace of Dec. 1997 and Jan. 1998 [3], but Dharmapurikar claims it was because the older generation of router architecture. Anyway, even if packet reordering is not serious in normal traffic, reassembling all the traffic is a common solution in IDS/IPS. Another solution is traffic normalizers, which are located in networks to remove ambiguous traffic before being exposed to IDS/IPS [7]. A normalizer still needs to maintain the state of each connection and to buffer out-of-sequence packets. Both the method of reassembly and normalizer require a large quantity of resources, thus bring potential bottleneck when working with high speed networks, and make IDS/IPS vulnerable to memory exhausting attacks [10].

There have been several attempts detecting attacks without reassembly or normalizers. George Varghese etal. proposed an approach named Split-Detect[9]. It split the signature into pieces. Detection of any piece will cause the fast path of IDS divert the TCP flow to the slow path. The processing and storage requirements of Split-Detect can be 10% of that required by a conventional IDS. Split-Detect is set out to solve the same problem as this paper does, but it does not completely avoid reassembly. It just offloads the reassembly from fast path to slow path, so that not every flow needs to be reassembled. The slow path still needs to reassemble packets. Moreover, according to the paper, Split-Detect requires three assumptions: a small modification to TCP receivers, a change in the definition of signature detection, and a restriction to exact signatures or regular expressions with a fixed exact length. All of the three assumptions are difficult to satisfy.

Another related paper is [12], in which an algorithm named On-Line Reassembly (OLR) is proposed. It utilizes a DAWG to store patterns and records the automaton state to avoid reassembly. This idea is very like our algorithm, but this paper does not consider the situation that a coming packet exactly fits a hole, thus it is logically incomplete. Moreover, our algorithm uses AC automaton, which is faster and is the most popular string matching algorithm in modern IDS. Our algorithm is also compatible with other AC-based string matching algorithms, which makes our algorithm more practical.

In order to develop a buffer-free scheme to process strings across the payload of packets, and to solve the reordering problem, we proposed a novel scheme using a suffix tree in the matching process. Dharmapurikar has used the concept of *hole* to measure the reordering issue. As described in Fig. 1, A hole is defined as a sequence gap which occurs in the TCP stream when a packet arrives with a sequence number greater than the expected one [6]. The size (the number of packets delayed) and the duration (the number of packets till the hole is filled) of the hole can largely affect the buffer needed for reassembly. But our scheme is not sensitive to the size or duration of the holes; buffer size is only related to the number of *successive blocks* in current flow. Each buffer entry for one *successive block* is just 28 Bytes, which makes the memory usage under control even in the worst case. In the common case, when packets come in order, the temporal complexity is the same as AC because there is no chance to use suffix tree.

Standard suffix tree has a tremendous space usage. In order to compress the size of the suffix tree, we proposed a data structure combining suffix tree and suffix array. Such data structure has the same spatial complexity as AC state machine, yet has the same temporal complexity as the standard suffix tree structure.

The remainder of this paper is organized as follows: Section 2 gives the formulation and notations of the string matching problem, and introduces AC and suffix tree algorithm. Section 3 describes our scheme in detail. Section 4 analyzes the performance of the algorithm. Experiments have been taken for both spatial and temporal performance evaluation and results are discussed. As a summary, in Section 5, we state our conclusion.

2. BACKGROUND

Before introducing our scheme, some notations must be defined to help description. The classic algorithms of AC Automaton (ACA) and Pattern Suffix Tree (PST) should also be introduced because they are important components in our scheme described later.

2.1 Notations

In this paper, a flow is considered as a complete string, and a packet of the flow is treated as a segment of it. A misuse pattern in IDS/IPS can be of various forms, such as byte string, regular expression, and so on. In this paper, we focus on string patterns that are signatures of intrusion. IDS/IPS should detect/block flows with these patterns to protect the digital assets behind it.

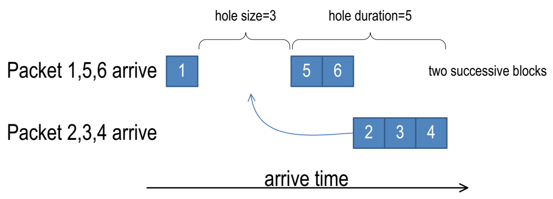


Figure 1: Concept of *hole* and *successive blocks*

This paper uses the notations in the book *Algorithms on Strings* [5] :

A **string** is a finite sequence of elements of a **alphabet** $\mathcal{A}$, which is a finite nonempty set whose elements are called **letters**. A zero letter sequence is called an **empty string** and is denoted by ε. The set of all the strings on the alphabet $\mathcal{A}$ is denoted by $\mathcal{A}^*$. The **length** of a string x is denoted by $|x|$, and the letter at index i (begin with 0) of x is denoted by $x[i]$, where $i \in \{0, 1, \ldots, |x| - 1\}$. The **product** – also called the **concatenation** – of two strings x and y is the string composed of the letters of x followed by the letters of y. It is denoted by xy.

A string x is a **factor** of a string y if there exist two strings u and v such that $y = uxv$. When $u = \varepsilon$, x is a **prefix** of y; and when $v = \varepsilon$, x is a **suffix** of y. A factor x of a string y is **proper** if $x \neq y$. It is denoted respectively as $x \preceq_{\text{fact}} y$, $x \prec_{\text{fact}} y$, $x \preceq_{\text{pref}} y$, $x \prec_{\text{pref}} y$, $x \preceq_{\text{suff}} y$ and $x \prec_{\text{suff}} y$ when x is a factor, a proper factor, a prefix, a proper prefix, a suffix and a proper suffix of y.

Besides the notations above, some new notations about segment are introduced to help explanation:

A segmented string (denoted by Y) is a string in the form of $y_1 y_2 \cdots y_n$. Here $n \geqslant 2$, and $y_i \in \mathcal{A}^*$, $y_i \neq \varepsilon$ for $i = 1, 2, \ldots, n$. A pattern set (denoted by X) is a set of pattern strings that IDS/IPS inspect against.

$y_1, y_2, \ldots, y_n$ are called segments of Y, and they may come in undetermined order. For each segment y_i, it can get through or get lost in network links. The segments get through can be combined to several successive blocks of Y. If no pattern $x \in X$ can be found in these successive blocks, it can be determined no threats are in this flow.

2.2 AC Automaton

The AC algorithm is one of the most popular multiple pattern matching algorithms. It compiles the pattern set to a deterministic finite automaton (DFA). Each state stands for a letter in the pattern set, and one state may belong to multiple patterns. The AC algorithm sequentially reads the input string and search along the DFA. It has linear performance to the length of the input string, regardless of the

```
typedef struct{
  uint_32    NextState[ ALPHABET_SIZE ];
  uint_32    FailState;
  AC_PATTERN *MatchList;
}AC_NODE;
```

Figure 2: Data structure of *AC_NODE*

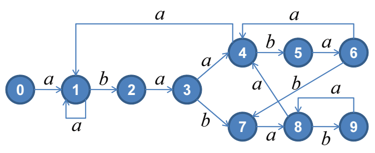


Figure 3: DFA of pattern set $X = \{abaaba,\ ababab\}$

Table 1: Output Function of X

state	6	9	others
output(state)	$\{abaaba\}$	$\{ababab\}$	$\emptyset$

number and length of the search strings.

The implementation of AC can be divided into preprocessing and searching stages. During the preprocessing stage, a DFA is built according to the pattern set. Such DFA contains some *goto functions* which determines the next state for current input letter, some *failure functions* which indicates the next state when there is no goto function for current input letter, and some *output functions* which output successful matches on current state.

The data structure of AC state node is shown in Fig. 2. The `ALPHABET_SIZE` is 256, so the size of the data structure is 1032 Bytes.

Fig. 3 is an example of DFA for pattern set $X = \{abaaba,\ ababab\}$. Its output function is shown in Table 1.

The searching stage is described in Function 1. The return value contains the final state and a list of matched patterns.

2.3 Pattern Suffix Tree

A Pattern Suffix Tree (**PST**) of a pattern set X is a trie which is built from the proper suffix set of X. For example, let $X = \{abaaba,\ ababab\}$, the proper suffix set of X is

$$\{a,\ ba,\ aba,\ aaba,\ baaba,\ b,\ ab,\ bab,\ abab,\ babab\}$$

an example of PST is shown in Fig. 4.

A PST is an automaton whose state transition function is described in Function 2. The procedure of constructing suffix tree is similar to DFA construction in AC algorithm. The failure function and output function in AC is not needed here. The return value contains the stop state and a "$fact$" mark. Once the input string is not finished but there is no available next state, $fact$ is **false**; and once the input string is finished but PST is not finished, $fact$ is **true**. So $fact$ = **true** means *str* is a proper factor of some patterns in X

Function 1 $ACA(str,\ state)$

```
$match \leftarrow \emptyset$
for $i = 0$ to $length(str) - 1$ do
  if there is an arc $(state,\ t)$ labeled $str[i]$ in ACA then
    $state \leftarrow t$
    $match \leftarrow match \cup output(state)$
  else
    $state \leftarrow failstate(state)$
  end if
end for
return $(state,\ match)$
```

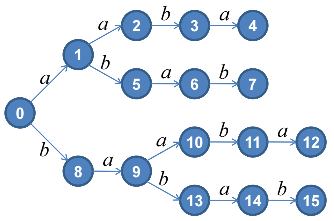


Figure 4: PST of pattern set $X = \{abaaba,\ ababab\}$

```
typedef struct{
  uint_32 NextState[ ALPHABET_SIZE ];
  uint_32 PreState; // parent node of
                    // this node
  uchar   PreChar;  // the path from parent
                    // node to itself
  uint_32 count;    // help to compress PST
}PST_NODE;
```

Figure 5: Data structure of *PST_NODE*

The data structure of PST state node is shown in Fig. 5. The `ALPHABET_SIZE` is 256. `PreChar` is the path from parent node to current node. `count` is used in the process of compression. Consider the alignment issue of the compiler, the size of the data structure is 1036 Bytes.

Our algorithm needs to trace back from the current state to the root node, retrieve the string in this path. Hence, the function $path(state)$ constructed to returns the string of path, i.e. the letters from root to current state. For example, $path(11)$ is *baab* on the PST shown in Fig. 4.

3. PROPOSED ALGORITHM

3.1 AC-Suffix-Tree Algorithm

Fig. 6 shows a simple situation of two packets' reordering. When packet y_2 comes first, a pattern may exist between the two packets only if some prefix of y_2 is one suffix of the patterns. So the PST can be used to determine whether a suffix of patterns exits at the beginning of y_2. If PST returns successfully, record the PST state. When y_1 comes, get the *path* from the PST state recorded, add the path to the end of y_1, then the pattern can be matched.

Now dive into more details of the AC-Suffix-Tree algorithm. A pattern x's occurrences in a segmented string $Y = y_1 y_2 \cdots y_n$ can be divided into two cases. One is that x only exists

Function 2 $PST(str,\ state)$

```
for $i = 0$ to $length(str) - 1$ do
    if there is an arc $(state,\ t)$ labeled $str[i]$ in PST then
        $state \leftarrow t$
    else
        return $(state,$ false$)$
    end if
end for
return $(state,$ true$)$
```

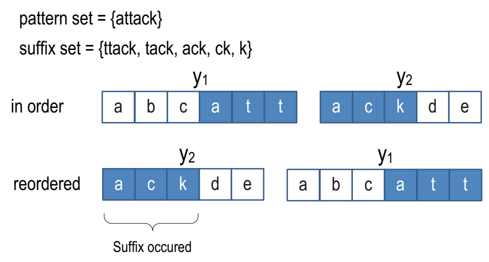


Figure 6: Examples of two packets' reordering

in single segment,

$$\exists i \in \{1, 2, \ldots, n\},\ x \preceq_{\text{fact}} y_i \tag{1}$$

The other is x exists across several segments,

$$\forall i \in \{1, 2, \ldots, n\},\ x \not\preceq_{\text{fact}} y_i \tag{2}$$

For the first case, the occurrence of x can be find out by passing y_i to the ACA of x. Notice that whatever the initial state of the ACA is, it can find out all the occurrences of x in y_i. This can be done whenever y_i is received.

The second case is what this paper focuses on, $\exists i,\ j \in \{1, 2, \ldots, n\}$, $i<j$ so that

$$x \preceq_{\text{fact}} y_i y_{i+1} \cdots y_j \tag{3}$$

but

$$x \not\preceq_{\text{fact}} y_{i+1} y_{i+2} \cdots y_j,\ x \not\preceq_{\text{fact}} y_i y_{i+1} \cdots y_{j-1} \tag{4}$$

In order to deal with this case without buffering the input string $y_1, y_2, \ldots, y_n$, more work should be done:

First, begin with a simple example: assume there are only two segments, y_i and y_j, $j = i + 1$. In this case, $\exists u, v \in \mathcal{A}^*$ so that $x = uv$ and $u \preceq_{\text{suff}} y_i,\ v \preceq_{\text{pref}} y_j$.

If y_i comes first, pass it to the ACA of x and then save the final state of the ACA as s_1. When y_j comes, pass it to the ACA with initial state s_1, the occurrence of x can be found. For example, suppose $X = \{abaaba,\ ababab\}$, $y_i = aaba$, $y_j = abaa$, there is an occurrence of $abaaba$ in $y_i y_j$. When y_i comes, it is passed to the ACA of X (as shown in Fig. 3), and the final state of the ACA is 3. Save $s_1 = 3$, and let packet y_i pass through. When y_j comes, recover the ACA state 3 and go on searching, it will find out the occurrence of $abaaba$. So y_j is marked up and this input string is recognized as threatening. Because only y_i gets through, i.e., there is only a part of $abaaba$ pass the check, the threats cannot take effects and are successfully blocked.

If y_j comes first, it is passed to the PST of x and the stop state of the PST is saved as s_2. When y_i comes, the passed string y_i is appended by the path of s_2 in PST before input to the ACA, the occurrence of x can also be found out. For example, assuming $X = \{abaaba,\ ababab\}$, $y_i = aaba$, $y_j = abaa$, there is an occurrence of $abaaba$ in $y_i y_j$. When y_j comes, it is passed to the PST of X (as shown in Fig. 4), and the stop state of the PST is 6. Save $s_2 = 6$, and let y_j pass through. When y_i comes, it is passed to the ACA of X appended by $path(6) = aba$. Then the ACA does pattern matching on $aabaaba$. Then the ACA finds out the occurrence of $abaaba$. So y_i is dropped and this data is marked up and this input string is recognized as threatening. Because only y_j gets through, i.e., there is only a part of $abaaba$ in it, the threats are successfully blocked, too.

Considering the possibility of packet retransmission, this algorithm need one more step to avoid evasion attempt. For example, if the pattern is divided into 3 packets, packets 2,3 arrive first and pass the system. When packet 1 arrives, it will be dropped. The pattern state is cleared. The sender then retransmit packet 1. It will pass the IDS, and all 3 packets will get to the destination. In order to avoid this situation, one solution is to send RST packets to both sides of the connection once a pattern is detected, so the receiver will terminate this connection and no longer accepts the retransmitted packet.

In a word, the ACA of x can be used to keep the suffix information of a segment, while the PST of x can be used to keep the prefix information of a segment.

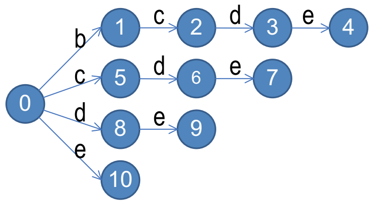
What if the pattern x crosses more than two segments? A *information merging* mechanism is used to merge the PST state records in successive blocks. Suppose there are $y_i y_{i+1} \cdots y_j$, $j > i + 1$. $\exists u, v \in \mathcal{A}^*$ so that $x = u y_{i+1} y_{i+2} \cdots y_{j-1} v$ and $u \preceq_{\text{suff}} y_i,\ v \preceq_{\text{pref}} y_j$. In this case, the return value "$fact$" of PST is used to identify the **proper factor** of x. $fact =$ **true** means the entire segment is a proper factor of x, thus needs to merge the PST state with the predecessor segment.

The detailed pseudo code description of our algorithm is shown in Appendix A. The data structure of *Buffer* is shown in Fig. 7. `Next` is the pointer to the next connection record for resolving hash collisions. The size of the data structure is 28 Bytes.

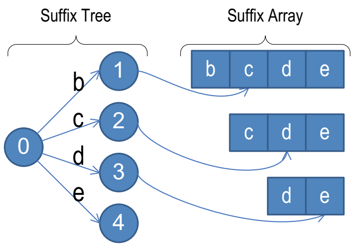
Timeout mechanism is not included in the algorithm in order for brevity. In the real network deployment, timeout mechanism is necessary because some flows may miss FIN or RESET packets. For example, in our test of Lincoln Lab traces [1], about 0.37% of the packets are never terminated by FIN or RESET packets. Such flows must be flushed from the buffer after it has silenced for a long time.

3.2 An Example

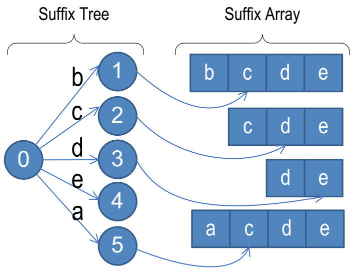
Here is an example for easy understanding of our algorithm. Suppose the pattern set $X = \{abaaba,\ ababab\}$ and the segmented string $Y = y_1 y_2 y_3 y_4$, where $y_1 = bbaa$, $y_2 =$



(a) PST of pattern *abcde*



(b) Suffix array of pattern *abcde*



(c) Suffix array of pattern *abcde* and *bacde*

Figure 8: Examples of PST and Suffix array

```
typedef struct _bufStruct{
  uint_32     fid;
  uint_32     seq;
  uint_32     len;
  uint_32     s1;
  uint_32     s2;
  uint_32     fact;
  _bufStruct* Next;  //pointer to the next
         // connection record for resolving
         // hash collisions (32 bits)
}BUFFER_STRUCT;
```

Figure 7: Data structure of $BUFFER\_STRUCT$

$baba$, $y_3 = baab$, $y_4 = aabb$. The coming order is y_3, y_1, y_4, y_2, flow ID is 1.

Firstly, an ACA and a PST is generated respectively for X as shown in Fig. 3 and Fig. 4, respectively. Initially we set $Buffer = \emptyset$. Then we begin to handle the input segments.

1. Suppose the first input segment is $y_3 = baab$.
   Because y_3 is the first coming segment of flow 1, there is no record in the $Buffer$ for flow 1. Passing y_3 to ACA and PST both with initial state 0, the return value $s1 = 2,\ s2 = 11,\ match = \emptyset,\ fact = \textbf{true}$.
   These are implemented by the pseudo codes at line 33 and 52. After the process, $Buffer$ contains one entry: $(1, 8, 4, 2, 11, \textbf{true})$.

2. The second input segment is $y_1 = bbaa$.
   Because there is neither y_1's predecessor nor successor in the $Buffer$, the process for y_1 is the same as y_3. Passing y_1 to ACA and PST both with initial state 0, the return value $s1 = 1,\ s2 = 8,\ match = \emptyset,\ fact = \textbf{false}$.
   After the process, $Buffer$ contains two entries: $(1, 0, 4, 1, 8, \textbf{false})$ and $(1, 8, 4, 2, 11, \textbf{true})$.

3. The third input segment is $y_4 = aabb$.
   Because $Buffer$ contains the information of y_4's predecessor – $(1, 8, 4, 2, 11, \textbf{true})$, y_4 is passed to ACA with initial state 2. The return value $s1 = 0,\ match = \emptyset$. Since the "$fact$" mark of y_4's predecessor is **true**, y_4 is passed to PST with initial state 11. The return value $s2 = 12,\ fact = \textbf{false}$.
   These are implemented by the pseudo codes at line 26, 33 and 44. After information merging, $Buffer$ contains two entries: $(1, 0, 4, 1, 8, \textbf{false})$ and $(1, 8, 8, 0, 12, \textbf{false})$.

4. The final input segment is $y_2 = baba$.
   Because $Buffer$ contains the information of both y_2's predecessor and successor – $(1, 0, 4, 1, 8, \textbf{false})$ and $(1, 8, 8, 0, 12, \textbf{false})$, $path(12) = baaba$ is appended to y_2's tail and $str = babab aaba$ is passed to ACA with initial state 1. The return value $s1 = 6,\ match = \{abaaba,\ ababab\}$. Then y_2 is dropped, all records with $fid = 1$ is cleaned from $Buffer$.
   These are implemented by the pseudo codes at line 26, 29, 33 and 35-38. After the process, $Buffer = \emptyset$.

Notice that even if y_1, y_3, y_4 has got through, they don't contain the full malicious string, thus will not cause damage.

3.3 Compression of Suffix Tree

As shown above, the AC suffix tree algorithm has advantages in reducing the memory usage on reordered packets. But it also incurs new memory usage. A standard Suffix Tree requires tremendous memory usage. If the length of each pattern is n_i, the upper bound of memory occupation is

$$\sum_i (n_i - 1)(n_i - 2)/2 \tag{5}$$

so that it is $O(n^2)$ spatial complexity. For example, a pattern *abcde*'s suffix tree is shown in Fig. 8(a). Though the string length is only 5 letters, the state number reaches 10 with counting the transitions. For the real pattern set, the size of suffix tree is usually 10 times larger than AC state machine, which hinder the practical use of our algorithm.

Consider a single pattern such as *abcde*, the arrangements of the suffixes $\{bcde,\ cde,\ de,\ e\}$ are very regular. A natural idea of compression is using a suffix array instead of a tree. The suffix array of pattern *abcde* can be shown as Fig. 8(b). There is only a root node and four children nodes, the other nodes are stored in a single pattern string with some pointers point to them.

Following is how to compress a single pattern's suffix tree to a suffix array. Each state node has a counter. While building the suffix tree, every accessed or new created state's counter increases by one. After the preprocessing stage, perform a *depth first search* to the tree. Once a state is found the counter is 1, point the next state of this node to the suffix array, and release all the state nodes below it.

When added more patterns, the suffixes of multiple patterns may be duplicated, which can cause unwillingly increment of the counters. So it's necessary to list all the suffixes and deduplicate them, then build the suffix tree. For example, the suffixes of *abcde* and *bacde* after deduplication are $\{bcde,\ acde,\ cde,\ de,\ e\}$. The suffix array of the two patterns is shown in Fig. 8(c).

The searching stage is similar to that of standard PST described in Section 2.3. After the input string is processed without failure, if the matching stopped in state machine, then return the state number; if the matching stopped in arrays, then return the array and the index in it. The *path()* function is also like that of PST if the matching stopped in state machine, and if the matching stopped in arrays, just return the sub string before current index in the array.

The preprocessing stage time can be several times longer than that of AC due to the compression. But for network security devices the preprocessing speed is not important, they focus more on the searching speed.

After the compression, the upper bound of state number is $\sum_i (n_i - 1)$, which is the same scale as AC Automaton. Note that one state node is 1036 Bytes, and one letter in the array is just 1 Byte. So the memory usage of the array can be largely ignored. The temporal complexity is the same as the standard PST, the only difference is that the matching step is divided into two parts: the state machine part and the array part.

4. PERFORMANCE ANALYSIS

4.1 Experiments

The AC Suffix Tree algorithm is implemented as a process in network processing platform to evaluate its performance. To alleviate the input speed bottleneck, AC suffix tree process feed the tcpdump trace files stored in ramdisk. When there is a match, a log message will be written into a file.

The pattern set is chosen from the latest rules of Snort 2.8.6.1[8], released on 22 Jul, 2010. All the patterns are fixed strings, no regular expressions included. The pattern set is divided into two parts according to pattern length with the threshold length set to 8. The reason for doing such classification is observation of the impact of pattern length to the memory usage. The detailed parameters of the pattern set is shown in Table 2. The upper bound of AC automaton size is calculated by

$$\sum_i n_i \times \mathrm{sizeof}(AC\_NODE) \qquad (6)$$

and the upper bound of PST size is calculated by

$$\sum_i \frac{(n_i - 1)(n_i - 2)}{2} \times \mathrm{sizeof}(PST\_NODE) \qquad (7)$$

where n_i is the length of pattern i.

The network traces are generated by a small program. We don't use captured traces because the severity of the reordering and number of sessions are not under control. Using these generated traces it is easy to observe the time and memory usage under different kinds of reordering.

Table 2: Parameters of Snort pattern sets

	Short Set	Long Set
Entries	786	1485
Max pattern length	8	127
Average pattern length	5.01	26.15
Upper Bound of AC size	4.0 MB	40.0 MB
Upper Bound of PST size	9.5 MB	637.2 MB

Four traces are used in this experiment. They are all unidirectional and the Ethernet frame size is 1518. Each of them contains 10,000 sessions, and each session has 10 packets. The packets come like this: the first packets of the 10,000 sessions come, then the second, the third$\cdots$. Trace One is not reordered, all packets comes with continual sequence number. Trace Two has one hole with the duration of one packet for each session, that is, for example, packets come in the order of $\{1, 3, 2, 4, 5, 6, 7, 8, 9, 10\}$. Trace Three has one hole with duration of 8 packets and size of 2, for example, $\{1, 4, 5, 6, 7, 8, 9, 10, 2, 3\}$. Trace Four has two holes, both with duration 6, for example, $\{1, 3, 4, 6, 7, 8, 9, 2, 10, 5\}$.

The experiments will compare the AC-Suffix-Tree algorithm with standard AC algorithm with reassembling scheme. The reassembling scheme only buffers packets that are reordered, not all the packets.

The following experiments are running on a PC with one Pentium Dual-Core CPU E5300 at 2.60GHz. The system has 4 GB of DDR2 RAM and runs Windows XP 32-bit SP3. The development environment is Visual Studio 6.0 SP6 with winpcap 4.1.2.

4.2 Temporal Complexity

According to the algorithm, when a packet p comes, the processing includes 5 steps:

1. Find p's predecessor and successor.
   Time taken by this process depends on the data structure of $Buffer$. Our implementation uses a hash table to store flow entries, and resolve hash collisions by chaining the colliding records in a linked list. The space is enough to maintain 1M connection records. Even if there are 100K concurrent connections, the theoretical memory accesses required for a successful search are only $1 + (0.1M - 1)/(2 \times 1M) \approx 1.05$ [4].

   After the flow is located, a linear search is performed to find p's predecessor and successor. The temporal complexity is $O(\log l)$ under binary search, where l is the number of *successive blocks* in this flow.

2. Get information from p's predecessor and successor.
   There's a path-searching operation in this process (at line 29 of pseudo code). Time needed to do this is $O(m - 1)$ if the returned state is in suffix tree, here $m = \max_{x \in X} |x|$. That is, m is the length of the longest pattern, and $m - 1$ is the depth of PST. If the returned

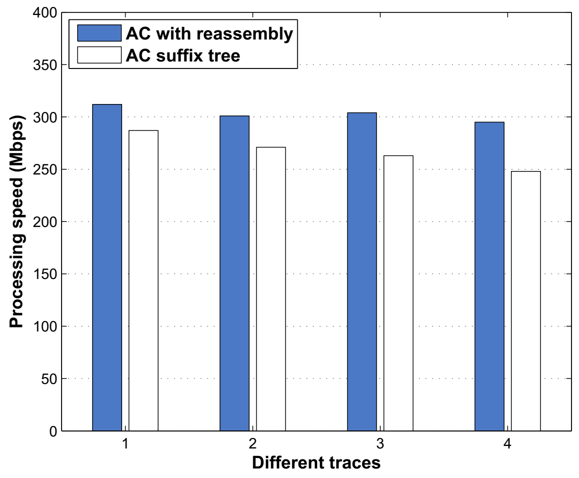


Figure 9: Processing speed for different traces with Long Set

state is in suffix array, the path can be immediately returned in $O(1)$.

3. Do pattern matching with ACA.
   Time taken by this process is $O(|y_i| + m - 1)$.

4. Merge information
   A group of state transitions in PST (at line 44 and 52) is $O(m - 1)$.

5. Update $Buffer$
   Time taken by this process is $O(1)$.

To sum up, the temporal complexity of the algorithm is $O(|Y| + nm)$, here n is the number of segments in the data flow and m is the length of the longest pattern.

Note that $O(|Y| + nm)$ is just an upper bound. In particular, when the segments come in order, the whole process is $O(|Y|)$ because there is no chance to use PST.

Fig. 9 shows the processing speed to the four traces. The speed (in Mbps) is the quotient of tcpdump file size (in bits) by the processing time (in seconds). For Trace One the speed of AC-Suffix-Tree is almost the same as AC with reassembly, even it has some extra operations like comparisons for each packet. For Trace Two, Three and Four, the speed of AC-Suffix-Tree algorithm is slower, because the PST is used. The more reordered packets, the more frequently the PST is used. But the speed is still acceptable. Recall that the severe reordering is not a normal condition, so the speed should not be too bad for normal traces.

4.3 Spatial Complexity

There are two aspects of spatial complexity. The first is the size of AC and Suffix Tree/Array, which is generated during the preprocessing stage. The second is the size of buffer needed during the searching stage.

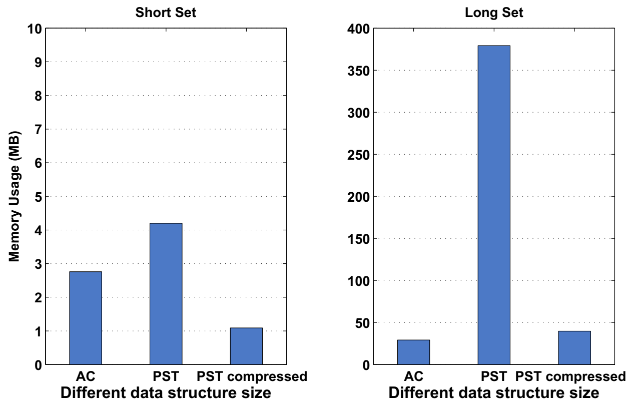


Figure 10: Memory usage of AC and suffix tree

4.3.1 Preprocessing Stage

Though the upper bounds of AC automaton and PST size have been analyzed in Section 3.3, they are just the worst cases and can hardly being reached. In the practical pattern set, different patterns may contain same substrings, so the states can be reused. The chance they share the same substrings becomes larger when the patterns are short, thus the AC automaton and suffix tree can be much smaller than the upper bond.

Fig. 10 shows the size of AC automaton and suffix tree (before and after compression). As we can see, the compression successfully reduces the size of suffix tree to 1/4 - 1/10 of the original size. The longer the patterns are, the larger the original PST is, and the higher compression ratio is. After the compression, the size of suffix tree is about the same as AC. For shorter patterns, the suffix tree is even smaller than AC, which means the patterns share many common substrings.

4.3.2 Searching Stage

According to the information merging mechanism, the memory needed by one data flow depends on not how many segments in the flow, but how many *successive blocks* received. For example, Trace Four has the packet sequence of $\{y_1, y_3, y_4, y_6, y_7, y_8, y_9, y_2, y_{10}, y_5\}$. Since y_3, y_4 are successive, their information is merged. And when y_6, y_7, y_8, y_9 come, their information is merged into another record. So the maximum number of buffer entries is 3, one contains information of block y_1, the second contains information of block y_3, y_4, and the last contains information of y_6, y_7, y_8, y_9. When y_2 come, the first two records are merged, when y_{10} come, it merges with the last record; when y_5 finally comes, the leaving two records are merged into one.

Therefore, the number of buffer entries is only related to the number of current *successive blocks*. A data flow consists of n segments needs $\lceil n/2 \rceil$ buffer entries in the worst case. In particular, when packets come in order, only one buffer entry is needed.

Besides, the size of each entry is only 28 Bytes, it is independent with the content of y_i. So when the size of every

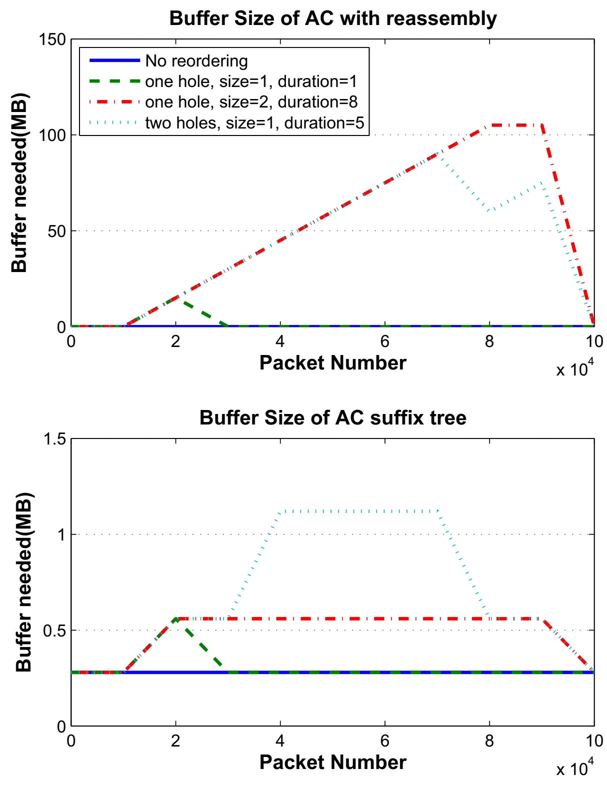


Figure 11: Buffer size while processing packets with Long Set

segment is relatively large, each entry takes much less memory than buffering the content of y_i.

In conclusion, the memory consumption by one data flow is proportional to the number of *successive blocks* in the data flow.

Fig. 11 shows the comparison of buffer needed in the two schemes: AC with reassembly scheme, and AC-Suffix-Tree. The horizontal axis shows the number of packets, the ordinate axis shows the buffer size when the current packet is processed. It shows the main advantage of AC-Suffix-Tree– low memory usage, especially when the reordering is severe. Moreover, in Trace Three, though the duration of the hole is 8 and length of the hole is 2, the entries of AC-Suffix-Tree are no more than 2. This shows the insensitive to the hole's size and duration. From Trace Three and Trace Four, it can be seen that the entries in the buffer is only related by the number of *successive blocks*.

5. CONCLUSION AND FUTURE WORK

In this paper, a string matching algorithm called AC-Suffix-Tree is proposed. The algorithm utilizes an ACA and a PST to help recording information of fragmented data flows, thus the memory usage is much less than packet reassembly. The algorithm is fast enough because it is based on AC, and has the same time performance as AC when there is no reordering. Another advantage of our algorithm is insensitivity to the size and duration of *holes* in network flows, which makes IDS robust against memory exhausting DoS attack.

Future work includes porting this scheme to other string matching algorithms. Our scheme is based on AC, but it is theoretically compatible with other automaton based string matching algorithms, such as CIAC [11]. Such variation algorithms can provide specific performance improvement, such as reducing the size of state machine.

6. REFERENCES

[1] DARPA Intrusion Detection Data Sets. *HTTP://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html*.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333 – 340, 1975.

[3] J. C. R. Bennett. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking*, 7(6):789 – 798, 1999.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Prentice Hall, 1998.

[5] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[6] S. Dharmapurikar and V. Paxson. Robust tcp stream reassembly in the presence of adversaries. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 5–5, 2005.

[7] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 9–9, 2001.

[8] M. Roesch. Snort – lightweight intrusion detection for networks. In *the 13th USENIX Conference on System Administration*, 1999.

[9] G. Varghese, J. A. Fingerhut, and F. Bonomi. Detecting evasion attacks at high speeds without reassembly. *SIGCOMM Comput. Commun. Rev.*, 36:327–338, August 2006.

[10] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and robust tcp stream normalization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 96 –110, May 2008.

[11] J. Yu, Y. Xue, and J. Li. Memory efficient string matching algorithm for network intrusion management system. *Tsinghua Science and Technology*, 12(5):585 – 593, 2007.

[12] M. Zhang and J.-b. Ju. Space-economical reassembly for intrusion detection system. In *Information and Communications Security*, volume 2836 of *Lecture Notes in Computer Science*, pages 393–404. Springer Berlin / Heidelberg.

APPENDIX

A. PSEUDO CODE OF AC-SUFFIX-TREE

```
$Buffer \leftarrow \emptyset$

**while** here comes a packet $p$ **do**
  $fid = HASH(p.srcip, p.dstip, p.srcport, p.dstport)$
  $seq \leftarrow$ sequence number of $p$
  $len \leftarrow$ length of payload of $p$
  $str \leftarrow$ payload of $p$

  /* find $p$'s predecessor and successor */
  $pre \leftarrow NULL$
  $suc \leftarrow NULL$
  **for all** $rec \in Buffer$ that $rec.fid$ equals to $fid$ **do**
    **if** $seq$ equals to $(rec.seq + rec.len)$ **then**
      $pre \leftarrow rec$
    **else if** $(seq + len)$ equals to $rec.seq$ **then**
      $suc \leftarrow rec$
    **end if**
  **end for**

  $s1 \leftarrow 0$ /* state of ACA */
  $s2 \leftarrow 0$ /* state of PST */
  $fact \leftarrow$ **false** /* a mark for PST */

  /* get information from $pre$ and $suc$ */
  **if** $pre \neq NULL$ **then**
    $s1 \leftarrow pre.s1$
  **end if**
  **if** $suc \neq NULL$ **then**
    $str \leftarrow strcat(str,\ path(suc.s2))$
  **end if**

  /* do pattern matching */
  $(s1,\ match) \leftarrow ACA(str,\ s1)$
  **if** $match \neq \emptyset$ **then**
    drop $p$
    send RST packets to both sizes
    remove all records of $fid$ from $Buffer$
    **continue**
  **end if**

  /* information merging */
  **if** $pre \neq NULL$ **then**
    **if** $pre.fact$ **then**
      $(s2,\ fact) \leftarrow PST(str,\ pre.s2)$
    **else**
      $s2 \leftarrow pre.s2$
    **end if**
    $seq \leftarrow pre.seq$
    $len \leftarrow len + pre.len$
    remove $pre$ from $Buffer$
  **else**
    $(s2,\ fact) \leftarrow PST(str,\ 0)$
  **end if**
  **if** $suc \neq NULL$ **then**
    **if not** $suc.fact$ **then**
      $s1 \leftarrow suc.s1$
      $fact \leftarrow$ **false**
    **end if**
    $len \leftarrow len + suc.len$
    remove $suc$ from $Buffer$
  **end if**

  /* store merged information to $Buffer$ */
  $Buffer = Buffer \cup \{(fid,\ seq,\ len,\ s1,\ s2,\ fact)\}$
  let $p$ get through
**end while**
```