Eagle+: A Fast Incremental Approach to Automaton and Table Online Updates for Cloud Services
Jianxin Li\textsuperscript{a}, Hao Peng\textsuperscript{a}, Erica Yang\textsuperscript{b}, Chunming Hu\textsuperscript{a,\ast}, Shenghai Zhong\textsuperscript{a}, Lihong Wang\textsuperscript{c,\ast}

\textsuperscript{a}School of Computer Science and Engineering, Beihang University, Beijing, 100191, China
\textsuperscript{b}Scientific Computing Department, STFC Rutherford Appleton Laboratory, Oxfordshire, UK
\textsuperscript{c}National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China

Highlights:

1. We propose Eagle+, an incremental approach for updating the matching Automaton and Table whilst avoiding recalculating the whole patterns after each change.
2. We implement three atomic operations, adding, updating and deleting, for achieving accurately incremental updating.
3. Eagle+ can save nearly 72\% - 92\% of the time consumption in AC automaton and SOBM automaton.
4. Eagle+ can perform 100X faster in WM table.
5. The matching accuracy of Eagle+ is same as existing approaches.
Eagle+: A Fast Incremental Approach to Automaton and Table Online Updates for Cloud Services

Jianxin Li\textsuperscript{a}, Hao Peng\textsuperscript{a}, Erica Yang\textsuperscript{b}, Chunming Hu\textsuperscript{a,}\textsuperscript{*}, Shenghai Zhong\textsuperscript{a}, Lihong Wang\textsuperscript{c,}\textsuperscript{*}

\textsuperscript{a}School of computer Science & Engineering  \hspace{1em} \textsuperscript{b}Scientific Computing Department STFC
Beihang University, Beijing, China  \hspace{1em} Rutherford Appleton Laboratory, Oxfordshire, UK
E-mail: \{lijx, penghao, hucm, zhongsh\}@act.buaa.edu.cn  \hspace{1em} E-mail: erica.yang@stfc.ac.uk
\textsuperscript{c}National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China
E-mail: wlh@isc.org.cn

Abstract

Automaton or table-based multi-pattern matching methods have been widely used in cloud services, i.e., virtual Firewall service, virtual IDS service, etc. In cloud, a large scale of patterns in such services are frequently updated causing by users’ joining or quitting and adjustment of security and management policies. Therefore, how to quickly and accurately update the Automaton and Table becomes an important issue. In this paper, we propose Eagle+, an incremental approach for updating the matching Automaton and Table whilst avoiding recalculating the whole patterns after each change. In Eagle+, we attain efficiency by computing only the latest update set of patterns when updating the Automaton and Table. Moreover, Eagle+ achieves accurately local updating based on three atomic operations, adding, updating and deleting, each of which modifies values on classical Aho-Corasick (AC) automaton, Set Backward Oracle Matching (SBOM) automaton and Wu-Manber (WM) table. Compared with existing pattern updating methods, Eagle+ reduces the computation complexity from $O(n^2)$ to $O(n)$. The experimental results show that Eagle+ can save nearly 72% - 92% of the time consumption in updating automatons and perform 100X faster in WM table.

Key words: Cloud Service; Multi-pattern Matching; Table and Automaton; Incremental Updating;

1. INTRODUCTION

Public clouds, taking advantage of cluster computing, virtualization and Internet-based computing model, have become a popular processing resource for many organizations. Increasingly, cloud technologies are being widely used in many service fields\textsuperscript{[1][2]} to offer vast amount of computation capabilities, and also to build a service platform to accommodate a huge number of concurrent computations for different application usages. The simultaneous presence of a large amount of concurrent data streams and hugely or frequently varied computational requirements introduces significant complexity in managing, deploying and validating dynamic services in the clouds.

Many cloud services, for example, virtual firewall, web server, and intrusion detection systems\textsuperscript{[3][4]} extensively employ pattern matching methods to deal with such complexity. In these services, new patterns are frequently added, and existing patterns are constantly modified, according to the needs of dynamic services in the cloud environment, for example, dynamic security filtering\textsuperscript{[5][6]}. In practice, a frequently arise situation is to perform updates for multi-pattern matching, where multiple updates of patterns are concurrently performed in one or more cloud services. Thus, the efficiency of updating pattern set becomes critical for multi-pattern matching in cloud services.

In clouds, there exist two classic approaches for multi-pattern matching, automaton or table based methods\textsuperscript{[7]}. AC and SBOM are two typical automaton methods, where the former is based on prefix searching, whilst the latter is based on factor searching. WM, a typical table-based multi-pattern matching method, uses character block-based suffix searching\textsuperscript{[8][9]}. All of the supply links, \(\sigma - \text{shifts}\) and values go in global transversal order through the hierarchical tree to build matching automaton or character
block list to build matching table. It is, therefore, computational expensive to perform localized modifications of pattern set as this process requires global compilation and matching engine generated repeatedly.

Existing automaton or table-based multi-pattern matching methods are both too (computationally) expensive to be employed by dynamic clouds that involve frequently updated services for massively online tasks. More specifically, the classic automaton and table updating methods have two problems. 1) It can take a long time to recompile large pattern set, in the order of minutes for tens of hundreds of policies, when the environment changes. Thus, during the process, the cloud service cannot response to new requests, incurring a significantly degraded level of service. 2) Global recompilation needed for automaton or table-based methods consumes a huge amount of precious online computational resources of the cloud. In other words, a cloud platform needs to dedicate a large quantity of computing resources to satisfy the online computational requirements incurred by the expensive pattern matching process, which would subsequently reduce the availability and reliability of the cloud service. Thus, improving the efficiency of real-time multi-pattern matching and updating mechanisms becomes an important pathway to improve the service quality of cloud services.

**Contributions.** We propose a fast incremental approach, Eagle+, to provide a real-time updating in multi-pattern matching automaton and table for massively online users-oriented cloud services.

1. Eagle+ satisfies a great number of online users' pattern updating needs and meets the multi-pattern matching requirements of cloud services. It is an incremental approach that is able to perform online updates on partial states, links and values in both automaton and table.

2. Eagle+ adds, deletes or updates batches of patterns and compiles them in fast incremental operations, namely adding, updating and deleting, using depth-first traversal in Tries which are used in AC automaton, Factor Oracle structure-based SBOM automaton and List which is adopted in WM table. Moreover, the supplying connection information contained in AC automaton, shift information contained in SBOM automaton and the minimum sliding step information contained in WM table, are used together to incrementally update pattern set.

3. Eagle+ is not only applicable for AC automaton and SBOM automaton[10] but also for WM table. For all cases, a detailed theoretical analysis shows that Eagle+ can reduce the computational complexity of updating patterns from $O(n^2)$ to $O(n)$. Furthermore, our experimental results demonstrate that Eagle+ decreases nearly 72% - 92% of time consumption in AC automaton and SBOM automaton and performs 100X faster in WM table. Thus, Eagle+ is the first incremental updating approach for automaton and table, whilst enabling the throughput rate of high-speed and highly-concurrent cloud services that serve massive number of online users.

Our paper contains four further sections. Section II surveys related work on multi-pattern matching algorithms and compilation methods used by automaton and table-based approaches. Section III explains the concept and its operations implemented for updating automaton and table. Section IV gives a practical case to demonstrate the process about updating pattern set in Eagle+. Section V presents the experimental results, analysis, and discussion with existing global and local updating matching approaches and Eagle+.

## 2. Related Work

In multi-pattern matching based cloud services[11][12], string patterns are compiled in automaton or table. Generally, for AC automaton and SBOM automaton, complicated status nodes and supply links are built in different types of prefix or factor oracle searching mechanisms[13]. And WM table, the characters block-based efficient suffix searching algorithm is an extended version of BM (Boyer-Moore) under
multi-pattern matching. Here we present the current works of multi-pattern matching and compiling
approaches in pattern matching-based cloud services.

A. Current Status of Multi-Pattern Matching Algorithms

AC automaton and SBOM automaton are two classic automaton-based multi-pattern matching algo-
rithms. AC automaton is the extension of prefix searching KMP algorithm[14] under multi-pattern
matching, which constructs a special automaton based on patterns. The algorithm goes in traversal order
through Trie to build supply links $S_{AC}$[15]. So the time complexity of establishing an AC automaton is
$O(nm)$, in which $n$ is the quantity of patterns and $m$ is the average length of patterns[13].

SBOM algorithm employs the Factor Oracle-based automaton. Factor Oracle-based automaton builds
Factor Oracle from the reversed sub-string of the first $l_{\text{min}}$ length of characters for each pattern, in which
$l_{\text{min}}$ refers to the minimal length of patterns, and, for each node, calculates the existing $\sigma$–shift path
and current state along the path[16]. So the time complexity of establishing SBOM automaton is
$O(nl_{\text{min}}m)$ in which $n$ is the quantity of patterns and $l_{\text{min}}$ is the minimal length of patterns[17].

WM table, a characters block-based suffix searching algorithm, is an extended version of BM under
multi-pattern matching computing. There is a special jumping sliding window, whose length is the same
as the minimum length of pattern. In the processioning, it needs to construct a Shift table and a Hash
table with fixed block size. Hash table stores the corresponding relationship between end of plain pat-
tern characters and pattern's index, and Shift table stores sliding step for every block of characters in
sliding windows. The time complexity of establishing WM table is $O(nm)$, in which $n$ is the quantity of
patterns and $m$ is also the average length of patterns. In general terms, as string pattern sets change,
three algorithms, AC, SBOM and WM, need to recompile for establishing the new automaton or table to
replace the existing.

B. Current Status of Automatons and Tables Updating Approaches

Updating string pattern set means, by the traversal order through the Trie, rebuilding the supply paths,
$\sigma$–shifts and values for automaton and minimizing sliding windows for table. So, the global compila-
tion are unnecessary when patterns are constantly changing. However, automaton and table, in cloud
services, are sophisticated to be modified from thousands to millions times traversal order by traditional
global compiling methods[18][19].

Liu-Dynamic approach[20] is a typically fast approach for adding and deleting patterns in automaton
and table, but it can not guarantee matching speed of recompiling automaton and table and consumes
more memory. Basically, this algorithm can not guarantee the global optimal solution for supply links,
$\sigma$–shifts and sliding steps in table value. The matching speed drops significantly in [20] and our
cloud experimental environment.

In this paper, based on the multi-pattern matching algorithms in cloud services, we present an accu-
rately and rapidly incremental compiling approach to update structure and value in automaton and table.
Specially, all operations of our compiling approach are based on the local optimal strategy, which not
decrease matching efficiency. Moreover, using the novel operations of adding, updating and deleting in
complicated of automatons and tables, our approach shortens the time-consumption of updating.

3. Incremental Updating Approach for Automatons and Tables

Different automatons or tables have different pattern updating approaches. For example, AC autom-
on adopts Trie and SBOM automaton applies Factor Oracle structure, while WM table uses the byte
block shifting. These approaches, during the realization of building automaton or table, calculate the
global optimal solution for all nodes which is the minimum value chosen from the new value and pre-
vious value.

In order to achieve the incremental updating for automaton and table, the local operations are implemented by *depth-first traversal*. Such incremental updating operations executed on each slave node in cloud, *adding*, *updating* and *deleting*, are self-adaptive in the pattern set. Figure 1 shows the process of frequently updating pattern set including a batch of tasks from various users and environments in cloud. The master node collects and sorts out the pattern set and delivers it to each slave node. Then each slave node incremental updates its Automaton and Table. Next, we will illustrate the realization of the three kinds of operations in detail.

![Figure 1: Add, Update and Delete string pattern set into the automaton and table in a cloud](image_url)

**A. Notations and Definitions**

In this section, we illustrate the detailed notations and definitions of basic concepts for multi-pattern matching and updating approaches in automaton and table.

**Basic Concepts.** Given a text $T = t_1t_2\cdots t_n$ where $t_n$ is extracted from a finite alphabet set $\Sigma$, multi-pattern matching problem is to search simultaneously for a set of string pattern $P = \{p_1, p_2, \ldots, p_m\}$, where $p^i = p^i_1p^i_2\cdots p^i_{m^i}$ is a string, and $p^i_{m^i}$ is extracted from $\Sigma$. In precise, we summarize all symbols (functions) and their explanation as Table 1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma$</td>
<td>The current character in text</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>The alphabet set</td>
</tr>
<tr>
<td>$</td>
<td>P</td>
</tr>
<tr>
<td>$l_{\min}$</td>
<td>The minimum length of pattern in $P$</td>
</tr>
<tr>
<td>$l_{\max}$</td>
<td>The maximum length of pattern in $P$</td>
</tr>
<tr>
<td>$B$</td>
<td>The size of block for splitting pattern in $P$, formally $\log_2(2 \cdot l_{\min} \cdot r)$</td>
</tr>
<tr>
<td>$L(\sigma)$</td>
<td>The longest suffix of current node in automaton</td>
</tr>
<tr>
<td>$S_{AC}$</td>
<td>Supply function</td>
</tr>
<tr>
<td>$N_i$</td>
<td>The $i$-th node</td>
</tr>
<tr>
<td>$\text{Reverse}()$</td>
<td>Reverse string pattern, formally $P^{rev} = (p^1)^{rev} \ldots (p^r)^{rev}$</td>
</tr>
<tr>
<td>$\text{Shift}$</td>
<td>Shift steps, formally $s[q, pos, k] = \min(\max(d_1[q], d_3[t_{pos-k}, k]), d_2[q])$</td>
</tr>
</tbody>
</table>

Table 1: Symbol notations

For AC, SBOM and WM, there are three basic searching methods, given as follows. 1) Prefix searching, on the pattern set, builds automaton $\mathcal{A}$ by forward matching the characters one by one in the text $T$. 2) Suffix searching is implemented by backward matching, where the position is sliding along the text
T. The pos is shifted according to next probable position of the suffix read in P. 3) Factor Oracle searching is implemented by sliding matching in the text T with a position, from which the backward factor, minimal size \( l_{\text{min}} \) of the pattern string in P, is read. In this paper, as traditional way, AC automation uses the prefix searching method and SBOM automaton adopt suffix searching, while WM table utilize Factor searching and suffix searching methods. And the incremental updating operations are brought into AC, SBOM and WM to perform fast matching in text T.

**Automatons and Tables.** In this paper, automatons represent AC and SBOM, built on P. On the one hand, AC automaton is a Trie of P amplified ‘supply function’ \( S_{AC} \). Formally, \( q \) represents a state in Trie, and \( L(q) \) represents the label of the path from initial state to \( q \). The reached state means that automaton reads the longest suffix of \( L(q) \) which is also a prefix of some \( p^i \in P \). A supply link goes from each state \( q \) to \( S_{AC}(q) \), and the supply path is a chain of supply links. On the other hand, our approach creates, in addition, a transition labeled by character \( \sigma \) from each state on the supply path to the state where original transitions occur. The states of the factor oracle are of the Trie including the initial state \( l \) and the terminal states. Our approach calculates outgoing connections by formula \( \delta(\text{Current}, \sigma) = \delta(S_{AC}(\text{Current}), \sigma) \) for each new \( \sigma \). Hence, the factor oracle has at most \(|P| + 1\) states, including the initial state. As before, Trie is a prefix tree used to store dynamic or associative strings, and Factor Oracle is a finite state automaton to search sub-strings in a body of text.

![AC automaton](a) AC automaton ![SBOM automaton](b) SBOM automaton

Figure 2(a) shows the AC automaton contained pattern strings \{she, his, her\}. Our approach calculates ‘supply function’ for every node in Trie. For example, \( N_s \) points to node \( N_5 \) for the longest prefix sub-string ‘he’. The dashed links represent the state-to-state supply function. It is same for other nodes pointing to corresponding longest prefix sub-string nodes. Note that the red nodes represent final matching status.

<table>
<thead>
<tr>
<th>SHIFT</th>
<th>HASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>value</td>
</tr>
<tr>
<td>H(l)</td>
<td>1</td>
</tr>
<tr>
<td>H(no)</td>
<td>3</td>
</tr>
<tr>
<td>H(ou)</td>
<td>3</td>
</tr>
<tr>
<td>H(an)</td>
<td>4</td>
</tr>
<tr>
<td>H(un)</td>
<td>1</td>
</tr>
<tr>
<td>H(nc)</td>
<td>1</td>
</tr>
<tr>
<td>H(ua)</td>
<td>0</td>
</tr>
<tr>
<td>H(al)</td>
<td>0</td>
</tr>
<tr>
<td>H(ly)</td>
<td>0</td>
</tr>
<tr>
<td>H(nn)</td>
<td>2</td>
</tr>
<tr>
<td>H(nu)</td>
<td>2</td>
</tr>
<tr>
<td>H(œ)</td>
<td>0</td>
</tr>
<tr>
<td>H(+)</td>
<td>5</td>
</tr>
</tbody>
</table>
Table 2: Shift table and Hash table in WM

The SBOM automaton shown in Figure 2(b) implies the suffix Trie of multi-pattern set \{app, ape, pla\}. The multi-pattern may be the reverse set of \{app, aperitif, plain\}, or \{app, apec, play\} can also be. The state goes down the supply links from the parent of current state for an outgoing transition labeled with the same character as between current and its parent, creating it if it does not exist. For example, \(N_3\) points to \(N_6\) for the same suffix sub-string 'p'. Similarly, red nodes represent terminal nodes, and it matches the remained suffix to determine results. Dash lines are \(\sigma - shift\) pointing to shift nodes for efficient matching.

In addition, WM table is another important structure for matching pattern. WM table consists of Hash table and Shift table. Shift table stores the minimum of the shifts of the blocks \(B_l\) such that \(h_1(B_l)\). Precisely, our approach initializes Shift table by hashing \(l_{\text{min}} - B + 1\) characters in blocks. When the value in shift is zero, the string on the left of the search position may be one pattern string. So, we use a new hash table \(HASH\) to store index of pattern set. In Table 2, the pattern set is \{announce, annual, annually\} and the size of best matching block is 2. Please refer to [21] for more information about building AC automaton and SBOM automaton, and WM table.

B. Adding operation in Automaton or Table.

The procession of adding string pattern into \textit{automaton} and \textit{table} by incremental operations is involved in the operation, \textit{adding}, in Algorithm 1, which includes initialization, inserting and \textit{depth-first traversal} constructing for \textit{automaton} and \textit{table}.

\begin{algorithm}
\caption{Adding operation in AC automaton, SBOM automaton and WM table}
\begin{algorithmic}
\Require \text{added Pattern data set} \(D\), AC automaton or SBOM automaton or WM table
\Ensure Updated AC automaton or SBOM automaton or WM table
\Function{Adding}{\(D\), AC, SBOM, WM}
\State Initialize every value in \(D\) to the Trie value \textit{Branches} or character value \textit{Blocks};
\State Locate and Insert \textit{Branches} or \textit{Blocks} into AC automaton or SBOM automaton or WM table;
\If{Concurrent Architecture == AC || Concurrent Architecture == SBOM}
\For{\(i = 0 \to \text{Branches.Size}\)}
\State Add new supply path, \(\sigma - \text{shift}\) and values into automaton;
\EndFor
\Else
\If{Concurrent Alternative Architecture == WM}
\For{\(k = 0 \to \text{Blocks.Size}\)}
\State Select smaller values between existing and new added for \textit{Blocks};
\EndFor
\EndIf
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}
Determine the added agile (Line 4-13). To perform a depth-first traversal for building new supply paths, \( \sigma - \text{shifts} \) and values, we need to firstly retrieve Branches or Blocks in existing trie and table rather than doing a transversal order from root node. For any node \( v \in \text{Branches} \) or \( v \in \text{Blocks} \), \( P_v^d \) denotes the depth-first traversal optimal solution, and \( P_v^t \) denotes the transversal order's. The final result \( P_v^d \) and \( P_v^t \) depend on current value of nodes in Trie or Table, so the hypothesis that \( P_v^d \) is equal to \( P_v^t \) for some optimal solutions, is tenable. For the depth-first searching, the total time complexity of this steps is \( O(n) \), where \( n \) refers the length of Branches or Blocks.

C. Deleting operation in Automaton or Table

The procession of deleting string pattern from automaton or table by the incremental operation is also involved in initialization and deleting in Algorithm 2. Here, we will minutely illustrate the realization of deleting operation in automaton or table.

Determine the deleted agile (Line 4-13). Firstly, all non-shared nodes and supplying connections that point to deleted nodes are deleted in Trie. Secondly, the exclusive blocks, such as key and value, are removed from Shift table and Hash table. Finally, all nodes that point to deleted nodes will not be used for matching. The Algorithm 2 describes deleting realization for deleting pattern from automaton and table. Such as, deleting pattern 'Durian' from automaton or table, when the unique sub-string is 'ian'. So we firstly delete all supplying connections that point to nodes 'I', 'a' and 'n' in automaton. Then, our approach deletes all supplying connections that point from these located and deleted nodes. In WM table, the unique Blocks = \{ri, ia, an\}, when the size of sliding window uses the default settings 2. We can delete all Blocks values in Shift table and corresponding index value 'Durian' in Hash table. And then match and limited depth compute \{Du, ur\} in remaining pattern. Lastly, we choose the minimum sliding step in current pattern set for character blocks \{Du, ur\}. Thus, the total time complexity of this steps for freeing illegal pointers and nodes is also \( O(n) \).

Algorithm 2 Deleting operation in AC automaton or SBOM automaton or WM

<table>
<thead>
<tr>
<th>Input:</th>
<th>deleted Pattern data set D, AC automaton or SBOM automaton or WM table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>Updated AC automaton or SBOM automaton or WM table</td>
</tr>
</tbody>
</table>

1: function DELETING(D, AC, SBOM, WM)

2: Initialize every value in D to the Trie value Branches or character value Blocks;

3: Locate and Insert Branches or Blocks into AC automaton or SBOM automaton or WM table;

4: if Concurrent Architecture == AC | Concurrent Architecture == SBOM then

5: for \( i = 0 \rightarrow \text{Branches.Size} \) do

6: Delete all supply path, \( \sigma - \text{shift} \) and values into automaton;

7: end for

8: end if

9: if Concurrent Alternative Architecture == WM then

10: for \( k = 0 \rightarrow \text{Blocks.Size} \) do

11: Select smaller values between existing and deleted for Blocks;

12: end for

13: end if

return AC, SBOM, WM

end function
D. Updating operation in Automaton or Table

Both adding and deleting string pattern in automaton and table are involved in the operation updating in Algorithm 3. In addition, we can decompose the updating string pattern into local deleting and adding pattern in Algorithm 1 and Algorithm 2.

Determine the updated agile (Line 2-11). On the one hand, the algorithm finds the optimal solution and new node is pointed to new Branches or Blocks during the process of adding pattern. On the other hand, we rebuild connection among nodes affected by deleting operations in automaton, or local calculate Shift in table. Such as, new supply paths pointing to ‘r’, ‘t’ and ‘a’ nodes or deleted supply paths pointing to these nodes need to be updated in automaton. Thus, the time complexity of this step is $O(n)$. In WM table, the realization of updating the best slide step from updated shift value and limited depth decomposing shift value is the summation of Algorithm 1 and Algorithm 2. The time complexity of limited depth decomposed is $O(n)$. For example, the deleted pattern contains character block {ur}. Our approach searches for pattern {ur} and calculates shift value in current matched pattern.

Algorithm 3  Updating operation in AC automaton or SBOM automaton or WM table

<table>
<thead>
<tr>
<th></th>
<th>Input: Initialized string Pattern data set, Branches, or Blocks</th>
<th>Output: Updated AC automaton or SBOM automaton or WM table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>function UPDATAING(D, AC, SBOM, WM)</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>if Concurrent Architecture == AC</td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>for i=0 \rightarrow Branches.Size do</td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>Select optimum supply path, $\sigma$ – shift t and values between existing and potential pointing to Branches nodes'</td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td>end for</td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>7:</td>
<td>if Concurrent Alternative Architecture == WM then</td>
<td></td>
</tr>
<tr>
<td>8:</td>
<td>for k=0 \rightarrow Blocks.Size do</td>
<td></td>
</tr>
<tr>
<td>9:</td>
<td>Select smaller or higher values between existing and add or delete for Blocks;</td>
<td></td>
</tr>
<tr>
<td>10:</td>
<td>end for</td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>12:</td>
<td>return AC, SBOM, WM</td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>end function</td>
<td></td>
</tr>
</tbody>
</table>

Reducing Order Mechanism. In adding, updating and deleting operations, the time-consumption is $O(n)$ and all operations are chosen separately. Thus, the time-consumption of Eagle+ is $O(n)$. Comparing Eagle+ with traditional global updating approaches, it reduces the computational complexity of updating automaton and table from $O(n^2)$ to $O(n)$. In Table 3, $n$ represents the quantity of patterns, and $m$ represents the average length of patterns and $l$ represents the minimal length of patterns.

<table>
<thead>
<tr>
<th>Items</th>
<th>AC</th>
<th>SBOM</th>
<th>WM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global approach</td>
<td>$O(nm)$</td>
<td>$O(nl)$</td>
<td>$O(nl)$</td>
</tr>
<tr>
<td>Eagle+ approach</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
4. A Case Study of Updating in Automaton and Table

In this section, we present a detailed example about updating operations, adding, deleting and updating in AC automaton, SBOM automaton and WM table individually.

A. Updating pattern in AC automaton

As is shown in Figure 3(a), AC automaton contains patterns \{annual, annually, announce\}. The red nodes represent terminal status, which mean a successfully matching process. The dash lines are supply links meaning where the current node goes to when matching failures.

1. Adding pattern into AC automaton

As is shown in Figure 3(b), the new pattern, like \{nonce\}, will be added into AC automaton. Firstly, our approach compares \{nonce\} with existing pattern and locates the last prefix matched node in Trie, which is null in current case. Null common prefix means root \(N_0\), as a new node \{nonce\}. Then, the remaining characters of the new pattern are added into Trie beginning of the shared \(N_0\). Next, each supply path of nodes, ‘n’, ‘o’, ‘n’, ‘c’ and ‘e’, is calculated through depth-first traversal. For example, nodes \(N_{14}\), \(N_{15}\), \(N_{17}\) and \(N_{18}\) point to the root node, and node \(N_{16}\) points to node \(N_{14}\), as shown in Figure 3(b). Finally, the existence of some new supply paths between old nodes and new nodes is checked, and the deeper path is chosen from old supply link and new supply link that points to new nodes. The time complexity of limited depth-first traversal is \(O(n)\). For example, there are old supply links from nodes \(N_2\), \(N_3\), \(N_5\) and \(N_9\) to node \(N_0\) and new supply links from nodes \(N_2\), \(N_3\), \(N_5\) and \(N_9\) to nodes \(N_{14}\) and \(N_{15}\), so our approach chooses the latter links as the optimal path. All optimal supply paths and new nodes are recorded into automaton.

2. Deleting pattern from AC automaton

When our approach needs to delete pattern \{nonce\}, the process of deleting is shown in Figure 3(c). Firstly, our approach locates non-shared nodes corresponding to \{nonce\} in Trie. Then, all nodes following the shared root (e.g., node \(N_{14}\), \(N_{15}\), \(N_{16}\) and \(N_{17}\)) are deleted, and the supply links corresponding deleted nodes are removed. Lastly, our approach recalculates the supply links that point from remaining nodes to deleted nodes, and stores links into automaton. For example, nodes \(N_2\), \(N_3\), \(N_5\) and \(N_9\) point to the deleted nodes, \(N_{14}\) and \(N_{15}\), and the latest supply link pointing to node \(N_0\) are updated, as shown by green dash lines.
B. Updating pattern in SBOM automaton

The original SBOM automaton is shown in Figure 4(a), which represents the Factor Oracle structure of patterns \{annually, annual, announce\}. The reverse string of the first \(l_{\text{min}}\) elements of patterns are \{launna, nuonna\}, in which \(l_{\text{min}}\) is equal to 6 referring to the shortest length of pattern \{annual\}. Red nodes mean terminal nodes, and the dash lines are the \(\sigma - shift\) from each node on the supply path to next node.

1. Adding pattern into SBOM automaton

The new prefixal \{snnosc\} patterns will be added into SBOM automaton. Firstly, our approach reverses the pattern string and truncates the first \(l_{\text{min}}\) characters which are discarded as \{csonns\} shown in Figure 4(b). Secondly, our approach locates the shared root from the root node, and adds 'c', 's', 'o', 'n', 'n' and 's' nodes into the automaton. Then the \(\sigma - shift\) for new added nodes is calculated by depth-first traversal. For example, the \(\sigma - shift\) of node \(N_{14}\) points to node \(N_4\). Finally, the existence of some new optimal paths between old nodes and new nodes is checked. For example, there are new supply links from nodes \(N_2\) and \(N_{10}\) to node \(N_{18}\), so we choose the latter link as the optimal path. All optimal supply paths and new nodes are recorded into Factor Oracle structure automaton.

2. Deleting pattern from SBOM automaton

The original patterns are represented as \{announce, annual, annually, snnosc\}. Two patterns, \{annoual\} and \{annually\}, will be deleted. Firstly, we reverse pattern string and truncate the first \(l_{\text{min}}\) elements, which is \{lauonna\}. Finally, our approach locates the shared root (node \(N_2\)) and deletes corresponding nodes following the shared root, such as node \(N_2, N_4, N_6, N_{16}, N_{10}\) and \(N_{12}\), and all the related \(\sigma - shift\) are drawn by red lines. Then, the new target nodes for \(\sigma - shift\) that point to the deleted nodes are recalculated in automaton. For example, nodes \(N_{16}\) and \(N_{17}\) point to nodes \(N_9\) and \(N_{12}\), as shown in Figure 4(c).
Table 6: Deleted WM Shift table and Hash table

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>H(sh)</td>
<td>1</td>
<td>H(sh)</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>H(he)</td>
<td>0</td>
<td>H(he)</td>
<td>1(she)</td>
</tr>
<tr>
<td>H(hi)</td>
<td>1</td>
<td>H(hi)</td>
<td>5(shi)</td>
</tr>
<tr>
<td>H(er)</td>
<td>0</td>
<td>H(er)</td>
<td>2(hers)</td>
</tr>
<tr>
<td>...</td>
<td>2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>H(ki)</td>
<td>1</td>
<td>H(ki)</td>
<td>...</td>
</tr>
<tr>
<td>H(is)</td>
<td>0</td>
<td>H(is)</td>
<td>3(his)4(kiss)6(niss)</td>
</tr>
<tr>
<td>...</td>
<td>2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>H(ha)</td>
<td>1</td>
<td>H(ha)</td>
<td>...</td>
</tr>
<tr>
<td>H(as)</td>
<td>0</td>
<td>H(as)</td>
<td>5(has)</td>
</tr>
<tr>
<td>H(ni)</td>
<td>1</td>
<td>H(ni)</td>
<td>...</td>
</tr>
</tbody>
</table>

C. Updating pattern in WM table

Table 4 records the original WM shift table and Hash table, which contains patterns [she, hers, his, kiss]. While the size of block is set to 2 in current scenario, the best matching will perform. The Hash table lists the corresponding relation of the last block of characters and indexes value of patterns. And the shift table stores the important moving steps for every block during the process of character matching. H is a mapping function representing moving steps and indexes relationship in SHIFT table and HASH table.

1. Adding pattern into WM table

The new patterns [shi, niss] will be added into WM table. Firstly, our approach adds the exclusive last block of characters into hash table and indexes value i.e. [(H (hi), 5), (H (is), 6)], as is shown in Table 5. Then, the moving steps for new block of characters i.e. [(H (ni), 1)] are added into table, and the shift table is updated by minimum new moving step i.e. from [(H (hi), 1)] to [(H (hi), 0)]. The values of hash table and shift table are the same with calculated by global compiling.

2. Deleting pattern from WM table

Table 6 represents the process of deleting patterns [niss] from [she, hers, his, kiss, has, niss]. Since WM tables have some same characters on different patterns, our approach just traverses segmented bytes and search minimum shift value from remaining patterns. Firstly, our approach removes the deleted pattern's value from hash table. For example, the pattern [H(is), niss] will be removed from Hash table. The deleted or updated index and value set in Shift table can not be removed by only character matching with fixed block. Lastly, the approach traverse remained patterns, segmented characters by deleted patterns, with limited depth, and choose a minimum shift value. If the minimum shift equals 0, then remove same with current's index and value from Hash table, as shown in Table 6 with red bold style.

5. Experiments and Results

In this section, we evaluate the performance of Eagle+ approach via simulating multi-tenant controlled cloud matching services.

Experimental Setup. In order to verify the validity of accurately incremental updating in Eagle+ via reducing many-many backtracking to a limited depth of one-many and many-one backtracking, our approach frequently adds or deletes batch of patterns into or from automaton and table. Thus, Eagle+ is applied to cloud gateway device and open network intrusion detection system of a multi-user and mul-
ti-services oriented cloud platform. Core hardware configuration includes Intel Xeon E5-240 1.9GHz CPU, 64GB DDR3 memory, 10Gbit network card in Mid-range firewall, cloud gateway and cloud engine switch. And the main programming language is C++.

According to the discussion in [23][24][25], Eagle+, in the process of incremental building automaton, preserves consistency as the level order traversal based on all data. Therefore, we only consider two benchmark, speed and acceleration.

**Experimental data set.** We choose two different types of experimental data set for simulating cloud security services. One data set is a flooded streaming captured in network and mixes with a variety of harmful string traffic, while its maximal peaking is up to 128 Gbps in hybrid cloud platform. The other data set consists by the rules that contain patterns, e.g., Snort and ModSecurity, URLs of phishing websites, harmful porn websites. However, the requirements being URLs of phishing websites and harmful porn websites vary greatly to 10000 different simulate cloud-users.

Simulated user-behavior about adding, deleting, updating harmful patterns is evenly distribution, and is linearly dependent with users-flow. In cloud security services, the maximum parallel of filter engine increased by peak flow is up to 100. Table 7, 8 and 9 record the time consumption of the Eagle+, Liu-Dynamic approach and global updating approach to add or delete patterns in AC automaton, SBOM automaton and WM table.

**Comparison in AC Automaton.** Table 7 shows the time consuming of random adding and deleting 20 types of patterns under various quantity in AC automaton. Comparing the Eagle+ with Global updating approach and Liu-Dynamic approach under the same quantity of patterns, the time consuming of all approaches increases when the quantity of patterns is rising. However, the time consuming of global updating approach is far more than other approaches. In addition, the time consuming of Liu-Dynamic is a little less than our approach, even though the complexity of Liu-Dynamic and Eagle+ is \(O(n)\). 

<table>
<thead>
<tr>
<th>Quantity of patterns</th>
<th>Global adding approach</th>
<th>Liu-Dynamic adding approach</th>
<th>Eagle+ adding approach</th>
<th>Global deleting approach</th>
<th>Liu-Dynamic deleting approach</th>
<th>Eagle+ deleting approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0144</td>
<td>0.00058</td>
<td>0.001352</td>
<td>0.0127</td>
<td>0.0003036</td>
<td>0.002244</td>
</tr>
<tr>
<td>1000</td>
<td>0.12096</td>
<td>0.005461</td>
<td>0.025772</td>
<td>0.11563</td>
<td>0.0028188</td>
<td>0.019512</td>
</tr>
<tr>
<td>10000</td>
<td>1.1602</td>
<td>0.051632</td>
<td>0.28944</td>
<td>1.1475</td>
<td>0.0235383</td>
<td>0.14472</td>
</tr>
<tr>
<td>10000</td>
<td>10.89011</td>
<td>0.31335</td>
<td>1.960044</td>
<td>10.8486</td>
<td>0.1817513</td>
<td>1.220344</td>
</tr>
<tr>
<td>100000</td>
<td>74.09892</td>
<td>1.470087</td>
<td>9.443248</td>
<td>74.01473</td>
<td>0.8337808</td>
<td>5.214928</td>
</tr>
</tbody>
</table>

Table 7: Time consumption among the Eagle+, the global approach and Liu-Dynamic approach in AC automaton

Figure 5(a) shows the matching speed of different approaches under various quantity of patterns in AC automaton. Note that Y-axis describes the scale of log. Obviously, Eagle+ and global approach keep a stable matching speed, but the Liu-dynamic approach cannot guarantee the matching speed when the quantity of patterns increases. And both global approach and Liu-dynamic approach are not suitable for matching vast amount of patterns which are frequently updating in cloud. For Eagle+, the experimental results of speedup are shown in Figure 5(b). When the quantity of existing patterns is too less or too more, the speedup for AC automaton is more significant. The deleting operation implemented in Eagle+ can accelerate 92% of matching speed by incremental learning, while the matching speed of the add operation rises 72%.
Comparison in SBOM Automaton. The time consuming of random adding and deleting 20 types of patterns under various quantity in SBOM automaton is shown in Table 8. The time consumption rises with the increase of the size of pattern string. Comparing with AC automaton, SBOM automaton has less computational cost because of shorter length of structure limited by $l_{\text{min}}$. However, we can see that Liu-dynamic adding and deleting approaches consume a lot of time.

<table>
<thead>
<tr>
<th>Quantity of patterns</th>
<th>Global adding approach</th>
<th>Liu-Dynamic adding approach</th>
<th>Eagle+ adding approach</th>
<th>_global deleting approach</th>
<th>Liu-Dynamic deleting approach</th>
<th>Eagle+ deleting approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0064</td>
<td>0.00121</td>
<td>0.0051</td>
<td>0.000313</td>
<td>0.000112</td>
<td>0.00136</td>
</tr>
<tr>
<td>1000</td>
<td>0.04032</td>
<td>0.008699</td>
<td>0.03786</td>
<td>0.002584</td>
<td>0.0013488</td>
<td>0.009485</td>
</tr>
<tr>
<td>10000</td>
<td>0.43204</td>
<td>0.10884</td>
<td>0.42092</td>
<td>0.025057</td>
<td>0.007356</td>
<td>0.089244</td>
</tr>
<tr>
<td>100000</td>
<td>2.97003</td>
<td>0.70161</td>
<td>2.94639</td>
<td>0.158163</td>
<td>0.0287964</td>
<td>0.53176</td>
</tr>
<tr>
<td>1000000</td>
<td>24.69964</td>
<td>4.862568</td>
<td>24.64238</td>
<td>0.658248</td>
<td>0.5025548</td>
<td>2.184632</td>
</tr>
</tbody>
</table>

Table 8: Time consumption among the Eagle+, the global approach and Liu-Dynamic approach in SBOM automaton

The matching speed of different approaches under changing pattern requirements in SBOM automaton is shown in Figure 6(a). Note that Y-axis also describes the scale of log. Obviously, Eagle+ and global approach keep a stable matching speed, but the Liu-dynamic approach cannot guarantee the matching speed because it is not optimization and global $\sigma$ – shift affected nodes in updating. Both global approach and Liu-dynamic approach are not suitable for matching vast amount of patterns with frequently updating in cloud.

Figure 6(b) shows the speedup of Eagle+ in SBOM updating. X-axis demonstrates the size of pattern.
that describes the log scale. The speedup for SBOM automaton rises to 91% in deleting approach. To further arises the speedup in deleting and adding approaches, SBOM automaton employs Factor Oracle structure, while the AC automaton is built by Trie.

<table>
<thead>
<tr>
<th>Quantity of patterns</th>
<th>Global adding approach</th>
<th>Liu-Dynamic adding approach</th>
<th>Eagle+ adding approach</th>
<th>Global deleting approach</th>
<th>Liu-Dynamic deleting approach</th>
<th>Eagle+ deleting approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0006</td>
<td>1.2 × 10^{-7}</td>
<td>1.2 × 10^{-7}</td>
<td>0.0005</td>
<td>3.2 × 10^{-8}</td>
<td>4.8 × 10^{-7}</td>
</tr>
<tr>
<td>1000</td>
<td>0.0063</td>
<td>1.76 × 10^{-7}</td>
<td>1.7 × 10^{-7}</td>
<td>0.0061</td>
<td>4.3 × 10^{-8}</td>
<td>5.44 × 10^{-7}</td>
</tr>
<tr>
<td>10000</td>
<td>0.0600</td>
<td>1.76 × 10^{-7}</td>
<td>2.2 × 10^{-7}</td>
<td>0.0596</td>
<td>1.76 × 10^{-7}</td>
<td>8.8 × 10^{-7}</td>
</tr>
<tr>
<td>100000</td>
<td>0.5799</td>
<td>2.48 × 10^{-7}</td>
<td>3.1 × 10^{-6}</td>
<td>0.5742</td>
<td>2.24 × 10^{-6}</td>
<td>1.14 × 10^{-6}</td>
</tr>
<tr>
<td>1000000</td>
<td>5.5443</td>
<td>4.86 × 10^{-7}</td>
<td>5.4 × 10^{-5}</td>
<td>5.5329</td>
<td>1.16 × 10^{-5}</td>
<td>2.1 × 10^{-5}</td>
</tr>
</tbody>
</table>

Table 9: Time consumption among the Eagle+, the global approach and Liu-Dynamic approach in WM table

(a) Matching speed comparison of WM table after global updating approach, Liu-Dynamic approach, and Eagle+
(b) Speedup for Eagle+ in WM table

Figure 7: Performance of WM table

**Comparison in WM Tables.** The time consumption of random adding and deleting 20 types of patterns under various quantity in WM tables is shown in Table 9. The time consumption rises with the increase of the size of pattern string. Comparing with AC automaton and SBOM automaton, WM table has less time consumption because of smaller computational complexity in selecting smaller slide step. Both Liu-dynamic and Eagle+ approaches have shorten updating time consumption than global approach, and perform about 100X faster in table.

From Figure 7(a), there are significantly decrease in Liu-dynamic add approach. Thus, Eagle+ in WM approaches can be applied into frequently updating patterns for matching-based cloud services. Since the difference between Eagle+ adding and deleting in time consumption is relatively smaller than global approach, the speedup curves of Eagle+ approaches are adjacent, as in Figure 7(b).

Comparing with global approach, the experimental results show that Eagle+ reduces 72% - 92% time consumption in automatons and performs 100X faster in matching table. Meanwhile, the time consumption of deleting operation in our approach is even less in AC and SBOM automatons, and the time consumption of adding operation is less in WM tables. Not only can Eagle+ be applied to multi-pattern matching based cloud services, also it can be applied to other network security filtering, such as traditional DPI/IDS/IP/IPS/NBA, etc. Besides, the experimental results show that, in WM algorithm, sometimes there is no need considering the time consumption for Eagle+. Our approach successfully reduces the computational complexity of adding and deleting patterns from $O(n^2)$ to $O(n)$, and, as shown in Table 7, 8 and 9, Eagle+ maintains a high match efficiency under a high-speed and high-volume cloud network environment.

**6. CONCLUSION**

In this paper we present a fast incremental approach Eagle+ to automaton and table online updates
for cloud services. Eagle+ performs adding, updating and deleting operations for batch of new patterns in AC automaton and SBOM automaton and WM table. To demonstrate the effectiveness and efficiency, we performed systematic evaluation on both updating time consumption and matching speed. The results show that our incremental approach is significantly faster than global updating, and performs more stable matching efficiency. In theory, our approach successfully reduces the computational complexity of updating multi-pattern set from $O(n^2)$ to $O(n)$. The natural future work is to extend our approach to other advanced incremental DFA and NFA [22] matching for more widely used cloud services.

As far as we know, this paper is the second one to give a detailed illustration on operations, adding, deleting and updating, in automaton and table to update patterns. In [10], we only give a general concept of updating in automaton. In the future, the proposed approach will be applied to some more complex and various cloud services environment, and refine in regular expressions matching algorithms.

**Acknowledgements**

This work was supported by China MOST project (No.2012BAH46B04).

**References**


Eagle+: A Fast Incremental Approach to Automaton and Table Online Updates for Cloud Services
Jianxin Li\textsuperscript{a}, Hao Peng \textsuperscript{a}, Erica Yang\textsuperscript{b}, Chunming Hu\textsuperscript{a},\textsuperscript{*}, Shenghai Zhong\textsuperscript{a}, Lihong Wang\textsuperscript{c}\textsuperscript{*}

\textsuperscript{a}School of Computer Science and Engineering, Beihang University, Beijing, 100191, China
\textsuperscript{b}Scientific Computing Department, STFC Rutherford Appleton Laboratory, Oxfordshire, UK
\textsuperscript{c}National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China

Biographies

\textit{Jianxin Li}, associate professor in School of Computer Science and Engineering, Beihang University. His current research interests include big data, distributed system, virtualization, trustworthy computing and network security. Email: lijx@act.buaa.edu.cn.

\textit{Hao Peng}, is currently a Ph.D. candidate at the School of Computer Science and Engineering in Beihang University (BUAA), China. His research interests include network security, machine learning, cloud computing. Email: penghao@act.buaa.edu.cn.

\textit{Erica Yang}, is a senior computer scientist with Scientific Computing Department, STFC Rutherford Appleton Laboratory. Her current research interests include high throughput systems, data management, high dimensional visualization, semantics analytics and big data mining. Email: erica.yang@stfc.ac.uk.

\textit{Chunming Hu}, associate professor in School of Computer Science and Engineering, Beihang University. His current research interests include network security, cloud computing, big data mining and analytics. The corresponding author. Email: hucm@act.buaa.edu.cn.

\textit{Shenghai Zhong}, is currently a Ph.D. candidate at the School of Computer Science and Engineering in Beihang University (BUAA), China. His research interests include privacy preserving, big data mining and statistical learning. Email: zhongsh@act.buaa.edu.cn.

\textit{Lihong Wang}, professor in National Computer Network Emergency Response Technical Team/Coordination Center of China. Her current research interests include information security, cloud computing, big data mining and analytics, Information retrieval and data mining. The corresponding author. Email: wlh@isc.org.cn.
Eagle+: A Fast Incremental Approach to Automaton and Table Online Updates for Cloud Services

Jianxin Li\textsuperscript{a}, Hao Peng\textsuperscript{a}, Erica Yang\textsuperscript{b}, Chunming Hu\textsuperscript{a*}, Shenghai Zhong\textsuperscript{a}, Lihong Wang\textsuperscript{c*}

\textsuperscript{a}School of Computer Science and Engineering, Beihang University, Beijing, 100191, China
\textsuperscript{b}Scientific Computing Department, STFC Rutherford Appleton Laboratory, Oxfordshire, UK
\textsuperscript{c}National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China

Biographies

\textit{Jianxin Li}

\textit{Hao Peng}

\textit{Erica Yang}