An Efficient Regular Expressions Compression Algorithm From A New Perspective

Tingwen Liu, Yong Sun, Yifu Yang, Li Guo, Binxing Fang
Institute of Computing Technology, Chinese Academy of Sciences, 100190
Email:liutingwen@software.ict.ac.cn

Abstract—Network security applications use more regular expressions to represent patterns to perform deep packet inspection. Standard DFA engine is usually used to implement regular expressions matching, because it only need $O(1)$ time to process one input symbol. However, DFAs of regular expression sets require large amount of memory, which limits the practical application of regular expressions in the high-speed networks. Some compression algorithms have been proposed to address this issue in recent works.

In this paper, we reconsider this problem from a new perspective, namely observing the characteristic of transition distribution inside each state, which is different from previous schemes that observe the characteristic among states. Furthermore, we introduce a new compression algorithm which can reduce 95% memory usage of DFA stably without significant impact on matching speed. Moreover, our work is orthogonal to previous compression algorithms, such as D2FA, δFA. Our experiment results shows that applying our work to them will have several times memory reduction, and matching speed of up to dozens of times comparing with original δFA in software implementation.

I. INTRODUCTION

Increasingly, modern network security services inspect incoming and outgoing packets not only by the fields of packet headers, but also by the content of packet payloads. Deep packet inspection (DPI) is widely recognized as a powerful and important technology used in network security and application-specific services, such as Intrusion Detection/Prevention Systems, Firewalls, traffic monitoring as well as packet classifier. Traditionally, string-based signatures have taken root as a predominant presentation of patterns. Currently, regular expressions are replacing exact strings to describe patterns in most popular software tools–including Snort[1], Bro[2], L7-filter[3], and some devices by many companies.

The widespread use is due to their expressive power, simplicity and flexibility for expressing signatures.

Deterministic Finite Automata (DFA) is a commonly used engine of regular expressions to match packet payloads. In the pattern matching context, DFAs have two major advantages: it requires only a single table lookup for each incoming character, and it is possible to compile multiple regular expressions into a composite DFA that recognizes all patterns in a single pass over the inputs. Unfortunately, the states of the composite DFA may experience exponential growth in size, which may be much larger than the total states of individual DFAs. For example, the composite DFA of L7-filter set require more than 16 GB memory space, while all individual DFAs only consume less than 9 MB memory.

On the other hand, we can use another classical engine, called Nondeterministic Finite Automata (NFA), to help inspect packet payloads. NFAs have modest memory requirements, but its matching speed is very slow. Thus, neither DFAs or NFAs is suitable to be used directly in high-speed network environments. Many states and transitions compression algorithms for DFAs have been proposed in order to reduce memory usage to a manageable complexity. However, most of them achieve their goals by either requiring hardware support, depending on specialized architectures, or at the cost of serious degradation of matching speed.

In this paper, we focus on reducing memory usage of composite DFAs by compressing transitions. Previous works are based on the observation that there are many common outgoing transitions for the same character among some states. So they are one-dimension algorithms which only account for the transitions redundancy between states. However, in this paper we try to obtain memory reduction by exploiting transitions redundancy among states and transitions distribution inside states. To the best of our knowledge, we are the first to rethink transitions compression from two-dimensions. We start our work with the observation that although each state may have relatively high number of unique 'next-states', its transitions concentretively transfer to a two clusters concentratively. Based on the observation, we adjust transitions in a cluster separately for each state by extracting base value to create more common transitions between states. Moreover, the work is orthogonal to previous compression algorithms, such as D2FA [4] and δFA [5], and their combination can obtain higher compression rates.

In summary, the major contribution of our work is the notion of solving memory usage in a DFA using transition compression from two-dimensions. To this end, we make the following specific contributions:

- We are the first to observe the transitions distribution inside state, and argue that it is essentially correct for all types of regular expressions [6], while previous observations only are effective on some subsets.
- We introduce a new transitions compression algorithm from two-dimensions.
- We extend our observation to previous algorithms, and introduce some optimizations to obtain positive improvement both in memory usage and matching performance.
- We present an analysis of our scheme, and perform a systematic experimental study comparing our work with...
II. RELATED WORK

Regular expression matching is a classical problem of computer science and technology. Previous works have [7], [8], [9] made fruitful steps to promote the research of regular expression matching in algorithms and theories. With the widespread use of regular expressions in many network applications, people pay more attention to its memory consumption and matching performance in practice. In this context, how to reduce the memory usage of DFAs is the hotspot of related researches.

Fang Yu et al. study the complexity of DFAs for typical patterns used in real-world packet payload scanning applications, and classify them into five types [6]. Then they propose two rewriting rules for the last two types whose DFAs experience quadratic and exponential growth in the number of states. However, the rewriting rules are only effective on some special regular expressions. For instance, it can rewrite $\text{AUTH} \backslash \{[^{\wedge}a][^{\wedge}b]\} \{1, 4\}$ (the example in [6], which detects an IMAP authentication overflow attack in Snort). But if we append some characters in the tail, the rewrite rules will be invalid, such as the pattern of iMesh in L7-filter [3].

Kumar et al. introduce a data structure called D2FA to reduce storage space of DFA [4]. It is an extension of a standard DFA, where if two states have common transitions, one of the states is made to "point" to the other by adding a default transition without consuming an input character. D2FA can greatly reduce the number of transitions, its experiments show that it can reduce an average 95% of transitions. But it is possible that an input character can lead to multiple default transitions before it is consumed along a normal transition, which entails a memory bandwidth increase to evaluate regular expressions.

Ficara et al. present a new representation of DFAs, called Delta Finite Automata (δFA) [5], which is based on the observation that most adjacent states share a large part of the same transitions. It constructs equivalent states (states have the same identical transition set) by removing the same transitions between parent state and child state. For current state $s$ and the input character $c$, if $s$ share the same transition with its all the parent states for $c$, we can omit to store the transition. δFA can achieve very good compression effect, but it need $O(|\Sigma|)$ ($\Sigma$ is the character set) time to update transitions of current state, which is very time-consuming.

Smith et al. propose extended finite automata (XFAs) [10], [11] to solve the state explosion problem described in [6], which augments traditional DFAs with finite scratch memory and instructions to manipulate this memory. This method works well to have matching speeds approaching DFAs yet memory requirements similar to NFAs. But it has the following three shortcomings: (1) for each state, it potentially need to update the scratch memory after processing each input character; (2) it solve the problem with the premise of no overlap between $S$ and $S'$ for regular expressions of the form $(^*S)^*$; (3) it only takes into account of length restrictions on a single character, but frequently sub-expressions may repeat several times in regular expressions, such as $(^*[a][^*b])\{1, 4\}$, XFAs cannot handle this case.

Recently, alphabet compression technique is employed to reduce memory requirement [12], [13], [14]. The technique can dramatically reduce alphabet size by mapping the set of symbols found in an alphabet to a smaller set by grouping characters that label the same transitions everywhere in the automaton. Kong et al. refine the technique by introducing multiple alphabet compression tables, which partitions the set of states in a DFA and creates compression tables for each subset in a way that yields further reductions in memory usage[15]. In fact, these solutions exploit the transitions redundancy among all states or subset states.

Multi-stride DFAs were proposed in [12] as a way to increase processing throughput in the context of small DFAs. Specifically, a stride-$k$ DFA consumes $k$ characters per state transition rather than just one, thus it yields a $k$-fold performance increase. Nan Hua et al. introduce variable-stride multi-pattern matching for deep packet inspection [16], which is inspired by the Winnowing scheme originated developed for document fingerprinting. However, this scheme cannot hand regular expressions of small length.

III. TRANSITIONS IN DFAS

In this section, we conclude transition characteristics in DFAs from three aspects and give an explanation of our observation.

A. transition characteristics

1) characteristic when matching: All the transitions can be classified into four categories according to their different functions[17]: basic transitions (successfully accept pattern set from start state), cross transitions (transfer from one pattern to another or one part to another part within one pattern), restartable transitions (transfer current state to states next to start state) and failure transitions (transfer current state to start state). Typically Aho-Corasick(AC) and its derivative algorithms build a DFA from string pattern set which has large proportion of failure transitions and restartable transitions. Michela Becchi extends this observation to regular expressions matching and argue that the vast majority of transitions between states lead back to the start state or its near neighbors [13], we call them magic states in this paper. As Table I shows, for different regular expression sets (see section 6 for more details on sets) the percentage of transitions that transfer to magic states in DFAs (Column 4, PT$_{dfa}$). From Table I we know that above 90% of transitions in DFAs transfer to magic states for type-1 set (illustrated in [6]), which is only composed of exact strings, such as $(^*abcd^*)$. However, the proportion is not that high for some regular expressions sets of the Snort and Bro, namely the observation is not very suitable for regular expressions.

We generate a 1 MB text for each pattern set, which can match successfully every regular expression in it 2 times. When matching the text with corresponding pattern set, we
find that a majority of transitions during the matching process transfer to magic states. Mostly the proportion is above 99%, as shown in Table I (Column 6, $P^\star_{\text{matching}}$). We can use this observation to speedup the matching performance of regular expressions. The first column is the rank of each protocol, the third is the proportion of respective traffic for each protocol, and the last is the total proportion.

2) characteristic among states: Many states, sometimes all states, have the same outgoing transitions for the same character usually in DFAs, this observation was first illustrated in [18]. Many literatures [4], [5], [19], [15] compress DFAs by retrieving and exploiting the intrinsic transitions redundancy among specific state pairs, such as adjacent states.

3) characteristic inside states: For a given state of DFAs, its outgoing transitions transfer to many distinct "next-states". Fortunately, this observation is useless to reduce memory usage of DFAs, because the number of distinct "next-states" is more than 25 on average, as shown in Table II (Column 2).

Fortunately, we observe that the average number of distinct clusters is usually less than 5 (Column 3), which is much smaller than that of distinct "next-states". Furthermore, the transitions of each state concentivate transfer to the maximum two clusters (Top-2 clusters for short). Column 4 is the total proportion of transitions that transfer to Top-2 clusters for the DFA of each regular expression set. We can find that these transitions occupy more than 95% of all transitions from TableII.

<table>
<thead>
<tr>
<th>pattern set</th>
<th>average distinct &quot;next-states&quot;</th>
<th>average distinct clusters</th>
<th>% transitions in Top-2 clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>snort-24</td>
<td>14.49</td>
<td>5.06</td>
<td>97.89</td>
</tr>
<tr>
<td>snort-31</td>
<td>12.24</td>
<td>4.75</td>
<td>97.92</td>
</tr>
<tr>
<td>snort-34</td>
<td>13.11</td>
<td>4.69</td>
<td>98.21</td>
</tr>
<tr>
<td>bro-217</td>
<td>54.29</td>
<td>4.23</td>
<td>97.95</td>
</tr>
<tr>
<td>type-1</td>
<td>47.79</td>
<td>1.93</td>
<td>99.96</td>
</tr>
</tbody>
</table>

B. Description

We have introduced transition characteristic inside states for DFAs, in this subsection we give a description in detail for this observation using regular expression $^*A\{2\}CD$ as an example. Fig. 1 show the DFA for the example.

Obviously, DFA is a digraph, thus we can get a unique determinate trie-tree after traversing DFA by level (breadth first traversal), if we stipulate that we traverse the son states by the label from small to large. Fig 2 shows the trie-tree by level traversing DFA of $^*A\{2\}CD$. At the beginning of this work, we give the definition of cluster first.

Definition 1. In the trie-tree, if state $r$ has a transition to state $s$, we call $r$ is the father state of $s$, conversely $s$ is the son state of $r$. A states set is called a cluster if it is composed of all the son states of a certain state.

The start state is not any state’s son, so it doesn’t belong to any cluster. However, we can think it there is a separate cluster, which has only one state: the start state. Then each state in trie-tree belongs to only one cluster. The proof is trivial. Each state is the son of a certain state, so it belong to the cluster that is composed of all the sons of its father. Furthermore, it cannot occurs in more than one cluster simultaneously, otherwise it has at least two distinct father states, which is not impossible in trie-tree. Thus, cluster is a mutually exclusive partition of nodes of trie-tree in essence. According to the definition of cluster, the start state set $\{0\}$ is a cluster. The son states set of state 0 is $\{1\}$, so $\{1\}$ is also a cluster. By the same token, the state sets $\{2, 3\}$, $\{4, 5\}$, $\{6, 7\}$, $\{8\}$ $\sim \{13\}$ are clusters too. Thus we can get a unique determinate cluster partition from the trie-tree, which is also a partition of all DFA states in actual.

In DFAs, each state has $|\Sigma|$ transitions ($\Sigma$ is the input symbols set, it contains $2^8$ symbols for the extended ASCII code), and each transition transfer to a certain determinate state, thus it transfer to a determinate cluster from above analysis. Each state has fewer distinct clusters than distinct "next-state", because each cluster contains at least one state. We can divide all the transitions and store them into three different matrices $T_1$, $T_2$, $T_3$ as shown in Fig.3. For each state, the maximum transitions that transfer to the same cluster were put into matrix $T_1$. Similarly, the second maximum transitions into matrix $T_2$, and the remaining transitions into matrix $T_3$. For example, all the transitions of state 1 transfer to state 2 or 3, both of which belong to cluster $\{2, 3\}$, thus they are all put into $T_1$, and $T_2$ and $T_3$ have no transitions for
We divide the DFA matrix as described in the last section: put the transitions that transfer to the TOP-2 clusters into two separate matrixes T1 and T2, and the remaining transitions into T3 (the transitions of the same state in T3 may transfer to more than one cluster).

2) Matrix Compressing: Matrixes T1 and T2 have the same structure: the transitions of each state transfer into the same cluster, so we use the same algorithm to compress them, and use another different algorithm to compress T3. Obviously matrix T3 is a sparse matrix, in this paper we do not discuss how to compress it but use classical sparse matrix compression algorithm. Before describing compression algorithm of matrix T1 and T2, we introduce a new definition for 

\textit{combinative row.} \textbf{Definition 2.} In matrix \(M (M \text{ is } T1 \text{ or } T2), \text{ for row } s, \text{ if there is a row } r \text{ which meets the following conditions: } \forall c \in \Sigma, M[r, c] = X \text{ or } M[s, c] = X \text{ or } M[r, c] = M[s, c], \text{ we say that row } r \text{ is a combinative row of row } s, \text{ or rows } r \text{ and } s \text{ are combinative row.} \}

If rows \(r \text{ and } s \) are combinative row, we process them according to the rules as follows: \(1) \) for character \( c \in \Sigma, M'[r, c] = M[r, c] \text{ or } M'[s, c] = M[r, c] \text{ or } M'[r, c] = M[s, c], \) \(M'[r, c] = M[s, c], \) \(c \in \Sigma, \) \(2) \) add a new index array \(equal, \) and set \(equal[r] = s, \) \(\text{ meaning that row } r \text{ now equals with row } s, \) \(\text{ and we can get the value of row } r \text{ from row } s \) \(\text{ equally; } \) \(3) \) delete row \( r \) \(\text{ in matrix } M. \) \n
The main idea of compressing matrixes T1 and T2 is: convert the matrix into an offset matrix, which can generate many combinative rows, and then merge them in order to reduce memory usage. As mentioned above, after rearranging DFA level by level, the state number in the same cluster is in a continuous sequence, assuming it as \((a, a + 1, a + 2, \ldots, a + n, n \leq |\Sigma|). \) We regard \(a \) as the base value of the cluster, and the sequence turns into \((0, 1, 2, \ldots, n) \) after extracting the base value \(a. \) All the transitions in each row transfer to the same cluster, so through extracting base value we can convert them into offset matrixes. This step can increase the number of combinative rows, because for row \( r = (a + t_0, a + t_1, a + t_2, \ldots, a + t_i) \) and row \( s = (b + t_0, b + t_1, b + t_2, \ldots, b + t_i), \) the symbols for effective elements are same, they are not combinative row by the definition. After extracting base value of each cluster, row \( r \) turns into \( \vec{r} \) \((\vec{r} = (t_0, t_1, t_2, \ldots, \) \(1) \text{ Supposing compress matrix T3 with tri-array structure [18]}

![Fig. 3. Division of DFA matrix for regular expression “A(2)CD”. Character "*" indicates all the symbols in \(\Sigma\) except A, C and D.](image-url)
In this paper, we use \( SCR \) (Spatial Compression Ratio) to evaluate the compression effect, which is defined as the ratio of \( CSS \) (Compressed Storage Space) and \( UCSS \) (Un-Compressed Storage Space). Obviously, the smaller the value of \( SCR \), the better the compression algorithm.

In order to compute \( SCR \), we introduce some new variables: \( n \) is the number of DFA states, namely the row number of matrix \( T \); \( n1 \) (\( n2 \)) represents the row number of matrix \( R1(R2) \). \( R1 \) and \( R2 \) are offset matrixes, its maximum value is \( n1 \) and \( n2 \), respectively.

Algorithm 1 Pseudo-code for the steps of algorithm to compress matrix \( M \) (\( M \) is \( T1 \) or \( T2 \))

1. for \( i \leftarrow 0, N − 1 \) do
2. \( base[i] \leftarrow \) the min state number of the cluster that contains state \( i \);
3. for \( c \leftarrow 0, C − 1 \) do
4. if \( M[i, c] \neq \mathcal{X}' \) then
5. \( M[i, c] \leftarrow M[i, c] − base[i] \)
6. end if
7. end for
8. end for
9. rowsize \( \leftarrow 0 \)
10. for \( i \leftarrow 0, N − 1 \) do
11. \( found \leftarrow −1, j \leftarrow 0 \)
12. while \( found == −1 \) and \( j < \) rowsize do
13. if \( M[i] \) and \( R[j] \) are combinative rows then
14. \( found \leftarrow j \)
15. for \( c \leftarrow 0, C − 1 \) do
16. if \( M[i, c] \neq \mathcal{X}' \) and \( R[j, c] == \mathcal{X}' \) then
17. \( R[j, c] \leftarrow M[i, c] \)
18. end if
19. end for
20. end if
21. end while
22. if \( found \geq 0 \) then
23. \( equal[i] \leftarrow found \)
24. else
25. \( equal[i] \leftarrow rowsize \)
26. \( R[\text{rowsize}] = M[i] \)
27. \( \text{rowsize} \leftarrow \text{rowsize} + 1 \)
28. end if
29. end for

Algorithm 2 Pseudo-code for getting next state of CSCA for current state \( cur \) and the input symbol \( c \)

1. if \( \text{bitmap}[,][c] == 1 \) then
2. return \( R1(\text{equal1}[,][c] + \text{base1}[,]) \)
3. else if \( (\text{temp} = R3[,][c]) \neq \mathcal{X}' \) then
4. return \( \text{temp} \)
5. else
6. return \( R2(\text{equal2}[,][c] + \text{base2}[,]) \)
7. end if

T2), array \text{base2} of T2, array \text{equal2} of T2; matrix \text{R3} (store the compressed T3).
is less than or equals to the size of character set (|Σ|). If the character set is ASCII (256 characters), we can store each element in one byte (8 bits); base1 (base2) is a int-type array, whose row number is n; array equal1 (equal2) has n rows, its element can be stored in \( \log_2 n \) (\( \log_2 n^2 \)) bits; matrix T3 is a sparse matrix, its ultimate storage space is related to the number of its effective elements, therefore we only statistic the ratio of effective elements in T3, represented by r. According to the above variables, we can conclude that

\[
SCR = \frac{R1 + R2}{T} + \frac{equal1 + equal2}{T} + \frac{base1 + base2}{T} + \frac{bitmap1}{T} + r
\]

\[
= \frac{n1 + n2}{4n} + \frac{\log_2 n1 + \log_2 n2}{256 + 32} + 0.039 + r \quad (1)
\]

Thus we can compute spatial compression ratio of CSCA by obtaining the variables n1, n2, n, r.

V. ORTHOGONAL TO PREVIOUS SCHEMES

One advantage of our work is that it is orthogonal to many previous compression schemes. Because our work focuses on utilizing the transition characteristic inside states and reducing memory usage by extracting the base value of each cluster, while previous schemes almost are based on the transition characteristic among states. Furthermore, we argue that this thought can introduce more transitions redundancy between states.

Definition 3. Let T be a DFA matrix. If state r has the same next state with state s on symbol c, we say that there is transitions redundancy between r and s on symbol c. For \( \forall c \in \alpha, \alpha \subseteq \Sigma \), if r and s have transitions redundancy on c, we say that r and s have transitions redundancy on the subset \( \alpha \).

Theorem 1: Let T be a DFA matrix. If state r transfer to cluster i on subset \( \alpha (\alpha \subseteq \Sigma) \) while state s transfer to cluster j on \( \alpha \), then r and s have no transitions redundancy on the subset \( \alpha \).

Proof: The proof is quite easy. Because each state belongs to only one cluster as we mentioned above, there is no duplicate states in cluster i and j. Thus state r does not have any same next state with state s on any symbol of \( \alpha \). By the definition of transition redundancy, we know the theorem is true.

Theorem 2: Let T be a DFA matrix. If matrix \( \bar{T} \) is transformed from T by extracting the base value for each element in T, then matrix \( \bar{T} \) has at least the same transitions redundancy between states as matrix T.

Proof: Supposing state r transfer to cluster i on subset \( \alpha \), state s transfer to cluster j on subset \( \beta \), r and s have transitions redundancy on \( \gamma = \alpha \cap \beta \) in matrix T. From above theorem we can know that r and s have transitions redundancy on \( \alpha \setminus \beta \) and \( \beta \setminus \alpha \) in matrix T. Because they transfer to the same cluster on \( \gamma \), they have the same base value. Then r and s also have transitions redundancy on \( \gamma = \alpha \cap \beta \) in matrix \( \bar{T} \).

From theorem 2 we can conclude that all transitions redundancy among states are kept after extracting the base value of each cluster. Fig. 5 (a) shows the process of extracting base value for DFA matrix of Fig 1. In each state, we extract the minimum state number in each cluster for the transitions that transfer to the TOP-2 clusters; for the remaining transitions, we extract the minimum state number of TOP-1st cluster. The elements in red denote that they are extracted by the base value of TOP-1st cluster, the remaining ones are extracted by the base value of TOP-2nd cluster.

As showed in Fig. 5 (a), there are more transitions redundancy after extracting base value. For example, all the transitions of state 1 transfer to cluster \{2, 3\}, state 2 transfers to cluster \{4, 5\} on \( \Sigma \), we know that state 1 and 2 have no transitions redundancy. However, after extracting base value they have transitions redundancy on all the symbols of \( \Sigma \).

Fig. 5 (b) shows the D2FA [4], [13] for \( .^{*}A.\{2\}CD \). The main idea is to reduce the memory footprint of states by storing only a limited number of transitions for each state and a default transition to be taken for all input symbols for which a transition is not defined. For example, in Fig. 5 (b) if current state is 8 and the input symbol is A, the default transition from state 8 to state 5 is taken. State 5 also does not know which state to go to upon the symbol A, the default transition from state 5 to state 3 is taken. State 3 know which state to go to, thus the next state of state 8 upon \( A \) is state 6. Fig. 5 (c) shows the offset-D2FA for the same regular expression. The offset-D2FA is constructed from the matrix \( T_{offset} \) in Fig. 5 (a). For current state 8 and input symbol A, the offset-D2FA jumps to state 0 by following two default transitions. We can get the next state 6 by plus the base value of state 8’s TOP-1st cluster (\( \{6, 7\} \)) to 0. In this example, the total number of transitions in D2FA is 22, while in offset-D2FA the number is reduced to 21.

The \( \delta \)FA of \( .^{*}A.\{2\}CD \) is depicted in Fig. 5 (d), which has 37 transitions. The main idea of \( \delta \)FA is to reduce the memory footprint of states by storing only a limited number of transitions for each state and the transition of its father state to be taken for all input symbols for which a transition is not defined. This requires a supplementary structure \( t_{loc} \) to locally store the transitions of current state in the matching process. The offset-\( \delta \)FA is shown in Fig. 5 (e), which has only 22 transitions.

VI. EXPERIMENT RESULTS

In order to make our experiment more reasonable, we do not use our own DFA generator, but choose to use the opensource tool — regex-tool, which is provided by Michela Becchi in [20].

A. Experiment setup

To prove that our compression algorithm has a wider range of applicability, we not only focus on regular expressions used in networking security applications. In detail, we design the following four complex pattern sets, as shown in Table III. The first pattern set is extracted from L7-filter which use regular expressions to classify network traffic; the second set is from the Snort system which contains 3 groups regular expressions;
the third set is from Bro system with a total number of 217 regular expressions for intrusion detection; and the last set contains 6 groups which are classified by Fang Yu [6]. All the DFAs are generated by regex-tool, which can group the regular expressions set automatically if the composite DFA is too large. For example, it divides the L7filter pattern set into 8 groups L7filter-1 ∼L7filter-8. (We only show the result of L7filter-1, L7filter-3 and L7filter-4 in our experiments).

**TABLE III**

<table>
<thead>
<tr>
<th>Group</th>
<th># of regex</th>
<th># of states</th>
<th>length range</th>
</tr>
</thead>
<tbody>
<tr>
<td>L7filter-1</td>
<td>26</td>
<td>3172</td>
<td>11–256</td>
</tr>
<tr>
<td>L7filter-3</td>
<td>6</td>
<td>30135</td>
<td>21–219</td>
</tr>
<tr>
<td>L7filter-4</td>
<td>13</td>
<td>22608</td>
<td>6–202</td>
</tr>
<tr>
<td>snort-24</td>
<td>24</td>
<td>13882</td>
<td>15–98</td>
</tr>
<tr>
<td>snort-31</td>
<td>31</td>
<td>19522</td>
<td>15–263</td>
</tr>
<tr>
<td>snort-34</td>
<td>34</td>
<td>13834</td>
<td>19–115</td>
</tr>
<tr>
<td>bro-217</td>
<td>217</td>
<td>6533</td>
<td>3–211</td>
</tr>
<tr>
<td>type-1</td>
<td>50</td>
<td>248</td>
<td>ABCD</td>
</tr>
<tr>
<td>type-2</td>
<td>10</td>
<td>78337</td>
<td>A, B, CD</td>
</tr>
<tr>
<td>type-3</td>
<td>50</td>
<td>83398</td>
<td>A, B, CD</td>
</tr>
<tr>
<td>type-4</td>
<td>10</td>
<td>5290</td>
<td>A, B, CD</td>
</tr>
<tr>
<td>type-5</td>
<td>50</td>
<td>7828</td>
<td>A, B, CD</td>
</tr>
<tr>
<td>type-6</td>
<td>50</td>
<td>14496</td>
<td>A, B, CD</td>
</tr>
</tbody>
</table>

**B. Effect of cluster-based compression**

We will evaluate cluster-based splitting compression algorithm (CSCA) with spatial compression ratio (SCR) and matching speed in software implementation.

1) **Comparison in SCR:** We apply our compression algorithm presented in Section 4 to the four pattern sets mentioned above. In order to show the compression effectiveness comparably, we compare our algorithm with δFA [5] and Default_Row [18] (a variation of tri-array algorithm, which converts matrix into sparse matrix by extracting the most frequent element of each row, and then compress the sparse matrix).

The experiment result is shown in Table IV, the number in bold is the minimum SCR of the three compression methods: CSCA, δFA and Default_Row. The SCR of original DFA is set to 1.0, because it is not compressed at all.

After comparing the value of n1 and n2 with that of n, we can find that matrixes T1 and T2 have a number of combinative rows after converting them into offset matrixes. The row number of the final compressed matrix does not exceed 1% of the original matrix ($\frac{n_1+n_2}{n}$), in L7filter-3 group the value is even less than 0.1%. r is so small that we can ensure that matrix T3 is a sparse matrix after extracting transfer edges in matrix T1 and T2. The results validates the correctness of the summary of the characteristic of DFAs in Section 3 for multi regular expressions experimentally.

The SCR of CSCA in most groups is less than that of other algorithms. Most importantly, its spatial compression ratio is relatively stable, and the value is around 5%. In the worst case (L7filter-4 group), the SCR is less than 10%. δFA algorithm is superior to others in the groups of Snort pattern set while relatively poor in other groups. For example, in L7filter-3 group it only reduces no more than 10% storage space. The SCR of Default_Row algorithm is least in type-2, type-3, type-4, and type-5 groups. This result shows that, for multi regular expressions of the same type, its DFA matrix can be converted into a sparse matrix by extracting the most frequent element for each row.

2) **Comparison in matching speed:** We compare our compression algorithm with the original DFA, Default_Row, D2FA...
and δFA in matching speed. We have 18 pattern groups, and we produce 5 streams of 100 MB with a probability $p_m$ of experiencing malicious traffic (When generate a byte of the input stream, either a forward transition is taken with probability $p_m$ or a random character is selected with probability $1- p_m$. More detail in [21]). High values of $p_m$ are used to model the likelihood of malicious traffic.

The matching speed varies slightly for different pattern groups, but it doesn’t affect the relativity of different compression algorithms, so we only show the results of l7filter-1 and type-2 pattern groups due to space limitation. Because the former contains relatively more complex regular expressions, and the latter has the largest number of DFA states. The comparison of matching speed comparison is shown in Fig 6.

From Fig 6, we can conclude that CSCA not only achieves good compression effect, but also has no significant reduction in matching speed comparing with original DFA (about 40%~50% loss, equivalent to 2 memory-access per transition when matching). As we mentioned in section 4.3, we need at least 4 memory-access. The reason may be that `bitmap`, `equal1`, `R1`, and `base1` cost too little memory, and they need to be accessed per transition, so our algorithm can access them from Cache memory. On the other hand, D2FA have about 60%~70% loss of matching speed and δFA has more than 99% loss in software implementation. Although Default_Row has the highest matching speed among the compression algorithms, it only get a better compression for the same type of regular expressions.

C. Effect of offset-D2FA and offset-δFA

In this section, we make an experiment of combining our work with previous schemes and evaluate its effect. We extract base value for DFA matrixes of regular expression groups to get the corresponding offset-matrixes, and then we compress DFA matrixes and offset-matrixes with previous compression schemes, such as D2FA and δFA. The D2FA algorithm we use is in [13], which adds a limitation: all the default transitions in D2FA transfer from a state of high depth to a state of low depth. This limitation can ensure that any string of length $N$ will require at most $2^N$ state traversals to be processed.

The result is shown in Table V, the value in which means the ratio of effect transitions after compressing DFAs. For l7filter group, offset-D2FA has nearly five times transitions redundancy than D2FA while offset-δFA has 10 times redundancy than δFA. From the table we can conclude that offset-D2FA and offset-δFA really introduce more transition redundancy between states for all groups. This proves our argument in Section 5 from experiments.

### Table IV

<table>
<thead>
<tr>
<th>Group name</th>
<th>original DFA</th>
<th>our compression algorithm (CSCA)</th>
<th>SCR of Default_Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>l7filter-1</td>
<td>3172.0</td>
<td>0.024621</td>
<td>0.232905</td>
</tr>
<tr>
<td>l7filter-4</td>
<td>30135.0</td>
<td>0.011429</td>
<td>0.356860</td>
</tr>
<tr>
<td>snort-24</td>
<td>13882.0</td>
<td>0.021074</td>
<td>0.108468</td>
</tr>
<tr>
<td>snort-31</td>
<td>19522.0</td>
<td>0.020840</td>
<td>0.061309</td>
</tr>
<tr>
<td>snort-34</td>
<td>13834.0</td>
<td>0.017938</td>
<td>0.060473</td>
</tr>
<tr>
<td>bro-217</td>
<td>6533.0</td>
<td>0.020456</td>
<td>0.224820</td>
</tr>
<tr>
<td>type-1</td>
<td>249.0</td>
<td>0.000016</td>
<td>0.186697</td>
</tr>
<tr>
<td>type-2</td>
<td>78337.0</td>
<td>0.001012</td>
<td>0.030254</td>
</tr>
<tr>
<td>type-3</td>
<td>8338.0</td>
<td>0.002395</td>
<td>0.018575</td>
</tr>
<tr>
<td>type-4</td>
<td>5290.0</td>
<td>0.012469</td>
<td>0.046357</td>
</tr>
<tr>
<td>type-5</td>
<td>7828.0</td>
<td>0.002451</td>
<td>0.019762</td>
</tr>
<tr>
<td>type-6</td>
<td>14496.0</td>
<td>0.002300</td>
<td>0.173284</td>
</tr>
</tbody>
</table>

### Table V

<table>
<thead>
<tr>
<th>Group name</th>
<th>D2FA</th>
<th>offset-D2FA</th>
<th>δFA</th>
<th>offset-δFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>l7filter-1</td>
<td>0.39234</td>
<td>0.02381</td>
<td>0.63496</td>
<td>0.12401</td>
</tr>
<tr>
<td>l7filter-3</td>
<td>0.47871</td>
<td>0.00508</td>
<td>0.96098</td>
<td>0.02499</td>
</tr>
<tr>
<td>l7filter-4</td>
<td>0.03322</td>
<td>0.00706</td>
<td>0.09717</td>
<td>0.04521</td>
</tr>
<tr>
<td>snort-24</td>
<td>0.00893</td>
<td>0.00567</td>
<td>0.03751</td>
<td>0.02508</td>
</tr>
<tr>
<td>snort-31</td>
<td>0.01203</td>
<td>0.00783</td>
<td>0.05358</td>
<td>0.03328</td>
</tr>
<tr>
<td>snort-34</td>
<td>0.00963</td>
<td>0.00440</td>
<td>0.03225</td>
<td>0.02116</td>
</tr>
<tr>
<td>bro-217</td>
<td>0.00657</td>
<td>0.00626</td>
<td>0.06181</td>
<td>0.06118</td>
</tr>
</tbody>
</table>

We measuring their matching speeds with type-2 pattern group, and show the result in Table VI. From the table, we can find that offset-δFA is dozens of times higher than δFA, and offset-D2FA is somewhat higher than D2FA. This shows that our optimizations are effective.

### Table V

<table>
<thead>
<tr>
<th>Group name</th>
<th>D2FA</th>
<th>offset-D2FA</th>
<th>δFA</th>
<th>offset-δFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>l7filter-1</td>
<td>0.39234</td>
<td>0.02381</td>
<td>0.63496</td>
<td>0.12401</td>
</tr>
<tr>
<td>l7filter-3</td>
<td>0.47871</td>
<td>0.00508</td>
<td>0.96098</td>
<td>0.02499</td>
</tr>
<tr>
<td>l7filter-4</td>
<td>0.03322</td>
<td>0.00706</td>
<td>0.09717</td>
<td>0.04521</td>
</tr>
<tr>
<td>snort-24</td>
<td>0.00893</td>
<td>0.00567</td>
<td>0.03751</td>
<td>0.02508</td>
</tr>
<tr>
<td>snort-31</td>
<td>0.01203</td>
<td>0.00783</td>
<td>0.05358</td>
<td>0.03328</td>
</tr>
<tr>
<td>snort-34</td>
<td>0.00963</td>
<td>0.00440</td>
<td>0.03225</td>
<td>0.02116</td>
</tr>
<tr>
<td>bro-217</td>
<td>0.00657</td>
<td>0.00626</td>
<td>0.06181</td>
<td>0.06118</td>
</tr>
</tbody>
</table>

We are measuring their matching speeds with type-2 pattern group, and show the result in Table VI. From the table, we can find that offset-δFA is dozens of times higher than δFA, and offset-D2FA is somewhat higher than D2FA. This shows that our optimizations are effective.

### VIII. Conclusions and Future Work

In this paper, we have introduced a new efficient compression algorithm to reduce the memory usage of DFAs of multi regular expressions from a new perspective. The algorithm is based on the observation that most transitions of each state does not transfer randomly but concentratively to a two clusters, although the number of distinct “next-state” are above 25 averagely. So we split the DFA matrix into three parts,
and compress them respectively with different compression strategies according to their characteristics. The experiment shows that our algorithm can considerably reduce the number of states and transitions, and save memory space requirement by 95% stably with 40% loss of matching speed comparing with original DFA in software implementation. However, this loss can be compensated by hardware support easily. We will finish this thought in our future work.

Furthermore, our work is orthogonal to many previous schemes, such as D2FA, δFA. We prove that extracting base value of each cluster can introduce more transitions redundancy among states, which can improve the compression effect of present schemes. Our experiments show that offset-δFA and offset-D2FA work better than δFA and D2FA both in memory consumption and matching speed.

REFERENCES


