An Efficient Regular Expressions Compression Algorithm From A New Perspective

Tingwen Liu[1,2], Yifu Yang[1], Yanbing Liu[1,3] Yong Sun[1,3], Li Guo[1,3]
[1]Institute of Computing Technology, Chinese Academy of Sciences, 100190
[2]Graduate University of Chinese Academy of Sciences, Beijing, 100039
[3]National Engineering Laboratory for Information Security Technologies, China
E-mail:liutingwen@software.ict.ac.cn

***Abstract*—Deep packet inspection plays a increasingly important role in network security devices and applications, which use more regular expressions to depict patterns. DFA engine is usually used as a classical representation for regular expressions to perform pattern matching, because it only need $O(1)$ time to process one input character. However, DFAs of regular expression sets require large amount of memory, which limits the practical application of regular expressions in high-speed networks. Some compression algorithms have been proposed to address this issue in recent literatures.**

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 algorithms that observe transition 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 show 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 as well as Network Managers. Traditionally, string-based signatures have taken root as a predominant representation of patterns. Currently, regular expressions are replacing exact strings to depict patterns in most popular software tools, such as Snort[1], Bro[2], L7-filter[3], and some devices by many companies. The widespread use is due to the expressive power, simplicity and flexibility of regular expressions in expressing patterns.

Deterministic Finite Automata (DFA) and Nondeterministic Finite Automata (NFA) are two classical equivalent representation of regular expressions. In the pattern matching context, DFA matching technique has two major advantages when compared to NFA matching technique: it triggers only one state transition (one corresponding memory access) for each input symbol processed, and it is possible to compile multiple regular expressions into a composite DFA which can inspect the input in a single pass. Therefore, DFA is the preferred representation to perform deep packet inspection in high-speed network environments.

Unfortunately, the use of DFAs demand for a large memory space to store state transition tables for current sets of regular expressions. For example, the composite DFA of L7-filter regular expressions set require more than 16 GB memory space. Obviously, a compression algorithm is needed to reduce the memory requirement in order to achieve wire-speed DFA matching in high-speed network environments. Many compression algorithms have been proposed in order to achieve the goal. However, most of them either require hardware support or depend on specialized architectures, and at the cost of serious degradation in 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 state pairs have many identical outgoing transitions for the same character. Obviously 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 between 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 different next-states, its transitions concentratively transfer to a two clusters concentratively. Based on the observation, we adjust transitions in every cluster separately by its extracting base value for each state, which can introduce more identical 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 ratios and matching speed.

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 for some subsets.

This work is supported by National Basic Research Program 973 of China (2007CB311100) and National Natural Science Foundation of China (61003295).

978-1-4244-9921-2/11/$26.00 ©2011 IEEE 2129

- 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 a careful analysis of our work, and make a systematic experiments to compare our work with previous compression algorithms.

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 security applications, people pay more attention to its memory consumption and matching performance in a real environment. 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 for some special regular expressions. For instance, it can rewrite $AUTH\backslash s[^\wedge\backslash n]\{100\}$ (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 memory requirement of DFA [4], which is an extension of classical DFAs. If two states have identical outgoing transitions, one state 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 an input character may 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 identical 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 $O(|\Sigma|)$ (Σ is the character set) time to update transitions of current state as a new state is reached, 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 classical DFAs with finite scratch memory and instructions to avoid some duplication of states. This method works well to have matching speeds approaching DFAs yet memory requirements similar to NFAs. However, the time required to update the scratch memory after each transition may be significant.

Recently, alphabet compression technique is employed to reduce memory requirement [12], [13], [14]. This 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 can partition the set of DFA states and create a compression table for each subset in a way that yields further reduction in memory usage[15]. In fact, all these solutions exploit transition redundancy among all states or subset of states.

Multi-stride DFAs is proposed in [12] as a way to increase processing throughout 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 growth. Nan Hua *et al.* introduce variable-stride multi-pattern matching for deep packet inspection [16], which is inspired by the winnowing scheme originally developed for document fingerprinting. However, this scheme can hand only exact strings.

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) Transitions to Magic States: All transitions in DFAs can be classified into four categories according to their different functions[17]: basic transitions (successfully accept pattern set from the initial 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 the initial state) and failure transitions (transfer current state to the initial state). Typically Aho-Corasick(AC) and its derivative algorithms build a DFA from string patterns 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 lead back to the initial state or its near neighbors [13], we call them magic states in this paper. Table I shows, for different regular expression sets (see section VI for more details about the 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. However, the proportion is not that high for some regular expressions sets of 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

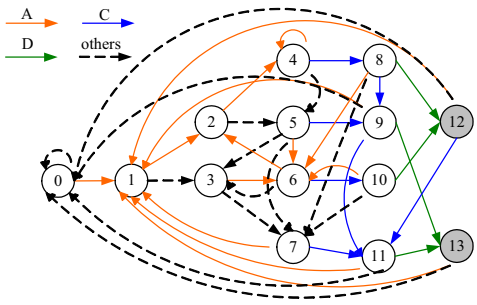

Fig. 1. The DFA of regular expression $.^*A.\{2\}CD$. Accepting states are in grey. Non-labeled symbols (others) transfer along the transitions in dashed, which are used for readability.

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, $\text{PT}_{matching}$). We can use this observation to speedup the matching performance of regular expressions.

2) Characteristic Among States: State pairs, sometimes all states, usually have the identical outgoing transition for the same character in DFAs, this observation was first illustrated in [18]. Many literatures [4], [5], [15], [19] 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 in DFAs, its outing transitions transfer to many distinct "next-states". Unfortunately, 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).

We observe that the average number of distinct clusters is usually less than 5 (Column 3 in Table II), which is much smaller than that of distinct "next-states". Furthermore, the transitions of each state concentratively 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% transitions from TableII.

TABLE II
TRANSITION CHARACTERISTIC INSIDE STATES

pattern set	average distinct "next-states"	average distinct clusters	% transitions in Top-2 clusters
snort-24	14.49	5.06	97.89
snort-31	12.24	4.75	97.92
snort-34	13.11	4.69	98.21
bro-217	54.29	4.23	97.95
type-1	47.79	1.93	99.96

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 shows the DFA of our 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 character from small to large. Fig 2 shows the trie-tree by level traversing the DFA of $.^*A.\{2\}CD$. Before introducing our work, we give the definition of cluster first.

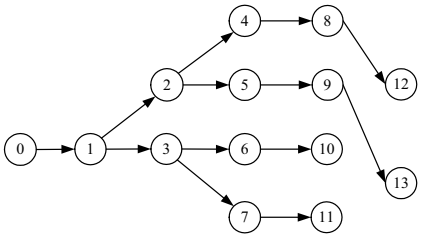

Fig. 2. The trie-tree for DFA in Fig 1

Definition 1. In the trie-tree, if state r has a transition to state s, we call r 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 son states of a certain state.

The initial state is not any state's son, so it doesn't belong to any cluster. However, we can think that there is a separate cluster, which has the initial state only. So each state in a trie-tree belongs to only one cluster. The proof is trivial. Each state belongs to the cluster that is composed of all the sons of its father. Furthermore, it cannot occur in more than one cluster simultaneously, otherwise it has at least two distinct father states, which is impossible in a trie-tree. Thus, cluster is a mutually exclusive partition of nodes of trie-tree in essence. According to the definition of cluster, the initial 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 (Σ is the input character set, it contains 2^8 characters for the extended ASCII code), and each transition transfer to a determinate next-state, thus it transfers to a determinate cluster from above analysis. Obviously 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 matrixes T1, T2, T3 as shown in Fig.3. For each state, the maximum transitions that transfer to the same cluster were put into matrix T1. Similarly, the second maximum transitions into matrix T2, and the remaining transitions into matrix T3. For example, all the outgoing transitions from state 1 transfer to state 2 or 3, both of which belong to cluster $\{2, 3\}$, thus all transitions are put into T1. There is no transitions in matrix T2 and T3 for state 1, we use 'x' to represent non-existent element. We can find that there are very few transitions in matrix T3 (the proportion is $\frac{6}{14*256} \approx 0.17\%$ for the example), most transitions transfer to the TOP-2 clusters (in matrix T1 or T2), which is the transition characteristic inside state as we mentioned above.

TABLE I
PERCENTAGE OF TRANSITIONS POINTS TO MAGIC STATES IN DFAS AND WHEN MATCHING

pattern set	# of regex	# of states	PT$_{dfa}$(%)	# of text bytes	PT$_{matching}$ (%)
snort-24	24	13882	3.84	1048576	99.92
snort-31	31	19522	7.94	1048576	99.97
snort-34	34	13834	4.95	1048576	99.18
bro-217	217	6533	48.55	1048576	99.39
type-1($.^*abcd.^*$)	50	248	99.62	1048576	99.89

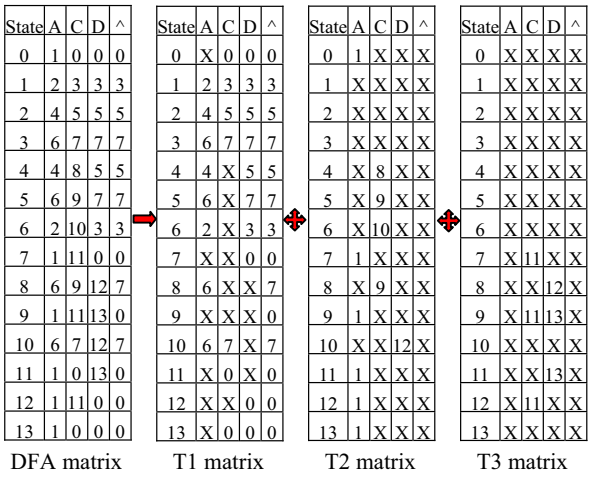

Fig. 3. Division of DFA matrix for regular expression $.^*A.\{2\}CD$. Character ^ indicates all the symbols in Σ except A, C and D

IV. CLUSTER-BASED SPLITTING COMPRESSION ALGORITHM

We introduce a new method, named Cluster-based Splitting Compression Algorithm (CSCA), that uses the transition characteristic inside state to compress DFA. It first rearranges DFA level by level from the initial state. Then it divides *T* (the transition matrix of DFA) into three matrixes. Finally it compresses them respectively.

A. Compression Process

1) Rearranging and Dividing: We need to rearrange DFA level by level according with the character sequence in Σ. After this step, the trie-tree of DFA has the following properties: for

$$\forall\ state\ r\ and\ s,\ \forall\ c \in \Sigma,\ T[r,\ c] \leq T[s,\ c],\ if\ r \leq s$$

$$\forall\ state\ r,\ \forall\ c\ ,\ i \in \Sigma,\ T[r,\ c] \leq T[r,\ i],\ if\ c \leq i$$

The benefit is that all son states of the same state is in a continuous sequence after rearranging. Thus the state number in the same cluster is continuous, and the minus of the maximum and minimum state number is less than $|\Sigma|$. This step can be combined with subset construction (the NFA-to-DFA transformation) function, fortunately many construction methods really keep above properties.

We divide the DFA matrix as describing 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 outgoing transitions from 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 T1 and T2, and use the other 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 [1]. Before describing the compression algorithm of matrix T1 and T2, we introduce a new definition of *combinative row*.

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

If rows r and s are combinative row, we process them according to the rules as follows: (1) for character c in Σ, if $M[s,\ c]$ = 'X', reset $M[s\,c] = M[r,\ c]$; if $M[s,\ c] = M[r,\ c]$ or $M[r,\ c]$ = 'X', keep $M[s,\ c]$ unchanged; (2) add a new index array $equal$, and set $equal[r] = s$, meaning that row r now equals with row s, and we can get the value of row r from row s equally; (3) delete row r in matrix M.

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, ... , a + n, $n \leq |\Sigma|$). We regard a as the base value of the cluster, and the sequence turns into (0, 1, 2, ... , n) after extracting the base value a. All the transitions of the same state in T1 and T2 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 = ($a+t_0$, $a+t_1$, $a+t_2$, ... , $a+t_i$) and row s = ($b+t_0$, $b+t_1$, $b+t_2$, ... , $b+t_i$), the characters for effective elements are same, they are not combinative row by the definition. After extracting base value of each cluster, row r turns into $\bar{r}$ ($\bar{r}$ = (t_0, t_1, t_2, ... , t_i)), s turns into $\bar{s}$ ($\bar{s}$ = (t_0, t_1, t_2, ... , t_i)). Obviously $\bar{r}$ and $\bar{s}$ are combinative row.

Fig. 4 shows the compressing process for matrix T1. The first matrix is T1, the second is the matrix after extracting

[1] supposing compress matrix T3 with tri-array structure [18]

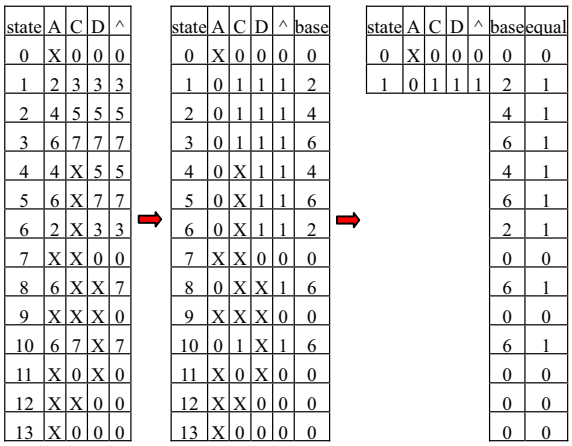

Fig. 4. Compressing process of matrix T1

base value for each row in T1, and the third is the matrix after merging combinative rows.

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

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

Algorithm 1 shows the pseudo-code for compressing T1 and T2. Line 1∼8 describe the process of extracting base value, which is the minimum state number of a cluster in our algorithm. Line 9∼30 are the process of merging combinative rows. In the process, we use matrix R to store the merged rows. For row i in matrix M, if there is a row j in matrix R that is a combinative row of row i, we make appropriate amendments to row j (line 17 and 18) and set row i equals row j (line 24). If the combinative row doesn't exist, we add a new row in matrix R and set row i equals the new row (line 26∼29).

B. Lookup

The lookup function is to get the next state for current state cur and the input character c. Because CSCA splits DFA matrix into three parts and compress them separately, the function need to decide which matrix the next state is in first. We can quickly get the information by adding two bitmaps to distinguish three parts. Fortunately, matrix T3 is compressed by sparse matrix compression algorithms, which can determine whether the element in T3 is effective or not by itself, thus we can save a bitmap by accessing T3 before T2, as shown in Algorithm 2.

Algorithm 2 Pseudo-code for getting next state of CSCA for current state cur and the input character c

```
1: if bitmap[cur][c] == 1 then
2:    return R1[equal1[cur]][c] + base1[cur]
3: else if (temp = R3[cur][c]) ≠'X' then
4:    return temp
5: else
6:    return R2[equal2[cur]][c] + base2[cur]
7: end if
```

C. Complexity Analysis

From Algorithm 2 we can know that when getting the next state the algorithm needs 3 memory-access operations (next-state in matrix T1), 6 memory-access operations (next-state in matrix T2), and 4 memory-access operations (next-state in matrix T3) [2].

Now we make an analysis of space complexity for CSCA. The memory consumption is composed of three parts: matrix $R1$ (store the compressed T1), array $base1$ of T1, array $equal1$ of T1, matrix $bitmap$ of T1; matrix $R2$ (store the compressed T2), array $base2$ of T2, array $equal2$ of T2; matrix $R3$ (store the compressed T3).

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

[2]the $base$ value and $equal$ value for each state are stored in a structure so as to read them in a single memory-access in our implementation.

of matrix T; $n1$ ($n2$) represents the row number of matrix $R1(R2)$. $R1$ and $R2$ are offset matrixes, its maximum value is less than or equals to the size of character set ($|\Sigma|$). 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 $\log_2 n1$ ($\log_2 n2$) 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

$$\begin{aligned} SCR &= \frac{R1+R2}{T} + \frac{equal1+equal2}{T} + \frac{base1+base2}{T} \\ &\quad + \frac{bitmap1}{T} + r \\ &= \frac{n1+n2}{4n} + \frac{\log_2 n1 + \log_2 n2}{256*32} + 0.039 + r \end{aligned} \quad (1)$$

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 α.

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 α, then r and s have no transitions redundancy on the subset α.

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 α. 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 α, state s transfer to cluster i on subset β, r and s have transitions redundancy on γ ($\gamma = \alpha \cap \beta$) in matrix T. From Theorem 1 we can know that r and s have transitions redundancy on $\alpha \backslash \beta$ and $\beta \backslash \alpha$ in matrix T. Because they transfer to the same cluster on γ, they have the same base value. Then r and s also have transitions redundancy on γ ($\gamma = \alpha \cap \beta$) in matrix $\bar{T}$. ■

From theorem 2 we can conclude that all transitions redundancy among states are kept after extracting the base value. Fig.5 (a) shows the process of extracting base value for DFA matrix in 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 outgoing transitions of state 1 transfer to cluster $\{2, 3\}$, state 2 transfers to cluster $\{4, 5\}$ on Σ, we know that state 1 and 2 have no transitions redundancy. However, after extracting base value they have transitions redundancy on the whole alphabet Σ.

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 on 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 plussing the base value of state 8's TOP-1st cluster ($\{6, 7\}$) with 0. In this example, the total number of transitions in D2FA is 22, while in offset-D2FA the number is reduced to 21.

The δFA of $.^*A.\{2\}CD$ is depicted in Fig.5 (d), which has 37 transitions. The main idea of δFA 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-δFA 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 constructor, 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

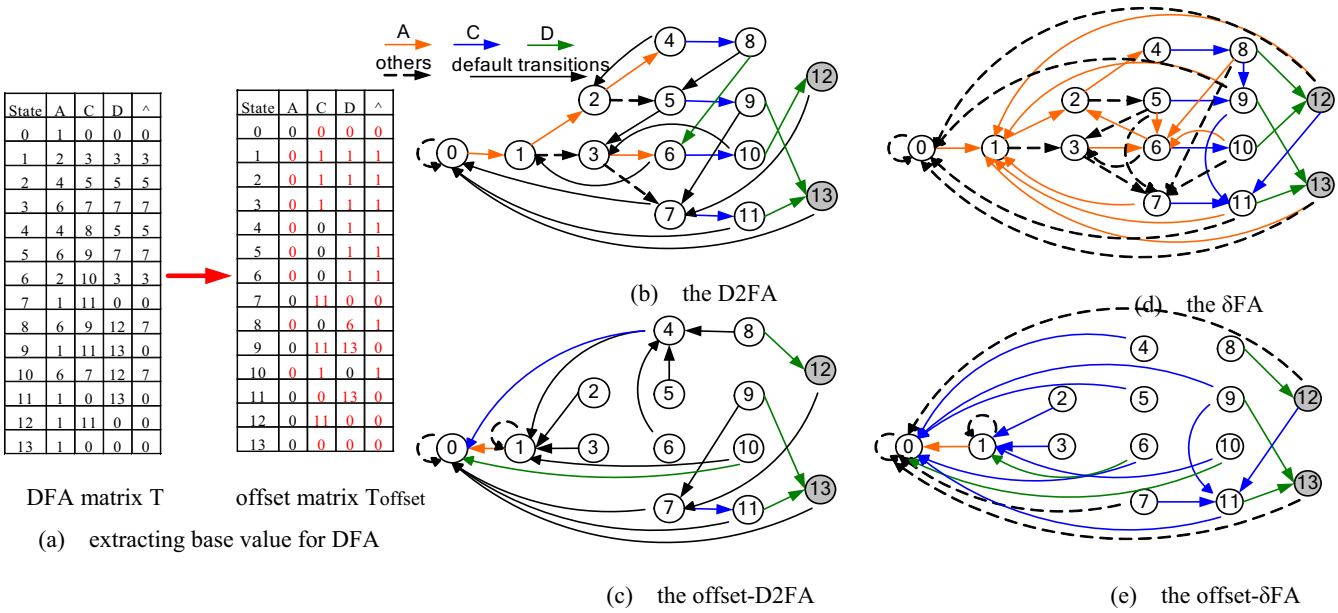

Fig. 5. Compressing process for T1 matrix

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 L7-filter 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
PRIMARY INFORMATION OF PATTERN SETS

Group	# of regex	# of states	length range
l7filter-1	26	3172	11–256
l7filter-3	6	30135	21–219
l7filter-4	13	22608	6–202
snort-24	24	13882	15–98
snort-31	31	19522	15–263
snort-34	34	13834	19–115
bro-217	217	6533	3–211
type-1	50	248	ABCD
type-2	10	78337	AB.*CD
type-3	50	8338	^AB{0,j}CD
type-4	10	5290	^A+[A-Z]{j}B
type-5	50	7828	^AB{j}CD
type-6	50	14496	A[A-Z]{j}B

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 ($\frac{n1+n2}{n}$), 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

TABLE IV
COMPARISON IN TERMS OF SPATIAL COMPRESSION RATIO

Group name	original DFA		our compression algorithm (CSCA)				SCR of δ FA	SCR of Default_Row
	n	SCR	$n1$	$n2$	r	SCR		
l7filter-1	3172	1.0	52	22	0.024621	**0.070756**	0.634964	0.232905
l7filter-3	30135	1.0	3	21	0.011429	**0.051420**	0.960985	0.356860
l7filter-4	22608	1.0	7	44	0.054823	**0.095459**	0.097177	0.381078
snort-24	13882	1.0	13	34	0.021074	0.062055	**0.037515**	0.108468
snort-31	19522	1.0	7	34	0.020840	0.061391	**0.053581**	0.061309
snort-34	13834	1.0	6	17	0.017938	0.058231	**0.032259**	0.060473
bro-217	6533	1.0	1	14	0.020456	**0.060557**	0.061814	0.224820
type-1	249	1.0	1	2	0.000016	**0.042212**	0.111281	0.186697
type-2	78337	1.0	512	2	0.000102	0.042026	0.099659	**0.030254**
type-3	8338	1.0	43	5	0.002395	0.043842	0.948123	**0.018575**
type-4	5290	1.0	236	4	0.012469	0.064080	0.990808	**0.046357**
type-5	7828	1.0	1	22	0.002451	0.041732	0.947048	**0.019762**
type-6	14496	1.0	9	22	0.002300	**0.042061**	0.973929	0.173284

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%$\sim$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%$\sim$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 $2N$ 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 V
EFFECT OF EXTRACTING BASE VALUE (%)

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

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

VII. 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

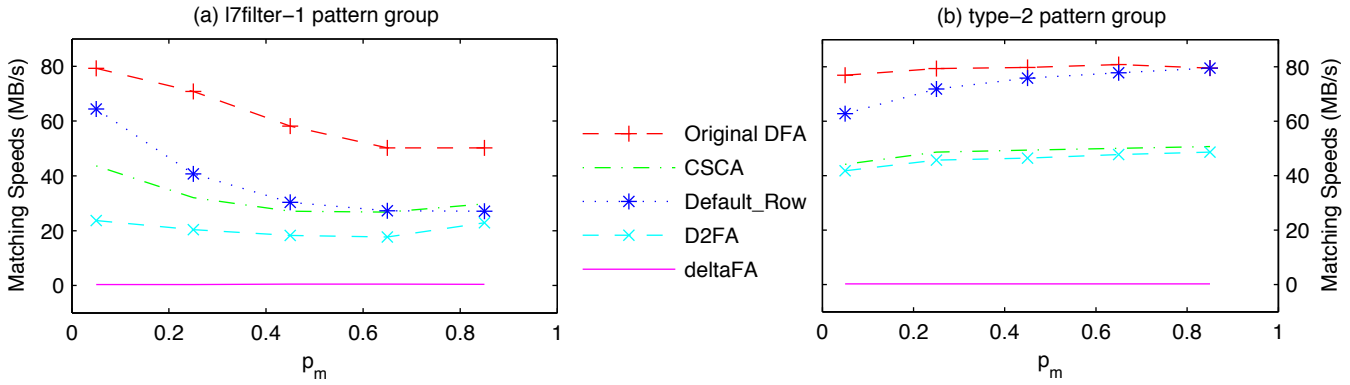

Fig. 6. Comparison in terms of matching speed (compressing sparse matrix with tri-array)

TABLE VI
COMPARISON IN MATCHING SPEED (MB/S)

compression algorithm	p_m				
	0.05	0.25	0.45	0.65	0.85
δFA	0.28	0.28	0.28	0.27	0.27
offset-δFA	19.8	33.5	36.5	42	53
D2FA	41.8	45.7	46.4	47.7	48.6
offset-D2FA	43.3	58	57.6	56.8	58.1

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 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

[1] Roesch and Martin. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, pages 229–238, 1999.

[2] Paxson Vern. Bro: a system for detecting network intruders in real-time. In *Proceedings of the 7th conference on USENIX Security Symposium*, pages 3–3, 1998.

[3] J. Levandoski, E. Sommer, and M. Strait. Application layer packet classifier for linux.

[4] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the 2006 ACM SIGCOMM conference*, pages 339–350, 2006.

[5] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved DFA for fast regular expression matching. In *Proceedings of the 2008 ACM SIGCOMM conference*, 38(5):29–40, 2008.

[6] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE ANCS conference*, pages 93–102, 2006.

[7] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[8] Gene Myers. A four russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):432–448, 1992.

[9] Gonzalo Navarro and Mathieu Raffinot. Compact DFA representation for fast regular expression search. In *Proceedings of the 5th International Workshop on Algorithm Engineering*, pages 1–12, 2001.

[10] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proceedings of the ACM SIGCOMM 2008 conference*, pages 207–218, 2008.

[11] Randy Smith, Cristian Estan, and Somesh Jha. Xfa: Faster signature matching with extended automata. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 187–201, 2008.

[12] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Comput. Archit. News*, 34(2):191–202, 2006.

[13] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 145–154, 2007.

[14] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 50–59, 2008.

[15] Shijin Kong, Randy Smith, and Cristian Estan. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, pages 1–10, 2008.

[16] Nan Hua, Haoyu Song, and T.V. Lakshman. Variable-stride multi-pattern matching for scalable deep packet inspection. In *INFOCOM 2009, IEEE*, pages 415–423.

[17] Tian Song, Wei Zhang, Dongsheng Wang, and Yibo Xue. A memory efficient multiple pattern matching architecture for network security. In *INFOCOM 2008. IEEE*, pages 166–170, 2008.

[18] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[19] Nen-Fu Huang, Yen-Ming Chu, Chi-Hung Tsai, Chen-Ying Hsieh, and Yih-Jou Tzang. A novel algorithm and architecture for high speed pattern matching in resource-limited silicon solution. In *ICC 2007. IEEE*, pages 1286–1291, 2007.

[20] Michela Becchi. Regular expression processor. http://regex.wustl.edu/.

[21] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *Proc. IEEE Int. Symp. Workload Characterization IISWC 2008*, pages 79–89, 2008.