PAPER Special Section on New Technologies and their Applications of the Internet

Fast and Memory-Efficient Regular Expression Matching Using Transition Sharing

Shuzhuang ZHANG[†a)], *Nonmember*, **Hao LUO**[††], *Member*, **Binxing FANG**[†], *and* **Xiaochun YUN**[††], *Nonmembers*

SUMMARY Scanning packet payload at a high speed has become a crucial task in modern network management due to its wide variety applications on network security and application-specific services. Traditionally, Deterministic finite automatons (DFAs) are used to perform this operation in linear time. However, the memory requirements of DFAs are prohibitively high for patterns used in practical packet scanning, especially when many patterns are compiled into a single DFA. Existing solutions for memory blow-up are making a trade-off between memory requirement and memory access of processing per input character. In this paper we proposed a novel method to drastically reduce the memory requirements of DFAs while still maintain the high matching speed and provide worst-case guarantees. We removed the duplicate transitions between states by dividing all the DFA states into a number of groups and making each group of states share a merged transition table. We also proposed an efficient algorithm for transition sharing between states. The high efficiency in time and space made our approach adapted to frequently updated DFAs. We performed several experiments on real world rule sets. Overall, for all rule sets and approach evaluated, our approach offers the best memory versus run-time trade-offs.

key words: *regular expression, memory reduction, deep packet inspection, transition sharing*

1. Introduction

Deep packet inspection matches the packet content against a group of given patterns (usually called rule sets) to identify specific attacks, viruses, protocol definitions, etc. Traditionally, the rule sets and patterns used in matching are explicit strings. However, regular expressions replace the strings and become the pattern matching language due to their powerful expressiveness and flexibility. For example, all the patterns are the regular expressions in L7 (Linux layer-7 protocol classifier) system [1]. Snort [2] and Bro [3] intrusion detection systems also migrate their content inspection rule sets to regular expressions.

The traditional way to perform regular expressions matching is to use deterministic finite automatons (DFAs) [4], [5]. However, memory requirements are high in practical deep packet inspection. For practical networking applications, since we don't have any prior knowledge of whether/where a matching substring may appear, most applications compile the rules together into a single one-pass search mode DFA [6] to achieve O (1) processing complexity. This means ".*" is pre-pended to each pattern without a "^", thus each state has transitions for all input characters and these patterns can be matched any where in the input. Since Network traffic is composed of ASCII alphabet, each state has 256 transitions. Further more, interactions among different rules may make the number of states increase sharply or even exponentially increased in some cases. So the real world rule sets may result in "huge" DFAs. For example, when 40 patterns of L7 were compiled into a composite DFA, the total number of states grows over 136,786. Huge memory requirements make traditional DFAs infeasible for the real word rule sets.

In order to apply DFAs in practical networking applications, we must reduce their memory requirements. However the memory reduction procedure also should be fast. A new attack or worm may affect the global internet in few hours or less time, so some practical systems need to update their rule sets in time to adapt those emergency events. For example, snort rule sets has 203 new rules added and 83 rules modified in 15 days (08/26/2008–09/09/2008). Regular expressions can not be added into a DFA incrementally, the memory reduction procedure will be called whenever rule sets are updated. So the reduction procedure should be fast to satisfy frequently updated rule sets.

In this paper we proposed a novel method to reduce the memory requirement of DFAs, especially those constructed from the real world rule sets. In our scheme, states were selected and formed into a number of groups. For each group, we merged all the transition tables and made states share the merged transition table. Because the duplicate transitions are removed during the merging procedure, the memory requirement of DFA was reduced drastically. In our approach, the states with same destination transitions share the same transition table. After sharing the transition tables, each state had a pointer that pointed to a true transition table. Compared to D^2FA, our method need travel only one state to get a valid transition destination during the matching. Compared to state merging, it avoids introducing the additional bits to retrieve an original state from a merged state. Another advantage of our approach is that it was efficient in time and space complexity, which made it more suitable and feasible for the frequently updated DFAs used in practical network applications.

The remainder of the paper was organized as follows: in Sect. 2, we discussed the related work. In Sect. 3, we proposed transitions sharing scheme by use of an example, and

Manuscript received December 26, 2008.
Manuscript revised May 21, 2009.

[†]The authors are with Research Centre of Computer Network and Information Security Technology, Harbin Institute of Technology, Harbin 150001, China.
[††]The authors are with Institute of Computing Technology, Chinese Academy of Science, Beijing 100190, China.
a) E-mail: zhangshuzhuang@pact518.hit.edu.cn

DOI: 10.1587/transinf.E92.D.1953

formalized our transitions sharing algorithm. We presented the experimental results in Sect. 4, and concluded the paper in Sect. 5.

2. Related Work

Since the memory requirement of a DFA is proportional to its number of states and transitions between these states, recent research on reducing the memory requirement of DFAs can be divided into two categories.

The first category aims at avoiding the number of states blow-up by dividing rule sets into multiple groups instead of compiling them into one group. In [6] Fang Yu et al. systematically analyze the regular expressions which are commonly used in networking applications. They analyze individual regular expressions that will lead DFAs with quadratic size or exponential size, and show that traditional methods are prohibitively high for patterns used in packet scanning applications. They also propose a rewrite technique which can guarantee left-first matching in one-pass search mode. Further more they develop a scheme which can strategically compile a set of regular expressions into several engines to improve the matching speed without much increasing in memory usage. However, this scheme requires big memory bandwidth. For a given memory bandwidth, if we divide the rule sets into n groups, the processing time will be n times of that of a single DFA.

The second category leverages the observation that each state has a total of 256 transitions and most of memory of a DFA is occupied by transition tables. Since there are many transitions with same destination in the transition tables, exceed 90% substantial space can be saved if this redundancy can be avoid. Using bitmaps is an efficient method to reduce memory requirements. It has been used in the packet classification [11] and Aho-corasick state machines [12]. But the basic bitmap-based data structures do not take advantage of duplicate entries in the transition table. Michela Becchi et al. extend the bitmap structure [10]. A pointer indirection table is introduced between the bitmap and transition table, thus the transition table contains only distinct next state entries. However, both of the two above data structures can only remove the redundancy transitions in a single state, there are still a lot of duplicate transitions between states.

In [7], Kumar et al. propose a new representation for regular expressions, called Delayed Input DFA (D^2FA). If two states S and T make transitions to the same set of states for some set of input characters ({C}), D^2FA eliminate these transitions from one state, suppose S, and introduces a default transition from S to T. S now only maintains unique next states for those transitions not common to T. Upon traveling S, if the input is in {C}, we follow the default transition to T and get the real destination state. D^2FA requires two states have same destinations for the same input characters. It compress memory at the expense of latency due to introducing default transition, which means more than one state may be traversed for an input character.

Two methods are proposed to improve D^2FA's matching efficiency. In [8] Kumar et al. proposed a method that use of content labels as state identifiers, called CD^2FA. CD^2FA allows one memory access per state traversal (acts as an uncompressed original DFA). The content label's fields consists of a state discriminator, the list of characters for which a labeled transition is defined and an identifier for the default transition state. The size of a content label depends on the number of labeled transitions defined for the corresponding state. This scheme might be effective only for those states which are highly compressed. Further more, it is bound to a precise state encoding, not a general and broadly applicable. The second improvement is proposed in [9]. They first define the depth of states, and then add the constraint that the default transitions can be oriented only in the direction of deceasing depth. It can be proved that this constraint makes any string of length N require at most 2N states traversals to be processed. They also describe an alphabet reduction scheme for DFA-based structures to reduce the data structure size.

It is worth noting that D^2FA and its improvements all need new data structure to reduce memory requirements of DFAs, such as bitmap based structure or linked list of pair (character, next_state). That means traveling each state may access memory more than one times.

Michela Becchi et al. [10] propose state merging method that can both reduce the number of states and transitions. This method merges several even "non-equivalent" states by introducing labels. Labels identify which portion of the merged state should be accessed during DFA traversal. This method transfers information from the states to the transitions of the DFA. With the extended data structure, the number of new states and transitions is dramatic reduced. However, it must introduce additional bits to retrieve the original state from a merged state.

3. Transitions Sharing in DFAs

3.1 An Example

In order to illustrate our method clearly, we used a simple example shown in Fig. 1. The data structure used to represent this DFA is shown in Fig. 2. We also showed the pointer indirection table and unique transition table of state 3 in Fig. 2. The example and data structure are all same as that of in [10]. In order to avoid cluster, all our figures exclude the transitions leading to the starting state (state 0) and the bitmap of each state.

Now let us introduce our method using this example. Consider two states that have same transitions, if we merge their transition tables and make the two states share the merged transition table, the duplicate transitions of the two states can be removed. We call this operation *transition sharing*. In our scheme, we recursively make states of the original DFA share their transitions with others.

First step of making two states or two group of states share their transitions is to merge their transition tables. We

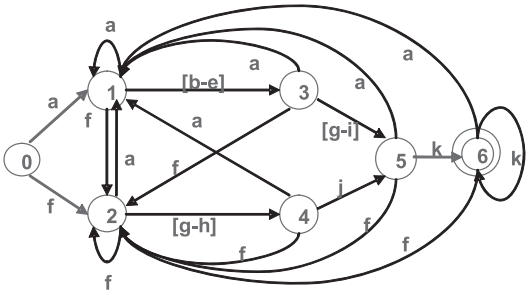
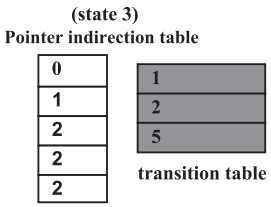


Fig. 1 DFA for expression $((a[b-e][g-i]|f[g-h]j)k+)$. All transitions not have shown lead to state 0.




```
struct DFA_state {
RegExList *accepted_regex;
unsigned char *ptr_indirection;
DFA_state *transition_table[num_unique_transitions];
DFA_state * failure;
}
```

Fig. 2 The data structure representing a state of DFA, the higher part shows the pointer indirection table and transition table of state 3.

removed all the duplicate transitions when merging tables, so each transition in the merged table was unique. The second step is to update the pointer indirection tables of all the states that share the merged transition table. Because all the transitions in the merged table come from different tables, some of them may not have the same index as in the original transition table. We must update the pointer direction tables to make sure the modified DFAs are equal to the original ones. To a pointer indirection table, supposed that the original value of the ith item is m, and this value has been changed to n after updating procedure, it should meet that the nth transition of the merged transition table is the same as the mth item of original transition table. If the pointer indirection table's item width is n bit, it can index 2^n transition items, so if the number of items in a merged table is less than 2^n, we can repeat fetch other states to share their table until the merged table is full. This avoids wasting expression capacity of the indirection table.

Figure 3 shows a group of states prior to and after the merging of their transition tables and the sharing of the merged table. In the merged transition table, there are five unique jump destinations. The first three items come form state 3 and state 4, while the last two items come from state 1 and state 2. As we can see, the first three items have the same index as in the original transition table, and the pointer directions tables of state 3 and state 4 needn't to be update. The indexes of last two items are different with their original value, so the item 2 to item 5 of pointer indirection table of state 1 should be updated to 3 and the last two items of state 2's pointer indirection table should be updated to 4.

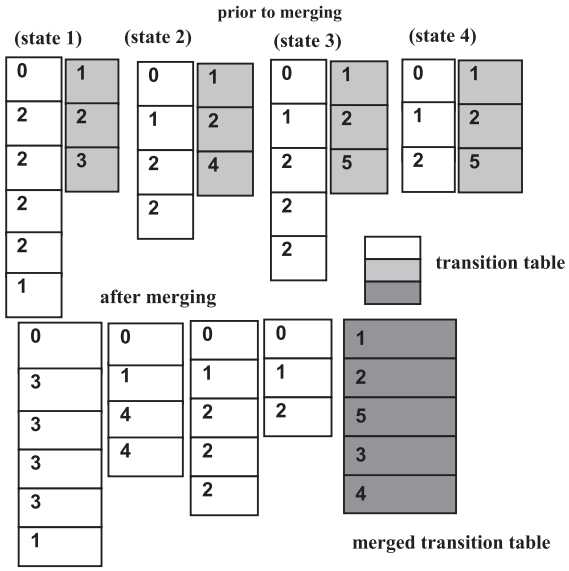


Fig. 3 prior to and after the merging of four transition tables.

Compare to the memory compression approaches described in Sect. 2, this method have three advantages: first, it can eliminate all the redundancy transitions within a group. If a group of states share a merged transition table, all the duplicate transitions can be removed. Second, we needn't to introduce additional bits such as label to index the original state. Third, each state's transition table pointer points to a "true" table, so we just need travel one state to get the jump destination. In additional, sharing the transition tables also creates more common destinations for other states. For example, if states A, B have same destination transitions and states B, C has same destination transitions too. After A and B sharing the same transition table, C will be able to share this table too.

3.2 Transitions Sharing Algorithm

Two or more transition tables can be merged and shared by a group of states. After transitions sharing, each state's transition table pointer points to a merged transition table which consists of the union of all individual transition tables in the same group. We use the terms "original tables" and "merged tables" to refer to the transition tables prior to merging and after merging respectively. Note that, after merging, some original tables may remain as in the original DFA.

We now describe our transitions sharing algorithm. The goal of this algorithm is to divide all the states into groups and merge each group's transition tables thereby to remove the duplicate transitions. For transition table s and t, $metric(s,t)$ is a measure of memory savings when s and t are merged. In our implementation, the $metric(s,t)$ is equivalent to the number of transitions that have the same destination. Merging operation on two transition tables can cause the metric to change to other transition tables, especially those have the same destinations with both of s and t.

The following terminology is useful in understanding

the pseudo code of the algorithm. D is the DFA being processed, assumed to be a global variable. L is a list which contains all the transition tables. *merge_table*(s, t) is an operation that merges the two given tables and updates the pointer indirection tables. *Insert*(L, t) and *remove*(L, s) are the standard operations on a list. *deletemax*(L) is an operation that gets and deletes the table which has the most unique transitions from a list. The pseudo code of this algorithm is shown in the Fig. 4.

Algorithm VI.1 shows the core procedure of our method. The first level loop terminates when there is no table in L. The second level loop terminates when there is not exist a table which can be merged with the given one. Algorithm VI.2 first constructs the list which contains all the tables of the given DFA, then sorts the list according to number of each table's item. Algorithm VI.3 is to find a table from the list which can save most memory if merged with the given one. Notice that after two transition tables are merged, the number of unique transitions of merged transition table is limited to 256, which means 8 bits will be enough to present an item of pointer indirection table.

3.3 Algorithm Analysis and Discussion

It is difficult to analyze and formulate an optimum algorithm because the metric between two tables are changing every step, therefore our algorithm just makes the best choice in each step. We all know that select a best pair elements from a list need O (n^2) iterations while find a best one for a given element need only O (n) iterations. If we just select a proper transition at the beginning of creating a new group, this can dramatically decrease the complexity of our algorithm. Usually, the more items a table has, the bigger probability it has to save more memory when merged with other tables. In order to take this observation, we sorted the list L according to the number of unique transitions of each table. We selected a table from L which has the most unique transitions as the first one at the beginning of creating a new group, then select appropriate table to merge with it in each step.

```
Algorithm VI.1 SHARE_TRANSITIONS (DFA D)
transition table s, t;
init (L)
t ← deletemax(L)
while (t <> NULL and L <> NULL)
    s← getmaxnode(L,t)
    while(s)
        t ← merge_table(t,s)
        remove(L,s)
        getmaxnode(L,t, s)
    t ← deletemax(L)

Algorithm VI.2: INIT (L)
transition table t;
for each (t in D)
    insert (L, t)
quicksort(L)

Algorithm VI.3: getmaxnode (L,e,t)
transition table s; int m;
m ← 0, t ← NULL;
for each (s in L)
    if(metric(e,s) > m and
    ((e.num_of_unique_transitions+
    s.num_of_unique_transition-metric(s,e)) < 256))
        m ← metric(e,s);
        t ← s;
```

Fig. 4 Transition sharing algorithm.

We now present a complexity analysis of our algorithm. The original DFA is assumed to have n states. First, we analyze the complexity of algorithm VI.1. Here is one call to the sorted list initialization function of algorithm V.2. And for each iteration there are a *deletemax* operation and several calls to the *getmaxnode* function of algorithm VI.3. Since inserting all the transition tables needs O (n) complexity and the quick sort operation takes O ($n^2/2$) complexity in an array, so building the sorted list leads to a total O ($n^2/2$) complexity. Next, a single *deletemax* operation on a sorted list takes O (1) complexity, while a single *getmaxnode* operation in algorithm VI.3 needs O (n) complexity. However, after each call to the *deletemax* or *getmaxnode* operation, the total number of elements in the list will reduce by one, so the operations in each step will decrease sharply. For n tables, the needed operation will be $n, n-1 \ldots 1$. The worst maximum possible iterations occur when all the transition tables can't merge with another one, and the total will be $n(n+1)/2$. From the above analyses, we can know that the overall complexity of the algorithm is O (n^2). And all the assistant space is a list L, which leads an O (n) complexity.

Since the regular expressions can not be added into a DFA incrementally, updates to rule sets mean constructing a new DFA form the updated rule sets. In order to apply DFAs in practical networking application, their memory requirements must be reduced first, thus in the practical deep packet inspection, constructing a DFA needs two steps: constructing a normal DFA and reducing the memory requirement of the normal DFA. Since the first step has classic solutions, we will focus on the second step. Suppose a DFA has n states, D^{2}FA needs a space reduction graph which is defined on the same vertex (state) as the original DFA. This graph has n^2 edges at worst case. The reduction procedure is constructing a maximum weight panning tree with specified bounded diameter from the graph. State merging needs a heap which may have n^2 elements as well as a weight graph. When two states are merged, data structures of other states in the DFA, as well as weights of edges in the weight graph must be updated. The total space and time complexities of each approach are showed in Table 1. We can ob-

Table 1 Space and time complexity comparisons of various approaches.

approach	Space complexity	Time complexity
Transitions sharing	$O(n)$	$O(n^2)$
D^{2}FA	$O(n^2)$	$O(n^2 \log n)$
state merging	$O(n^2)$	$(1\text{-}1/max\ labels)n^3 \log n$

Table 2 Summary of characteristics of the regular expressions groups.

Source	#of RE	Avg. ASCII length	% expressions using single wildcards (*,+,?)	% expressions length Restrictions {,+,k}	Total # of states	Total # of transitions	space requirement (KB)
Bro	77	40	0	0	986	252416	986
Bro	173	33	0	0	5066	1311488	5066
Snort	26	37	96.15	46.15	8967	475904	8967
Snort	45	46	93.3	88.8	9744	949248	9744
L7	5	52	80	20	4762	2008576	4762
L7	10	57	80	10	21386	8064512	21386

Table 3 Space and time requirements comparisons of various approaches.

Source	space requirement of Transition sharing (KB)	Time requirement of Transition sharing(s)	space requirement of D^{2}FA (KB)	Time requirement of D^{2}FA(s)	space requirement of state merging （KB)	Time requirement of state merging (s)
Bro(77)	986	6.74	59606.71	16.74	16208.39	1271.00
Bro(173)	5066	443.104	495829.52	450.289	406071.56	38130.56
Snort(26)	8967	46.18	399837.19	1359.03	1265327.77	18284.29
Snort(45)	9744	135.66	400645.44	1739.57	1493268.00	63995.23
L7(5)	4762	35.20	391099.34	590.317	359084.56	42129.75
L7(10)	21386	2249.19	607908.27	10562.557	>2000000.00	

serve that our approach is more efficient than other two approaches on both space and time complexity, this makes our approach more suitable and feasible for practical network applications.

4. Experimental Results

In this section, we performed experiments on rule sets used in a wide variety of networking applications to evaluate the space reductions and the matching performance of our approach. We present experimental results on security rulesets: Bro and Snort as well as the protocol identification patterns of L7 system. We evaluate the benefits of our scheme over the D^{2}FA and state merging.

4.1 Experimental Setup and Memory Representation

We implemented our approach and state merging approach, while the D^{2}FA approach was obtained from [13]. The width of transition tables is set to 32 bits, amenable to a software implementation. The width of pointer indirection tables is set to 8 bits. Bitmaps are set to 256 bits. Failure pointers were set to the mostly frequently occurring transitions. In order to compute memory reduction clearly, we build an independent label table instead of appending the labels to the pointer indirection table in the state merging implementation. The width of label table is set to 8 bits. Since the data structure used in D^{2}FA is a naive implementation (each state has full 256 transitions), we translate the states of D^{2}FAs into data structure we describe in Sect. 3 before computing the final memory reduction.

In order to do a fully comparison, for each rule set we constructed D^{2}FA using the refined version of spanning tree with the diameter 4, 8 and 16. We also run the state merging approach with 4, 8 and 16 as the *max_labels* (the maximum number of sub-states a merged state can consist of) respectively (the parameter of each approach is in the brackets).

4.2 Rule Database

The rule sets we used consist of security rules of Bro, snort, and protocol identification patterns of L7 system. The rule sets of bro and snort are obtained form [13], which is published with D^{2}FA approach. In the case of L7 patterns, we selected two groups from the entire 109 patterns [1] randomly. These rule sets include variety type regular expressions, wildcard, counting constraints, etc. The key properties of our representative rule sets are summarized in Table 2.

In order to better illustrate the memory reduction of our approach and make comparison with other approaches, for each group regular expressions, we compiled them into a single DFA instead of using the set splitting techniques proposed in [10]. The space and time requirements of each approach are showed in the Table 3. We can observe that transition sharing is more efficient both on time and space complexity. Compared with other two approaches, it can finish the compression procedure in shorter time with less space, which means it is more suitable and feasible for the frequently updated DFAs used in practical network applications. For the last group rule sets in the Table 2, State merging didn't finish the reduction procedure in our implementation because of huge space requirements.

4.3 Results on Some Rule Sets

A: *Memory Requirement*
For the data structure described in Sect. 3, the basic memory requirement of a state includes bitmap, pointer indirection table and transition table, apart from this, D^{2}FA needs default transition pointers and state merging needs labels. So a big factor reduction in the number of states or transitions may translate into a smaller reduction in actual memory usage because of overheads in the other parts of data structure.

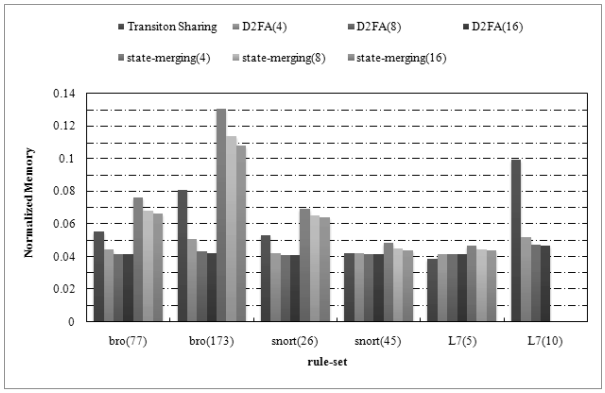


Fig. 5 Memory reduction of transition sharing, D2FA with diameter 4, 8, 16 and state merging with *max_label* 4, 8, 16.

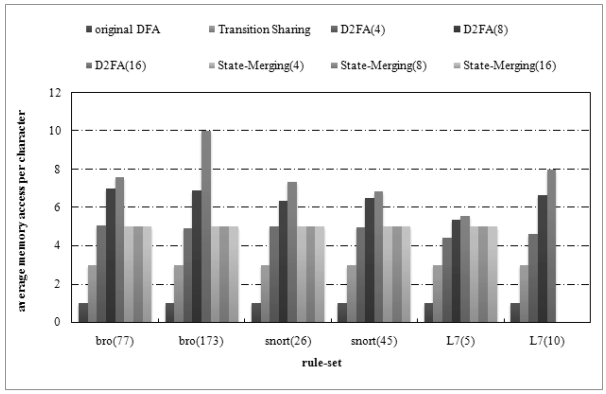


Fig. 6 Average memory access times per character of each resulting DFA constructed by each approach.

Since our goal is to reduce memory requirements of DFAs in practical, we will compute the memory requirements of both basic and special parts of states in each approach instead of just counting transition tables. We compared the memory requirement of transitions sharing, D^{2}FA and state merging in Fig. 5. The data are all normalized to the memory requirement of the naive data structure.

We can observe that the final memory requirements deceased by a factor 10 or even more for most of the rule sets. Transitions Sharing and D^{2}FA do better than the state merging approach on all the rule sets, and D^{2}FA do better than Transitions Sharing on most of rule sets. The memory reduction of D^{2}FA is increasing as the diameter of spanning tree, while the memory reduction of state merging is increasing as the *max_labels*. However, to each approach, the memory reduction is different for each rule sets even when two rule sets come from the same application. That means memory reduction of each approach is related to the character of rule sets.

These three approaches all aim at eliminating the duplicate transitions between states. However, the D^{2}FA and state merging found the best pair of states in a DFA at every step, while our approach just selected the state that has the most transitions as the first one of a new group. When selected the states to share their transitions in our approach, we use the constraint that the number of items of a combined transition table must not be more than 256, so it can take full use of each bit of pointer indirection index, while the *max_labels* of state merging is assigned by user, after the states were merged, they may waste some representing capacity of their pointer indirection index. State merging also reduces the number of states. However, it has to introduce additional memory: labels as the discriminator to indicate which portion of the merged state should be accessed during DFA traversal.

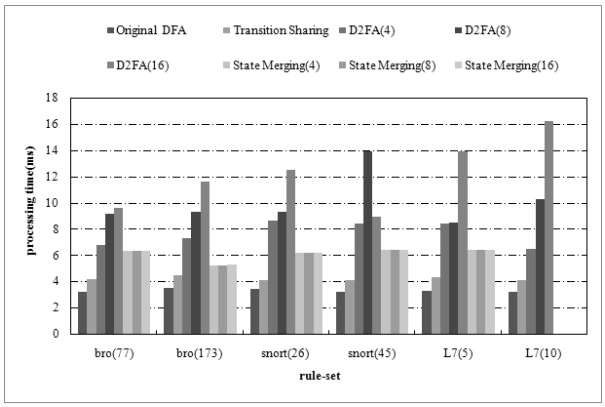
B: *Matching Performance*

As mentioned, reducing memory requirements of DFAs is a trade-off with memory access per input character. To the data structure we used, traveling a state needs three times memory access (ignore the bitmap counting): first, get the transition index from the pointer indirection table, second, load the transition table, and at last get the next state form the table using the transition index. For each input character, our approach needed to travel one state as a normal DFA. D^{2}FA may need travel more than one state to process an input character. While in the case of state merging, since our implementation has an independent label table for the state merging approach, traveling a merged state needs five times memory access due to its additional operations to get the destination state and label from a merged state (note that the memory access for processing one input character can be decreased using other implementation for state merging). We statistics all the states of result DFAs constructed by each approach from each rule set. The average of memory access for one input character of different approaches is shown in the Fig. 6.

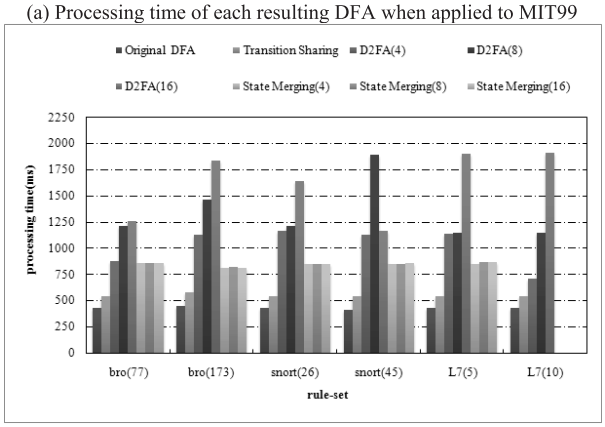
We can observe that the memory access for processing an input character in our approach is fixed at three times, and it fixes at five times in the state merging. This indicates that both of the two approaches processing an input character need travel only one state. For the D^{2}FA, the memory access is increasing as the diameter of spanning tree. However, growth rates are different and related to the rule sets' properties.

To evaluate the performance of each approach in practical environments, we test all the resulted DFAs by two network traffic. The first traffic data (MIT99) is downloaded from MIT Lincoln laboratory [14], and the other one (RT101) is captured from the gateway of a company LAN. Figure 7 shows the total processing time of each resulting DFA. (a) shows the results of processing MIT99 and (b) shows the results of processing RT101.

We can observe that most of real processing time is consistent with the statistics of the compressed DFA. For transitions sharing, all processing times are nearly the same when processing the same traffic no matter the rule sets. For State merging, the processing times are also fixed when process the same traffic no matter the rule set and parameter. While to D^{2}FA, most of the processing times are increasing as the diameter of spanning tree. However, to both of



(a) Processing time of each resulting DFA when applied to MIT99



(b) Processing time of each resulting DFA when applied to RT101

Fig. 7 Processing time of each resulting DFA on real word traffics.

the two traffic data, the processing time of D^2FA (16) is fewer than D^2FA (8) when the D^2FA were construct from the rule set snort (45). It means that to the D^2FA, the traveling paths are also affected by the character of the input data. Due to the paper limitation, we do not elaborate on the details in this paper. It is worth noting that the compressed DFA's processing times didn't increase proportional to their memory access times when compared with original DFA. This is because the original DFA's big memory requirement would lead to poor cache performance, while the compressed DFA's were more cache-efficient because of their small size.

5. Conclusions

DPI using the regular expressions is becoming a more and more important task in the modern network management. However, the memory requirements of DFAs make them impractical for the real world rule sets. In this paper, we proposed a novel method to drastically reduce the memory requirements of DFAs while still maintain the high matching speed and provide worst-case guarantees. We removed the duplicate transitions between states by dividing all the states into a number of groups and making each group of DFA states share a merged transition table. Huge memory will be saved after transition sharing. Processing an input character only needs travel one state in our scheme. By reducing the memory requirement of a DFA, the matching engine can be easily integrated into any practical applications. Another advantage of our approach is that our algorithm is more efficient on both of the time and space complexity.

We performed experiments on real world rule sets used by snort, bro and L7 systems. The results show that for those practical rule sets, the performance of our scheme is better than that of the state merging in memory compressions, and it is faster than D^2FA in the matching process. Overall, our approach offers the best memory versus run-time trade-offs. This means it is more suitable and feasible for practical network applications.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the National High-Tech Development 863 Program of China (Nos. 2007AA01Z467) and Major State Basic Research Development Program (Nos. 2007CB311100).

References

[1] J. Levandoski, E. Sommer, and M. Strait, "Application layer packet classifier for Linux," http://l7-filter.sourceforge.net/
[2] "SNORT network intrusion detection system," http://www.snort.org
[3] "Bro intrusion detection system," http://bro-ids.org/Overview.html
[4] J.E. Hopcroft, R. Motwani, and J.D. Ullman, Automata Theory, Languages and Compilation, 3rd ed., Addison Wesley, 2004.
[5] J. Hopcroft, "An nlogn algorithm for minimizing states in a finite automaton," in Theory of Machines and Computation, ed. J. Kohavi, pp.189–196, Academic, New York, 1971.
[6] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman, and R.H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," IEEE/ACM ANCS, pp.93–102, Dec. 2006.
[7] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," ACM SIGCOMM, pp.339–350, Sept. 2006.
[8] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," IEEE/ACM ANCS, pp.81–92, Dec. 2006.
[9] M. Becchi and S. Cadambi, "An improved algorithm to accelerate regular expression evaluation," IEEE/ACM ANCS, pp.145–154, Dec. 2007.
[10] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," IEEE Infocom, pp.1064–1072, 2007.
[11] F. Baboescu and G. Varghese, "Scalable packet classification," IEEE/ACM Trans. Netw., vol.13, no.1, pp.2–14, Feb. 2005.
[12] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," IEEE Infocom, pp.333–340, March 2004.
[13] http://regex.wustl.edu/files/regex_1.2.tar
[14] MIT DARPA Intrusion Detection Data Sets, http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1999/testing/week4/thursday/outside.tcpdump.gz l, 1999.

Shuzhuang Zhang received the B.S. and M.S. degrees in Computer Science & Technology from YanShan University. During 2004–2006, he stayed in Computer Architecture Research Laboratory. He now is a Ph.D Candidates of Research Centre of Computer Network and Information Security Technology of Harbin Institute of Technology, his primary research focus lies in network security and Information Content Security.

Hao Luo is a PhD and Assistant Professor of Research Center of Information Intelligent and Information Security of Institute of Computing Technology, Chinese Academy of Sciences. His research interest is network security.

Binxing Fang is a professor of Department of Computer Science and Engineering, Harbin Institute of Technology. He is academician of Chinese Academy of Engineering. His research interests include computer network and information security.

Xiaochun Yun is a professor of the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include network security and information security.