This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE ICC 2010 proceedings

Pattern-Based DFA for Memory-Efficient and Scalable Multiple Regular Expression Matching

Junchen Jiang[1], Yang Xu[2], Tian Pan[1], Yi Tang[1], Bin Liu[1]

[1]Tsinghua National Laboratory for Information Science and Technology Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China
livejc@gmail.com, platinum127@gmail.com, tangyi@ymail.com, lmyujie@gmail.com

[2]Polytechnic Institute of NYU, USA
yangxu@poly.edu

***Abstract*— In Network Intrusion Detection System, Deterministic Finite Automaton (DFA) is widely used to compare packet content at a constant speed against a set of patterns specified in regular expressions (regexes). However, combining many regex patterns into a single DFA causes a serious state explosion. Partitioning the pattern set into several subsets, each of which produces a small DFA, is a practical way to deflate the state explosion.**

In this paper, we propose a regex pattern grouping scheme based on a new DFA model called Pattern-Based DFA (P-DFA) which supports efficient pattern-based operations, such as insertion, deletion, and etc. By using these basic operations, one can easily measure the state explosion when combining a set of regex patterns into a single DFA. Based on the privilege, we develop regex grouping algorithms for mitigating the state explosion in parallel and sequential matching environments, respectively. The evaluation shows that under the same constraints, our approach requires only half the number of groups compared with the most well-known algorithms.

***Key Words*—Deterministic Finite Automata (DFA), Deep Packet Inspection (DPI), Regular Expression**

I. INTRODUCTION

Nowadays, Deep Packet Inspection (DPI) has become a crucial technique to network security detection and application identification, where the payload of packets in traffic streams is matched against a set of patterns to identify specific viruses, attacks and protocols. Regular expression (regex) is the de facto standard to represent these patterns for its expressiveness, flexibility and matching efficiency [1]. For example, in Linux Application Protocol Classifier (L7-filter) [2], all the 112 protocol are expressed in regexes. In addition, the famous open source NIDS tool, Snort [3], adopts more than 1100 regex signatures.

It is well-known that multiple regex patterns can be combined into a single DFA to achieve a constant scanning speed which is independent to the number of regex patterns [4]. However, the resulting DFA could be extremely large because of the severe state explosion, and therefore, hard to fit into state-of-the-art memories. A practical solution of resolving the state explosion is to divide the set of regex patterns into several groups, and generate an individual DFA for each group. The most well-known regex pattern grouping algorithms [4] have to check the chance of state explosion between every two regex patterns, and only group patterns without interactivity together. These grouping approaches have two critical defects:

1. **Raw granularity**: They are not applicable to the complicated patterns. Actually, interactivity widely exists among regex patterns. In our evaluation, only 23.6% pattern pairs in L7-filter and about 5% pattern pairs in Snort web patterns will not cause state explosion when combined together. Hence, the traditional grouping methods of checking if there is a state explosion between any two regex patterns are too raw in granularity to achieve a refined partition and would therefore be degenerated to random pattern grouping scheme.
2. **High time complexity**: they require DFA reconstruction to obtain explosion coefficient for every two regex pattern combination. The time complexity on computation for any two regex patterns is $O(N^2)$, where N is the number of states after combination. Therefore, such approaches are time-consuming and infeasible for large pattern set.

To our understanding, pattern grouping is essentially a composite operation consisting of pattern insertions and deletions. The conventional DFA is designed for the purpose of fast matching and compact representation of regex patterns, and therefore doesn't store the specific information of each pattern. As a result, it is unable to dynamically measure the state explosion for a set of regex patterns in which new/old regex patterns are incrementally inserted and deleted.

In this paper, we propose a novel DFA model called *Pattern-Based DFA* (P-DFA). Generally, a P-DFA records the information of every regex pattern in each of its state, and supports the operations like pattern insertion and deletion. For better hardware utilization, we save such information in a data structure called *Pattern-Based Structure* (PBS). Through treating pattern grouping as composite operation of insertions and deletions, we are able to predict the state explosion without virtually executing the grouping. Based on the prediction, algorithms are further developed to group regex patterns to create compact DFAs. Our contributions are summarized as follows:

1. We propose a new DFA structure called P-DFA which preserves the information of every regex pattern in each state and can accurately measure the explosion of pattern grouping.
2. We develop techniques for pattern grouping which are

978-1-4244-6404-3/10/$26.00 ©2010 IEEE

more time and memory-efficient compared with existing methods in both theoretical and experimental evaluation. Our experiment shows that, under the same constraints, our approach produces only half the number of groups in comparison to the most well-known algorithms.

This paper is organized as follows. After going over the related works and problem statement in Section II, we give the detailed presentation of P-DFA and PBS in Section III. The grouping algorithms based on P-DFA and PBS are provided and analyzed in Section IV. The experimental analysis is given in Section V. Finally, Section VI concludes the paper.

II. Related Work and Problem Statement

A. DFA Compression Methods

Most of the researches today in regular expression matching focus on abating the memory requirement of DFA [5-12] by means of reducing transitions or states. D^{2}FA [8] is one of the most famous methods using transitions reduction, where the authors compress DFA through adding default transitions, which save the memory at the cost of increasing the memory access times per input character. Following D^{2}FA, many papers [7][9] were proposed to improve its worst case performance. For deflation of DFAs state, the state-of-art work named XFA [6][10] uses auxiliary memory to reduce the DFA state explosion. However, it involves a lot of manual work on regex patterns which is error-prone and of low efficiency.

Our proposed method does not conflict with above works. After generating small DFA for each group, the single DFA compression methods mentioned above can be further employed to compress each DFA.

B. Problem Statement

We now classify the pattern grouping problem into two categories in terms of implementation and storage architecture, and formalize the requirements in each category. In Section IV we design two specific algorithms for the two scenarios.

In general purpose processor architecture, the processor sequentially executes multiple composite DFAs stored in a shared memory. The objective of pattern grouping in this circumstance is to group all regex patterns into the smallest number of groups under the constraint that the total storage cost should not exceed the size of available shared memory.

For multi-parallel processor architecture: In FPGA, multiple groups of regex patterns can be processed in parallel with exterior SRAM or DRAM chips. Due to the limitation of FPGA pins, the number of parallel processing units cannot be too large (generally, < 10). Moreover, the memory size for each processing unit is also limited. Therefore, the objective of pattern grouping in this circumstance is to divide regex patterns into a smallest number of groups under the constraint that the storage cost of each group does not exceed the size of available memory for each parallel unit.

Mathematically, given a set of k regex patterns $P = \{p_1, \dots, p_k\}$, we denote the size of its corresponding DFA as $f(P)$. The problem of regex pattern grouping is to partition the regex pattern set P into t subsets $A_1 \dots A_t$ so that t is as small as possible under the constraints that, for **general purpose processor architecture,** total memory $f(A_1) + f(A_2) + \cdots + f(A_t)$ does not exceed the shared memory size L; for **multi-parallel processor architecture,** each of $f(A_1), \dots, f(A_t)$ does not exceed the memory upper bound L for each parallel unit. Given that the memory size of DFA is bounded by the number of states, we simply define $f(P)$ as the number of states, and L as the corresponding upper bound.

III. Our Approach

A. Pattern-Based DFA (P-DFA)

Typically, given a pattern set $P = \{p_1, \dots, p_k\}$ with k patterns, an NFA over the whole set is first generated and is then converted to a DFA using subset construction [14]. In subset construction, the states of the resulting DFA (before being renumbered) are subsets of the state set of the original NFA (the elements are called sub-states) (Figure 1-1-(a) and (b) describe the typical NFA and DFA respectively). In our approach, unlike the traditional DFA construction which compiles all regex patterns directly into one DFA, k DFA $M_1, \dots, M_k$ on k regex patterns are generated separately. By treating the k DFAs as an NFA, we then convert it to one DFA, namely *Pattern-Based DFA* (*P-DFA*).

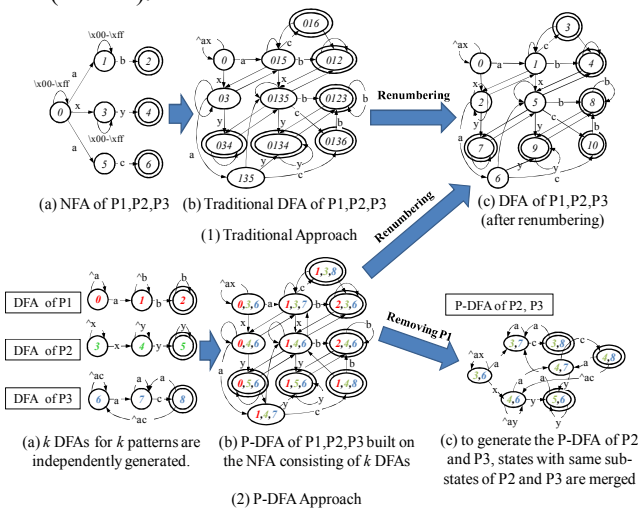


Figure 1 Comparison between P-DFA and DFA of a regex set {P1=.*a.*b, P2=.*x.*y, P3=.*ac} (some transitions are omitted).

Figure 1 compares the construction of a traditional DFA (Figure 1-1) with that of a P-DFA (Figure 1-2) for a sampling regex pattern set P={p_1=.*a.*b, p_2=.*x.*y, p_3 =.*ac}[1]. In Figure 1-2-(a), k (=3) DFAs for k regex patterns are separately generated first, and then combined to one P-DFA in Figure 1-2-(b). A comparison between Figure 1-1-(b) and Figure 1-2-(b) shows that P-DFA and DFA are equivalent except for their state IDs. In fact, it is easy to prove that the P-DFA and the DFA generated from the same pattern set are mutually isomorphic (before minimization). Hence after being renumbered, they should be identical (Figure 1-1-(c) and 1-2-(b)). Also it is evident that each state in P-DFA contains k sub-states, each of which is derived from one regex pattern.

[1] Here "*" refers to closure and "." refers to wildcard (i.e., any byte).

Figure 1-2-(c) shows another important feature of P-DFA which is formally described as:

Property 1: For any subset $A = \{p_{j_1}, \ldots, p_{j_t}\}$ *of a regex pattern set* $P = \{p_1, \ldots, p_k\}$*, the P-DFA after removing A from P can be obtained by deleting the sub-states of* $p_{j_1}, \ldots, p_{j_t}$ *in all states of the P-DFA of P, and merging identical states (i.e., having the same set of subsets) together with transitions.*

As an example, in Figure 1-2-(c), when p_1 is removed, we can get the P-DFA of the two remaining patterns by removing the sub-states of p_1 (in red) and merging the same states together with their transitions. For instance, states {0,3,6},{2,3,6} have identical sub-states on p_2 (sub-state 3) and p_3 (sub-state 6), so they can be merged when p_1 is removed. Hence the P-DFA on a regex pattern set can be easily updated once some regex pattern is removed from the regex pattern set. Also, when a new regex pattern is added to the pattern set, we first compile the new regex pattern to a DFA (treated as the P-DFA of a single regex pattern) and combine it into the P-DFA of the existing pattern set.

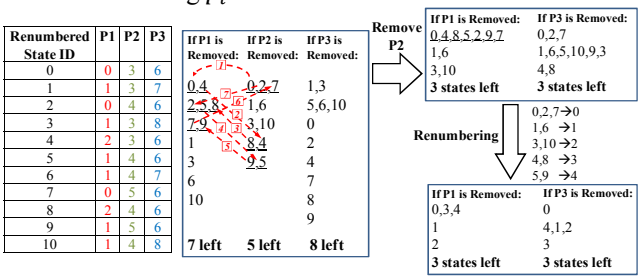
To sum up, P-DFA represents an identical DFA structure with efficient support for operations such as adding new regex pattern as well as removing an existing regex pattern. However, these properties are no longer available after being renumbered. To this end, we give another structure for saving the additional information in software so that P-DFA is renumbered and implemented in hardware with no lose on information of sub-states.

B. Pattern-Based Structure (PBS)

To indicate which states of P-DFA can be merged when a regex pattern is removed, we introduce a data structure called *Pattern-Based Structure* (PBS). Essentially, of the P-DFA for k regex patterns, its PBS consists of k *sub-PBS*s $PBS_1, \ldots, PBS_k$. In more details, the states that will be merged when p_i is removed are stored in a linked list in PBS_i which is actually a group of such linked lists. After building the PBS, we renumber the state IDs (Figure 2(a)) in P-DFA and PBS. Figure 2(b) illustrates the PBS of the P-DFA in Figure 1-2-(b). For instance, state 0 and 4 represent states {0,3,6}and {2,3,6} of P-DFA in Figure 1-2-(b), respectively, whose sub-states of p_2 and p_3 are identical (sub-states3 and 6), so they belong to the same linked list in PBS_1.

The P-DFA and its PBS can be generated simultaneously. For any new generated state in P-DFA construction, we compare it with each existing state to see whether they can be merged once p_i is removed (i.e., whether their sub-states derived from regex patterns other than p_i are the same) and store it in the corresponding position of sub-PBS PBS_i.

Now, if we want to add or remove a regex pattern, it is necessary to update the PBS. We devise two procedures: `Combine` and `Delete` respectively for the operations. The procedure `Combine` treats the P-DFA of existing pattern set and P-DFA of the newly added pattern as an NFA and converts them to one P-DFA using the method in the previous subsection and update the PBS at the same time. For brevity, we only present the pseudo code of the procedure `Delete` in Algorithm 1 which updates PBS when one regex pattern is removed. We observe that if two states are merged after removing p_i and p_j, they shall be in the same linked list either in PBS_i or in PBS_j. So for every state i, we recursively search the states that can be merged with it. As an example, Figure 2(b) and 2(c) depict the original PBS and resulting PBS when p_2 is removed. The red dashed arrows show the recursion sequence for searching the states that can be merged with state 0. The complexity analysis is presented in Subsection IV-C. It is demonstrated in Figure 2(b) that by calculating $|PBS_i|$ (i.e., the number of linked lists) one can predict the number of states after removing p_i.



(a) Renumbering state IDs (b) PBS of P-DFA on the complete pattern set (Renumbered state IDs are used) (c) PBS of P-DFA on the pattern set after removing P2 (After renumbering)

Figure 2 PBS of P-DFA in Figure 1(2) and a sample update of rule deletion.

Algorithm 1: Updating P-DFA M of regex pattern set $P = \{p_1, \ldots, p_k\}$ and its $PBS = \{PBS_1, \ldots, PBS_k\}$ once removing one pattern p_i. Q is the state set of P-DFA.

```
Search(q, i, j)        /* Search the state that can be merged with j in PBS_i */
1:  T = the set of states that can be merged with q in PBS_i
2:  if T = {q} then q.Merged() = true; return;
3:  else for all s ∈ T \ {q} do
4:          s.Merged() = true;
5:          all transition in M to s will be changed to q
6:          all transition in M from s will be changed to from q
7:          Search(s, j, i)
8:      end of for
9:  end of if

Delete(M, i)
10:  remove PBS_i from the PBS of M
11:  for j = 1,...,i−1,i+1,...,k do
12:      for q ∈ Q do  q.Merged() = false  end of for
13:      while ∃q ∈ PBS_i, q.Merged() = false
14:          Search(q, i, j)
15:      end of while
16:  end of for
17:  update the set of states so that merged states are labeled with the same ID;
18:  for j = 1,...,i−1,i+1,...,k do apply the new state ID's in PBS_j  end of for
```

IV. Grouping Algorithm using PBS

A. Overview of Our Scheme

We present an overview of our grouping algorithms and the major distinction compared with existing ones in this subsection. Previous analysis reveals the fact that one can predict the number of states when removing any regex pattern. Based on it, we define an operation called `Reduce` which takes two P-DFAs and one threshold L as input. By using the PBS to predict the size of the resting set, the operation always moves a pattern from one P-DFA to another to keep the size of the former as close as possible to L. Algorithm 2 gives the details of `Reduce`.

By using `Reduce` as a basic operation, Figure 3 presents a general grouping procedure which is essentially different from the existing approaches. We first compile the given set of regex patterns to a complete P-DFA and a PBS , and then employ greedy algorithms (see the next subsection) to *split* the P-DFA with the aid of PBS, rather than start with *adding*

regex patterns to an empty set which is shown ineffective in Section I. The efficiency of our approach will be analyzed after introducing the two algorithms designed for the two problems stated in Section II.

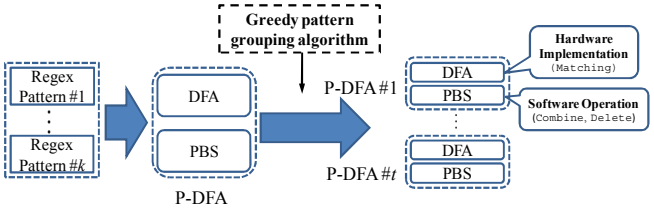


Figure 3 General Scheme of pattern grouping using P-DFA and PBS.

B. Design of Grouping Algorithms

For general purpose processor architecture (refer to Subsection II-B): `Group1` (in Algorithm 2) minimizes the number of groups while keeping the total number of states limited. After generating the complete P-DFA, `Group1` keeps calling `Reduce` to split the current largest group in size into two small groups until the sum of all groups' size is smaller than the given limit (see line 10). To minimize the number of groups, we intend to execute the loop between lines 10 to 18 as few times as possible since each loop produces one new group. Therefore, every time a group is split, we seek to maximize the reduction in total size. Since almost all patterns are mutually inflated (see Section I), a group with M states will be decreased by at least $M/2$ states by calling `Reduce`. Therefore, it is reasonable to split the largest group in each loop (see line 12).

For multi-parallel processor architecture (refer to Subsection II-B): `Group2` (in Algorithm 2) minimizes the number of groups while keeping the number of states in each group limited. Since almost all patterns are mutually inflated (see Section I), the group number will be smaller when each group has as many patterns as possible. In other words, the size of P-DFA over each group should be as close to the limit as possible. `Group2` is designed to meet this requirement. For one group, when the size of its P-DFA is larger than the limit, we extract a pattern from the group by calling `Reduce`, so that the size of P-DFA is as close to the limit as possible.

C. Practical Improvements

Table 1 lists the complexity in storage and processing compared with existing approaches. Construction of P-DFA and PBS can be regarded as repeatedly executing `Combine`. According to Algorithm 1, `Delete` has a processing complexity of $O(kN)$ since every state in P-DFA is processed once for each rest PBS. It reveals that our approach has a same storage and processing complexity of construction procedure while more valuable information is preserved. It is shown that the complexity of grouping algorithms is bounded by the complexity of construction, which implies that our approach does not increase the overall computing and storage complexity after utilizing grouping algorithms.

Strictly speaking, our approach assumes that enough information is stored during the construction of a complete P-DFA. However, it is infeasible due to the prohibitively large number of states.

Since the feasibility of our approach is strongly related to maximum accessible memory size (denoted as *MAX*), we incrementally add patterns to the P-DFA and PBS by calling "`Combine`" until memory size becomes larger than *MAX* and then `Group1` or `Group2` are called to further split the P-DFA and its PBS. We use this method (called "pre-grouping technique") in the evaluation, and it works well in our experiments. Algorithm 2 below gives the application of pre-grouping technique, namely `Group1'`, on `Group1`. `Group2` can be adjusted similarly to `Group2'`.

Algorithm 2: Grouping algorithm to minimize the number of groups

```
Reduce (M, M', L)
1:   M.PBS = {PBS_1,..., PBS_k};
2:   let |PBS_i| be the smallest in M.PBS
                          /* |PBS_i| is the # of states after removing p_i */
3:   if |PBS_i| < L then
4:      let |PBS_t| be the largest one so that |PBS_i| ≤ L  Finish = true
5:   end of if
6:   Delete(M, i)        /* remove p_i from M */
7:   Combine(M', p_i); /*add p_i to M' and its PBS'*/

Group1(P = {p_1,..., p_k}, L)   /* with limit L on overall size*/
8:   generate P-DFA M and M.PBS of P;
9:   Group = {M};
10:  while Σ_{M∈Group} M.size()> L   /* M.size() is # of states in P-DFA M */
11:     Finish = false;
12:     find the M in Group with the largest M.size();  Group = Group \ {M};
13:     new Temp is an empty P-DFA;
14:     while not Finish
15:        Reduce (M, Temp, Temp.size());
16:     end of while
17:     Group = Group ∪ {M, Temp};
18:  end of while

Group2(P = {p_1,..., p_k}, L)   /* with limit L on each group size*/
19:  generate P-DFA M of M.PBS of P;
20:  Group = ∅;
21:  while M.size() < L
22:     new Temp is an empty P-DFA;
23:     while (!Finish)
24:        Reduce (M, Temp, L);
25:     Group = Group ∪ {Temp};
26:     end of while
27:  end of while
28:  Group = Group ∪ {M};
```

Table 1 The complexity analysis of PBS and Grouping algorithms (N the number of states in P-DFA; k is the number of patterns; Σ is the size of alphabet set; m is the number of patterns removed in the loop of line 13,14 in Group2 and line 21,22 in Group1, t is the number of groups $m \ll k < \Sigma$)

	Traditional construction of DFA	Updating PBS and P-DFA	
		`Combine`	`Delete`
Storage complexity	$O(\Sigma N)$	$O(\Sigma N + kN)$	$O(\Sigma N + kN)$
Processing complexity	$O(N^2)$	$O(N^2)$	$O(kN)$

	`Reduce`	Grouping algorithms		Using pre-grouping technique
		`Group1`	`Group2`	
Storage complexity	0	From $O(\Sigma N + kN)$ to negligible	From $O(\Sigma N + kN)$ to negligible	$< MAX$
Processing complexity	$O(mN^2)$	$O(tmN^2)$	$O(tmN^2)$	$O(tmN^2)$

Algorithm 2': Applying pre-grouping technique on Group1

```
Group1'(P = {p_1,..., p_k}, L, MAX)
/*MAX is the limit of accessible memory*/
1:   M is an empty P-DFA;
2:   while (P ≠ ∅)
3:      while M.size() < MAX   /* M.size() is # of states in P-DFA M */
4:         randomly choose p from P and remove it from P;
5:         Combine(M, p);
6:      Delete(M, p);
7:      Group = {M};
8:      .....(same to line 10 to 18 of Group1)
9:      end of while
10:  end of while
```

V. EVALUATION

A. Experimental Setup

To evaluate the efficiency of P-DFA, we performed experiments based on ruleset extracted from L7-filter (28-May-2009 [2]), and Snort (V2.8.4: 17-June-2009 [3]). We took all 112 regex patterns from L7-filter and randomly selected 300 regex patterns (denoted as Snort1 and Snort2) from Snort's web pcre ruleset as our evaluation datasets.

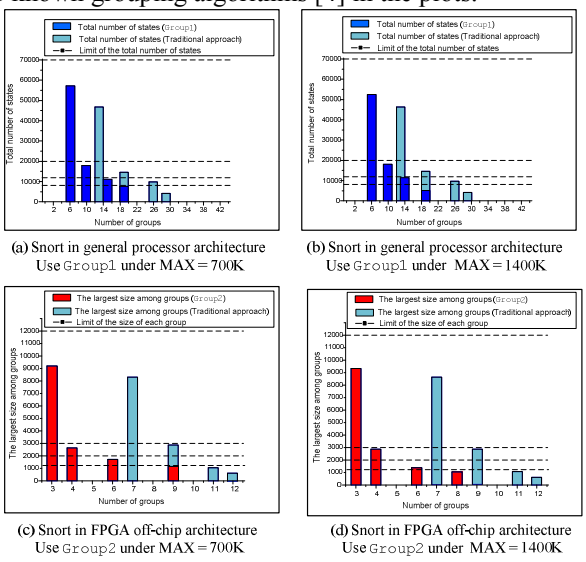
We first investigated the interactivity between any two patterns. We said that a pattern pair (x, y) causes state explosion if $f(\{x\}) + f(\{y\}) < f(\{x, y\})$. As shown in Table 2, nearly 95% pattern pairs from Snort ruleset had state explosion after being combined. The reason might be that few state character ("^") appeared in Snort patterns. For L7-filter patterns, about 76% of pattern pairs had state explosion once combined.

Table 2 Explosion probability of different regex pattern sets.

Rule Set	Pattern Number	Probability of Explosion (pattern pair)
Snort1	150	95.3%
Snort2	150	92.5%
L7-filter	112	76.4%

B. Efficiency of Pattern Grouping Algorithms

The performance of pattern grouping algorithm was evaluated by the latest Snort ruleset. To test all patterns, we employed the pre-grouping technique (mentioned in Section IV-C). The result of the experiment is illustrated in Figure 4. For the purpose of comparison, we also give the performance of the most well-known grouping algorithms [4] in the plots.



(a) Snort in general processor architecture Use `Group1` under MAX = 700K

(b) Snort in general processor architecture Use `Group1` under MAX = 1400K

(c) Snort in FPGA off-chip architecture Use `Group2` under MAX = 700K

(d) Snort in FPGA off-chip architecture Use `Group2` under MAX = 1400K

Figure 4 Performance of pattern grouping algorithm on Snort ruleset (`Group1` and `Group2`).

In Figure 4 (a) and (b), we use `Group1'` (proposed in Section IV-C) to minimize the number of groups, when the total number of states is limited. It can be observed that `Group1'` has a stable performance under various scales of limit. Similarly, (c) and (d) also indicate stable performance of `Group2'` when conducted under various off-chip SRAM scenarios. In addition, it can also be seen that pre-grouping algorithm provides similar results under different MAX's, suggesting that the restraint in maximum memory size imposes little impact on the performance. A comparison between our approach and the most well-known algorithm shows that our approach reduces almost half the amount of groups.

VI. ACKNOWLEDGEMENTS

This work is supported by NSFC (60625201, 60873250, 60903182), 973 project (2007CB310701), 863 high-tech project (2007AA01Z216) and Tsinghua University Initiative Scientific Research Program.

VII. CONCLUSION

In this paper, we propose a novel DFA model named P-DFA which is able to precisely quantize the explosion of states when multiple regex patterns are combined. Based on P-DFA, we develop regex pattern grouping algorithms which achieve significant improvements on both time and memory complexities compared with the existing approaches. Generally speaking, our approach takes the merits of storing enough information during the construction of a P-DFA and PBS.

VIII. REFERENCES

[1] R. Sommer and V. Paxson, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context" ACM CCS 2003

[2] "Application Layer Packet Classifier for Linux." http://l7-filter.sourceforge.net/

[3] Snort. http://www.snort.org/, 2009

[4] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman and R. H. Katz, "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection", in ANCS 2006

[5] S. Kumar, B. Chandrasekaran, J. Turner, G. Varghese, "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia" , in ANCS 2007

[6] R. Smith, C. Estan, S. Jha "XFAs: Faster signature matching with extended automata", IEEE Symposium on Security and Privacy (Oakland), May 2008

[7] S. Kumar, J. Turner and J. Williams, "Advanced Algorithms for Fast and Scalable Deep Packet Inspection", in ANCS 2006

[8] S. Kumar et alt., "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection" in ACM SIGCOMM, Sept 2006.

[9] M. Becchi and P. Crowley, "An Improved Algorithm to Accelerate Regular Expression Evaluation", ANCS 2007.

[10] R. Smith, C. Estan, S. Jha, S. Kong, "Deflating the Big Bang: fast and scalable deep packet inspection with variable-extended automata" SIGCOMM, August 2008

[11] C. L. Hayes and Y. Luo: "DPICO: a high speed deep packet inspection engine using compact finite automata" ANCS 2007

[12] "Ragel State Machine Compiler", http://www.complang.org/ragel/

[13] "JFlex - The Fast Scanner Generator for Java", http://jflex.de/

[14] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to Automata Theory, Languages and Computation", Addison Wesley, 2001.

[15] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In the 23rd Conference of the IEEE Communications Society (Infocomm), Mar. 2004.