PLOS ONE

RESEARCH ARTICLE

Exploring efficient grouping algorithms in regular expression matching

Chengcheng Xu[1☯¤*], **Jinshu Su**[1,2☯], **Shuhui Chen**[1☯]

1 College of Computer, National University of Defense Technology, Changsha, Hunan, China, **2** National Key Laboratory for Parallel and Distributed Processing, Changsha, Hunan, China

☯ These authors contributed equally to this work.
¤ Current address: National Key Laboratory of Science and Technology on Vessel Integrated Power System, Naval University of Engineering, Wuhan, Hubei, China
* xuchengcheng@nudt.edu.cn

Citation: Xu C, Su J, Chen S (2018) Exploring efficient grouping algorithms in regular expression matching. PLoS ONE 13(10): e0206068. https://doi.org/10.1371/journal.pone.0206068

Editor: Xiangtao Li, Northeast Normal University, CHINA

Received: November 26, 2017

Accepted: October 6, 2018

Published: October 24, 2018

Data Availability Statement: All relevant data are within the paper and its Supporting Information files.

Funding: This work was supported by the National Natural Science Foundation of China under Grant 61379148 (http://www.nsfc.gov.cn/). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exist.

Abstract

Background

Regular expression matching (REM) is widely employed as the major tool for deep packet inspection (DPI) applications. For automatic processing, the regular expression patterns need to be converted to a deterministic finite automata (DFA). However, with the ever-increasing scale and complexity of pattern sets, state explosion problem has brought a great challenge to the DFA based regular expression matching. Rule grouping is a direct method to solve the state explosion problem. The original rule set is divided into multiple disjoint groups, and each group is compiled to a separate DFA, thus to significantly restrain the severe state explosion problem when compiling all the rules to a single DFA.

Objective

For practical implementation, the total number of DFA states should be as few as possible, thus the data structures of these DFAs can be deployed on fast on-chip memories for rapid access. In addition, to support fast pattern update in some applications, the time cost for grouping should be as small as possible. In this study, we aimed to propose an efficient grouping method, which generates as few states as possible with as little time overhead as possible.

Methods

When compiling multiple patterns into a single DFA, the number of DFA states is usually greater than the total number of states when compiling each pattern to a separate DFA. This is mainly caused by the semantic overlaps among different rules. By quantifying the interaction values for each pair of rules, the rule grouping problem can be reduced to the maximum *k*-cut graph partitioning problem. Then, we propose a heuristic algorithm called the one-step greedy (OSG) algorithm to solve this NP-hard problem. What's more, a sub-routine named the heuristic initialization (HI) algorithm is devised to further optimize the grouping algorithms.

Results

We employed three practical rule sets for the experimental evaluation. Results show that the OSG algorithm outperforms the state-of-the-art grouping solutions regarding both the total number of DFA states and time cost for grouping. The HI subroutine also demonstrates its significant optimization effect on the grouping algorithms.

Conclusions

The DFA state explosion problem has became the most challenging issue in the regular expression matching applications. Rule grouping is a practical direction by dividing the original rule sets into multiple disjoint groups. In this paper, we investigate the current grouping solutions, and propose a compact and efficient grouping algorithm. Experiments conducted on practical rule sets demonstrate the superiority of our proposal.

Introduction

Modern network services increasingly rely on the processing of stream payloads. These services leverage the features identified from the payloads to perform content-aware network applications, such as traffic billing, application protocol identification, load balancing, and network intrusion detection, etc. [1] Deep packet inspection (DPI) is the core component for the identification process, here the "deep" means the inspection checks not only the header part but also the payload part. In the DPI process, the packet payload is compared with hundreds of predefined signatures byte by byte, to detect out whether the payload matches any signature(s).

In the early days, signatures were mainly described with simple character strings. Classical string matching algorithms such as Aho-Corasick (AC) algorithm [2] and SBOM algorithm [3] could provide efficient matching speed with linear space consumption. With the development of network applications, exact strings were incompetent for signature description. Then, the regular expression is introduced as a main tool for its powerful and flexible description ability. Now the regular expression (RE) has been widely used in the network applications such as the Linux application protocol classifier (L7-filter) [4], the network intrusion detection system of Snort [5] and Bro [6], as well as the network devices such as the Cavium matching engines [7] and the IBM PowerEN processor [8].

For automatic processing on the platforms, the regular expression rules need to be converted to an equivalent finite state automaton (FSA) first. Each state in the FSA represent a different matching progress, and the matching process is an input byte driven state traversal process. The matching process starts from the initial state, in each step, the matching engine reads one byte from the payload sequentially. Based on the current state(s) and input byte, it inquires the FSA to achieve the next state(s). Then the achieved state(s) will be regarded as the current active state(s) for the processing of the next input byte. This process loops until the last byte of the payload. During the traversal process, any accessed final state denotes the identification of its corresponding regular expression rules.

There are two kinds of traditional FSAs, namely the nondeterministic finite automata (NFA) and the deterministic finite automata (DFA), and they are equivalent in the description ability. Usually the RE rules are first compiled to the NFA, then the NFA is converted to an equivalent DFA with the subset construction algorithm. The NFA and DFA have the exact

opposite behavior on the space cost and matching efficiency. In the DFA matching procedure, there is only one active state at any time. Thus for each input symbol the matching engine only needs one memory access to achieve the next active state, and the time complexity is the fixed *O(1)* for each byte processing. While for the NFA matching procedure, a set of NFA states maybe active concurrently, thus the matching engine needs to inquire the NFA multiple times to achieve the next active NFA state set. In the worst case, all NFA states maybe active concurrently.

For the excellent matching efficiency of the DFA, it is much more widely used in the memory-centric architectures. However, the subset construction algorithm usually introduce much state expansion even state explosion for the DFA. In fact, each DFA state represents a set of NFA states which maybe active concurrently during matching. Thus, when converting a NFA with n states to a DFA, the number of DFA states could be 2^n in the worst case.

For practical implementation, the space cost of the DFA storage should be as small as possible. The time cost of processing a given payload is equal to the payload length times the memory access latency. As the payload length is fixed, for better performance the DFA should be deployed on fast on-chip memories, such as caches and SRAMs. However, the space of these memories is usually very small, thus reducing the space cost of DFA is the main issue for DFA based matching.

As countermeasures, current works mainly focus on compressing the space requirement of DFA. The main data structure of the DFA is a two-dimensional matrix, where the rows represent the DFA states and columns represent the input symbols. Each element in the matrix is called a transition which denotes the next state for the corresponding state and input symbol. Due to the deterministic features of DFA, there exists much redundancy among the transition. Compression operation can be implemented from different dimensions, such as the state merging solutions [9] on the state dimension and alphabet re-encoding proposals [10–13] on the input character dimension. These solutions perform well on simple and small RE sets, while most other research focuses on the transition compression aspect. D^2FA [14] is the most representative one, and many transition compression algorithms [10, 15–18] are based on the D^2FA. A representative D^2FA based solution, such as the A-DFA [10] proposed by Becchi, can achieve a compression ratio of more than 90% with no more than two memory accesses on average for each byte processing.

With the ever-increasing complexity and scale of the RE rule set, the state explosion has became an inevitable problem, which usually makes the DFA unavailable on moderate platforms. Though these compression solutions are very efficient, they would be inapplicable in practice because most compression solutions rely on the original DFA, while on the other hand the DFA is unavailable.

In this work, we focus on the solutions to solve the state explosion problem. More precisely, we aim at the so-called rule grouping method which was firstly proposed by Yu [19]. In the rule grouping method, the RE rule sets are divided into k disjoint sets and compiled to k separate DFAs to avoid the state explosion. In this way, the k DFAs can work concurrently on the same stream, thus it is extremely suitable for parallel platforms such as multi-core processors, GPUs, etc. Furthermore, as the grouping result is a set of standard DFAs, all the above mentioned compression solutions can be combined with the grouping solutions to further reduce the space cost.

Unfortunately, Yu's method [19] is quite inefficient as it usually costs much time especially when the rule set is large. However, somewhat surprisingly, since the invention of rule grouping, only few subsequent works have been found in the literature. It is worth noting that Rohrer also proposed a rule grouping method based on simulated annealing (SA) algorithm [20] but the resulting performance was not good enough compared with Yu's method despite

that it improved the grouping efficiency. Similar to the rule grouping idea, Luchaup [21] proposed a novel and efficient structure called DFA-trees to solve the state explosion problem. This work allows fitting large rulesets in small groups by repeatedly approximating grouped DFAs. To some extent, the DFA-trees method and rule grouping method are orthogonal. Our proposed grouping algorithms could be combined with Luchaup's work to further reducing the grouping time and improving the grouping results during the DFA-trees construction.

Our contributions

Based on some empirical assumptions [20], the rule grouping problem could be reduced to the maximum k-cut graph partitioning problem. As a corresponding solution, we then propose a compact and efficient algorithm called the one-step greedy (OSG) algorithm, which achieves desirable performance in both the space cost and time cost. Furthermore, we devise a subroutine called heuristic initialization (HI) algorithm. The HI algorithm can generate relatively good initial solutions, which could make significant improvements for these grouping algorithms. Experimental results demonstrate that our OSG algorithm outperforms other solutions in terms of the space cost and the time cost in most cases, and the HI algorithm also works well for the grouping algorithms.

Analysis of the rule grouping problem

According to [22], DFA state explosion mainly raises from two kind of patterns, namely the patterns with "dot-star"-like features (such as ".*", ". +", "[^c]*") and the patterns with "counting constraints" features (such as ".{n}", "[^c]{n}"). A single RE with counting constraints may cause severe state expansion, but a single "dot-star" pattern never cause state expansion, the expansion arises only when compiling the "dot-star" pattern with other patterns together. Take the patterns "*ab.*cd*" and "*ef.*gh*" as example, there is no any expansion when compiling them separately, as shown in Fig 1A and 1B. But when compiling them together, state

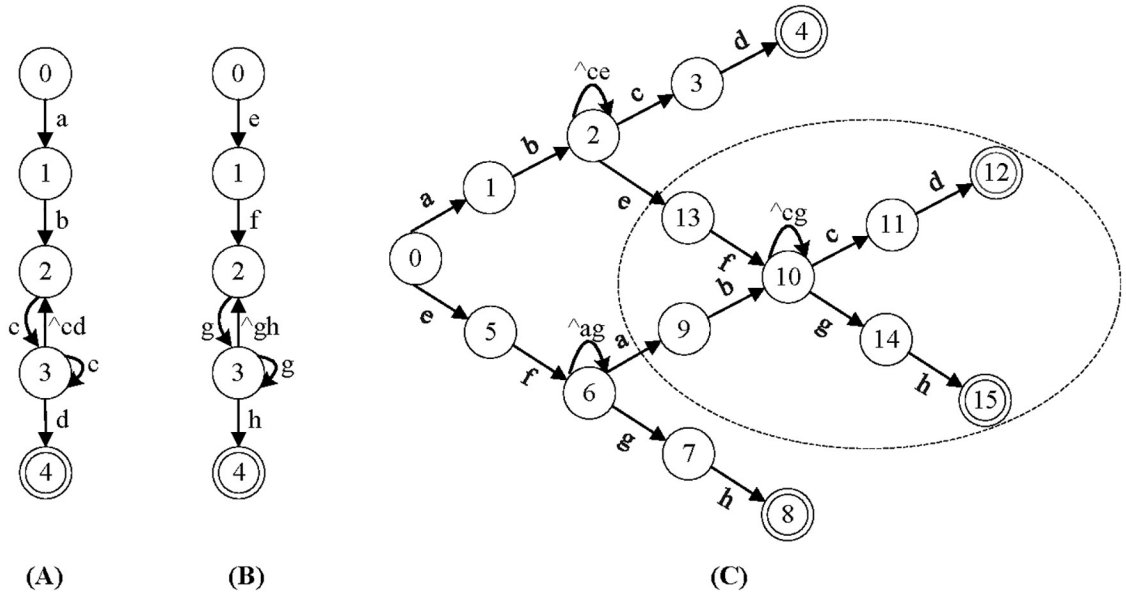

Fig 1. The illustration of the DFA state expansion caused by ".*". (A) DFA for pattern "*ab.*cd*", (B) DFA for pattern "*ef.*gh*", (C) the composite DFA for the patterns of "*ab.*cd*" and "*ef.*gh*". For simplicity, some transitions are omitted in these figures.

https://doi.org/10.1371/journal.pone.0206068.g001

expansion occurs as shown in the ellipse of Fig 1C. As ".*" can match any sequence, extra states are needed to record the partial matching of these patterns. For example, state 10 represents both the prefixes "*ab*" and "*ef*" have been matched. When the number of ".*" rule grows, the combinations of the prefixes also increase exponentially, which would require huge number of DFA states.

Rule grouping is a natural method to restrain the expansion, because rules from different groups would not incur state expansion anymore. For a given RE set $RE_{set} = \{r_1, r_2, \cdots, r_n\}$, rule grouping is to divide the rules into k disjoint subsets and compile each subset to a DFA. These DFAs can work concurrently on parallel platforms. For practical implementations, the time and space cost of grouping should be as small as possible.

Yu [19] defined the *interaction* as whether state expansion exists when compiling two rules together. Based on the interaction relationships of each pair of rules, Yu proposed a simple heuristic algorithm, and the main idea is to set an upper space limitation. As long as the current group does not exceed the limits, it chooses a rule which has least interactions with the current group, and adds the rule to the current group.

Yu only judges whether two rules have interaction. Rohrer [20] takes one step further, he also quantifies the interaction value. For the rule r_i and r_j, I_{ij} in Eq (1) means the interaction value between r_i and r_j, where S denotes the number of states when converting its subscripted rule(s) to a DFA. For example, S_{ij} means the corresponding number of states when compiling r_i and r_j to a DFA.

$$I_{ij} = S_{ij} - S_i - S_j \tag{1}$$

Empirical results reveal that, when adding a new rule r_m to the set r_i and r_j, the increased number of DFA states can be estimated as $S_m + I_{mi} + I_{mj}$, namely

$$S_{ijm} = S_{ij} + S_m + I_{mi} + I_{mj} \tag{2}$$

Combined with Eq (1), we can get Eq (3).

$$S_{ijm} = S_i + S_j + S_m + I_{ij} + I_{mi} + I_{mj} \tag{3}$$

Then for the *RE* set with n rules, we have

$$S_{RE} = \sum_{i=1}^{n} S_i + \sum_{i=2}^{n} \sum_{j=1}^{i-1} I_{ij} \tag{4}$$

When grouped into k subsets, the total number of states can be described as Eq (5), where RE_l denotes the lth group.

$$S_{RE} = \sum_{i=1}^{n} S_i + \sum_{l=1}^{k} \sum_{\substack{i=2 \\ i \in RE_l}}^{n} \sum_{\substack{j=1 \\ j \in RE_l}}^{i-1} I_{ij} \tag{5}$$

Now, our target is to find the best grouping solution to minimize the S_{RE}. For a given rule set, the left item $\sum_{i=1}^{n} S_i$ is fixed, thus it is equivalent to minimize the right item of the plus sign. A direct method is to exploit every possible grouping solution, and choose the best one as the result. However, there exist $O(k^n)$ grouping solutions, the time cost is unacceptable even for several tens of rules.

Rohrer [20] further transforms the rule grouping to an equivalent graph partitioning problem. For a given RE rule set RE_{set}, we can construct a weighted undirected graph $G = (V, E)$, where each node represents a rule, and the edge weight denotes the interaction value for the

corresponding pair of rules. Based on previous analysis, the problem is equivalent to find the graph partitioning solution whose total edge weight inside each group is minimum. As the total weight of the whole graph is fixed, it means that the total weight among groups for the solution should be maximum. This is a typical max k-cut graph partitioning problem, but it is NP-hard [23]. Therefore, it is very hard to find an optimal solution.

Heuristic algorithms for the rule grouping problem

Wheeler [24] classified the solutions for max k-cut graph partitioning into three categories according to their effectiveness, namely the exact methods, the approximation methods with performance guarantees and the inexact methods without guarantees. The first two kinds of methods are too time-consuming to be applied in the rule grouping application, thus we employ the third kind of methods. Heuristic algorithms seem to be a possible direction for the rule grouping problem.

Through investigating research [25–27] for graph partitioning problem, we learned that the general combinational optimization algorithms such as simulated annealing (SA) algorithm and genetic algorithm (GA), etc. and the classical specific graph partitioning algorithm such as Kernighan-Lin (KL) method [28] are the most recommended algorithms. We have implemented these three algorithms, but we found that the grouping results of these algorithms are much worse than or even not comparable to the current best rule grouping algorithm [19]. Combined the above research and our evaluations, we learned that a given algorithm may have different degrees of adaptability to different applications or data sets. Thus, we are motivated to design new grouping methods for the specific rule grouping application. In this section, we devise a compact and efficient algorithm to handle the partitioning problem.

Inspiration from the state expansion

Before explaining the one-step greedy algorithm, we would first introduce a subroutine called the heuristic initialization (HI) algorithm which is designed for improving the grouping algorithms. Note that the main purpose of rule grouping is to eliminate the interactions of rules from different groups. The "dot-star"-like characteristics such as ".*", ".+", "[^c]*" make a great contribution to state expansion, especially to the rule interactions. Different rules have different number of ".*"-like characteristics, thus they have different power to cause the state expansion. Intuitively, we should divide the rules with great expansion power into different groups as possible, in order to reduce the superposed expansion from different rules.

This provides us the inspiration to improve the grouping algorithms. For each rule r_i, we use EP_i as in Eq (6) to quantify its expansion power. The EP_i can reflect the average expansion power of r_i when combining it with other rules.

$$EP_i = \frac{1}{n}\sum_{j=1}^{n}\frac{I_{ij}}{S_j} \tag{6}$$

With the assist of the quantified expansion power for each rule, we design a heuristic algorithm as in Algorithm 1 to generate an initial solution. This solution is not so good as the best grouping solution, but it is much better than a random grouping solution.

Algorithm 1 The heuristic initialization algorithm

```
for i = 1 to re_num do
  Compute the expansion power EP_i for rule r_i;
end for
Sort the EPs in descending order, assumed as EP_{k_1}, EP_{k_2}, ..., EP_{k_{re_num}};
```

```
5: for i = 1 to group_num do
6:   Add rule r_{k_i} to group G_i;
7: end for
8: for i = group_num + 1 to re_num do
9:   Find the group with the least total EP, assumed as G_j;
10:   Add rule r_{k_i} to group G_j;
11:   Update the total EP value for G_j;
12: end for
13: return the current grouping solution;
```

The HI algorithm works as follows. First, the RE rules are sorted according to their expansion power in descending order (Step 1 to Step 4). Here, each subscript k_i denotes a specific rule identifier. Then, the first k rules are distributed to k groups separately (step 5 to step 7). For the remaining rules, in each step, the next rule is added to the group with the least expansion power, until all the rules have been grouped (step 8 to step 12). We will leverage the returned solution in step 13 to improve the grouping algorithms, with the expectation that a good initial solution would make contributions to the convergence rate and the final result.

In order to better explain the HI algorithm, we present a simple example as shown in Fig 2. Suppose we need to divide 8 rules into 3 groups. With the given rules, first, we compute the interaction values for each pair of rules, as presented in the interaction matrix. Then, the average expansion power for each rule can be achieved as in the *EP* array with Eq (6). Finally, the *EP* array is ordered, and the rules are grouped according to Algorithm 1 (step 5 to step 12). The total state number of DFA would be 25503 if compiling these rules together, while after grouping the total state number of the grouped DFAs is only 443. This is a pretty good result, considering the simplicity and efficiency of the HI algorithm.

The one-step greedy algorithm

The OSG in Algorithm 2 is a greedy algorithm for the rule grouping problem, and the core procedure is to search and execute the one-step move which provides the biggest gain, namely, the biggest cutsize increase. Here, the one-step move means moving a rule from its original group to another group. The OSG algorithm starts from a random initial solution. Then, it computes the cutsize increase matrix (step 2 to step 6), where each element C_{ij} denotes the increase of the total cutsize after moving rule r_i to group G_j. This matrix is the main data structure in the grouping algorithm, and it will be updated along with the grouping process.

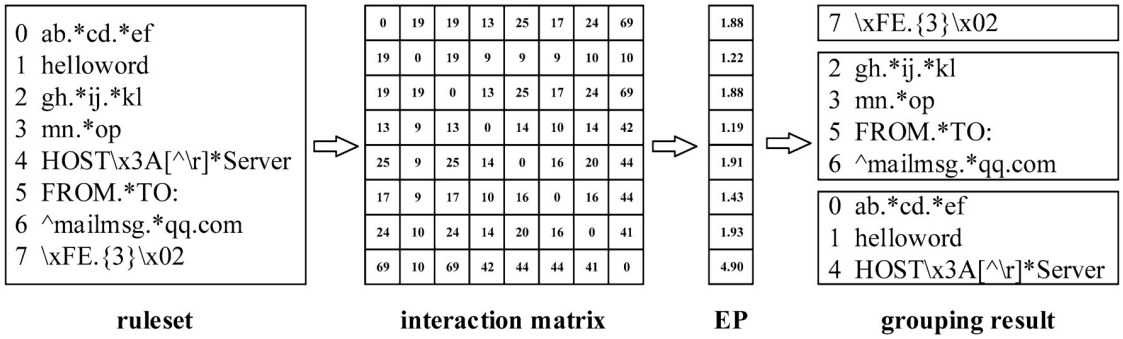

Fig 2. An example for the grouping procedures with the HI algorithm.

https://doi.org/10.1371/journal.pone.0206068.g002

Algorithm 2 The one-step greedy algorithm for the rule grouping problem

```
1: Generate a random initial solution;
2: for i = 1 to re_num do
3:   for j = 1 to group_num do
4:     Compute cutsize increase value C_ij if move rule r_i to group G_j;
5:   end for
6: end for
7: repeat
8:   repeat
9:     Find the biggest cutsize increase value, assumed as C_gh;
10:    Move the rule r_g to group G_h;
11:    Mark the rule r_g as moved;
12:    Update all the cutsize increase values in C;
13:  untill every rule has been marked or all the cutsize are not
positive
14:  Reset all rules as unmarked;
15: untill no more improvement
16: return the current solution;
```

The main process contains a nested loop. In the inner loop, each time it finds the one-step move with the biggest cutsize increase (step 9), and executes the corresponding move (step 10). Then the moved rule will be marked (step 11), and the matrix needs to be recomputed (step 12) due to the changes of the current grouping solution. The inner loop terminates until all rules have been moved or no positive cutsize increase exists in the matrix. While the outer loop works on the basis of the last inner loop, and it stops until no more improvements can be achieved with the execution of one-step move.

Here, we also employ the ruleset in Fig 3 to illustrate how the OSG algorithm works for grouping. With an initial random grouping solution, it is easy to compute the cutsize increase matrix, where each element denotes the cutsize achievement when moving a corresponding rule to a corresponding group. The grouping process passes through 4 outer loops, and each outer loop involves several inner loop iterations, but for space limitation, we only present the results for outer loop iterations. In the inner loop, for each iteration, the algorithm searches for the one-step move with biggest cutsize achievement and executes this move. The inner loop ends until all the rules have been moved once or no positive one-step move can be found. Then for the next round of outer loop, all the rules will be set as unmoved. Finally, we get the grouping results as (1, 4, 5, 6; 2, 3; 0, 7), and the total state number of the grouped DFAs is 394.

Obviously, the one-step greedy strategy ensures that the result can always march forward to the better direction. We can simply prove this conclusion as follows.

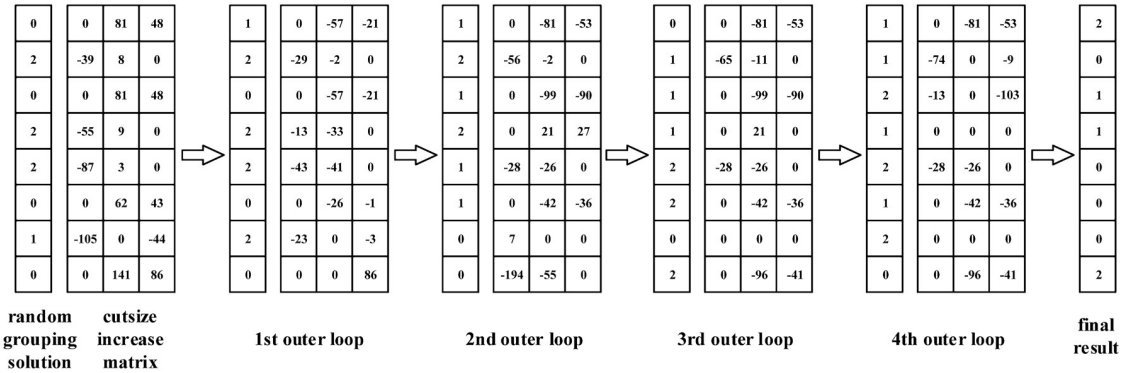

Fig 3. An example for the grouping procedures with the OSG algorithm.

https://doi.org/10.1371/journal.pone.0206068.g003

Proof. The nested loop is the core of the OSG algorithm. For the outer loop, the initial solution of the current loop iteration is based on the returned solution of the previous loop iteration. The current loop iteration would not be adopted if it could not achieve a better solution than its previous loop iteration. Thus, each outer loop iteration yields a better at least not worse solution than the previous loop iteration. For the inner loop, the OSG algorithm repeats executing the one-step move which has biggest cutsize increase. Further, step 13 can guarantee that the value of the biggest cutsize increase in each loop iteration would be positive, which means that each loop iteration results in a bigger cutsize namely a better solution than the previous loop iteration. To sum up, the OSG algorithm would guarantee that the solution could always march forward to the better direction during loop iterations.

As all the actions of the OSG algorithm are based on the perspective of the simple one-step move, this may cause the omission of the global optimal solution. However, the outer loop of the OSG algorithm can guarantee that it would not fall into inferior local optima, because the outer loop will exploit any possible one-step move if this move can lead to a better solution.

Experimental evaluation

Based on Becchi's open-source RE compiler [29], we implemented the above mentioned algorithms. A random grouping algorithm is also implemented for comparison. Experiments are conducted on an Intel Xeon CPU E5-2630 (CPU: 2.3GHz, Memory: 32GB). Three RE rule sets were tested, including 109 RE rules from the L7-filter [4] protocol identification signatures, 127 RE rules from the backdoor file and 250 RE rules from the spyware-put file, the latter two sets are from the Snort [5] NIDS. As these rule sets are complex enough, none of them can be compiled to a single DFA on our platform.

For practical implementation in DPI applications, we divided each rule set into five to eight groups, and recorded the time cost for grouping and the space cost of the grouped DFAs for each algorithm. Except Yu's algorithm, all other algorithms have random factors during the grouping procedure. Thus, a given algorithm would get various results for the same rule set. For reasonable comparisons, we repeated each of these algorithms ten times and compute the average time cost and space cost for comparison.

Figs 4 and 5 illustrate the space cost and time cost for different rule sets and algorithms. Even with grouping, the state explosion still happened, especially when the number of groups is small. We set 3 million states as the upper limit of a single DFA, the numbers beside the symbols in Fig 4 record the number of state explosions (the number of a single DFA states exceeds 3 million) occurred during the 10 runs. As the space cost and time cost are too huge, the explosions were terminated manually, and these situations were not counted for comparison.

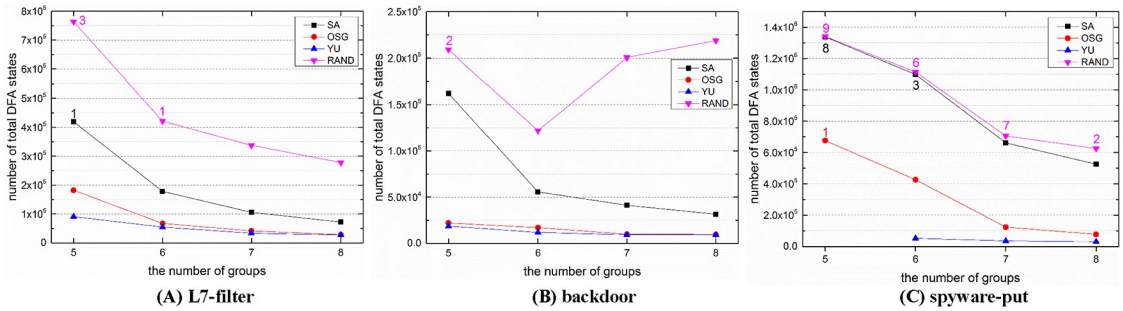

Fig 4. The space cost (total number of DFA states) for different solutions when dividing the rule sets into 5 to 8 groups. Each configuration was repeated ten times, and the figures reflect the average performance. The digital numbers beside the symbols in these figures denote the number of state explosions occurred during the ten trails.

https://doi.org/10.1371/journal.pone.0206068.g004

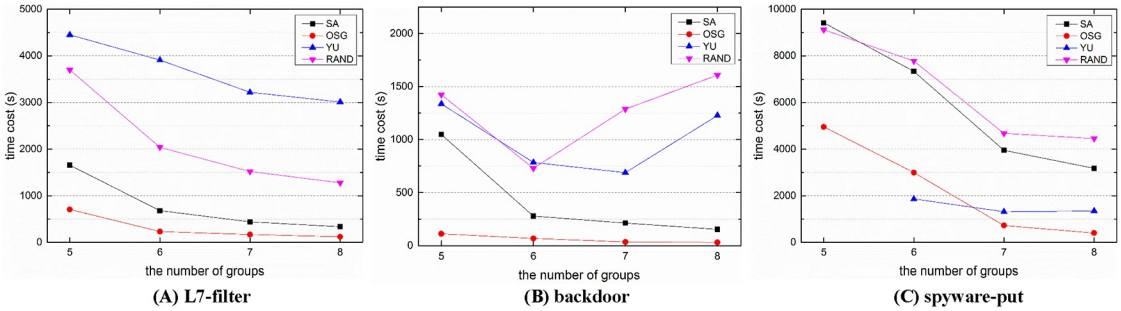

Fig 5. The time cost for different solutions when dividing the rule sets into 5 to 8 groups.

https://doi.org/10.1371/journal.pone.0206068.g005

For the space cost in Fig 4, it is obvious that Yu's algorithm is superior to other algorithms in most situations, and our OSG algorithm is slightly worse than Yu's algorithm but obviously better than the SA algorithm and the random algorithm. While on the other hand, the time cost of Yu's algorithm is about 14 times higher than the OSG algorithm on average. It is because Yu's algorithm requires lots of tests until it reaches the space limitation. And these tests would cost a lot of time for compilation. What's more, the number of groups can not be set beforehand in Yu's algorithm, thus the grouped number is unknown until the finish of grouping. For example, Yu's algorithm could not divide the spyware-put rules into five groups after ten trials.

Yu's algorithm is a classical grouping algorithm and achieves good grouping results. However, we still think that it is impractical to employ Yu's algorithm in practice for the following reasons. First, the grouped DFAs are usually executed on parallel platforms. Each processing unit represents a matching engine and is responsible for matching a grouped DFA, and a packet is processed in parallel on these engines. Because the number of parallel units in a given platform is fixed, the number of grouped DFAs is preferably the same as (or a divisor of) the number of matching engines to maximize the use of the parallel platform. In Yu's grouping algorithm, however, the number of grouped DFAs obtained by limiting the upper bound of the number of states in each group is uncertain and unpredictable. This will be very impractical because the engines are fixed in parallel platforms, and the uncertain number of groups would bring troubles for task scheduling. Second, the grouped DFA are deployed in the main memory, while the main memory is shared by all the matching engines. Our goal is that the overall space overhead is as small as possible and it is not usually required that the size of each grouped DFA should be similar. The same size for each grouped DFA does not make any sense for actual deployment and matching, so defining an upper limit on the number of states in each grouped DFA is not necessary.

Rohrer's SA algorithm is the state of the art grouping method, but it does not perform as well as the OSG algorithm. SA's space cost and time cost are separately 3.7 times and 4.4 times higher than the OSG algorithm on average. In addition, Rohrer's SA algorithm encounters more state explosions in our experiments. The random grouping algorithm performs the worst, both in the space cost and time cost.

As a general optimization algorithm, the SA algorithm can be used to solve a broad range of problems. However, mapping a real problem to the domain of the SA algorithm could be difficult and it requires the familiarity with the algorithm [30]. To be specific, it involves the adjustment of a series of parameters and strategies, including how to determine an appropriate cooling strategy, how to perturb the current solution to generate the neighbor solution, how to select a proper random number generator, and how to implement the algorithm properly. The

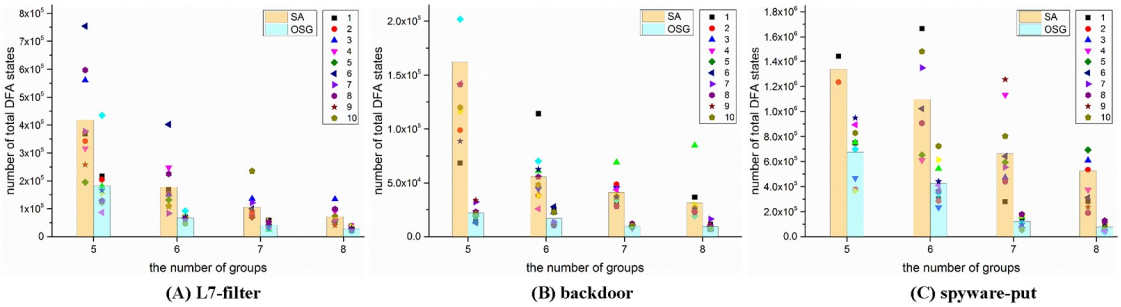

Fig 6. The comparison of grouping stability between the SA algorithm and the OSG algorithm. The column graphs reflect the average performance, and the symbols denote the specific grouping results.

https://doi.org/10.1371/journal.pone.0206068.g006

parameters depend on the specific data set. Thus to get better results, we need to constantly adjust the parameters for different data sets. In addition, both Johnson [31] and Williams [32] concluded that the SA algorithm can not perform very well for sparse graphs. In the domain of rule grouping, it means that the SA algorithm is not suitable for the rule set with many string-like patterns.

For the time cost in Fig 5, except Yu's algorithm we can observe that the time cost has the same tendency with the corresponding space cost in Fig 4. This is because the time cost is mainly composed of the grouping procedure and the DFA compilation procedure after grouping, and the compilation plays a dominant role in the time cost.

As Rohrer's SA algorithm is the state of the art method, we would make more comparisons between the SA algorithm and our OSG algorithm. Fig 6 displays the grouping results for ten different runs. We can observe that except for the better average space cost, the OSG algorithm also achieved better stability compared with the SA algorithm. In general, the space cost of the OSG algorithm is much more centralized than that of the SA algorithm. Even omitting the state explosion situations, the worst grouping result of the SA algorithm in spyware is even 6 times higher than its best one. We attribute the great variation in the SA algorithm to its various stochastic factors. There exist four random factors in the SA algorithm, namely the random initial grouping solution, the randomly selected neighbor numbers and positions, the random reset for neighbor solutions, and the uncertain probability to receive a bad neighbor solution. While the initial solution is the only random factor in the OSG algorithm. Thus the OSG algorithm has better stability on the whole.

As the initial grouping solution is a main influencing factor for the grouping results, a better initial solution may yield better results. To verify this assumption, we employ the HI algorithm to generate good initial solutions, and use these solutions to optimize the grouping algorithms. We tested another ten runs for the SA and OSG algorithm with the initial solutions from HI, as shown in Fig 7. The vertical black lines are called Y error lines, which denote the range of standard deviation for each solution. It is obvious to find the great improvements, especially for the SA algorithm. The average space cost of SA separately dropped by 15%, 11% and 22% for the three rule sets. Another achievement is the improved stability, from the Y error lines, we can find that the grouping results are much more centralized after employing the HI subroutine. What's more, with the HI's support fewer state explosions occurred in the SA algorithm. As for the OSG algorithm, the space cost decreased generally except for few situations, and it separately saved 20%, 1% and 8% space for the three sets. In addition, as the only random factor is removed, the OSG algorithm is a deterministic algorithm now. On the other

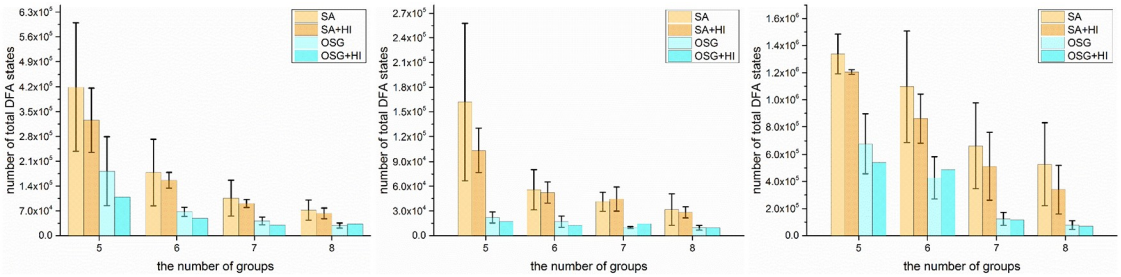

Fig 7. The HI's effect on the space cost of the SA algorithm and the OSG algorithm. The vertical black lines in these figures represent the Y error lines for each configuration.

https://doi.org/10.1371/journal.pone.0206068.g007

hand, as the HI subroutine did not involve the DFA compilation, the time cost would not increase much for the introduction of the HI subroutine.

Conclusion

Rule grouping is a natural method to avoid the state explosion in regular expression matching. However, few research are found in this direction, and current solutions can not perform well in both the space cost and time cost. In this paper, we proposed a novel heuristic grouping algorithm called the OSG algorithm. Experiments show that our OSG algorithm outperforms the state-of-the-art algorithms. In addition, we also proposed a subroutine, called the HI algorithm, to improve the grouping algorithms with no more time overhead.

Supporting information

S1 File. The tested dataset of L7 file from L7-filter.
(TXT)

S2 File. The tested dataset of backdoor file from Snort NIDS.
(TXT)

S3 File. The tested dataset of spyware-put file from Snort NIDS.
(TXT)

S4 File. The achieved results and figures for these results in our experiment, this file should be opened with the Origin software.
(OPJ)

Author Contributions

Conceptualization: Chengcheng Xu.

Formal analysis: Jinshu Su, Shuhui Chen.

Funding acquisition: Jinshu Su.

Investigation: Jinshu Su, Shuhui Chen.

Methodology: Chengcheng Xu, Jinshu Su, Shuhui Chen.

Project administration: Shuhui Chen.

Software: Chengcheng Xu.

Supervision: Jinshu Su, Shuhui Chen.

Validation: Chengcheng Xu.

Writing – original draft: Chengcheng Xu.

Writing – review & editing: Chengcheng Xu, Shuhui Chen.

References

1. Xu C, Chen S, Su J, Yiu SM, Hui LC. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. IEEE Communications Surveys & Tutorials. 2016; 18(4):2991–3029. https://doi.org/10.1109/COMST.2016.2566669
2. Aho AV, Corasick MJ. Efficient string matching: an aid to bibliographic search. Communications of the ACM. 1975; 18(6):333–340. https://doi.org/10.1145/360825.360855
3. Allauzen C, Raffinot M. Factor oracle of a set of words. Institute Gaspart-Monge, University de Marne-la-vallee, TR-99-11. 1999;.
4. Application layer packet classifier for linux. 2009;. http://l7-filter.sourceforge.net/.
5. Snort v2.9. 2014;. http://www.snort.org/.
6. Bro intrusion detection system. 2014;. http://www.bro.org/.
7. Cavium, OCTEON5860;. http://www.cavium.com/OCTEON_MIPS64.html/.
8. IBM, PowerEN PME Public Pattern Sets Wiki 2012;. https://www.ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/PowerEN%20PME%20Public%20Pattern%20Sets/page/Welcome/.
9. Becchi M, Cadambi S. Memory-efficient regular expression search using state merging. In: INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE. IEEE; 2007. p. 1064–1072.
10. Becchi M, Crowley P. A-DFA: a time-and space-efficient DFA compression algorithm for fast regular expression evaluation. ACM Transactions on Architecture and Code Optimization (TACO). 2013; 10(1):4.
11. Brodie BC, Taylor DE, Cytron RK. A scalable architecture for high-throughput regular-expression pattern matching. In: ACM SIGARCH Computer Architecture News. vol. 34. IEEE Computer Society; 2006. p. 191–202.
12. Kong S, Smith R, Estan C. Efficient signature matching with multiple alphabet compression tables. In: Proceedings of the 4th international conference on Security and privacy in communication netowrks. ACM; 2008. p. 1.
13. Becchi M, Crowley P. Efficient regular expression evaluation: theory to practice. In: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. ACM; 2008. p. 50–59.
14. Kumar S, Dharmapurikar S, Yu F, Crowley P, Turner J. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. ACM SIGCOMM Computer Communication Review. 2006; 36(4):339–350. https://doi.org/10.1145/1151659.1159952
15. Becchi M, Crowley P. An improved algorithm to accelerate regular expression evaluation. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems. ACM; 2007. p. 145–154.
16. Kumar S, Turner J, Williams J. Advanced algorithms for fast and scalable deep packet inspection. In: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems. ACM; 2006. p. 81–92.
17. Ficara D, Di Pietro A, Giordano S, Procissi G, Vitucci F, Antichi G. Differential encoding of DFAs for fast regular expression matching. Networking, IEEE/ACM Transactions on. 2011; 19(3):683–694. https://doi.org/10.1109/TNET.2010.2089639
18. Ficara D, Giordano S, Procissi G, Vitucci F, Antichi G, Di Pietro A. An improved DFA for fast regular expression matching. ACM SIGCOMM Computer Communication Review. 2008; 38(5):29–40. https://doi.org/10.1145/1452335.1452339
19. Yu F, Chen Z, Diao Y, Lakshman T, Katz RH. Fast and memory-efficient regular expression matching for deep packet inspection. In: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems. ACM; 2006. p. 93–102.

20. Rohrer J, Atasu K, van Lunteren J, Hagleitner C. Memory-efficient distribution of regular expressions for fast deep packet inspection. In: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis. ACM; 2009. p. 147–154.
21. Luchaup D, De Carli L, Jha S, Bach E. Deep packet inspection with DFA-trees and parametrized language overapproximation. In: INFOCOM, 2014 Proceedings IEEE. IEEE; 2014. p. 531–539.
22. Becchi M, Crowley P. A hybrid finite automaton for practical deep packet inspection. In: Proceedings of the 2007 ACM CoNEXT conference. ACM; 2007. p. 1.
23. Arora S, Karger D, Karpinski M. Polynomial time approximation schemes for dense instances of NP-hard problems. Journal of computer and system sciences. 1999; 58(1):193–210. https://doi.org/10.1006/jcss.1998.1605
24. Wheeler JW. An investigation of the max-cut problem. University of Iowa. 2004;.
25. Fjällström PO. Algorithms for graph partitioning: A survey. vol. 3. Linköping University Electronic Press Linköping; 1998.
26. Wheeler JW. An investigation of the max-cut problem. University of Iowa. 2004;.
27. Elsner U. Graph partitioning-a survey. 1997;.
28. Kernighan BW, Lin S. An efficient heuristic procedure for partitioning graphs. The Bell system technical journal. 1970; 49(2):291–307. https://doi.org/10.1002/j.1538-7305.1970.tb01770.x
29. regex tool;. http://regex.wustl.edu/index.php/Main_Page.
30. Ledesma S, Aviña G, Sanchez R. Practical considerations for simulated annealing implementation. In: Simulated annealing. InTech; 2008.
31. Johnson DS, Aragon CR, McGeoch LA, Schevon C. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. Operations research. 1989; 37(6):865–892. https://doi.org/10.1287/opre.37.6.865
32. Williams RD. Performance of dynamic load balancing algorithms for unstructured mesh calculations. Concurrency: Practice and experience. 1991; 3(5):457–481. https://doi.org/10.1002/cpe.4330030502