An Efficient Regular Expression Matching Method Based on Guess and Verification

Shuzhuang Zhang, Chen Xue, Zhigang Wu
Beijing University of Post and Telecommunications, Beijing, 100876, China
zhangshuzhuang@bupt.edu.cn, maxxue120@sohu.com, wuzhigang@bupt.edu.cn

Abstract

Nowadays, Deterministic Finite Automaton (DFA) has been widely used to compare packet contents at a constant speed against a set of regular expressions in network security inspections. However, combining multiple regular expressions into a single DFA may cause a serious state explosion, which makes them impractical on large-scale rule set. In order to address this issue, this paper proposed a matching method based on “guess and verification”. It first searches the sub-expressions of each rule with DFA, and then verifies the result with NFA once the previous guess is successful. This method takes advantage of the high processing efficiency of DFA and the compact representation of NFA. The result shows that this proposal can provide a high throughput with a moderate memory requirement.

Keywords: Deep packet Inspection, Regular expression, Finite automaton, Sub-expression; Guess and Verification

1. Introduction

Nowadays, Deep Packet Inspection (DPI) has been widely used and becomes a crucial task in network security applications, such as Intrusion Detection System (IDS), Intrusion Prevention System (IPS), and so on. DPI technique compares the payload of packets to a set of given patterns (as known as rule set). Each pattern describes a characteristic of a piece of an attack, a section of malicious code or an application. We call it a “hit” if the packet contents match a pattern. Each hit triggers a predefined action.

Matching means finding patterns of interest within a given data set. It is a classic problem of computer science and has been a well-studied area of research for many years. Though many excellent research results have been proposed in efficient matching, it is still a challenge to design high performance matching engine in network security domain where packet contents must be inspected against large rule sets at a high speed. The whole system’s efficiency is greatly affected by the matching algorithm’s performance. Measurements of the Snort [1] on a production network show that as much as 31% of total processing time is due to string-matching, and this cost increases to as much as 80% of total processing time in the case of Web-intensive traffic. The cost of matching will be more than 90% of the total processing time in Linux Application Protocol Classifier (L7)[2]. Therefore, improving the performance of matching algorithms carries very important significance on improving the efficiency of the security system.

Exact strings are used to define the characteristics of attacks and malicious codes in initial stage of network security, and some representative matching methods are AC[3], Wu-Manber[4], SBOM[5]. However, with the increasing of attacks and aggravation of network attack and defense, the exact strings are barely expressive for complex and large rules. Since regular expressions can give a concise description of a set of attacks, without having to list all elements, they are replacing the exact strings and becoming the main pattern matching language in research and commercial fields, such as Linux Application Protocol Classifier (L7), Snort, Bro[6], Cisco Adaptive Security Appliance[7] and Citrix Application Firewall[8]. In addition, regular expression matching is a central task in email scanning system and application level filtering.

There are two challenges for a regular expression matching engine in network applications:

1) Since the rule sets become larger and larger with the increase of attacks and applications to be identified, the matching engine must allow parallel match over multiple or all of the rules.
Some applications may be deployed on the edge of a network, and deal with a large amount of network traffics in real time. The matching engine must provide a high performance and be capable of dealing with packets at a high speed.

This paper proposed a regular expression matching method based on “guess and verification” which filters out the suspicious packets with DFA and makes verification on those suspicious packets with NFA. It takes advantages of the characteristics of the network security inspections that only few packets can hit a rule. This proposal combines the benefit of DFA and NFA and can provide a high throughput with a moderate memory requirement.

The rest of this paper is organized as follows. In Section 2 the related works are discussed. In Section 3 we give the principle of the method based on “guess and verification” and detail it. The experimental results are given in Section 4. Finally, Section 5 concludes the paper.

2. Related works

Currently, most of the studies take the finite automata (constructed from rule sets, referred as FA hereinafter) as the tool for regular expression matching, either in their deterministic or in their non-deterministic form (referred as DFA and NFA respectively)[9]. The matching operation is then equivalent to a FA traversal guided by the content of the input stream. The resources used in automata-based matching techniques are mainly of following two kinds:

1) Memory resources for storing data structures.
2) Computing resources for matching operations.

Each state of DFA has one and only one outgoing transition for each character of the alphabet. We can combine multiple regular expressions into a single DFA to achieve $O(1)$ matching speed independent to the size of the rule set. This property makes DFA more suitable for online real-time applications than NFA. However, if an NFA has $n$ states, the corresponding DFA can potentially have $2^n$ states, which is called state explosion. State explosion makes DFA hard to fit into state-of-art memories and hence impractical. Nowadays, the researches of regular expression matching with DFA focus on how to reduce the memory requirement of DFA. Those methods can be divided into two categories:

The first category reduces the DFA memory requirement by eliminating “redundant” structures. Its representative methods include: pre-coding of the input, combining outgoing transition, merging state, and so on.

Chen Shuhui et al proposed a set intersected precode(SI-precode) method[10], which encodes all input characters before the conversion from regular expressions to DFA and can save about 50% memory. Michela Becchi et al. presented an alphabet reduction scheme[11] for DFA-based structures which sets two characters into the same class if they are treated in the same way in all DFA states. Ficara. D et al proposed a state encoding scheme[12] in which states reached by a transition labeled with character $c$ would be given a relative-id, and then the shorter relative-ids replace absolute addresses in the state transition table to reduce memory. Huanyun Wang et al. proposed the Run Accumulation Length Encoding DFA (RALE-DFA) which introduces alphabet compression based on appearance character set to cut down the DFA memory requirement [13].

Kumar et al proposed a new representation for regular expressions, called Delayed Input DFA (D²FA)[14]. D²FA reduces the number of transitions between states by introducing a default transition. Two methods are proposed to improve D²FA’s matching efficiency: In [15] Kumar et al proposed a method using content labels as state identifiers, called CD²FA. CD²FA allows one memory-access in each state traversal (as in an uncompressed original DFA). The second improvement is proposed in [11]. It first defines the depth of states, and then adds the constraint that the default transitions can be done only in the direction of deceasing depth. It can be proved that this constraint makes any string of length $n$ require at most $2n$ time’s state traversals to be processed. Ficara. D et al presented a representation for deterministic finite automata, called δFA[12] in which a local fast memory is used as a cache to store neighbor states. It can delete most common transitions between adjacent states by taking into account the different ones only. Michela Becchi et al proposed state merging method [16] that can reduce both the number of states and transitions. This method merges several even “non-equivalent” states by introducing labels and transfers information from the states to the transitions of the DFA. With the extended data structure, the number of states and transitions is dramatically reduced. In [17] Zhang et al proposed a transition table sharing method which removes 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.

Although, those methods mentioned above can eliminate more than 90% of the memory requirement in DFA, it cannot completely solve the problem of state explosion. The number of DFA states may be increase exponentially, while the reduction efficiency is linear. Moreover, DFA cannot be built from many practical rule sets under the current memory conditions.

The second category uses the improved DFA structure or the hybrid structure of the NFA and DFA, thus to avoid the explosion of the DFA states.

Fang Yu et al developed a grouping scheme[18] that can strategically compile a set of regular expressions into several engines to get a remarkable improvement of regular expression matching speed without much increase in memory usage. Jiang et al proposed a regular expression pattern grouping scheme based on Pattern-Based DFA (P-DFA)[19] which supports efficient pattern-based operations, such as insertion, deletion. Compiling a set of regular expressions into several groups is a “general” form of NFA where the number of transitions for each character of each state equals to the number of groups.

Following-up studies promoted and refined above ideas. Huanyun Wang et al. proposed the Jump-CFA[20] which introduces jump strategy and counter to speed up the RE matching and reduce the DFA memory requirement of REs including Counting Constraints. In [21] Kumar et al proposed history-based FA, which consists of a finite state machine coupled with a history-data structure. To achieve a compact automaton, the history-data structure is used to determine which NFA-states to represent. XFA[22,23] is another work which also extends a finite automaton through external variables. The basic idea of XFA is to extend a finite automaton through external variables, namely history bits and counters. These variables are manipulated through simple instructions (set, reset, increment, test) associated either to the state transitions or to the states. Michela Becchi et al proposed a hybrid automaton (Hybrid-FA)[24] which is resulted from interrupting the subset construction operation at those NFA states whose expansion would cause state explosion to happen when converting a NFA into DFA. Furthermore, it gives the refinements to improve the worst case bound on memory bandwidth requirement.

Those methods falling into the second category divide the boundaries of “deterministic” and “non-deterministic” in a finite automaton by splitting the rule set. They are raw granularity and may not be functionally equivalent to the original NFA, which makes them not applicable in complicated rule sets.

3. Guess and Verification

Regular expressions in network security inspections are used to describe the characteristics of various attacks and malicious codes. They pay more attention to the logical relationships of the multi-stage of the attacks and the position relationships of the multi-partition of the malicious codes. Therefore, the rules used in network security inspections can be typically divided into several sub-expressions clearly, and these sub-expressions are connected by the concatenation operators, as shown in figure 1.

![Figure1. Exemplification of rules in network security inspections](image)

The structure of DFA is like a directed graph, where the states are the vertexes and the transitions are the edges. Assume that the depth of initial state is 0. Since few packets can hit a rule during matching in network security inspections, most traversals just access the shallow “vertexes” of finite automaton. Those deeper “vertexes” can be accessed only when the input can match a long prefix of or a whole rule.

DFA compiles all the regular expressions into a whole structure. However, the concatenation operators used to combine multi sub-expressions may cause state explosions during subset construction and hence make the resulted DFA cannot fit into state-of-art memory condition. Besides, combing the states frequently-accessed and the states rarely-accessed together cannot take advantage of the architecture of computer memory hierarchy. The main operation of matching is to access the transition tables, whose processing time depends on the times of memory access and the hit rate of cache.

To address the issue above, we proposed the matching method based on “guess and verification”. It firstly extracts the sub-expressions from each rule, and then guesses whether a packet can hit a whole
rule by those sub-expressions, at last makes verification on those suspicious packets that can hit one or
more sub-expressions of a rule. This paper mainly focuses on the following four concatenation
operators: ".*", "\{n\}”, "[^{c_1}c_2…c_k]*"-like”, "[^{c_1}c_2…c_k]\{m, n\}"-like” where the c indicates a
character(subscripts are used to distinguish different characters). Previous research has shown that
those four operators are the main cause of state explosions when converting the NFA to DFA.

Our method has two benefits: first, we can extract those sub-expressions without above operators,
therefore these sub-expressions will not cause state explosion when compiled into a single DFA. The
guess procedure can be finished fast with DFA. Second, we can use the NFA to make the verification
in that there will be few packets after guessing.

For description convenience: we denote the rule set by \( R \), each regular expression in \( R \) by \( r \)
(subscripts are used to distinguish different regular expressions), denote the \( k \)-th sub-expression of
regular expression \( r_j \) by \( s_{j,k} \), denote the sub-expressions set only belonging to \( r_j \) by \( S_{rj} \) and denote the
number of sub-expressions of each regular expression by \( M \), denote the set of sub-expressions by \( S \),
denote the NFA building from \( r_j \) by \( NFA_{rj} \), denote the set of all the NFA by \( S_{NFA} \), denote a string by \( T \). The pre-process and matching procedure of guess and verification are shown in Algorithm 1 and
Algorithm 2, respectively.

**Algorithm 1: pre_process(\( R \))**

1. \( S=\emptyset, S_{NFA} = \emptyset \)
2. for (each \( r_j \in R \)) do
3. \( S_{rj}=\text{extract_sub}(r_j) \)
4. \( S=S\cup S_{rj} \)
5. for (\( k=1 \) to \( M \)) do
6. if (\( s_{j,k} \in S_{rj} \)) then
7. \( \text{set_map(map}_{j},1,k) \)
8. end if
9. end for
10. \( NFA_{rj}=\text{build_NFA}(r_j) \)
11. \( S_{NFA}=S_{NFA}\cup NFA_{rj} \)
12. end for
13. \( DFA=\text{build_DFA}(S) \)

Algorithm 1 describes the pre-process. We first extract one or multiple sub-expressions from each
regular expression. For each \( r_j \), a bitmap \( \text{map}_{j} \) is used to record the sub-expressions used for DFA
construction: the \( k \)-th bit will be set to 1 only if the \( k \)-th sub-expression has been used. Second, we
build a NFA for \( r_j \). At last we compile all the sub-expressions into a single DFA. Notice that different
regular expressions may have common sub-expressions, we add two lists into each acceptance state of
DFA: the \( \text{rule_list} \) and the \( \text{index_list} \). The \( \text{rule_list} \) records the regular expressions containing the sub-
expressions accepted by this state, while the \( \text{index_list} \) records the sub-expression’s corresponding
bitmaps and its positions index in these bitmaps.

**Algorithm 2: Match (\( DFA, S_{NFA}, T \))**

1. for (each \( p \in T \))
2. \( \text{state}=\text{dfa_match}(DFA,p) \)
3. if (\( \text{state} \in DFA.\text{accept} \_\text{states} \)) then
4. for (each \( s_{NFA} \) in state.\text{rule} \_\text{list} \)
5. \( \text{set_bit} \_\text{map}(0, \text{state}.\text{index} \_\text{list}[j]) \)
6. if (\( \text{map}_{j}==0 \)) then
7. \( \text{nfa_match}(NFA_{rj},T) \)
8. end if
9. end for
10. end if
11. end for

Algorithm 2 describes the matching procedure. For a given string \( T \), we first process each character
\( p \) in it by the DFA obtained from Algorithm 1. If an acceptance state is accessed, we update all the
4. Experimental results and Evaluations

Memory requirements and processing speed are the two main measurements to evaluate regular expression matching schemes. In this section we will test the performance of our proposal (G-V) on some practical rule sets, and compare it with Hybrid-FA.

4.1 Rule sets

Our proposal mainly focuses on the four concatenation operators described above, since they are the main causes of state explosion in practical rule sets. The regular expressions in the experiment are from Snort intrusion detection system and the Linux Layer-7 filter (L7). Snort performs packet payload inspection only after header filtering (i.e. packet classification). Therefore, we clustered rules with common header and selected two groups in this experiment: Web-misc and Backdoor. Table 1 shows the characteristics of the rule sets.

<table>
<thead>
<tr>
<th>Rule set</th>
<th># of Rule</th>
<th># of special Term</th>
<th># of rules has at least one special term</th>
<th># sub-expressions used in guess</th>
<th># states of NFA/DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web-misc</td>
<td>55</td>
<td>3/75/0/31</td>
<td>53</td>
<td>79</td>
<td>7615/&gt;3000000</td>
</tr>
<tr>
<td>Backdoor</td>
<td>153</td>
<td>9/388/0/10</td>
<td>130</td>
<td>356</td>
<td>3364/&gt;3000000</td>
</tr>
<tr>
<td>L7</td>
<td>89</td>
<td>23/33/23/3</td>
<td>57</td>
<td>142</td>
<td>1450/&gt;3000000</td>
</tr>
</tbody>
</table>

In Table 1, the first column shows the number of rules in each set: the scale increases from 55 to about 153. The second column shows the numbers of occurrences of the four concatenation operators in the rule sets, which are separated by “/” in the following order: “.”, “.*”, “{n}”, “[c1c2…ck]*-like”, “[c1c2…ck]{m,n}-like”. As we can see, the most frequent operators are “[c1c2…ck]*-like” operators in Snort rule set. Especially, these operators occur nearly 3 times in each rule of the Backdoor rule set. The first three operators occur more frequently in L7 rule set. The third column shows the numbers of the rules in each rule set with at least one concatenation operator. As we can see, the occurrences of concatenation operators are very frequent in the practical rules, especially in the two rules set from Snort, where such kinds of rules account for more than 90% of the whole rules. The last column shows the numbers of DFA and NFA states directly constructed with respect to these rule sets. The number of DFA states is represented by a range in that we abandon one construction of automaton if the number of states in this construction exceeds 3,000,000. As we can see, the memory requirement of DFA is enormous if the expressions contain special concatenation operators. Note that although the Web-misc rule set is relatively smaller, yet the number of NFA states constructed from it is the biggest. This can be explained in the following way: Web-misc rules contain a lot of occurrences of the fourth type concatenation operator (“[c1c2…ck]n”), and the values of n are larger (with the maximum up to 1024), which leads to a lot of NFA states.

4.2 Experimental Results

In this subsection, we present the memory requirements and processing speeds of our method on experimental rule sets.

4.2.1 Memory requirements

<table>
<thead>
<tr>
<th>Method</th>
<th>#NFA states</th>
<th>#DFA states</th>
<th>#Total states</th>
</tr>
</thead>
<tbody>
<tr>
<td>G-V</td>
<td>7952</td>
<td>763</td>
<td>8715</td>
</tr>
<tr>
<td>Hybrid-FA</td>
<td>6714</td>
<td>29343</td>
<td>36057</td>
</tr>
</tbody>
</table>
Table 3. Memory requirements on Backdoor rule set

<table>
<thead>
<tr>
<th>Method</th>
<th>#NFA states</th>
<th>#DFA states</th>
<th>#Total states</th>
</tr>
</thead>
<tbody>
<tr>
<td>G-V</td>
<td>4725</td>
<td>2993</td>
<td>7718</td>
</tr>
<tr>
<td>Hybrid-FA</td>
<td>2539</td>
<td>30936</td>
<td>33475</td>
</tr>
</tbody>
</table>

Table 4. Memory requirements on L7 rule set

<table>
<thead>
<tr>
<th>Method</th>
<th>#NFA states</th>
<th>#DFA states</th>
<th>#Total states</th>
</tr>
</thead>
<tbody>
<tr>
<td>G-V</td>
<td>1842</td>
<td>2603</td>
<td>4445</td>
</tr>
<tr>
<td>Hybrid-FA</td>
<td>1091</td>
<td>8249</td>
<td>9340</td>
</tr>
</tbody>
</table>

Table 2, 3, 4 detail the memory requirements on the three rule sets of our proposal (G-V) and Hybrid-FA respectively. For our proposal, we give the number of DFA states, the number of all NFAs’ states and the total number of all states. For Hybrid-FA, we give the number of Head-DFA states, the total number of all Tail-NFA states and the total number of states in the whole Hybrid-FA. From the comparison, the following conclusions can be made:

(1) The memory requirement of a finite automaton is mainly depend on the characteristics of rule set, especially the kinds and frequency of special operators. For example, the Web-misc rule set has the least number of rules, while its corresponding FA has the most number of states. Compare to Table 1, we can observe that the number of state in a combined NFA is less than the sum of the number of state in the NFA for each regular expression, because those individual NFAs cannot share the common parts.

(2) The memory requirements of our proposal are less than that of the Hybrid-FA. For the L7 rule set, the requirements of our proposal is about 50% of that of Hybrid-FA and it becomes only about 25% for the two Snort rule sets. That is because Hybrid-FA only interrupts the subset construction when accounting a “special operator”, so it will make much more DFA states if the operators are near the end of regular expression.

4.2.2 Matching speed

The test data used in the experiment are generated by Regex[25]. Regex can generate test data with specified characteristics for regular expression rules. In order to make a comprehensive comparison, we generate 5 trace data for each rule set with different \( p_{seed} \) (\( p_{seed}=0.1, 0.35, 0.55, 0.75, 0.95 \)), where \( p_{seed} \) indicates the probability of accessing the deeper states relative to the current state during transversal. The probability becomes larger as \( p_{seed} \) increases.

The matching speeds of our proposal and Hybrid-FA are shown in Figure 2. As we can see, our proposal (G-V) has a higher speed on all three rule-sets, especially on the Web-misc and Backdoor rule set. This can be explained by the following facts: first, the operators in these two rule sets are more complex. The Hybrid-FA can build the Head-DFA using a short prefix of each rule, so the slow tail-NFAs are accessed frequently during matching. Second, the DFA used in our proposal is much smaller than the Head-DFA of Hybrid-FA and hence has fewer cache misses.
Our proposal shows lower speed on the L7 rule set than on Backdoor and Web-misc rule sets under the same conditions. That is because the rules in L7 are used to identify the applications, while the rules in Snort are used to detect attacks. The hit ratio on L7 rules will be higher than that on Snort rules, which indicates that matching on L7 rules will have to call more verification procedures and hence result in a relatively slower matching speed. Therefore our proposal is more suitable for network security inspections where only few packets can hit rules.

5. Conclusions

This paper analyzed the characteristics of rule sets and matching in network security inspection, and proposed a regular expression matching method which has two steps: guess and verification. The guesses are finished by a single DFA while the verifications are finished by a group of NFAs. However, we do not compile the DFA and NFAs together into a hybrid structure. Instead we just build a DFA for the guess and select the NFAs corresponding to the hit regular expressions for verification. The results show that this proposal can provide a high throughout with a moderate memory requirement and it is more suitable for network security inspections.

6. Acknowledgment

This work is partially supported by the National Key Technology R&D Program (NO.2012BAH37B02, NO.2012BAH42B02), the National High-Tech Development 863 Program of China (NO. 2012AA03001), and the 242 Program (NO. 2012A54, 2012A51, 2012A62).

7. References

An Efficient Regular Expression Matching Method Based on Guess and Verification
Shuzhuang Zhang, Chen Xue, Zhigang Wu


