Towards Fast Regular Expression Matching in Practice

Kai Wang*† and Jun Li†‡
*Department of Automation, Tsinghua University, Beijing, 10084, China
†Research Institute of Information Technology, Tsinghua University, Beijing, 10084, China
‡Tsinghua National Lab for Information Science and Technology, Beijing, 10084, China
wang-kai09@mails.tsinghua.edu.cn, junl@tsinghua.edu.cn

ABSTRACT

Regular expression matching is popular in today's network devices with deep inspection function, but due to lack of algorithmic scalability, it is still the performance bottleneck in practical network processing. To address this problem, our method first partition regular expression patterns into simple segments to avoid state explosion, and then compile these segments into a compact data structure to achieve fast matching. Preliminary experiments illustrate that our matching engine scales linearly with the size of the real-world pattern set, and outperforms state-of-the-art solutions.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: General—*Security and protection (e.g., firewalls)*

Keywords

Regular expression matching; deep inspection; DFA

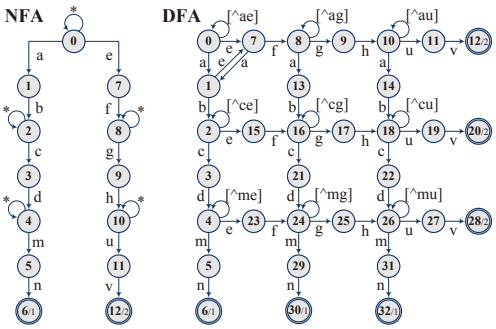
1. INTRODUCTION

Nowadays, regular expression matching is widely used in content-aware network processing, such as intrusion detection/prevention, antivirus, application identification, content filtering and so on. Given a set of regular expressions specifying the patterns of interest, the matching engine is to inspect the payload of every packet in each network flow, and find out all the patterns that occurred.

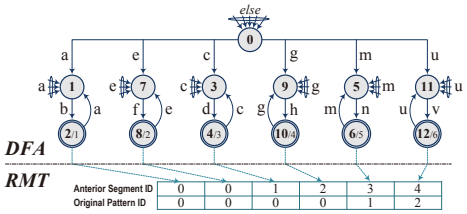
Regular expression matching is performed via either nondeterministic finite automata (NFA) or deterministic finite automata (DFA). Although NFA has a compact data structure, it demands high memory bandwidth and thus runs very slowly. DFA is always fast, on the contrary, but it often exhibits an exponentially growing size (i.e., state explosion) compared to NFA, leading to prohibitive memory usage.

To achieve fast regular expression matching in practice, prior work mainly focuses on deflating DFA by a variety of compression techniques [3–5]. However, their design comes at the cost of manyfold decreased matching speed. In this work, we present a fundamentally different solution, to address the entire problem through pattern partition.

SIGCOMM'13, August 12–16, 2013, Hong Kong, China.
ACM 978-1-4503-2056-6/13/08.



(a) NFA vs. DFA (some edges omitted for clarity)



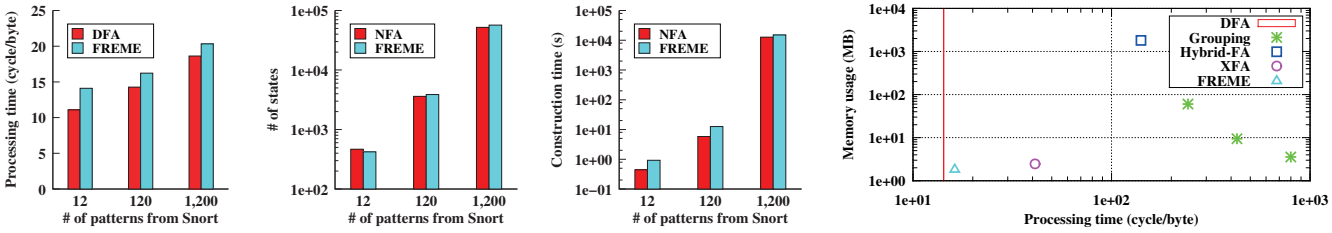
(b) FREME (some edges simplified for clarity)

Figure 1: The matching engines for the pattern set of <.*ab.*cd.*mn> and <.*ef.*gh.*uv>

2. THE MOTIVATION

It is known that, regular expression syntaxes containing character set with Kleene closure (e.g. .*) or counting constraint (e.g. .{n}), called *overlapping factors* (OFs) in this work, can cause corresponding DFA to explode [3–5]. For example, one can find that both the NFA and the DFA for the pattern set of <.*abcdmn> and <.*efghuv> have 13 states. In contrast, the NFA for the pattern set of <.*ab.*cd.*mn> and <.*ef.*gh.*uv> also has 13 states, but corresponding DFA has 33 states (see Figure 1a). The explosion reason can be explained as the possible semantic overlapping between the OF and the patterns [3, 4]. In the example above, the entire pattern <.*ef.*gh.*uv> could be fully covered by both two middle OFs <.*> of the pattern <.*ab.*cd.*mn> in regular expression semantics, and thus the states to represent <.*ef.*gh.*uv> (states 7-12) are duplicated (states 15-20 and states 23-28) at the places where the two middle <.*> of <.*ab.*cd.*mn> exist in DFA (at state 2 and state 4 in the DFA of Figure 1a).

Accordingly, the state explosion problem of DFA can be mitigated or solved, if all the patterns have no middle OFs (e.g. the above-mentioned <.*abcdmn> and <.*efghuv>). To this purpose, we propose the idea that first partition all the patterns of the given pattern set into simple segments, so that the resulting segments have no middle OFs, then



(a) Processing time (b) Number of states (c) Construction time (d) Time vs. space for 120 patterns from Snort

Figure 2: Performance comparison for DFA, NFA, Grouping [5], Hybrid-FA [3], XFA [4] and FREME

use all these segments to construct the fast regular expression matching engine (FREME) which can match original patterns without causing false positive and false negative.

3. PATTERN PARTITION BASED DESIGN

As the DFA part of Figure 1b implies, the example patterns <.*ab.*cd.*mn> and <.*ef.*gh.*uv> are partitioned, respectively, into <.*ab>, <^.*cd>, <^.*mn> and <.*ef>, <^.*gh>, <^.*uv>. Note that for each pattern, all the partitioned segments except the first one must be anchored, to guarantee the concatenate matching for adjacent segments. Particularly, <^.*> is equivalent to <.*> in syntax. Figure 1b shows that the DFA built for the resulting segments only has 13 states (like the NFA in Figure 1a).

In addition, the relations of each pair of adjacent segments (in each pattern) are recorded during pattern partition, meanwhile each segment is assigned a distinct segment ID. In Figure 1b, the segment IDs of <.*ab>, <.*cd> and <.*mn> are 1, 3 and 5 (see match state 2, 4 and 6), respectively. And the relation mapping table (RMT) part indicates the relations among the segments: the anterior segment IDs of segment 3 and 5 are 1 and 3 (i.e., segment 3/5 is at the back of segment 1/3), respectively. Besides, the original pattern ID of segment 5 identifies corresponding pattern 1, that is <.*ab.*cd.*mn>. Note that 0 means no corresponding ID in RMT. It is the ID of the matched segment that is used to index corresponding table entry in RMT.

The DFA part and the RMT part together make up the final matching engine FREME. The size of DFA part is $N_{State} \cdot 256 \cdot log_2 N_{State}$ bits. The size of RMT part is $N_{OF} \cdot 2 \cdot log_2 N_{OF}$ bits, where $N_{OF} < N_{State}$. One can find that the size of FREME mainly depends on the number of states in DFA part. In FREME, an original pattern is matched only when all its segments are processed and matched in sequence. This is guaranteed by activating a single state unit to keep track of current state number (based on a $log_2 N_{State}$-bit value) and the IDs of previously matched segments (based on a N_{OF}-bit bit vector) in the process of matching.

4. MATCHING TO INPUT

Take Figure 1b as example and suppose the content to inspect is "abmncdmn". The initial state unit consists of state number 0 and bit vector $0000b$. For input "ab", segment 1 is matched, then its ID 1 is used to index the 1st entry in RMT, where no anterior segment is required, thus this match is valid, and the 1st bit of bit vector is set (i.e., $0001b$). Next for input "mn", segment 5 is matched, the indexed 5th entry in RMT shows that segment 3 is expected to be previously matched, but the 3rd bit in bit vector is not set, thus this match is invalid and ignored. Next for input "cd", segment 3 is matched, and its anterior segment 1 in MRT is previously matched according to bit vector, thus this match is valid, and the 3rd bit of bit vector is set (i.e., $0101b$). Next for another input "mn", segment 5 is matched correctly this time because segment 3 has been matched before. Besides, the match of pattern 1 (i.e., <.*ab.*cd.*mn>) is found, because the original pattern ID of segment 5 is 1.

Note that in Figure 1b, all the match states (e.g. state 2) have the same next state as state 0 for identical input character. This guarantees that the matching of arbitrary segment must start from state 0 after the previous one is correctly matched. If an exception is meet in practice (although rarely), a new state unit with state number 0 and the ID of the segment just matched must be activated, and parallelly processed with the original one after then. In a lot of prior work, it is found that most state transitions in DFA lead back to state 0 or its neighbors. Consequently, even if the exception occurs, the original and new activated state units have a great chance of transiting to the same state in a short time, then they could be merged merely by the bitwise OR of their bit vectors. This means the number of state units in FREME can always converge to one in any case.

5. EXPERIMENTS AND CONCLUSION

Experiments are conducted on an Intel Xeon E5504 platform (CPU: 2.0GHz, L1 Cache: 32KB, L2 Cache: 4MB, Memory: 8GB), using real-world regular expression patterns (with hundreds of explosive OFs) collected from Snort IDS [1] and the published trace from MIT Lincoln Lab [2]. The results shown in Figure 2 verify that FREME is practically fast (like DFA) and scalable (like NFA), and outperforms state-of-the-art solutions. In summary, benefiting from pattern partition, our solution can defuse state explosion, meanwhile utilize DFA to achieve fast matching. We expect more efficient design with heuristic partition and grouping.

6. REFERENCES

[1] Snort, `http://www.snort.org/`.

[2] DARPA Intrusion Detection Data Sets, `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html`.

[3] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT Conference*, pages 1–12. ACM, 2007.

[4] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proceedings of the 2008 ACM SIGCOMM Conference*, pages 207–218. ACM, 2008.

[5] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2nd ACM/IEEE ANCS Conference*, pages 93–102. ACM, 2006.