Journal of Network and Computer Applications 55 (2015) 154–169

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

FREME: A pattern partition based engine for fast and scalable regular expression matching in practice

Kai Wang [a,*], Jun Li [a,b]

[a] Research Institute of Information Technology (RIIT), Tsinghua University, Beijing, China
[b] Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing, China

ARTICLE INFO

Article history:
Received 17 June 2014
Received in revised form
28 March 2015
Accepted 12 May 2015
Available online 4 June 2015

Keywords:
Regular expression matching
Deep inspection
Pattern partition
Deterministic finite automata (DFA)

ABSTRACT

Regular expression matching has been widely used in modern content-aware network devices, where the content of interest (i.e., patterns) is often specified by regular expressions. Due to the ever-increasing number of patterns, implementing fast and scalable regular expression matching becomes a big challenge. Practical solutions rely mainly on a variety of deterministic finite automata (DFA) deflation techniques, but cannot guarantee both high speed and linear scalability simultaneously.

To fully address the problem, in this paper, we present a fundamentally different design: (1) following principles to partition all regular expression patterns (in the given pattern set) into segments, so that state explosion never occurs when converting these segments to DFA, and (2) compiling the resulting segments and their syntagmatic relations, respectively, into DFA and relation mapping table (RMT), which together make up the final matching engine named FREME.

Despite the pattern partition, FREME does not sacrifice any matching correctness with the aid of RMT. Evaluation based on real-world pattern sets (open source and commercial) shows that FREME scales linearly with the size of pattern set, meanwhile keeps fast matching based on nonexplosive DFA. In contrast, FREME outperforms state-of-the-art matching engines up to two orders of magnitude.

1. Introduction

In TCP/IP network, given a set of patterns specified by regular expressions, the regular expression matching problem is to inspect the payload of every packet in each network flow, and find out all the patterns that occurred.

Modern content-aware network devices increasingly rely on regular expression matching to search the content of interest (i.e., patterns) in network flows. Particularly, by defining the patterns of application-layer attacks (e.g. SQL injection attack) or application-layer protocols (e.g. P2P filesharing protocol) through regular expressions (Sommer and Paxson, 2003), the regular expression matching component can provide the function of intrusion detection/prevention in network security devices (Roesch, 1999; www.snort.org; Paxson, 1999; http://www.bro.org; http://www.clamav.net), or protocol identification in network monitoring devices (http://l7-filter.clearfoundation.com).

However, due to lack of algorithmic scalability, regular expression matching is still a performance bottleneck in practical network processing. Even worse, with the rapid development of Internet applications, the number of developed regular expression patterns is increasing sharply. One can find that, in the popular Snort intrusion detection system (IDS) (Roesch, 1999; www.snort.org), the number of rules using regular expression patterns has evolved from 1131 (February 2006) to 13,605 (February 2014) over the past eight years. As a result, it is a big challenge at present to achieve fast and scalable regular expression matching.

1.1. Background

In theory, regular expression matching is performed using either nondeterministic finite automata (NFA) or deterministic finite automata (DFA) (Hopcroft et al., 2006). NFA is built from regular expression patterns by following McNaughton–Yamada (1960) construction or Thompson (1968) construction first and then NFA reduction (Hopcroft et al., 2006), and DFA is further constructed based on NFA by using subset construction (Hopcroft et al., 2006) first and then DFA minimization (Hopcroft, 1971).

NFA has a compact data structure, and its size (i.e., number of states) grows linearly with the size (i.e., number of patterns) of pattern set. This makes NFA an ideal solution to perform scalable regular expression matching in terms of memory usage. However, NFA demands high memory bandwidth because (1) it needs to access many concurrent active states for each input character, to

* Corresponding author at: Research Institute of Information Technology (RIIT), Tsinghua University, Beijing, China.
E-mail addresses: wang-kai09@mails.tsinghua.edu.cn (K. Wang), junl@tsinghua.edu.cn (J. Li).

http://dx.doi.org/10.1016/j.jnca.2015.05.012
1084-8045/

keep track of all candidate state transition paths leading to the match states, and (2) it must compare the input character with all fan-out transitions in each currently active state, to determine all next states to be activated, so that it suffers very slow matching speed when memory bandwidth is insufficient in practical devices. In worst case, the processing complexity of NFA is $O(N_P \cdot L_P^2)$ (where N_P is the number of patterns, L_P is the average length of patterns) (Yu et al., 2006). Thus, NFA does not suit high-bandwidth network environments.

On the contrary, DFA has a homogeneous data structure,[1] and only requires precisely one state transition lookup per input character (i.e., $O(1)$ processing complexity). This merit makes DFA always fast and the prior choice for practical network processing. Unfortunately, along with the increase of the size of pattern set, DFA often exhibits an exponentially growing size, and its worst-case space cost is $O(|\Sigma|^{N_P \cdot L_P})$ (where $|\Sigma|$ is the size of alphabet) (Yu et al., 2006). This is the well-known state explosion problem of DFA. Consequently, DFA tends to require prohibitive memory overhead, so that it is hard to be widely used.

1.2. Prior art

In order to implement high-performance regular expression matching and support large-scale pattern sets in practice, prior work mainly focuses on deflating the DFA by a variety of state compression techniques (Yu et al., 2006; Rohrer et al., 2009; Fu et al., 2014; Becchi and Crowley, 2007; Liu and Wu, 2013; Kumar et al., 2007; Becchi and Crowley, 2008; Smith et al., 2008a, 2008b; Yang and Prasanna, 2011).

According to the prior work, it is the semantic overlapping (or overlapping for short) of regular expression patterns that leads DFA to explode. Further, the inter-pattern overlapping can cause that given two or more patterns, the size of the composite DFA built from them together increases far more than the total size of the DFAs built from each of them independently (Becchi and Crowley, 2007; Smith et al., 2008b). In addition, the intra-pattern overlapping can make the size of the DFA merely built from one single pattern expand exponentially (Becchi and Crowley, 2007).

Yu et al. (2006) first propose two solutions to deflate DFA. One, called pattern grouping, uses greedy heuristics to divide the given patterns into fewest groups where all the patterns have limitative inter-pattern overlapping, and constructs independent DFA for each group of patterns. This can efficiently trade decreased speed by a factor of tens (due to multiple DFAs running in parallel) for reduced memory by several orders of magnitude (due to mitigated inter-pattern overlapping inside each group). However, the number of required groups to avoid semantic overlapping largely depends on the scale and the complexity of the given pattern set, making this solution unscalable in performance. The other, called pattern rewrite, is against the intra-pattern overlapping. Unfortunately, pattern rewrite not only breaks the semantic equivalence, but also fails to treat a number of patterns ignored by the authors.[2] Rohrer et al. (2009) convert the grouping problem into an energy minimization problem, and employ heuristic optimization algorithm to improve the spatial and temporal performance tradeoff. Fu et al. (2014) further improve the estimation method of state explosion among different patterns, and present the grouping scheme based on intelligent evolutionary algorithms to achieve less groups on the premise of the same deflation effect. Nevertheless, the aforementioned problems still exist.

Becchi and Crowley (2007) and http://regex.wustl.edu present the design of Hybrid Finite Automata (Hybrid-FA) afterwards. Hybrid-FA aims to obtain high speed by transforming partial NFA states (whose NFA-to-DFA conversion will not cause a great increase in DFA states) to a single head-DFA, and prevent the occurrence of state explosion by retaining other NFA states as a set of tail-NFAs. Such a design is effective when the process concentrates on the head-DFA, but will incur great performance reduction when the process is enslaved to the tail-NFAs. (This happens if a border state between head-DFA and tail-NFAs is activated.) Although the authors emphasize that the tail-NFAs could be further converted to tail-DFAs, and give the detailed demonstration as well, it can be proven that this conversion is infeasible if the pattern segments corresponding to a single tail-NFA still have semantic overlapping.[3] Liu and Wu (2013) designed the dual finite automata (dual FA), which is similar to Hybrid-FA in fact, but uses a linear finite automata (LFA) to represent the NFA states causing DFA state explosion and an extended DFA (EDFA) to represent the remainder. The dual FA has the same problem as Hybrid-FA.

Kumar et al. (2007) propose an alternative representation of patterns without incurring state explosion, named History based Finite Automata (H-FA). In comparison with DFA, H-FA brings in auxiliary flag/counter variables to replace the essential but duplicated states caused by inter-pattern overlapping, substitutes the variable calculation for the state transition of the replaced states, and uses variable values as well as an active state to track the matching history. Further, Becchi and Crowley (2008) extend the design of H-FA to Counting-FA, so as to support patterns with intra-pattern overlapping. However, as a key problem, the design of both H-FA and Counting-FA is ad-hoc for various patterns and has no uniform model for systematic construction, therefore it is limited to transform a large-scale pattern set to a general H-FA or Counting-FA with manageable number of variables and states.

Smith et al. (2008a,b) consider that incorporating state variables in DFA is a right direction, and propose a formal model called Extended Finite Automata (XFA). Compared with H-FA and Counting-FA, XFA has specific mathematic definition, and its auxiliary variables as well as manipulating instructions have unified coding, thus it can be systematically constructed and performed. The authors argue that XFA achieves good performance in terms of memory space versus run time tradeoff. However, it is not proven that XFA can deal with all kinds of intra-pattern overlapping. (The authors themselves also mention this problem in their original work Smith et al., 2008a.)[4] Furthermore, in terms of practicability, constructing XFA does need a prohibitive duration for large-scale pattern sets.[5] Although Pasetto et al. (2010) and Huang et al. (2013) respectively present the DotStar algorithm and TCAM-based CFA algorithm to further compress XFA, these methods do not solve the above problems XFA has.

Yang and Prasanna (2011) introduce the Semi-deterministic Finite Automata (SFA) recently, to avoid state explosion from a theoretical perspective. When constructing SFA from NFA, a "state grouping" heuristics is used to cluster all the NFA states into

[1] A data structure is said to be homogeneous if all of its elements are of the same data type.

[2] For example, pattern rewrite does not apply to a more general pattern like <auth\s[^\n]{100}suffix>, although it can treat the exemplified pattern <auth\s[^\n]{100}> as the authors argue in their work (Yu et al., 2006).

[3] As an example, for the Hybrid-FA built from two real-world patterns <filename=.*\x2e\x2e(\x2f|\x5c)> and <php.*\x3f[^\n]{256}> (www.snort.org), its two tail-NFAs correspond to the two pattern segments <\x2e\x2e(\x2f|\x5c)> and <\x3f[^\n]{256}>, respectively, where the latter still has intra-pattern overlapping and cannot be converted to tail-DFA.

[4] Note that the pattern <\ncmd[^\n]{200}> exemplified in their further work (Smith et al., 2008b) begins with a character <\n>, which is exclusive in the character set <[^\n]> of the pattern segment <[^\n]{200}>, and such a pattern has no intra-pattern overlapping (both its NFA and DFA have only 205 states, while the DFA of <\rcmd[^\n]{200}> has over 5×10^{28} states).

[5] According to the data provided by the authors (Smith et al., 2008a), one can figure out that the construction time of the given XFA is between 2.5 days and 25 days, even without counting the combination time.

fewest subsets where corresponding NFA states have no pair-wise conflicts (i.e., overlapping between state transitions), and SFA consists of the constituent DFAs (c-DFAs) converted from each subset of NFA states. SFA is a good theory model to find a best tradeoff between NFA and DFA, but the algorithm used to judge state conflicts is so complex (i.e., $O(N_P^2 \cdot L_P^2 \cdot |\Sigma|^{L_P})$ processing complexity) that using SFA to implement large-scale regular expression matching is infeasible in practice.

1.3. Our contribution

As a whole, although considerable contributions have been made in prior work, the proposed solutions still suffer unscalable algorithms or data structures, and thus cannot tackle the challenge comprehensively.

Recalling the entire problem, one can find that the root cause of state explosion lies in the syntaxes of regular expression patterns, rather than DFA itself, because DFA built from exact string patterns does not have this problem at all (Yu et al., 2006). This means that DFA deflation is not the best research orientation. Fundamentally different from prior work, in this paper, we focus on directly removing the semantic overlapping from patterns themselves, to lay a foundation for constructing nonexplosive DFA.

To this end, this paper presents heuristic pattern partition briefed in Wang and Li (2013), to split all the patterns of the given pattern set into pattern segments without semantic overlapping. Then, we propose the fast regular expression matching engine (FREME) make up of DFA and relation mapping table (RMT), based on the resulting segments as well as their syntagmatic relations.

The key contributions of this paper are fourfold:

- The principles and heuristics of pattern partition are presented, to provide the foundation of linear scalability for FREME.
- A DFA-based but compact data structure as the prototype of FREME is proposed on the premise of pattern partition, to gain fast regular expression matching in practice, and the mathematical description of FREME is formalized.
- A matching algorithm as the core of FREME is designed, to guarantee no false positive or false negative compared to original patterns.
- A set of optimization schemes based on the nature of FREME is introduced, to further improve the performance of FREME.

The evaluation for our solution uses large-scale sets of regular expression patterns obtained from real-world Snort rules (www.snort.org), Bro rules (http://www.bro.org), ClamAV rules (http://www.clamav.net), L7-filter rules (http://l7-filter.clearfoundation.com) and two commercial application-aware products. Experimental results show that FREME indeed approaches or even excels NFA in terms of both small memory usage and short construction time, meanwhile keeps fast matching speed comparable with DFA. Besides, FREME outperforms prior art (Yu et al., 2006; Becchi and Crowley, 2007; Smith et al., 2008b) up to two orders of magnitude in both spatial performance (i.e., smaller memory usage) and temporal performance (i.e., faster processing speed and shorter construction time).

1.4. Roadmap

The remainder of this paper is organized as follows: we discuss the motivation of our work and present the detailed pattern partition in Section 2. The design of matching engine and matching procedure is proposed in Section 3. Section 4 further introduces the optimization schemes for our solution. After the experimental results shown in Section 5, we conclude this paper in Section 6.

2. Heuristic pattern partition

In this section, we will describe the motivation and the approach of pattern partition, to solve the state explosion problem existing in patterns themselves.

2.1. Semantic overlapping

First of all, inorder to facilitate the textual description of this paper, it is necessary to formulate the regular expression syntaxes below using the prescribed terms:

- *Character set* (ϕ): The union set of characters,including the proper subset of the alphabet Σ (with form as $[c_1c_2\cdots c_k]$ or [\widehat$c_1c_2\cdots c_k$],where $c_k \in \Sigma$),and the universal set named wildcard (with form as .).
- *Boundless counting constraint*: The combination of character set and Kleene closure,namely the character set repeating unlimited times (with form as ϕ^*). In particular,the combination of wildcard and Kleene closure is called *dot-star* (i.e.,.*).
- *Constant counting constraint*: The combination of character set and fixed n times constraint (with form as $\phi\{n\}$,where $n \in Z^+$).
- *Infimum counting constraint*: The combination of character set and at least n times constraint (with form as $\phi\{n,\}$,where $n \in Z^+$).
- *Supremum counting constraint*: The combination of character set and finite m-to-n times constraint (with form as $\phi\{m,n\}$, where $m \in Z^*$, $n \in Z^+$, $m < n$).
- *Overlapping factor* (OF): The general term of ϕ^*, $\phi\{n\}$, $\phi\{n,\}$ and $\phi\{m,n\}$.

Compared with exact string patterns, regular expression patterns introduce two basic operators Kleene closure (*) and alternation (|), in addition to the identical operator concatenation (·). Therefore, they provide an expressive power that far exceeds exact string patterns. For example, c_1 * stands for a string with zero or more c_1 characters, and $[c_1c_2]$, i.e., $(c_1|c_2)$, corresponds to an optional set of characters c_1 and c_2. Based on the two additional operators, regular expression patterns can express ϕ^*, $\phi\{n\}$, $\phi\{n,\}$, and $\phi\{m,n\}$. In this paper, we refer to these four types of syntaxes as OFs.

As a lot of previous work summarizes, it must be the OFs that cause the DFA of corresponding regular expression patterns to explode (Yu et al., 2006; Becchi and Crowley, 2007; Smith et al., 2008b). On the contrary, due to no OF, exact string patterns never suffer such a state explosion problem.

Take Patterns 1 and 2 listed in Table 1 for example. The NFA for the pattern set of Patterns 1 and 2 has 19 states, but corresponding DFA expands to 36 states. In contrast, one can find that both the NFA and the DFA for the pattern set of string patterns <file=\.exe> and <path=\/bin> have 19 states. It is not hard to see that the only difference between these two pattern sets is the additional OFs <[^\r\n]*> existing in Patterns 1 and 2. Likewise, Table 1 shows the NFA for the pattern set of the single Pattern 3 only has 10 states, but corresponding DFA has up to 26 states.

The root cause of state explosion lies in the possible semantic overlapping between the OF and the pattern where the OF belongs (i.e., intra-pattern overlapping) or other patterns (i.e., inter-pattern overlapping) (Becchi and Crowley, 2007; Smith et al., 2008b). In this paper, we define semantic overlapping as follows.

Definition 1. Semantic overlapping means for given pattern set S_P, $\exists$ overlapping semantics Λ, the DFA for S_P must use additional state copies to represent Λ in comparison with corresponding NFA, otherwise, it can cause false negative in pattern matching.

Table 1
The analysis of semantic overlapping for the example set of regular expression patterns.

ID	Pattern	# of States				Overlapping semantics	
		NFA	DFA	NFA	DFA	Intra-pattern examples	Inter-pattern examples
1	file=[^\r\n]*\.exe	10	10	19	36	\	file=xxxpath=xxx/binxxx.exe
2	path=[^\r\n]*\/bin	10	10			\	path=xxxfile=xxx.exexxx/bin
3	cmd=[^\n]{5}	10	26	30	154	cmd=xcmd=xxxxx	cmd=www.xxx.org/xxx.shtml
4	www.*org\/[^\s]{3,5}\.shtml	21	48			www.xxx.org/www.shtml.org/xxx.shtml	www.xxx.org/cmd=x.shtml

A simple pattern example to illustrate semantic overlapping is <abcd[^d]*efg>, where the OF <[^d]*> can cause semantic overlapping coupled with the prefix <abc>, because this pattern must be able to recognize two types of semantics, i.e., whether or not the prefix <abc> occurs in the strings which match the OF. For the regular semantics Λ_1, it could be the string "abcdxxxxxxxefg" (where 'x' is any character except for 'd'), while for the overlapping semantics Λ_2, it could be the string "abcdxxxabcdefg". Hence, if Λ_2 cannot be represented together with Λ_1, then the valid match of "abcdefg" within Λ_2 will be missed.

To keep track of the overlapping semantics, NFA adopts non-deterministic transitions for each state, and thus can activate multiple states simultaneously during runtime matching. Unlike NFA, DFA employs extra duplicated states to represent all overlapping semantics beforehand, where each state records one milestone to match possible semantics to keep the deterministic one state transition per input character. For the above-mentioned example, the NFA and DFA have 8 and 11 states respectively, and the additional 3 states in the DFA are just added to track the overlapping semantics Λ_2, i.e., the prefix <[^d]*> occurs when the OF <abc> is matched. This indicates that semantic overlapping is the root of DFA state inflation. In the worst case, it will cause corresponding DFA to explode.

A necessary but insufficient condition for the occurrence of semantic overlapping is that $\exists$ *OF* and pattern *P* in pattern set S_P, *OF* intersects with *P* in regular expression semantics.

Consider pattern (segment) *X* and pattern (segment) *Y*, where *X*'s *k*th $(0 < k \le |X|)$ character (set) is ϕ_{Xk} $(\phi_{Xk} \subseteq \Sigma)$, and *Y*'s *k*th $(0 < k \le |Y|)$ character (set) is ϕ_{Yk} $(\phi_{Yk} \subseteq \Sigma)$. We define that *X* intersects with *Y*, or $X \vDash Y$ as below.

Definition 2. $\exists i\ (0 < i \le \min(|X|, |Y|))$, $\forall j \le i$, $\exists$ character $p \in \phi_{Xj}$ where $p \in \phi_{Yj}$, meanwhile $\exists$character $q \in \phi_{Xi}$ where $q \notin \phi_{Yi}$, then $X \vDash Y$.

In the example above, the OF <[^\r\n]*> of Pattern 1 can cover (i.e., intersect with) the entire Pattern 2 in regular expression semantics (see inter-pattern examples of Pattern 1 in Table 1). Therefore, their corresponding DFA requires extra states (i.e., the states with hollow circles in Fig. 1a) to identify all such overlapping semantics. As shown in the DFA of Fig. 1a, the states to represent Pattern 2 (states 10–18) are duplicated (states 27–35) at the place where the OF exist (at state 05). In the same way, the OF of Pattern 2 can also lead to the duplication of the states representing Pattern 1. This is just the polynomial inflation caused by inter-pattern overlapping.

Note that in Fig. 1, all the states in NFA and DFA (denoted as circle with number) and only the significant state transitions (denoted as arrow with character) are drawn. Taking the state transition, the state will get to the next-hop state. For example, in Fig. 1a, NFA will transfer to state 02 if the current state is state 01 and the current input character is 'i', and NFA will transfer to next-hop states 0 and 01 simultaneously if the current state is state 0 and the current input character is 'f' (the arrow with mark * means arbitrary character). However, in Fig. 1a, DFA can only transfer to a single state at any time. For example, DFA will transfer to next state 01 when the current state is state 0 and the input character is 'f'.

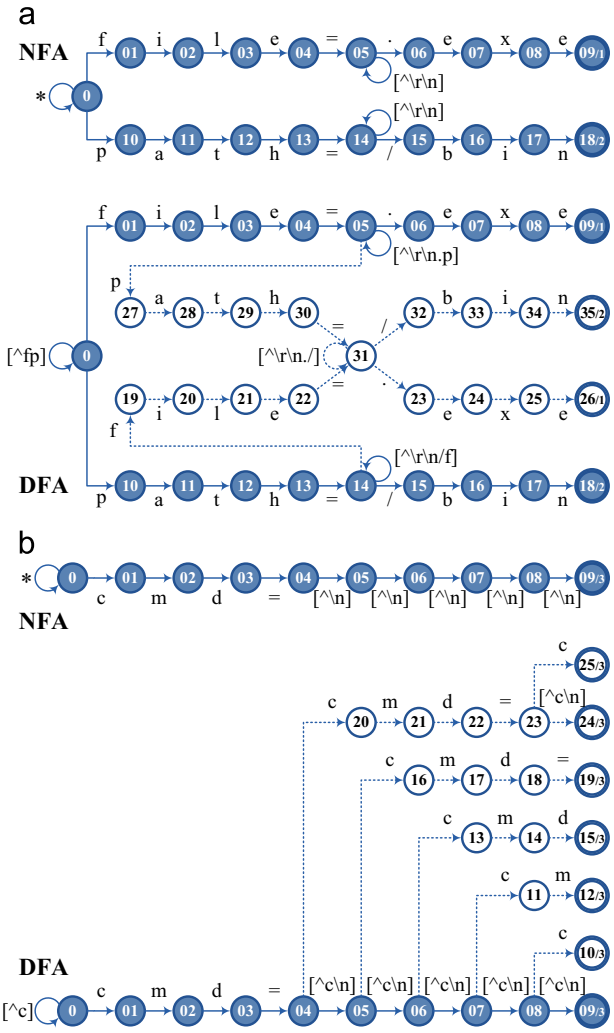

Fig. 1. NFA versus DFA (some state transition edges omitted for clarity): (a) for the pattern set of Patterns 1 and 2 and (b) for the pattern set of Pattern 3.

For Pattern 3, its OF <[^\n]{5}> can partially intersect with itself, resulting in the duplication of different lengths (e.g. states 16–19, the copies of states 01–04 in Fig. 1b) to represent the overlapping semantics (see intra-pattern examples of Pattern 3 in Table 1). This is the exponential explosion due to intra-pattern overlapping. Note that the OFs leading to intra-pattern overlapping can also cause inter-pattern overlapping to a great extent

(see inter-pattern examples of Pattern 3 in Table 1), and thus are the fatal factors of state explosion.

2.2. Motivating example

According to the previous analysis, the state explosion problem can be resolved if the OFs have no chance to intersect with patterns.

Intuitively, if Patterns 1 and 2 are respectively rewritten into <file=[^p\r\n]*\.exe> and <path=[^f\r\n]*/bin>, then the two modified OFs are impossible to intersect with the opposite pattern due to the exclusion of the heading character, and the DFA for the pattern set of the two modified patterns will have only 19 states, equal to corresponding NFA. Besides, if Pattern 3 is anchored at the beginning, that is <^cmd=[^\n]{5}>, then its OF cannot produce intra-pattern overlapping as well, because in this case, the prefix <^cmd=> of the OF can only occur at the start (i.e., never intersected with the position of the OF) of the string the pattern is applied to, and corresponding DFA will not explode.

One can find that the two rewriting ways both come at the expense of the sacrificed semantics of original patterns. But we are motivated to consider that first remove the semantic overlapping from the given patterns to defuse DFA, and eventually recover the complete semantics of original patterns based on the nonexplosive DFA. To this purpose, we present the idea of pattern partition.

Table 2 reveals the result of pattern partition for Patterns 1–4 listed in Table 1. Note that when a pattern is partitioned into multiple segments, all of them except the first one must be anchored at the beginning via a caret (e.g. <^e=[^\r\n]*\.exe>),[6] to guarantee the concatenate matching for adjacent segments (this will be further explained in Section 3.1). As Table 2 shows, Patterns 1–3 are all split into two segments. However, for a pattern with more OFs whose semantics are more extensive, multi-section partition is required to completely eliminate the possible overlapping stemmed from every OF. Therefore, Pattern 4, which has OF ϕ^* and OF $\phi\{m,n\}$, is necessarily split into four segments listed in Table 2. In particular, the segment <^.*org> is simplified to <org>, because "anchored dot-star" is equivalent to "unanchored" in regular expression syntax.

Further, the resulting segments are divided into two categories (i.e., Groups 1 and 2 in Table 2), where one contains all the first segments (no matter whether to be unanchored) of each pattern (e.g. <www>) as well as all unanchored non-first segments (e.g. <org>), and the other contains all the remaining anchored non-first segments (e.g. <^\/[^\s]{3,5}>). In other words, all these pattern segments could be complied into at most two DFAs.

According to the previously mentioned analysis, semantic overlapping of a pattern could be removed if the pattern is anchored. Fortunately, the non-first segments resulting from the partition must be anchored. Therefore, the OFs should naturally be kept in the anchored non-first segments. However, for arbitrary patterns having OFs, regardless of whether they are anchored or unanchored, they are very likely to cause semantic overlapping with the unanchored patterns (e.g. the segment <^e=[^\r\n]*\.exe> will cause semantic overlapping with the segment <fil>). This is why it is necessary to divide the partition-derived segments into two categories.

As Table 2 implies, the two DFAs corresponding to the segments of partitioned Patterns 1–4 only have 48 states in total, the same as the NFA corresponding to original Patterns 1–4, while the DFA for the pattern set of Patterns 1–4 has up to 1058 states (i.e., a state inflation factor of more than 20). It is thus clear that pattern partition is effective to defuse state explosion.

[6] The introduced caret for the non-first segments limits that its match must be at the start of the string the segment is applied to.

It is nontrivial to notice that the semantic equivalence between the resulting segments and the original patterns is recoverable, and this will be discussed in detail in Section 3.

2.3. Principles for explosion-free partition

Random partition cannot guarantee that the resulting segments for the partitioned patterns (with OFs) correspond to a nonexplosive DFA. For example, if Pattern 1 is partitioned into <file=[^\r\n]*\.> and <^exe>, rather than the segments shown in Table 2, then the first segment <file=[^\r\n]*\.> can still make corresponding DFA inflate when combining with the first segment <pat> of Pattern 2.

In order to achieve explosion-free pattern partition, the following four principles must be obeyed:

1. *Legal syntax*: Partition should first ensure the regular expression syntax of the resulting segments is legal.
2. *Concatenated relation*: Partition should ensure the syntagmatic relation of the adjacent segments from the same patterns is concatenation.
3. *Free of intra-segment overlapping*: Partition should ensure that for each resulting segment, its OF(s) cannot intersect with itself.
4. *Free of inter-segment overlapping*: Partition should ensure that for each resulting segment, its OF(s) cannot intersect with any other segments.

Principle 1 declares the essential requirement for a correct pattern partition without respect to its anti-explosion effect. As a part of the original regular expression patterns, each resulting segment must be also a valid regular expression, otherwise, these segments cannot be compiled into DFA normally.

Principle 2 can facilitate the concatenate matching of the resulting segments to preserve the complete semantics of the original patterns. It requires the partition to be at the position of literal concatenation in a regular expression pattern, meanwhile it cannot affect the underlying priority of concatenation. For example, if the pattern <ab.*cd|ef.*gh> is partitioned into <ab>, <cd|ef> and <gh>, then the priority of the concatenation is just reduced, because the resulting segments actually correspond to a different pattern <ab.*(cd|ef).*gh>. Thus, for such a pattern with lower-priority alternation, it should be written as multiple alternated sub-patterns with identical pattern ID, or else the partition may violate Principle 2. For example, the aforementioned pattern ought to be first written as two sub-patterns, namely <ab.*cd> and <ef.*gh>, and then enforce partition for each sub-pattern.

On the premise of Principles 1 and 2, to defuse state explosion, Principles 3 and 4 must be followed in order. Without loss of generality, Principle 3 requires that the OFs $\phi\{n\}$, $\phi\{n,\}$ and $\phi\{m,n\}$ must be kept in the anchored segments, and the unanchored segments must have none of them, because it is less likely for intra-segment overlapping to happen inside an anchored segment (e.g. the aforementioned <^cmd=[^\n]{5}>). Besides, for the anchored segments, the OFs $\phi\{n\}$, $\phi\{n,\}$ and $\phi\{m,n\}$ should have no prefix with the OF ϕ^*, meanwhile the OFs $\phi\{n,\}$ and $\phi\{m,n\}$ should have no suffix. Otherwise, the occurrence of the prefix or suffix of the OF could be not limited to the fixed position of the string the segment is applied to, so that the OF can still intersect with them (e.g. <^org\/[^\s]{3,5}\.shtml> or <^www.*org\/[^\s]{3,5}>).

Different from intra-segment overlapping, all the OFs can contribute to inter-segment overlapping. One can find that for a segment (no matter whether to be unanchored), as long as it has an OF whose character set ϕ contains most characters of the alphabet Σ, it can easily intersect with any other unanchored segments. Consequently, Principle 4 requires that the unanchored

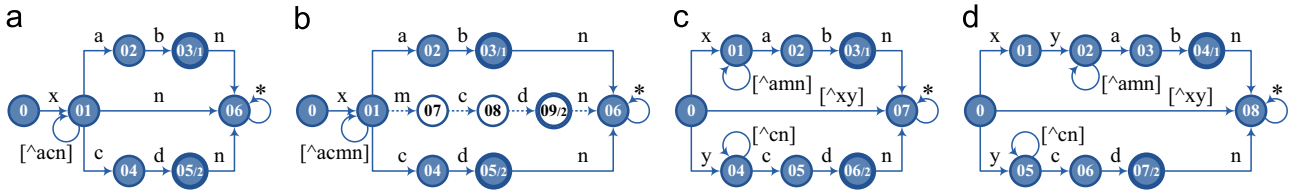

Fig. 2. The impact of the OFs as well as their prefixes on the DFA (only significant state transition edges are kept) for anchored patterns: (a) <^x[^n]*ab> and <^x[^n]*cd>; (b) <^x[^mn]*ab> and <^x[^n]*cd>; (c) <^x[^mn]*ab> and <^y[^n]*cd>; and (d) <^xy[^mn]*ab> and <^y[^n]*cd>.

Table 2
The effect of pattern partition for the example set of regular expression patterns.

ID	Pattern	# of States		Segments								# of states	
		NFA	DFA	ID	Group 1			ID	Group 2			DFA 1	DFA 2
1	file=[^\r\n]*\.exe	48	1058	1	fil			7	^e=[^\r\n]*\.exe			16	32
2	path=[^\r\n]*\/bin			2	pat			8	^h=[^\r\n]*\/bin				
3	cmd=[^\n]{5}			3	cmd			9	^=[^\n]{5}				
4	www.*org\/[^\s]{3,5}\.shtml			4	www	5	org	6	^\/[^\s]{3,5}	10	^\.shtml		

Table 3
The universal partition for pattern types with OFs.

Pattern type	Segment type	
	Group 1	Group 2
(^)STR_1.*STR_2	(^)STR_1, STR_2	
(^)$STR_1\phi^*$ STR_2	(^)STR_{11}	^$STR_{12}\phi^*$ STR_2
(^)$STR_1\phi\{n\}STR_2$	(^)STR_{11}	^$STR_{12}\phi\{n\}STR_2$
(^)$STR_1\phi\{n,\}STR_2$	(^)STR_{11}	^$STR_{12}\phi\{n,\}$, ^STR_2
(^)$STR_1\phi\{m,n\}$ STR_2	(^)STR_{11}	^$STR_{12}\phi\{m,n\}$, ^STR_2

segments cannot be anything but strings, and the anchored segments with OFs cannot be treated together with the unanchored ones. For the anchored segments, although they have less chance to form inter-segment overlapping, partial prefix (at least one character) should be retained for the OF of each segment, and the prefixes of the distinct OFs should be different.

In fact, if the OFs (in each anchored segment) are identical, no matter whether their prefixes are different, inter-pattern overlapping never happen between the anchored segments (see Fig. 2a). However, if the OFs are distinct, the disappearance of overlapping will depend on the difference of their prefixes, because the OFs can still intersect with the suffix of the opposite segment when they have the same prefix (see Fig. 2b), and on the contrary, different prefixes can make overlapping never occur (see Fig. 2c). In detail, the prefixes can be judged as different as long as they have distinct characters in the same position (see Fig. 2d), and the OFs are regarded to be distinct if either their forms or their character sets are different. This is why the anchored non-first segments of Patterns 1/2 and 3 have different prefixes (i.e., <^e=>/<^h=> and <^=>).

Particularly, for OF .*, the partition position should be exactly at the place of the OF (e.g. the aforementioned <org>). This is used to guarantee the existence of endless loop state in DFA (e.g. state 06 in Fig. 2a), and it will be introduced in Section 3.3. Besides, all the first segments of each pattern are treated together with the unanchored ones in Group 1, and thus should have no OFs according to Principles 3 and 4, meanwhile the remainder anchored segments are all in Group 2 and need to satisfy Principles 3 and 4.

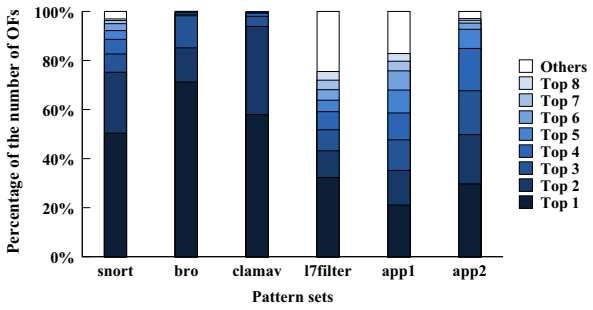

Fig. 3. The statistics of top eight distinct OFs in real-world pattern sets.

In summary, Table 3 lists the universal treatment of the typical pattern types (with the threat of explosion) summarized by Yu et al. (2006). Note that, for Group 1, both STR_{11} and ^STR_{11} belong to it, and for Group 2, ^STR_{12} of distinct OFs must be different.

2.4. Heuristic algorithm

Algorithm 1. Pattern partition.

Input:
Pattern set S_{pat}.
Output:
Segment set S_{seg1} and S_{seg2}; Relation set R_{seg}.
Algorithm:

```
1:  S_of = ∅; id = 1; S_seg1 = ∅; S_seg2 = ∅; R_seg = ∅;
2:  for all pat ∈ S_pat do
3:    // tag the partition position for each OF
4:    for all of ∈ pat.regex do
5:      // for .*, partition is at its head position
6:      if of == .* then
7:        pat.S_tag.push(1, of.head_position());
8:      else
9:        // for else, partition is at the head position of its pre-
          fix
10:       str = of.get_different_prefix(S_of);
11:       S_of.insert(of, ^str);
12:       pat.S_tag.push(2, str.head_position());
13:       // for φ{n,} or φ{m,n}, partition is also at its tail po-
          sition
14:       if of == φ{n,} || of == φ{m,n} then
15:         pat.S_tag.push(2, of.tail_position());
16:       end if
17:     end if
18:   end for
19:   pos = pat.regex.tail_position();
20:   // partition each pattern into segments according to the
      tags, meanwhile set segment ID and record syntagmatic
      relation
21:   while pat.S_tag != ∅ do
22:     tag = pat.S_tag.pop();
23:     seg = pat.regex.get_segment(tag.pos, pos);
24:     if tag.group == 1 then
25:       S_seg1.insert(id, seg);
26:     else
27:       S_seg2.insert(id, ^seg);
28:     end if
29:     pos = tag.pos;
30:     R_seg.push(id, ++id, pat.id);
31:   end while
32:   // handle the (remaining) first segment of each pattern
33:   seg = pat.regex.get_segment(pat.regex.head_position(),
          pos);
34:   S_seg1.insert(id, seg);
35:   R_seg.push(id, ++id, pat.id);
36: end for
```

Based on Principles 1–4, the pseudocode of the partition algorithm can be concluded as Algorithm 1.

In Algorithm 1, steps 3–14 are used to find the partition positions satisfying the explosion-free principles for different OFs according to Table 3. In these steps, steps 4–5 are used to tag the partition position for the special OF .*, steps 6–9 are used to tag the partition positions for other OFs, besides, steps 10–12 aim to tag the additional partition positions for the OFs $\phi\{n,\}$ and $\phi\{m,n\}$.

Particularly, step 7 is to find the shortest prefix *str* for the *of*, so that \widehat*str* is different from all other prefixes recorded in S_{of} whose corresponding *of*s are distinct. A simple and fast implementation of this function is as follows: construct a DFA for all the prefixes in S_{of} to match \widehat*str*, if no match occurs, then return *str*, else if the matched prefix corresponds to an OF identical with *of*, then return *str* likewise, or else extend *str* and repeat the matching of \widehat*str* in the DFA till find the available *str*. Note that the DFA for S_{of} can be incrementally updated when a new pair of *of* and *str* is inserted into S_{of}.

It is easy to find a different prefix for each distinct OF in practice, because the OFs that can cause explosion (note that the OFs <\s*>, <\d*> and so forth have very little chance to produce explosion due to the small-size character set) mainly concentrate on a small number of different types. Figure 3 displays the percentage of the maximum eight OFs that widely exist in real-world pattern sets, and they make up the majority of all the OFs in corresponding pattern sets. In addition, for the prefixes with *k*

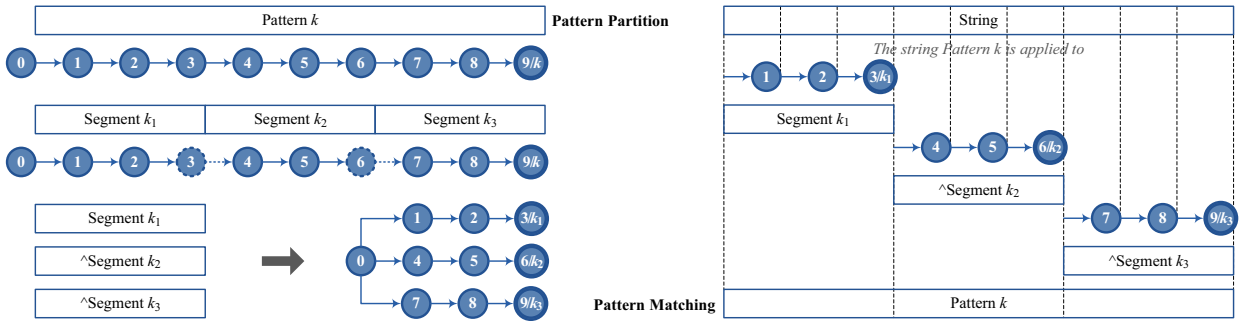

Fig. 4. The reassembly of the pattern segments to match the string the original pattern is applied to.

characters, the probability that they are different is $1-(|\Sigma|*(1/|\Sigma|)^2)^k = 1-1/|\Sigma|^k$. If $k=2$ and $|\Sigma|=256$, then $1-1/256^2 \approx 99.9985\%$.

Steps 15–29 are used to partition the original patterns into segments with distinct IDs according to the tags generated in steps 3–14, and add each resulting segment into one of the two segment sets according to whether it is an anchored and non-first segment. Besides, the syntagmatic relation of each segment is also recorded in the independent relation set. In these steps, steps 27–29 are particularly used for handling the first segments of all the patterns.

Besides, according to Algorithm 1, it is not essential to execute partition for every OF of a pattern. For example, the pattern $STR_1\phi\{n\}STR_2\phi\{n\}\cdots STR_k$ (where $k \in Z^+$) only needs to be partitioned into STR_{11} and \widehat$STR_{12}\phi\{n\}STR_2\ \phi\{n\}\cdots STR_k$. Therefore, in common case, splitting into two segments (i.e., doing partition for just one OF) is enough for a pattern.

Based on Algorithm 1, the overall partition result for the exemplified pattern set with Patterns 1–4 has been shown in Table 2. To guarantee no intra-segment overlapping, each OF in Patterns 1–4 should be partitioned into the anchored segments, therefore the OFs are all kept in the anchored segments of Group 2 (e.g. OF <[^\n]{5}> is kept in segment <^=[^\n]{5}>). Particularly, the anchored OF <.*> is equivalent to the unanchored, therefore the segment <^.*org> is equivalent to <org>. In addition, for Pattern 4 in Table 2, the OF <[^\s]{3,5}> may cause semantic overlapping with its suffix, thus its suffix should be also partitioned into an individual segment <^\.shtml>.

To guarantee no inter-segment overlapping, each OF in the anchored segments should have distinct prefixes, therefore the OFs <[^\r\n]*>, <[^\r\n]*>, <[^\n]{5}> and <[^\s]{3,5}> in Segments 6–9 respectively have different prefixes <e=>, <h=>, <=> and <\/>. Besides, because the anchored segments with OFs can also generate semantic overlapping with the unanchored segments (e.g. Segments 1 and 7), the anchored segments and the unanchored segments should add into different groups (i.e., Group 2 and Group 1). From Table 9, one can find that the total number of states in DFA 1 and DFA 2 is equal to that in original NFA (i.e., 48), and far less than that in original DFA (i.e., 1058).

To sum up, the state explosion problem can be resolved by heuristic pattern partition, and we can utilize the nonexplosive DFA to do fast regular expression matching. However, the complete semantics of the original patterns are broken into pieces, hence we need to construct the data structure which can recover the equivalent semantics on the basis of the pattern segments.

3. Fast regular expression matching engine (FREME)

In this section, we will propose a formal data structure, to represent the segments resulting from pattern partition without losing any semantics of their original patterns, meanwhile design the corresponding processing algorithm, to match the original patterns at very high speed without causing false positive and false negative.

3.1. Bipartite data structure

As Fig. 4 depicts, when original patterns are partitioned into overlapping-free segments, the syntagmatic relation of these segments turns from consecutive "concatenation" to independent "alternation", because the partition is at the concatenated location of adjacent segments according to Principle 2 (see the left subfigure). Hence, to simulate the direct matching of a complete pattern, the matching of the segments must be in succession and in sequence (see the right subfigure).

Note that, as Fig. 4 implies, the caret attached to the non-first segments (due to pattern partition) is used to guarantee the matching succession: the start of the matching of the posterior segment and the end of the matching of the anterior segment must be contiguous at the interval, otherwise false positive could occur. Further, to ensure the matching sequence, it is essential to rely on the syntagmatic relations of the segments. To this purpose, we propose the bipartite data structure containing DFA part and relation mapping table (RMT) part.

The DFA part consists of (at most) two nonexplosive DFAs, which are built from the overlapping-free segments. According to the previous analysis, the matching of each segment is separate in DFA part. To judge whether the matched segment is in correct sequence, RMT part is required. The RMT part maintains the syntagmatic relations of the segments in their original patterns, as well as the correspondence between the segments and their original patterns. In RMT part, the alternated segments belonging to the same pattern, should follow the order of their concatenation in their original pattern to be matched, or else although it is matched in DFA part, it will be regarded invalid and ignored.

This bipartite data structure together makes up the final matching engine named FREME. Based on the description above, FREME is defined formally as the following.

Definition 3. FREME is a 8-tuple $(Q, \Sigma, \delta, \Xi, G, \varphi, <\xi_0, q_0>, F)$, where

- Q is a finite set of states;
- Σ is a finite alphabet of input characters;
- $\delta : Q \times \Sigma \to Q$ is the state transition function for each input character;
- Ξ is a finite array of index keys;
- $G \subseteq Q$ is a finite set of match states;
- $\varphi : \Xi \times G \to <\Xi, Q>$ is the index update function for each match state;
- $<\xi_0, q_0>$ is the initial state unit which consists of an initial index key ξ_0 and an initial state number q_0;
- $F \subseteq \Xi \times G$ is the set of accepting state units.

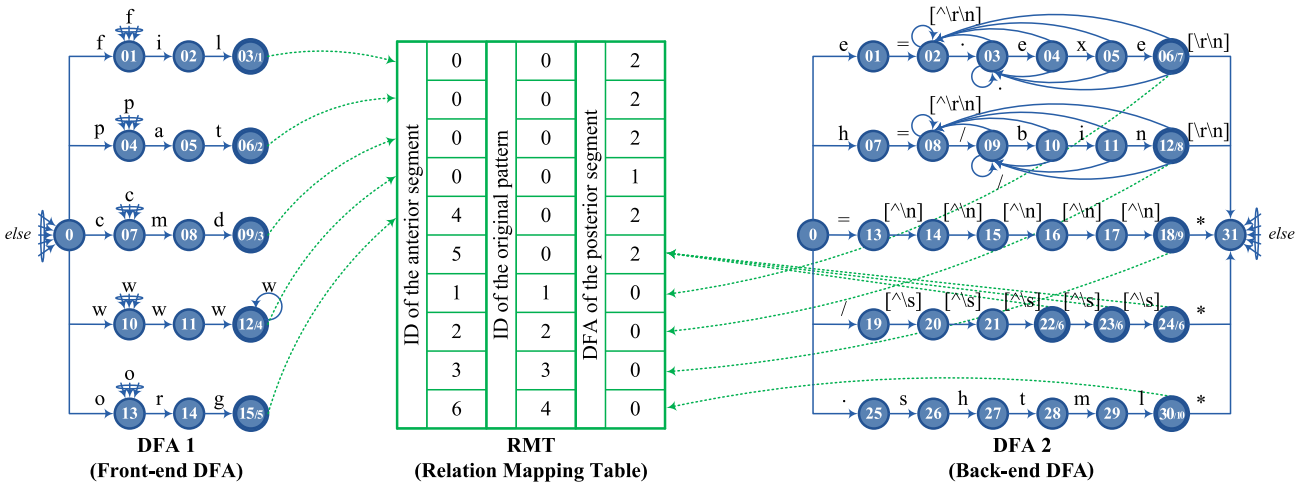

Fig. 5. FREME (some state transition edges omitted for clarity) for the pattern set of Patterns 1–4.

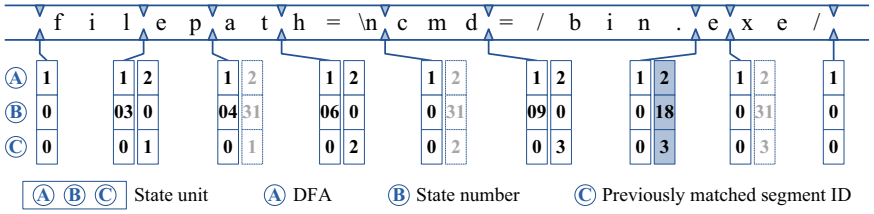

Fig. 6. The matching procedure of FREME for the example input content "filepath=\ncmd=/bin.exe/".

In this definition, Ξ records the index information of correctly matched segments. φ is used to judge whether the new matched segment follows the correct sequence in RMT part, and update the current Ξ and Q according to the result of the judgement. $F \subseteq \Xi \times G$ reveals that an original pattern is regarded as matched only when all its segments are processed and matched as expected. This is guaranteed by activating state units $< \Xi, Q >$ to keep track of the matching procedure.

3.2. Proof of concept

During pattern partition, the correspondence of each pair of adjacent segments (in each pattern) is recorded, meanwhile each segment is assigned a distinct segment ID (see Algorithm 1). Therefore, the combination and permutation of the segment IDs is unique for each different pattern, and every segment can be exactly determined based on the ID of its anterior (adjacent) segment. For example, in Table 2, the anterior segment of segment 7 ought to be segment 1 (corresponds to Pattern 1). Accordingly, the anterior segment ID can be used to judge whether the match of the current segment is valid. Therefore, a prototype of FREME is designed based on Definition 3.

As Fig. 5 demonstrates, in DFA part, each segment corresponds to a match state (e.g. state 03). When one segment is matched, its segment ID is used to index (indicated as dashed arrows) the corresponding table entry (in each row) in RMT part. For RMT part, the ID of anterior segment records the expected anterior segment of the matched one, which is used to judge the correctness of the match by comparing it with the practical anterior segment (i.e., the defined ξ). Besides, because there are two DFAs, the DFA of posterior segment should be designated, to identify where to match the next (adjacent) segment of current one. In the end, the ID of original pattern should also be marked for the tail segment of each pattern, because the match of tail segment is the essential condition to confirm that a pattern is completely matched.

In detail, illustrated in Fig. 5, the segment IDs of <www>, <org>, <^\/[^\s]{3,5}> and <^\.shtml> are 4, 5, 6, and 10 (see match states 12, 15 of DFA 1, and match states 24, 30 of DFA 2), respectively. The RMT part indicates the relations among these segments: the anterior segment IDs of segments 5, 6, and 10 are 4, 5, and 6 (i.e., segment 5/6/10 is at the back of segment 4/5/6), respectively. Besides, the posterior segments of segments 4, 5, and 6 are in DFA 1, DFA 2, and DFA 2 (i.e., segment 5/6/10 is in DFA 1/2/2), respectively. Moreover, the original pattern ID of segment 10 identifies corresponding Pattern 4, that is <www.*org\/[^\s]{3,5}\.shtml>. Note that 0 means no corresponding value in RMT part.

One can find that the original patterns can be uniquely reassembled based on the segments in DFA part and the relations in RMT part. For example, segments 4, 5, and 6 are respectively the anterior segment of segments 5, 6, and 10, and segment 4 is the first segment (due to no anterior segment), therefore, segments 4, 5, 6, and 10 correspond to a complete pattern in sequence, that is Pattern 4. This reversible procedure verifies that the FREME representation preserves the full semantics of original patterns, and we can further use the proper matching algorithm to guarantee no false positive or false negative.

3.3. Matching procedure

When using FREME to perform regular expression matching, active state units are required to keep track of intermediate state number and the IDs of previously matched segments for each input character. Algorithm 2 describes the matching procedure.

Algorithm 2. Matching procedure.

Input:
Data structure *freme*; Input content *input*.
Output:
Match result.
Algorithm:

```
1:  i = 0; S_active = {state_unit(1, 0, 0)}; S_update = ∅;
2:  while i < input.size() do
3:    c = input[i++];
4:    for all su ∈ S_active do
5:      // get to the next state in su.dfa
6:      su.state = freme.get_dfa_state(su.dfa, su.state, c);
7:      // if it is a endless loop state, remove su from S_active
8:      if freme.be_dead_state(su.dfa, su.state)
            == true then
9:        S_active.erase(su);
10:       continue;
11:     end if
12:     // if it is not a match state, do nothing for su
13:     if freme.be_match_state(su.dfa, su.state, &seg_id)
            == false then
14:       continue;
15:     end if
16:     te = freme.get_rmt_entry(seg_id);
17:     // for match state, if the segment is not matched in
        sequence and in succession, the match is invalid
18:     if te.ante_seg_id != su.ante_seg_id then
19:       continue;
20:     end if
21:     // if the validly matched segment is a tail segment, then
        the corresponding pattern is matched
22:     if te.orig_pat_id > 0 then
23:       output_match_pattern(te.orig_pat_id);
24:     else
25:       // or else, activate a new state unit for the validly
          matched segment
26:       S_update.insert(state_unit(te.post_seg_dfa, 0, seg_id));
27:     end if
28:   end for
29:   // if new state units are generated, add them into S_active
30:   if S_update != ∅ then
31:     S_active.insert(S_update);
32:     S_update.clear();
33:   end if
34: end while
```

In Algorithm 2, only steps 3 and 5 must be executed for each input character. If there are no new state units generated in step 20, then FREME is just equal to DFA. One can find that the premise of producing new state units is that neither of steps 10, 14, and 17 can be satisfied, which means a match state must be accessed and it should be an eligible match of a non-tail pattern segment. In detail, step 14 is used to judge whether the matched segment *seg_id* is eligible according to whether the expected anterior segment *te.ante_seg_id* of the matched segment *seg_id* has been matched, namely equal to *su.ante_seg_id*. Step 17 is executed to find out whether the matched segment is a tail one, and if so, a complete pattern is judged to be matched in step 18, or else a new state unit ought to activated.

As for step 6, it can be found in FREME that there is an endless loop state (e.g., state 31 of DFA 2 in Fig. 5), named dead state, which always exists in the DFA whose corresponding patterns are all anchored and have no dot-star. Every state in corresponding DFA can transit to this state for mismatched characters. Once the *state* of an active state unit *su* falls into such a dead state, no more segment match is able to happen for *su*, no matter what the input character is, because the *state* will never transit to any other match states but the dead state itself. Thus, such an active state unit *su* can be removed from S_{active} without affecting the matching procedure. This is just the purpose of steps 6–9.

To demonstrate Algorithm 2, we take Fig. 5 as example and suppose the input content to process is "filepath=\ncmd=/bin.

exe/". Figure 6 illustrates the detail of matching. The initial state unit consists of DFA 1, state number 0 and segment ID 0. For input "fil", segment 1 is matched in DFA 1, then its ID 1 is used to index the 1st entry in RMT, where no anterior segment is required (anterior segment ID is 0), thus this match is valid, and a new state unit with DFA 2, state number 0 and segment ID 1 is generated, parallelly processed with the original one after then. Next for input "ep", the state unit in DFA 2 enters dead state, thus it is removed (the number is grayed). Next for input "at", segment 2 is correctly matched, likewise, a corresponding state unit with DFA 2, state number 0 and segment ID 2 is generated. Next for input "h=\n", the new state unit is also deleted due to the transition to dead state. Next for input "cmd", segment 3 is matched in DFA 1, and another state unit is created. Next for input "=/bin.", segment 9 is matched in DFA 2, the indexed 9th entry in RMT shows that segment 3 is expected to be previously matched, and it had been matched in deed (according to current state unit), thus this is a valid match. Besides, the match of Pattern 3 (i.e., <cmd=[^\n]{5}>) is found (the background is colored), because the original pattern ID of segment 9 is 3. For the following input "exe/", the number of state units is further reduced to one in the end.

3.4. Performance advantages

In FREME, the size of DFA part is $(N_{S1}+N_{S2})\cdot 256\cdot \log_2 \max(N_{S1},N_{S2})$ bits, where N_{S1} and N_{S2} are the number of states in DFAs 1 and 2, respectively. Due to no state explosion, $N_{S1}+N_{S2}=N_P\cdot L_P$, and $N_P\cdot L_P/2\le \max(N_{S1},N_{S2})\le N_P\cdot L_P$. The size of RMT part is $(N_P+N_{OF})\cdot(1+\log_2 \max(N_{S1},N_{S2})+\log_2 N_{OF})$ bits, where N_{OF} is the number of OFs, and $N_{OF}<N_P\cdot L_P$. One can easily derive that the size of RMT part is smaller than 1/64 of the size of DFA part, thus the size of FREME mainly depends on the number of states in DFA part. Because the number of DFA states in FREME is close to the number of NFA states, FREME is as scalable as NFA in terms of size.

It is important to note that, although two DFAs could be built for FREME, the back-end DFA is rarely activated and executed. In fact, the front-end DFA plays a prefilter role, because only the first segment of any original pattern in front-end DFA is matched correctly, can the back-end DFA be activated to process the following segments belonging to the same pattern. Besides, even if the back-end DFA is triggered to match the new generated state units, it can be closed in a short time because the state unit has a big chance to be trapped in dead state. Therefore, its average-case matching complexity is $O(1)$. In worst case, its theoretical complexity is $O(N_{S1}+N_{S2})$, because at most all concurrent active state units correspond to the distinct state numbers in different DFAs, and for the state units with identical state number of the same DFA, they can be merged into one (this will be explained in Section 4.3.3). In fact, this can be further optimized according to Sections 4.3.1 and 4.3.2.

Furthermore, unlike the general data structure with prefilter (e.g. Snort IDS Roesch, 1999; www.snort.org), where the back-end matching engines of the prefilter are built from the complete patterns and required to process the input characters from the start when a match occurred in prefilter (i.e., multi-pass matching), the back-end DFA in FREME is also constructed from pattern segments, and only needs to process the subsequent input characters if it is executed (i.e., one-pass matching). Moreover, FREME has a homogeneous structure, because both DFA and RMT can be stored as static arrays, and thus its data access is more simple and time-efficient than Hybrid-FA (Becchi and Crowley, 2007) and XFA (Smith et al., 2008a, 2008b).

4. Optimization

In this section, we will introduce a series of optimization schemes from the perspective of pattern partition, data structure and matching procedure, to further improve the performance of FREME.

4.1. Towards pattern partition

4.1.1. Equivalent segments

In pattern partition, each segment is labeled with an unique ID, which implies that all the segments are distinctive by default. However, there are often many equivalent segments, which actually can be merged although they have different IDs.

For any two segments X_i and Y_j, where X_i is the ith segment in the pattern X which has I segments in total, and Y_j is the jth segment in the pattern Y which has J segments altogether, we say that X_i and Y_j are equivalent if the following conditions hold:

1. $i<I$, $j<J$, and $i=j$;
2. $\forall k\le \max(i,j)$, $X_k=Y_k$;
3. X_{i+1} and Y_{j+1} belong to the same group.

As an example, for the two patterns <ab.*cd.*ef> and <ab.*cd.*gh>, they can be split into $<$ab$>$, $<$cd$>$, $<$ef$>$ and $<$ab$>$, $<$cd$>$, $<$gh$>$. One can find that the two identical segments $<$cd$>$ (and $<$ab$>$) are equivalent indeed, and can be merged into just one because the remainder segments will correspond to a complete pattern <ab.*cd.*(ef|gh)>, which is equal to the original two patterns. According to our evaluation, such an optimization can reduce about 14% of the number of segments for the real-world pattern set from Snort IDS (www.snort.org).

All the equivalent segments must be matched at the same time, which can lead to the simultaneous activation of multiple state units during matching, therefore, the segment merger based on the equivalency will greatly reduce the number of active state units, and in turn improve the matching performance.

4.1.2. Identical OFs

In pattern partition, Principle 4 requires that for distinct OFs, their prefixes reserved for corresponding anchored segments should be different as well, in order to avoid the possible inter-segment overlapping. However, for identical OFs having the same type and character set, their prefixes are preferred to be the same for more compact DFA.

For the OFs $\phi\{n_1\}$, $\phi\{n_2,\}$ and $\phi\{m,n_3\}$, if their character sets ϕ and their prefixes STR are equal, then the states of the DFA for corresponding segments will be $\max(n_1,n_2,n_3)$, rather than $n_1+n_2+n_3$.

For example, the two patterns <abcdx[^\n]{64}> and <efghx[^\n]{32,72}> (whose composite NFA has 147 states) can be split into the non-first segments <^x[^\n]{64}> and <^x[^\n]{32,72}>. The two OFs have the same character set <[^\n]>, meanwhile the prefixes of two OFs are both <^x>, then the DFA for the two segments only has 75 states (corresponding FREME has 75+9=84 states, much less than the NFA). In contrast, if the former segment is changed into <^dx[^\n]{64}> based on another partition of the first pattern, then the DFA for the new two segments will have 75+66=141 states. For the real-world pattern set from Bro IDS (http://www.bro.org), our statistical result shows that about 21% of the number of states can be optimized away for FREME because of identical OFs.

It is thus clear that, on the premise of Principle 4, choosing equal prefixes for identical OFs will render corresponding DFA more succinct than the NFA of original patterns.

4.1.3. Trade space for time

According to Section 3.4, the back-end DFA can be triggered only when the front-end DFA has segments matched. If the possibility of the match taking place in the front-end DFA is reduced, then the frequency of the activation of the back-end DFA will be decreased, accordingly, the matching performance of FREME will be great improved.

Because longer segment can have fewer chance to be matched, we can trade less partition for longer segments of each pattern. To be practical, for the patterns whose OFs causing no or little state inflation, the partition position against the OFs can be neglected, thus the OFs can be retained for their front segments to lengthen them, at the expense of the possible (slight) state inflation of the front-end DFA caused by the reserved OFs in corresponding segments.

For example, <^authorization\x3a\s*basic[^\n]{256}>, a real-world pattern from Snort IDS (www.snort.org), can be partitioned into <^authorization\x3a\s*basi> and <^c[^\n]{256}>, because the OF <\s*> cannot cause too much state inflation even if it can intersect with other patterns. Then although the first segment could arouse small memory overhead, the temporal performance gets more efficient. In our experiments, for the pattern set of Linux L7-filter (http://l7-filter.clearfoundation.com), the speed of FREME can be increased by almost 50% in exchange for tripled memory footprint.

Consequently, this tradeoff can facilitate pattern partition to flexibly deal with different kinds of patterns, making FREME more practical. Note that, even for the special pattern <^search\s+[^\n]{1024}> mentioned by Yu et al. (2006), it can be split into <^search\s*> and <^\s[^\n]{1024}> to remove semantic overlapping, in the similar way as $\widehat{STR_1}\phi\{n\}\ STR_2$.

4.2. Towards data structure

4.2.1. DFA part

It is known that there are a lot of DFA-based compression algorithms (Tuck et al., 2004; Kumar et al., 2006; Brodie et al., 2006; Becchi and Cadambi, 2007; Becchi and Crowley, 2007; Ficara et al., 2008; Meiners et al., 2010; Qi et al., 2011; Peng et al., 2011; van Lunteren and Guanella, 2012) proposed in the past decade. They focus on eliminating the redundant identical transitions inside or among the states, to reduce the memory overhead of each state in the DFA, and only work on the premise that DFA can be generated (i.e., nonexplosive). However, one can find that these methods are orthogonal to ours, and can be used to reduce the number of state transitions for FREME, although the DFA part of FREME is very succinct in terms of the number of states. Coupled with them, FREME is expected to achieve further multiple times decrease in memory usage. The test shows that, based on the improved D^2FA algorithm (Becchi and Crowley, 2007), over 99% compression ratio can be achieved for the pattern set from ClamAV antivirus engine (http://www.clamav.net).

4.2.2. RMT part

It can be found from Fig. 5 that, in RMT of FREME, the columns of the ID of original pattern and the DFA of posterior segment can be combined into only one column, because only one value of them is simultaneously significant for each segment. After merged, original values can be identified by using positive number and negative number. This can decrease the space of RMT part, and more important, reduce the time to generate a state unit.

Besides, in FREME, each entry of RMT can be put in corresponding match state of DFA. Based on this memory layout, during the matching, the cache-line can simultaneously load the state transitions as well as corresponding RMT entries, and thus avoid one more memory access for fetching RMT entries (the access of RMT entries can be directly got from cache then).

4.3. Towards matching procedure

4.3.1. Unnecessary state units

Note that in Fig. 5, almost all the match states (e.g. state 03) in DFA 1 have the same next state as state 0 for identical input character. Because a new state unit is initialized with state number 0, for the next input character, if the old state unit (at match state) and the new state unit (at state 0) can transit to the same next state in the DFA, there will be no need to generate a new state unit, except to record the just matched segment in the old state unit.

As Fig. 7a shows, for the input “wwworg”, when the segment <www> is matched at match state 12 of DFA 1, it is unnecessary to generate a new state unit, because the DFA of the posterior segment <org> is still DFA 1, and the next state of state 12 is

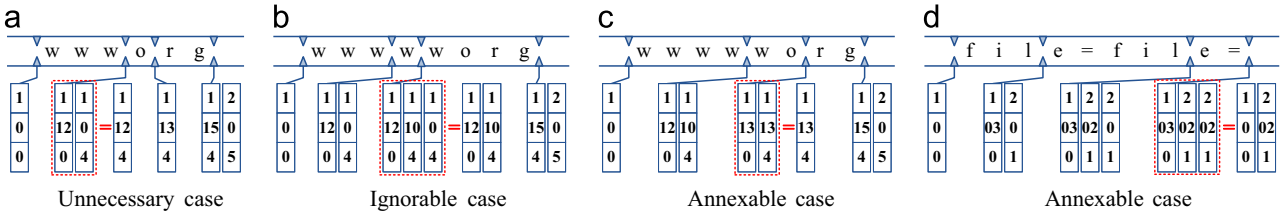

Fig. 7. The optimization for matching procedure (of FREME shown in Fig. 5) based on the activation of state units: (a) unnecessary case; (b) ignorable case; (c) annexable case; and (d) annexable case.

Table 4
The statistics of the real-world pattern sets used in our experiments.

Pattern set	# of patterns				# of OFs				# of states		# of segments	
	Total	Explosive	W/inter-overlapping	W/ intra-overlapping	Total	Different	W/ϕ^* form	w/$\phi\{\}$ form	NFA	DFA	Before optimi.	After optimi.
snort.set	1210	16	772	92	5099	54	4942	157	52379	⩾10M	2240	1365
bro.set	1238	42	233	47	344	7	51	293	23918	⩾10M	1578	1370
clamav.set	1292	15	980	964	4177	6	192	3985	73885	⩾10M	3743	2450
l7filter.set	111	0	71	22	255	34	116	139	2997	⩾10M	320	209
app1.set	1387	1	724	433	2730	28	1344	1386	31890	⩾10M	2492	1958
app2.set	5886	2	72	40	131	18	67	64	155038	⩾10M	6026	5967

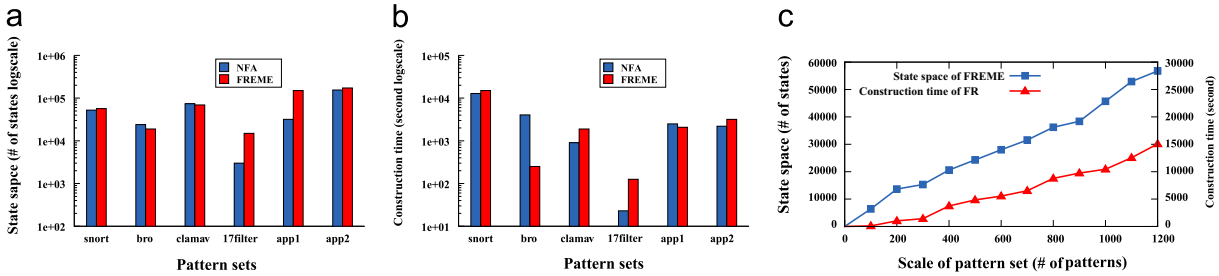

Fig. 8. The efficiency comparison between NFA and FREME, and the scalability evaluation of FREME in terms of state space and construction time: (a) state space; (b) construction time; (c) linear scalability for *snort.set*.

the same as the one of state 0 for the next input character "o" (i.e., state 13). However, the old state unit will be updated by appending the ID of the new matched segment (i.e., 4) to itself.

Although the exception that the match states have different next states with state 0 rarely occurs (e.g. for match state 12 and next input "w"), we need to judge whether it is necessary to generate a new state unit, by pre-fetching and pre-matching the next input character, and comparing the next state with the one of state 0, if not equal, then it is still necessary to create the new state unit.

4.3.2. Ignorable state units

For the segments whose posterior (adjacent) segments are in DFA 1 (i.e., the unanchored non-first segments), when they are matched once and corresponding new state units are generated, then when they are matched again, even though the match is still correct, it is ignorable, because the matching in DFA 1 is greedy. For this reason, one active state unit is enough for the match of such segments, no matter how many times their match happens. We can use a bit vector to mark whether these segments have been already matched, where the number of bits depends on the number of dot-star.

As Fig. 7b shows, for the input "wwwwworg", the segment <www> will be matched three times, but the second match can be ignored (the third match can refer to Section 4.3.1), because it does not affect the match of the segment <org> in DFA 1.

4.3.3. Annexable state units

It is found in prior work (Becchi and Crowley, 2007; Peng et al., 2011) that most state transitions lead back to the initial states (e.g. state 0 of DFA 1 or its neighbor states in Fig. 5) or the self-loop states (e.g. state 02 of DFA 2 or the dead state in Fig. 5). Consequently, even if multiple state units are activated, they have a great chance of leading back to the same state of corresponding DFA in a short time, and become annexable, as shown in Fig. 7c and d. If the IDs of previously matched segments are maintained in state unit based on the bit vector, where the number of bits is equal to the number of OFs, then the state units having identical state number of the same DFA could be merged merely by the bitwise OR of their bit vectors (Wang and Li, 2013). This means the number of state units in FREME can always converge to one in any case.

In addition, we can choose the proper time to merge the annexable state units as necessary as possible, rather than for every input character, because most of the time the number of state units is converged. Thus, the threshold of the number of active state units is used. Initially, a default threshold is given. Once the number of active state units achieves the threshold, the merger is executed. After that, if the number of current state units is much less than the threshold, then change the threshold to make it slightly larger than this number. If the number is not reduced too much, then we need to adjust the threshold to be slightly larger than its own value. In other cases, the threshold remains unchanged. By dynamically choosing the best time to merge, FREME can keep matching as fast as possible (e.g. about 6% processing performance improvement for the pattern set from Bro IDS http://www.bro.org).

5. Experimental evaluation

In this section, we will use real data and experiments to verify the effectiveness of our proposed solution.

5.1. Experiment environment and data sets

All the experiments are conducted on an Intel Xeon E5504 server (CPU: 2.0 GHz, L1 Cache: 32 KB, L2 Cache: 4 MB, DDR3 RAM: 800 MHz/8 GB, OS: 32bit Ubuntu Server 10.04 LTS).

The pattern sets used in our experiments are collected from the open source Snort (www.snort.org), Bro (http://www.bro.org), ClamAV (http://www.clamav.net), L7-filter (http://l7-filter.clearfoundation.com) and two commercial application-aware products. As Table 4 shows, all of them but *l7filter.set* contain thousands of regular expression patterns. Note that few prior work uses pattern sets of such a large scale to test the actual performance of the proposed solutions.

For every given pattern set, Table 4 also lists the number of explosive patterns each of which cannot be individually compiled into a single DFA within 10M (i.e., 10,000,000) states. Besides, the complex patterns with intra- and inter-pattern overlapping are counted. It can be found from Table 4 that for these pattern sets, none of them can be transformed into the corresponding DFA in practice, because the storage of the DFA with far over 10M states requires much more than 10 GB (i.e., >8 GB) memory space. In contrast, their NFAs are all available, and contain no more than 200K states (the NFA of *app2.set* has the maximum number of states, i.e., 155,038).

5.2. Partition statistics

Table 4 displays the statistics about OFs. Comparatively speaking, *snort.set*, *clamav.set* and *app1.set* are more complex due to significantly more, beyond a thousand, OFs (i.e., 2730–5099) and hundreds of patterns with semantic overlapping (i.e., >700). Especially, *clamav.set* and *app1.set* have thousands of OFs with form $\phi\{\}$ (i.e., 1386–3985), which can cause DFA state inflation individually. Besides, it is evident that the number of nonequivalent OFs is far less than the total (as mentioned earlier in Section 2.4). For example, for *clamav.set*, the number of nonequivalent OFs is only 6, although the total number of OFs is 4177. This facilitates the satisfaction of Principle 4 and optimization in Section 4.1.2.

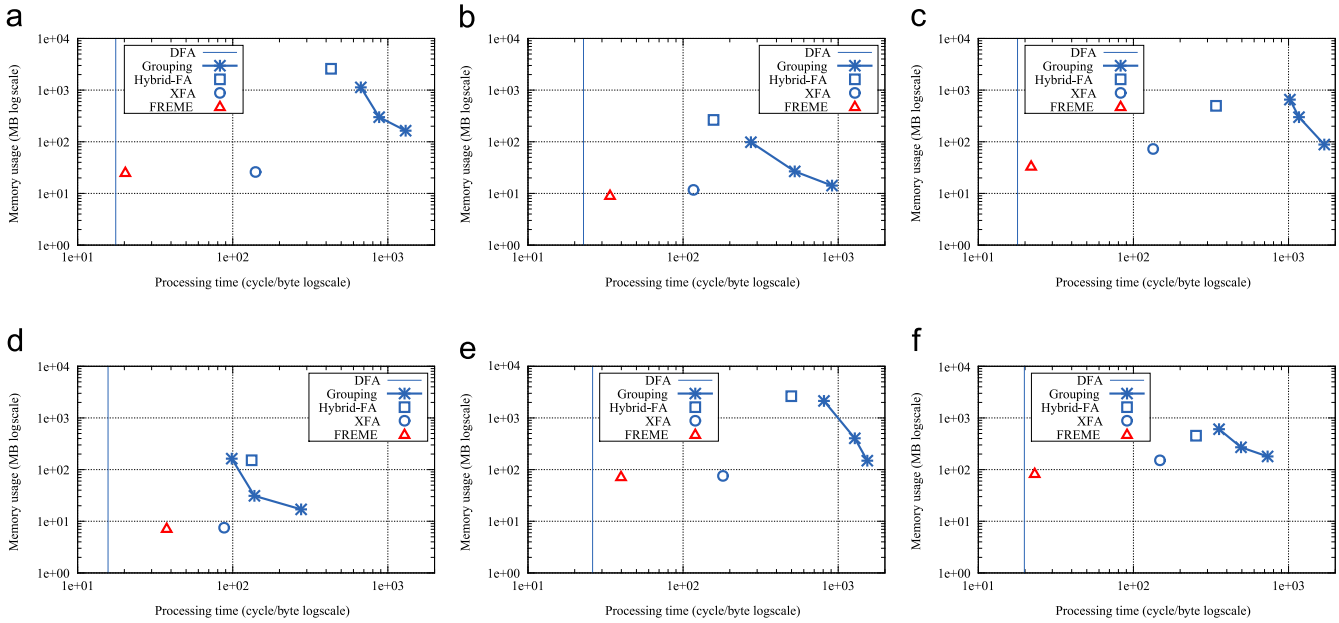

Fig. 9. The temporal versus spatial performance comparison of DFA, Grouping, Hybrid-FA, XFA and FREME: (a) for *snort.set*; (b) for *bro.set*; (c) for *clamav.set*; (d) for *l7filter.set*; (e) for *app1.set*; and (f) for *app2.set*.

Table 5
The comparison of the per-flow states for NFA, Hybrid-FA and FREME.

Pattern set	Patterns		NFA			Hybrid-FA			FREME		
	All #	Matched #	Max size/bytes	Max #	Avg. #	Max size/bytes	Max #	Avg. #	Max size/bytes	Max #	Avg. #
snort.set	1210	19	218	109	9	184	46	8	22	4	1.048
bro.set	1238	153	116	58	34	176	44	28	115	23	1.061
clamav.set	1292	12	376	94	9	128	32	7	46	3	1.053
l7filter.set	111	5	28	14	6	56	14	6	10	3	1.101
app1.set	1387	57	282	141	11	232	58	9	168	28	1.113
app2.set	5886	23	320	80	10	152	38	6	54	4	1.059

For the pattern sets with more OFs, such as *snort.set*, *clamav.set* and *app1.set*, splitting their patterns into more segments is required, to completely eliminate the semantic overlapping. Because each pattern is partitioned into at least one segment, the number of segments must be larger than the number of patterns. For example, for *snort.set* in Table 4, the number of original patterns is 1210, and the number of segments in the raw segment sets is 2240, larger than 1210. However, based on the optimization of Sections 4.1.1 and 4.1.3, the number of segments can be further reduced. As the last two columns of Table 4 show, for *snort.set*, the number of segments in the raw segment sets is 2240, but the number of distinguishing segments for the optimized segment sets is 1365. There is a large decrease (e.g. $2240-1365=875$ for *snort.set*) for the practical number of segments. Therefore, compared with original patterns, the number of segments does not increase too much in fact (e.g. no more than twice for all given pattern sets in Table 4), and thus its growth can be regarded to be near-linear.

Moreover, from Table 4, one can find that not all the OFs are partitioned in practice, because the number of segments is far less than the number of OFs. For example, the number of OFs in the pattern set *snort.set* is 5099, and the number of segments in the raw segment sets is 2240, much less than 5099. In fact, according to the analysis presented in Section 4.1.3, OFs with a small character set ϕ are unlikely to cause semantic overlapping; therefore, they are ignored for partition.

5.3. State space

According to Fig. 8a, FREME is very close or even superior to NFA in terms of fewer number of states. In fact, Principles 3–4 can make the segments resulting from partition free of semantic overlapping, and thus cause the state space of the built DFA part the same as NFA. However, for identical OFs having the same type and character set, the optimization in Section 4.1.2 makes their prefixes as identical as possible, therefore the built DFA part of FREME can be more succinct than NFA. For example, for *bro.set* in Fig. 8a, there are many patterns having identical OFs with the same prefix, the number of states in the NFA constructed from the original patterns is 23,905, but FREME built from the generated segments only have 18,935 states, less than NFA, because there are many patterns having identical OFs with the same prefix.

On the contrary, for *l7filter.set* and *app1.set* in Fig. 8a, the number of states in FREME is slightly more than NFA, because FREME ignores the partition of the OFs with small character set in exchange for fewer segments and better runtime performance, which results in little semantic overlapping. This is the

optimization of performance tradeoff mentioned in Section 4.1.3. For these pattern sets, although there are also many segments satisfying the condition of the optimization in Section 4.1.2, but the number of OFs ignored for partition is more, therefore the corresponding FREME is not better than NFA in state space.

Using the statistical numbers of patterns, OFs, segments and states of DFA part, we can figure out the size of RMT part based on the formula mentioned in Section 3.4, which definitely is negligible compared to the size of DFA part. As an example, for *snort.set*, the size of RMT part in FREME is $1365*(2\ B+2\ B)\approx 5.3$ KB, but the size of DFA part is $52,379*256*2\ B+1365*(2\ B+4\ B)\approx 25.6$ MB.

5.4. Construction time

For FREME, its construction contains two steps: pattern partition and segment compilation. The total construction time of FREME is mainly determined by the number of states in the DFA part (i.e., $N_{S1}+N_{S2}$), because the time consumed in compilation process is proportional to $N_{S1}+N_{S2}$, while the time spent in partition is proportional to N_{OF} and relatively insignificant ($N_{OF}<N_P\cdot L_P\le N_{S1}+N_{S2}$).

Figure 8b shows that, because of the nonexplosive state space, the construction time of FREME is also comparable with NFA. For example, for *bro.set*, the construction of FREME only requires about 250 s, but the construction of NFA costs over 2700 s. Even when FREME has more states than NFA, for example, for *app1.set*, building FREME still can cost less time than NFA. This is because the NFA reduction process is more time-consuming for the original patterns than the overlapping-free segments.

However, for *clamav.set* in Fig. 8b, NFA is slightly more efficient than FREME, although it has slightly more states than FREME. This is because both FREME and NFA spend almost the same time in the process of NFA reduction, but FREME requires more time for the processes of subset construction and DFA minimization. Even so, FREME is still close to NFA in terms of less construction time.

Because the existing state-of-the-art algorithms are mainly based on NFA, their preprocessing time will be not better (often be far worse) than NFA in practice. For pattern grouping (Yu et al., 2006), Hybrid-FA (Becchi and Crowley, 2007) and XFA (Smith et al., 2008b), they cost at least one order of magnitude (above two orders of magnitude in most cases) more construction time than FREME for real-world pattern sets (e.g. *snort.set*). The shorter preprocessing time can make FREME practically available within the tolerable update deadline.

5.5. Scalability

Because state explosion is defused by pattern partition, FREME increases proportionally in state space with the scale of the pattern sets, like NFA, as illustrated in Fig. 8c. When the number of patterns in *snort.set* increases from 100 to 1200, the growth of the number of states in corresponding FREME approximates linearity, meanwhile the construction time of FREME is also proportional to the scale of the pattern set. Besides, according to Section 5.2, the number of segments grows linearly with original patterns, thereby the size of RMT is proportional to the number of patterns as well (although ignorable). Because FREME consists of the DFA and RMT parts, in consequence, FREME scales linearly. Likewise, from Fig. 8c, one can find that the construction time has a similar trend of linear growth with the number of states for FREME. In contrast, the growth of the state space of DFA is exponential in most cases.

5.6. Matching performance

The network traffic used for statistical performance evaluation includes the 495 MB published trace from MIT Lincoln Lab (http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data), and the 309 MB real-world trace from the campus network of Tsinghua University (mixed into one trace for experiments). Besides, the state-of-the-art algorithms used for performance comparison include pattern grouping (Yu et al., 2006), Hybrid-FA (Becchi and Crowley, 2007) and XFA (Smith et al., 2008b).

Due to the existence of explosive patterns, neither of DFA, pattern grouping and XFA can deal with corresponding pattern sets in practice. However, to make these algorithms feasible for comparison, all the complex patterns beyond their processing capacity are rewritten by narrowing down the character sets of corresponding OFs (see Section 2.2) before their evaluation. From the perspective of regular expression semantics, only a subset of patterns in each pattern set is used to estimate the performance of these algorithms. For example, in *snort.set*, the pattern <auth\s[^\n]{100}> should be rewritten as <auth\s[^a\s\n]{100}> for pattern grouping and XFA, and obviously the rewritten pattern is only a subset of the original pattern in regular expression semantics. Therefore, it is nontrivial to note that the actual performance of them will be no better than the evaluated performance.

Figure 9 shows the results of performance comparison for the given pattern sets, and the shorter the processing time is, the faster the processing is. Although DFA has deterministic $O(1)$ processing complexity, its processing speed can be affected by its size to some extent (i.e., 16–24 cycle/byte, shown as vertical line in Fig. 9), due to the memory cache hit ratio. From Fig. 9, one can find that FREME is quite close to DFA in terms of lower matching time in all these cases, and its processing speed is relatively stable, because its processing time is kept between 20 and 40 cycle/byte (i.e., 400–800 Mbps per core for 2.0 GHz CPU).

For *l7filter.set* in Fig. 9d, the processing of FREME is much slower compared with DFA. This is because on the one hand, the number of segments is almost twice as much as the number of patterns in *l7filter.set* (i.e., 209 vs. 111 in Table 4), which means almost every pattern is partitioned, and thus the FREME often activate more state units in processing when one segment in some pattern is correctly matched, so that the processing speed of FREME is relatively slow, and one the other hand, the DFA is very small due to the aforementioned pattern rewrite. In fact, according to our simulated analysis, the performance of practical DFA (whose size is comparatively very large) should be over one order of magnitude slower than the DFA built on the premise of pattern rewrite.

Unlike *clamav.set*, for *clamav.set* in Fig. 9c, the number of segments is also almost twice as much as the number of original patterns (i.e., 2450 vs. 1292 in Table 4), but the segments are not frequently matched, therefore not too many state units are activated during processing, and the speed of corresponding FREME is still fast.

Besides, compared with the state-of-the-art solutions, FREME has the best temporal and spatial performance simultaneously (being more close to left bottom means better performance in Fig. 9). In general, FREME outperforms XFA for a factor of tens in terms of shorter processing time (XFA sacrifices the major runtime for the fetch and execution of the instructions attached with each accessed state), and exceeds Hybrid-FA for up to two orders of magnitude in terms of both shorter processing time and less memory space (the matching of Hybrid-FA is often trapped in the tail-NFAs). Note that the optimization of Section 4.2.1 is not used in our evaluation to ensure that the performance comparison is impartial.

In fact, the temporal and spatial performance of XFA should be lower because many patterns which it cannot handle are rewritten in the test. In other words, FREME is superior to XFA not only in processing time, but also in memory usage, although the advantage in spatial performance cannot be obviously observed in Fig. 9. Furthermore, both pattern grouping and Hybrid-FA focus on finding a best performance tradeoff between NFA and DFA, therefore, neither of them can create a data structure with states less than corresponding NFA, and keep the matching speed close to DFA. From Fig. 9 one can find that, for pattern grouping, it makes the memory usage less at the cost of great processing speed. Besides, for Hybrid-FA, to make Hybrid-FA practically available for various kinds of complex patterns, it also sacrifice much processing time compared with DFA.

In addition, Table 5 depicts the statistics of per-flow states for NFA, Hybrid-FA and FREME (Grouping and XFA are not included because the raw pattern sets should be used for this comparison) during the matching procedure. For FREME, the average number of per-flow states is indeed convergent and close to one (see the last column of Table 5). Compared with NFA and Hybrid-FA, FREME is superior in terms of both the less number and the smaller size of per-flow states.

5.7. Sampling-based smoothed analysis (SSA)

In practice, the probability for FREME to meet its worst case (see Section 3.4) is very low. To analyze the overall performance of FREME theoretically, SSA technique (Ren et al., 2013) is employed to test the efficiency of our algorithms more convincingly. According to the evaluation of SSA, the generic performance of FREME are only about 40–110% worse than DFA (in terms of cycle/byte).

6. Conclusion and future work

In this paper, we present FREME, a pattern partition based engine for fast and scalable regular expression matching in practice. In contrast to direct DFA deflation, FREME leverages on predefined partition procedure to disarm the semantic overlapping existing in the complex patterns from the root, and thus the corresponding DFA part will not suffer state explosion any longer. Meanwhile, FREME uses simple RMT part to preserve the complete semantics of original patterns which are broken by partition. Benefiting from pattern partition, FREME achieves fast matching based on the compact DFA part for large-scale pattern sets. Evaluation based on real-world pattern sets (open source and commercial) verifies that FREME is practically fast (like DFA) and scalable (like NFA). In contrast, FREME outperforms state-of-the-art matching engines up to two orders of magnitude.

As a foundation for future research, FREME is found on the software algorithm. However, its design in hardware platforms is worth of in-depth study, which can better utilize the pipelining and parallelization to accelerate matching, so as to be more applicable in high-bandwidth networks. Indeed, even with our current prototype, measurements demonstrate very significant performance improvements over previous solutions.

References

Application layer packet classifier for linux, ⟨http://l7-filter.clearfoundation.com/⟩, 2014.

Becchi M, Cadambi S. Memory-efficient regular expression search using state merging. In: IEEE INFOCOM; 2007. p. 1064–72.

Becchi M, Crowley P. A hybrid finite automaton for practical deep packet inspection. In: ACM CoNEXT; 2007. p. 1–12.

Becchi M, Crowley P. An improved algorithm to accelerate regular expression evaluation. In: ACM/IEEE symposium on architectures for networking and communications systems (ANCS); 2007. p. 145–54.

Becchi M, Crowley P. Extending finite automata to efficiently match perl-compatible regular expressions. In: ACM CoNEXT; 2008. p. 1–12.

Brodie BC, Taylor DE, Cytron RK. A scalable architecture for high-throughput regular-expression pattern matching. In: International symposium on computer architecture (ISCA); 2006. p. 191–202.

Clam antivirus, ⟨http://www.clamav.net/⟩, 2014.

Darpa intrusion detection data sets, ⟨http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data/⟩, 2014.

Ficara D, Giordano S, Procissi G, Vitucci F, Antichi G, Di Pietro A. An improved dfa for fast regular expression matching. ACM SIGCOMM Comput Commun Rev 2008;38(5):29–40.

Fu Z, Wang K, Cai L, Li J. Intelligent grouping algorithms for regular expressions in deep inspection. In: International conference on computer communication and networks (ICCCN); 2014. p. 1–8.

Hopcroft JE, Motwani R, Ullman JD. Introduction to automata theory, languages, and computation. 3rd ed. USA: Addison-Wesley; 2006.

Hopcroft J. An n log n algorithm for minimizing states in a finite automaton. Technical report. Stanford, CA: Stanford University; 1971.

Huang K, Ding L, Xie G, Zhang D, Liu AX, Salamatian K. Scalable tcam-based regular expression matching with compressed finite automata. In: ACM/IEEE symposium on architectures for networking and communications systems (ANCS); 2013. p. 83–94.

Kumar S, Dharmapurikar S, Yu F, Crowley P, Turner J. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: ACM SIGCOMM; 2006. p. 339–50.

Kumar S, Chandrasekaran B, Turner J, Varghese G. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In: ACM/IEEE symposium on architectures for networking and communications systems (ANCS); 2007. p. 155–64.

Liu C, Wu J. Fast deep packet inspection with a dual finite automata. IEEE Trans Comput 2013;62(2):310–21.

McNaughton R, Yamada H. Regular expressions and state graphs for automata. IRE Trans Electron Comput 1960;EC-9(1):39–47.

Meiners CR, Patel J, Norige E, Torng E, Liu AX. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In: USENIX conference on security; 2010. p. 1–16.

Pasetto D, Petrini F, Agarwal V. Tools for very fast regular expression matching. Computer 2010;43(3):50–8.

Paxson V. Bro: a system for detecting network intruders in real-time. Comput Netw 1999;31(23):2435–63.

Peng K, Tang S, Chen M, Dong Q. Chain-based dfa deflation for fast and scalable regular expression matching using tcam. In: ACM/IEEE symposium on architectures for networking and communications systems (ANCS); 2011. p. 24–35.

Qi Y, Wang K, Fong J, Xue Y, Li J, Jiang W, Prasanna V. Feacan: front-end acceleration for content-aware network processing. In: IEEE INFOCOM; 2011. p. 2114–22.

Regular expression processor, ⟨http://regex.wustl.edu/⟩, 2014.

Ren X, Liu Z, Qi Y, Li J, Teng S. Sampling-based smoothed analysis for network algorithm evaluation. In: Proceedings of IEEE GLOBECOM; 2013. p. 9–13.

Roesch M. Snort-lightweight intrusion detection for networks. In: USENIX conference on system administration (LISA); 1999. p. 229–38.

Rohrer J, Atasu K, van Lunteren J, Hagleitner C. Memory-efficient distribution of regular expressions for fast deep packet inspection. In: IEEE/ACM international conference on hardware/software codesign and system synthesis (CODES+ISSS); 2009. p. 147–54.

Smith R, Estan C, Jha S. Xfa: faster signature matching with extended automata. In: IEEE symposium on security and privacy (SSP); 2008. p. 187–201.

Smith R, Estan C, Jha S, Kong S. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In: ACM SIGCOMM; 2008. p. 207–18.

Snort, ⟨http://www.snort.org/⟩, 2014.

Sommer R, Paxson V. Enhancing byte-level network intrusion detection signatures with context. In: ACM conference on computer and communications security (CCS); 2003. p. 262–71.

The bro network security monitor, ⟨http://www.bro.org/⟩, 2014.

Thompson K. Regular expression search algorithm. Commun ACM 1968;11(6):419–22.

Tuck N, Sherwood T, Calder B, Varghese G. Deterministic memory-efficient string matching algorithms for intrusion detection. In: IEEE INFOCOM; 2004. p. 2628–39.

van Lunteren J, Guanella A. Hardware-accelerated regular expression matching at multiple tens of gb/s. In: IEEE INFOCOM; 2012. p. 1737–45.

Wang K, Li J. Towards fast regular expression matching in practice. In: ACM SIGCOMM; 2013. p. 531–2.

Yang YE, Prasanna VK. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In: IEEE INFOCOM; 2011. p. 1853–61.

Yu F, Chen Z, Diao Y, Lakshman TV, Katz RH. Fast and memory-efficient regular expression matching for deep packet inspection. In: ACM/IEEE symposium on architectures for networking and communications systems (ANCS); 2006. p. 93–102.