Contents lists available at ScienceDirect

Computer Communications

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

Practical regular expression matching free of scalability and performance barriers

Kai Wang[a,b,*], Zhe Fu[a,b], Xiaohe Hu[a], Jun Li[b,c]

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

A R T I C L E I N F O

Article history:
Received 14 March 2014
Received in revised form 3 July 2014
Accepted 10 August 2014
Available online 19 August 2014

Keywords:
Regular expression matching
Deterministic finite automata (DFA)
Signature-based intrusion detection

A B S T R A C T

Regular expressions (regexes) provide rich expressiveness to specify the signatures of intrusions and are widely used in contemporary network security systems for signature-based intrusion detection. To perform very fast regex matching, deterministic finite automata (DFA) has been the first choice because its time complexity is constant $O(1)$. Unfortunately, DFA often suffers the well known state explosion problem and, consequently, tends to require prohibitive memory overhead in practical applications. To address the problem, a wide variety of DFA compression techniques have been proposed; however, few can keep up with the ever increasing network traffic bandwidth and regex set complexity. This paper proposes that the DFA problem is rooted in regexes (rather than in DFA), i.e., semantic overlapping of regexes, and accordingly presents a complete algorithmic solution *PaCC* (Partition, Compression, and Combination), that can transform the given large-scale set of complex regexes into a compact and fast matching engine using DFA as its core. PaCC fundamentally defuses state explosion for DFA by partitioning complex regexes into overlapping-free segments. By exploiting the massive repetitiveness among the resulting segments, PaCC can further deflate corresponding DFA in terms of the number of states. Moreover, on the basis of the characteristics of these segments, PaCC takes a tailor-made compression approach and reduces over 96% of the state transitions for the corresponding DFA. In the final matching engine, the combination of DFA and a small relation mapping table, built from segments and their syntagmatic relations, respectively, guarantees high performance and semantic equivalence. Experimental evaluation shows that PaCC produces succinct matching engines with memory usage proportional to the size of the real-world Snort and Bro regex sets, with speeds of up to 1.7 Gbps per core on a HP Z220 SFF workstation with a 3.40 GHz Intel Core i7-3770.

1. Introduction

Due to their rich expressiveness, regular expressions (regexes) are flexible for specifying the ever changing signatures of intrusions. Hence, contemporary network security systems have migrated toward signature-based intrusion detection that leverages the full power of regex matching to detect the occurrence of all signatures (defined as regexes) hiding in packet payloads of malicious traffic [1–5]. Typically, in the popular Snort intrusion detection system (IDS) [3], the rule using regex "AUTH\s[^\n]{100}" describes the behavior of IMAP authentication overflow attempts, and it matches any input that contains "AUTH", then a white space, and no carriage return character in the following 100 bytes.

Given a set of regexes, they can be compiled into a nondeterministic finite automata (NFA) to perform regex matching [6]. NFA has a succinct data structure, and its size (i.e., number of states) grows linearly with the size (i.e., number of characters) of the regex set. However, NFA runs very slowly because it needs to process a mass of concurrent state transitions for each input character. Generally, it is necessary to further convert NFA into deterministic finite automata (DFA) [6]. Compared with NFA, DFA requires only one state transition lookup per input character (i.e., $O(1)$ time complexity). Consequently, DFA always run fast and is the first choice for practical applications. Unfortunately, along with the increased size of the regex set, DFA often exhibits exponential growth, i.e., the well known state explosion problem. For example, the DFA built from the typical regex "AUTH\s[^\n]{100}" contains

* Corresponding author at: Department of Automation, Tsinghua University, Beijing, China.

E-mail addresses: wang-kai09@mails.tsinghua.edu.cn (K. Wang), fu-z13@mails.tsinghua.edu.cn (Z. Fu), huxh10@mails.tsinghua.edu.cn (X. Hu), junl@tsinghua.edu.cn (J. Li).

http://dx.doi.org/10.1016/j.comcom.2014.08.005
0140-3664/© 2014 Elsevier B.V. All rights reserved.

over 10^{13} states. Therefore, DFA tends to require prohibitive memory overhead and implementations are significantly limited.

To achieve very fast regex matching in practice, a wide variety of DFA compression techniques have been proposed: deflating DFA by regex grouping [7–9], alternative representation [10–13], hybrid construction [14–17], and transition compression [18–29]. Although these solutions mitigate the DFA problem to some extent, they still suffer from unscalable algorithms or data structures. Thus, few solutions can keep up with the ever increasing network traffic bandwidth and regex set complexity. In addition, with rapid development of Internet applications, the number of regexes developed for application-layer intrusions is increasing sharply. In Snort [3], the total number of rules using regexes has grown from 1131 (February 2006) to 13,605 (February 2014). Therefore, overcoming the scalability and performance barriers for practical regex matching remains challenging.

One can find that the root of the DFA state explosion problem lies in regexes rather than DFA, because the DFA built from exact strings does not have this problem [7]. This indicates that directly deflating DFA is not the best research approach. Differing fundamentally from previous work, in this paper, we focus on eliminating the threat of state explosion at the source, i.e., regexes themselves, to lay the foundation for constructing scalable DFA. To this end, we dissect regex syntaxes in depth, and conclude and prove that the DFA state explosion problem is caused by semantic overlapping of regexes. Accordingly, we present *PaCC* (Partition, Compression, and Combination), a complete algorithmic solution that transforms the given large scale set of complex regexes into a fast and succinct matching engine, using a pair of twinned DFAs as its core.

Fig. 1 illustrates the PaCC solution in a nutshell. In contrast with direct DFA deflation, PaCC first partitions all regexes in the given set into two categories of segments (see the partition procedure in Fig. 1), where no semantic overlapping exists; consequently, state explosion never occurs when converting either category of segments into corresponding DFA. The syntagmatic relations of the resulting segments are also recorded for final restoration of the original regexes. Then, PaCC constructs non-explosive DFAs for the two categories of segments, respectively (see the compilation procedure in Fig. 1). Benefiting from the partition, there can also be massive repetitiveness among the generated segments; thus, PaCC can further deflate corresponding DFAs in terms of significantly fewer states. Furthermore, on the basis of the twinned DFAs natures, PaCC adopts a tailor-made compression method to reduce their redundant state transitions dramatically. Finally, PaCC relies on its matching engine (see the matching procedure in Fig. 1), the combination of the two compact DFAs and a small relation mapping table (RMT), to guarantee the high performance and semantic equivalence for runtime matching.

In summary, the main contributions of this paper are as follows.

1. The root of DFA state explosion is introduced and proved to be semantic overlapping of regexes. In addition, for the first time, quantitative descriptions of the state inflation caused by typical semantic overlapping are provided to help predict the number of states for the DFA of corresponding regexes.
2. A complete algorithmic framework, PaCC, which includes partition, compilation, and matching procedures, is presented. Differing from previous work, PaCC leverages a predefined partition procedure to defuse the DFA state explosion problem for the compilation procedure in advance.
3. Against semantic overlapping of regexes, a universal partition mechanism, which is described briefly in [30], is proposed. The partition can render the resulting segments free of semantic overlapping and make them as repetitive as possible; thus, corresponding DFAs have equal or even a far smaller number of states than the NFA built from the original regexes.
4. Against the transition duplication of DFAs, a tailor-made compression approach is designed. The compression can reduce the size of the DFAs built from the segments by a stable 96% or greater; thus, the resulting DFAs can be placed completely in modern CPU last-level cache (LLC) to facilitate speedup of matching.
5. Against the semantic separation resulting from the partition procedure, a semantics-equivalent matching engine is proposed. The engine guarantees no false positives or false negatives based on the RMT and achieves fast processing speed based on the twinned DFAs.

To verify effectiveness of the PaCC solution, our experimental evaluation uses real-world regex sets (20–2000 regexes) obtained from open source Snort [3] and Bro [5], as well as synthetic regex sets (5000 regexes) generated from the public regex processor [31]. The results demonstrate that PaCC outperforms NFA relative to memory footprint and construction time, and achieves matching speed that is comparable with DFA at the same time. Compared with the state-of-the-art and practical solution Hybrid-FA [14,31], PaCC leads by up to two orders of magnitude in spatial and temporal performance. In particular, PaCC produces compact matching engines (2–700 KB) for all real-world regex sets (0.2–500 KB) from Snort and Bro at up to 1.7 Gbps per core on a HP Z220 SFF workstation with a 3.40 GHz Intel Core i7-3770.

The remainder of this paper is organized as follows. We first summarize related work in Section 2. In Section 3, we introduce

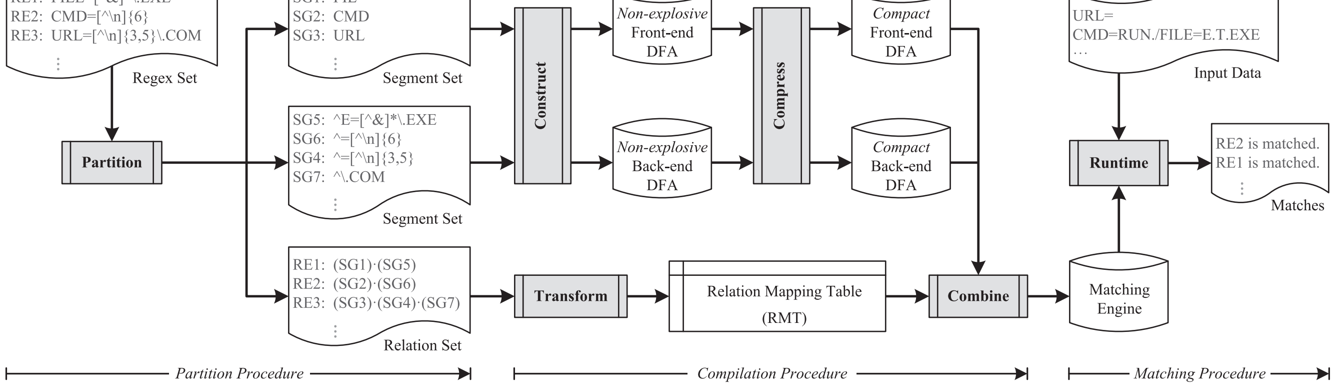


Fig. 1. An overview of PaCC solution for handling the exemplified regex set: “FILE=[ˆ&]*\.EXE”, “CMD=[ˆ\n]{6}”, and “URL=[ˆ\n]{3,5}\.COM”.

semantic overlapping of regexes and analyze its influence on DFA state explosion. In Sections 4–6, we describe the regex partition, engine compilation and runtime matching procedures of PaCC, respectively. Experimental results are shown in Section 7, and we present conclusions in Section 8.

2. Related work

In the past decade, related work in the area of regex matching has focused on addressing the speed problem of NFA or the memory problem of DFA. FPGA-based NFA implementation [32–39] exploits the parallelism of programmable logic arrays to speed up NFA. However, these approaches encode NFA in hardware logics; thus, the online update requirement is difficult to meet due to the need to re-synthesize the logics. There are also schemes that utilize the massive parallel processing power of GPUs to accelerate NFA [40,41]; however, their performance is limited to the I/O bandwidth. Thus, the scalability cannot be guaranteed.

Because DFA is fast and requires only a single state of execution, which reduces the per-flow state maintained due to the packet multiplexing in network links, it has attracted a considerable amount of attention. Generally, the related solutions can be categorized into four directions: regex grouping [7–9], alternative representation [10–13], hybrid construction [14–17], and transition compression [18–29].

Yu et al. [7] first proposed the regex grouping method. This method uses greedy heuristics to divide the given set of regexes into the fewest groups where the corresponding DFAs have tolerable state inflation, and runs independent DFA for each group of regexes. This can efficiently trade decreased speed by a factor of tens for reduced memory by several orders of magnitude. Rohrer et al. [9] converted the grouping problem into an energy minimization problem, and employed a simulated annealing algorithm to improve the spatial and temporal performance tradeoff. Majumder et al. [8] presented the grouping algorithms to reduce the runtime cache cost of the DFA corresponding to each group. However, regex grouping solutions are not scalable in performance because the number of groups required to avoid state explosion largely depends on the scale and complexity of the given regex set. Even more critical, they cannot handle the DFA state explosion problem caused by a single regex.

Kumar et al. [10] presented history based finite automata (H-FA), an alternative representation of regexes without incurring state explosion. H-FA incorporates auxiliary flag/counter variables to replace the propagated states in DFA, substitutes the variable calculation for the state transition of the replaced states, and uses variable values and an active state to track the matching history. Becchi et al. [11] extended H-FA to Counting-FA, to support regexes that cause DFA state explosion individually. Smith et al. [42,12] proposed extended finite automata (XFA), a formal model using state variables. Compared with H-FA and Counting-FA, XFA has specific mathematic definition, and its auxiliary variables and manipulating instructions have unified coding. Thus, it can be systematically constructed and executed. The alternative representations achieve good performance in terms of memory space versus run time tradeoff; however, they are all ad hoc for various regexes and have no comprehensive construction. Therefore, it is limited to transform a large scale regex set to general H-FA, or Counting-FA or XFA with a manageable number of variables and states.[1] Besides, in terms of practicability, their construction does require a prohibitive duration for large scale regex sets.

Becchi et al. [14] introduced hybrid finite automata (Hybrid-FA). 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 state explosion by retaining other NFA states as a set of tail-NFAs. Liu et al. [17] 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. Such designs are effective when the process concentrates on the DFA part, but will incur great performance reduction when the process is enslaved to the NFA/LFA part. Xu et al. [15] presented *i*-DFA, which splits all possible combinations of NFA active states into *i* subsets to minimize the total number of distinct combinations, and generates a single *i*-DFA for each subset. Yang et al. [16] presented semi-deterministic finite automata (SFA), which uses "state grouping" heuristics to cluster all NFA states into the fewest subsets where corresponding NFA states have no pair-wise conflicts. SFA consists of the constituent DFAs (c-DFAs) converted from each subset of NFA states. Both *i*-DFA and SFA are good models to find balanced combination of NFA and DFA; however, the algorithms used to judge state grouping is so complex that using them to implement large scale regex matching is infeasible in practice.

Transition compression algorithms [18–29] focus on eliminating redundant transitions, which are identical inside or among the states, to reduce the memory consumption of DFA. D^2FA proposed by Kumar et al. [19] and A-DFA proposed by Becchi et al. [21], which can achieve over 90% compression ratio, are the typical algorithms to remove the duplicated transitions among states by introducing the default transition. RCDFA proposed by Antonello et al. [29] is the representative algorithm to compress the consecutively identical transitions inside states into the single ranged transition, with over 97% compression ratio and no additional memory lookups. The transition compression solutions only work on the premise that DFA has no state explosion. In addition, their compression performance is not very stable and fluctuates with different regex sets.

Although considerable contributions have been made in prior work, previous solutions still suffer from unscalable algorithms or data structures, and thus cannot effectively tackle scalability and performance challenges comprehensively. Distinguished from prior work, we address the DFA problem from its root by partition, and thus build a foundation for both linear scalability and high performance. In addition, differing from current transition compression algorithms, our compression method fully exploits the features of partition to achieve a stable and high compression ratio. Table 1 summarizes the brief comparison between previous work and ours.

3. Regular expression dissection

Motivated by the fact that the DFA built from exact strings does not exhibit the state explosion problem at all [7], in this section, we analyze the characteristics of regexes in depth, and discuss the intrinsic relationship between regex semantics and DFA state explosion.

In advance, to make the textual description of this paper more clear, the relative notations and terms are listed in Table 2.

3.1. Semantic overlapping

Compared with exact strings, regexes introduce two operators Kleene closure (*) and alternation ($|$), in addition to the identical operator concatenation ($\cdot$). Therefore, they provide an expressive power that far exceeds exact strings. For example, α^* stands for a

[1] It is not proved that XFA can handle the DFA state explosion problem caused by a single regex [42]. Note that the regex "\ncmd[^\n]{200}" exemplified for XFA [12] does not have a state explosion problem (both its NFA and DFA have only 205 states, while the DFA of "\rcmd[^\n]{200}" has over 5×10^{28} states).

Table 1
Comprehensive Comparison for prior work and PaCC solution.

Solution	Main idea	Space consumption	Matching speed	Updating efficiency	Scalability	Practicality
FPGA-based NFA implementation [32–39]	Designing logic blocks to achieve parallel and pipelined NFA processing	Small, based on NFA	Fast for simple regex sets	Low, due to the requirement for re-placement-and-routing of logics	Limited to the number and fan-in/fan-out of logic gates	Limited to the updating efficiency
GPU-based NFA Acceleration [40,41]	Exploiting the parallelism of NFA processing to achieve massive-thread acceleration	Small, based on NFA	Fast for simple regex sets	Moderate, determined by the memory layout of NFA data structure	Limited to the cost of context switching and synchronization	Limited to the I/O bandwidth
Regex grouping [7–9]	Trading decreased speed for reduced space by dividing regexes into appropriate groups to build independent DFAs	Moderate, due to the mitigated state inflation in the DFA of each regex group	Moderate, due to the multiple DFAs running together	Low, due to the DFA construction for every single regex and pair of regexes	Limited to the scale and complexity of regex sets	Limited to the unsupported types of regexes, for example "AUTH\s[^\n]{100}"
Alternative representation [10–13]	Replacing the propagated states in DFA with auxiliary variables and instructions	Small, due to no state explosion	Moderate, due to massive instruction fetching and variable calculation	Low, due to the complex algorithms and data structures	Limited to the complexity of regex sets	Limited to the unsupported types of regexes, for example "AUTH\s[^\n]{100}", "SITE.*EXEC"
Hybrid construction [14–17]	Constructing the hybrid finite automata as a tradeoff between DFA and NFA	Moderate, due to the DFA part	Moderate, due to the NFA part or the multiple DFAs running together	Low, due to the calculation of pair-wise NFA state conflicts in preprocessing	Limited to the scale and complexity of regex sets	Practical, but may limited to the preprocessing complexity
Transition compression [18–29]	Eliminating the transition duplication inside and among states of DFA	Small for simple regex sets, due to great compression ratio	Guaranteed cost of additional memory lookups	Moderate, mainly relative to the DFA construction time	Limited to the complexity of regex sets	Limited to the premise that DFA must be non-explosive
PaCC	First defusing DFA state explosion at its source by partitioning the regexes, last representing the partitioned regexes based on DFA	Small, due to no state explosion and even less states than NFA	Fast, based on DFA	High, superior to NFA	Scalable, in terms of the space comparable with NFA and the speed close to DFA	Practical, due to low complexity

string with zero or more α characters, and $[\alpha\beta]$, i.e., $(\alpha|\beta)$, corresponds to an optional set of characters α and β.

Based on the two additional operators, regexes can express $\phi^*, \phi\{n\}, \phi\{m,n\}$, and $\phi\{n,\}$, where ϕ is the union set of characters in the alphabet Σ, and $m \in Z^*, n \in Z^+$, and $m < n$. In this paper, we refer to these four types of syntaxes as *overlapping factors* (OFs).

The regexes used to describe signatures of real-world intrusions regularly have the form $ES_1 \cdot OF_1 \cdot ES_2 \cdot OF_2 \cdots ES_K$, where $K \in Z^+$, and ES denotes exact sub-strings (e.g. the regex "AUTH\s[^\n]{100}"). This is because various attack behaviors or malicious code contain one or more partial stages or sections, and regexes can use ESs to denote the independent sub-signatures of each stage or section while leveraging OFs to specify the logic and position relationship of the sub-signatures [44]. Obviously, it is just the OFs generate the flexibility and rich expressiveness required to specify intrusions. However, according to prior work [14,12] and our analysis, OFs also produce side effects, i.e., semantic overlapping. Thus, regexes can cause corresponding DFA to suffer the state explosion problem. In this paper, we define semantic overlapping as follows:

Definition 1. Semantic overlapping means for given regex set S_{RE}, $\exists$ overlapping semantics Λ, the DFA for S_{RE} must use additional state copies to represent Λ in comparison with corresponding NFA. Otherwise, it can cause false negatives in regex matching.

A simple regex example to illustrate semantic overlapping is "ABCD[^D]*EFG", where the OF "[^D]*" can cause semantic overlapping coupled with the prefix ES "ABC", because this regex must be able to recognize two types of semantics, i.e., whether or not the ES 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 nondeterministic 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 ES "ABC" occurs when the OF "[^D]*" 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.

3.2. Relationship to state explosion

Without loss of generality, and to quantitatively describe the impact of semantic overlapping on DFA state inflation, six typical types of regexes with form $ES_1 \cdot OF \cdot ES_2$ are analyzed, where $|ES_1| > 0, |ES_2| \geqslant 0$, and all characters in both ES_1 and ES_2 are distinct by default.

3.2.1. Type 1: $ES_1 \cdot \phi^* \cdot ES_2$

Suppose all characters of ES_1 are included in the character set ϕ of the OF ϕ^*. The exact number of DFA states $F(^*)$ in this type is calculated as follows.

$$F(^*) = |ES_1| + |ES_2| + 1 \tag{1}$$

Table 2
Summary of the prescribed notations and terms.

Notation	Explanations
Σ	ASCII character set
α, β	The literal characters belonging to the alphabet Σ
$\cdot$	Concatenation operator in regex, $\alpha \cdot \beta$ is equivalent to the string $\alpha\beta$
$\vert$	Alternation operator in regex, $\alpha\vert\beta$ matches either the character α or the character β
*	Kleene Closure operator in regex, α^* matches zero or more consecutive α
[]	Character set operator in regex, $[\alpha\beta]$ is equivalent to $(\alpha\vert\beta)$
{}	Counting constraint operator in regex, $\alpha\{n\}$ matches n consecutive α; $\alpha\{n,\}$ matches at least n consecutive α; $\alpha\{m,n\}$ matches m-to-n consecutive α ($m \in Z^*, n \in Z^+$, and $m < n$)
.	Wildcard character in regex, .* matches arbitrary strings with any ASCII character
\	Escape character in regex, \n matches the carriage return character and \. only matches the dot mark
^	Anchor symbol in regex,^$\alpha\beta$ only matches the first segment $\alpha\beta$ of the string $\alpha\beta\alpha\beta$; [^$\alpha\beta$] matches any ASCII character except α and β
ϕ	Character set, including the proper subset (with form $[c_1c_2\cdots c_k]$ or $[\hat{}c_1c_2\cdots c_k]$, where $c_k \in \Sigma$) and the universal set (wildcard)
Λ	Regex semantics, the strings that the regex can match
$\vert\ \vert$	The size of the object, $\vert\Sigma\vert$ means the number of characters in Σ; $\vert$ES$\vert$ means the length of the exact string (ES)
▲, ▼	Tags used for regex partition, indicating which segment set the corresponding segment belongs to
$F()$	The exact number of states in minimized DFA [43]

Term	Descriptions
PaCCE	The final matching engine generated from PaCC
PaCCE-xC	PaCCE without executing transition compression for the DFA part
RMT	Relation Mapping Table, a part of PaCCE/PaCCE-xC
RE	Regex
SG	Segment
OF	Overlapping Factor, a part of regex only including $\phi^*, \phi\{n\}, \phi\{m,n\}$ and $\phi\{n,\}$
ES	Exact sub-String, a part of regex excluding OF
FDFA	Front-end DFA, generated from the segment set containing all the first segments of regexes as well as the unanchored non-first segments
BDFA	Back-end DFA, generated from the segment set containing the anchored non-first segments
SC	State Cluster in compact FDFA/BDFA
DS	Default State of the SC in compact FDFA/BDFA
LS	Labeled State of the SC in compact FDFA/BDFA
MT	Major Transition of the DS in compact FDFA/BDFA
DT	Default Transition of the LS in compact FDFA/BDFA
LT	Labeled Transition of the DS or LS in compact FDFA/BDFA
SU	State Unit, consisting of a state number and a bitset

In this case, DFA has the same number of states as NFA; thus, no state inflation occurs. However, if there exists a character α that is excluded in ϕ (α is the kth character of ES_1, and $1 < k \leqslant |ES_1|$), then $F(^*)$ will be changed as follows.

$$F(^*) = |ES_1| + |ES_2| + k \tag{2}$$

In this case, the OF ϕ^* can cause semantic overlapping coupled with ES_1; thus, constant state inflation occurs in corresponding DFA (e.g. the DFA of the regex "ABCD[^D]*EFG" has 11 states).

3.2.2. Type 2: $ES_1 \cdot \phi\{n\} \cdot ES_2$

Suppose all characters of ES_1 and ES_2 are included in the ϕ of the OF $\phi\{n\}$. The exact number of DFA states $F(n)$ in this type can be calculated according to the following formulas.

$$F(0) = |ES_1| + |ES_2| + 1 \tag{3}$$

When $1 \leqslant n < |ES_1|$,

$$F(n) = F(n-1) + n + 1 \tag{4}$$

If $|ES_1| < |ES_2|$, then when $|ES_1| \leqslant n < |ES_2|$,

$$F(n) = F(n-1) + F(n - |ES_1|) + n - |ES_2| \tag{5}$$

When $n \geqslant \max\{|ES_1|, |ES_2|\}$,

$$F(n) = F(n-1) + F(n - |ES_1|) \tag{6}$$

Although the explicit solution of $F(n)$ cannot be deduced directly, it can be calculated via a half-recursion program with viable computational complexity. For example, to calculate $F(100)$, the values from $F(50)$ to $F(50 + |ES_1| - 1)$ could be first calculated and buffered based on the recursive formulas above. Then the value $F(100)$ is calculated via the same recursion using the buffered intermediate values as the initial conditions.

For $n \geqslant |ES_2| \geqslant |ES_1| = 1$, the explicit solution of $F(n)$ can be derived as follows:

$$F(n) = 2^{n+2} - 2^{n-|ES_2|+1} \tag{7}$$

For the DFA of the regex in this type, the size increases exponentially with n (e.g. the DFA of the regex "AUTH\s[^\n]{100}" contains 10,343,812,679,475 states) because the overlapping semantics include every combination in which $|ES_1|$ fully or partially occurs at all different places in the string matching the OF $\phi\{n\}$. This is typical exponential state explosion caused by semantic overlapping.

Formulas 3–6 are valid if the first character in ES_2 is different from all characters of ES_1, and there are no identical prefixes and suffixes in ES_1. Otherwise, the actual $F(n)$ will be larger than calculated. By leveraging these formulas, it is easy to predetermine whether the DFA of a certain regex can be generated with a given size constraint.[2]

3.2.3. Type 3: $ES_1 \cdot \phi\{m,n\} \cdot ES_2$

Suppose all characters of ES_1 and ES_2 are included in the ϕ of the OF $\phi\{m,n\}$, and $|ES_1| > 1$ and $|ES_2| \geqslant 1$. The exact number of DFA states $F(m,n)$ in this type can be calculated using the following formulas.

[2] Interestingly, if $|ES_1| = 2, |ES_2| = 3$, for example "AB.{n}CDE", then $F(1) = 8, F(2) = 13, F(3) = 21, \ldots$, and such a sequence of $F(n)$ is equivalent to a Fibonacci Sequence.

$$F(-1,0) = |ES_1| + |ES_2| + 1; F(0,1) = |ES_1| + |ES_2| + 4 \quad (8)$$

When $1 \leqslant m+1 < |ES_1|$,

$$F(m,m+1) = F(m-1,m) + m + 2 \quad (9)$$

If $|ES_1| < |ES_2|$, then when $|ES_1| \leqslant m+1 < |ES_2|$,

$$F(m,m+1) = F(m-1,m) + F(m-|ES_1|) + m - |ES_2| \quad (10)$$

When $m+1 \geqslant \max\{|ES_1|, |ES_2|\}$,

$$F(m,m+1) = F(m-1,m) + F(m-|ES_1|, m-|ES_1|+1) \quad (11)$$

Then

$$F(m,n) = F(m,m+1) + (|ES_1| + |ES_2|)(n-m-1) \quad (12)$$

Like Type 2, Type 3 is another semantic overlapping that can cause DFA state explosion, and the explicit solution of $F(m,n)$ can also be calculated via a half-recursion program. Formula 3–6 hold when the first character in ES_2 is different from all characters of ES_1 and there are no identical suffixes and prefixes in ES_1. Otherwise, the actual $F(m,n)$ will be larger than calculated. From these formulas, it can be observed that $F(m,n)$ in Type 3 is not necessarily larger than $F(n)$ in Type 2 for the same n, although the regex of Type 3 can be divided into $n-m+1$ regexes of Type 2 in regex syntaxes (i.e., regexes from $ES_1 \cdot \phi\{m\} \cdot ES_2$ to $ES_1 \cdot \phi\{n\} \cdot ES_2$).

3.2.4. Type 4: $ES_1 \cdot \phi\{n,\} \cdot ES_2$

Suppose all characters of ES_1 are included in the ϕ of the OF $\phi\{n,\}$. The exact number of DFA states $F(n)$ in this type is calculated as follows.

$$F(n) = |ES_1| + |ES_2| + n + 1 \quad (13)$$

In this case, DFA has the same number of states as NFA; therefore, no state inflation occurs.[3] However, if there exists a character α that is excluded in ϕ (α is the kth character of ES_1, and $1 < k \leqslant |ES_1|$), then $F(n)$ is calculated as follows.

When $k \leqslant n$,

$$F(n) = |ES_1| + |ES_2| + k(2n-k+3)/2 \quad (14)$$

When $k > n$,

$$F(n) = |ES_1| + |ES_2| + k + n(n+1)/2 \quad (15)$$

In these two cases, the OF $\phi\{n,\}$ can cause semantic overlapping coupled with ES_1, and thus linear or polynomial state inflation occurs in corresponding DFA.

3.2.5. Type 5: $\hat{}ES_1 \cdot \alpha^* \cdot \phi\{n\} \cdot ES_2$

Suppose all characters of ES_2 and the character α are included in the ϕ of the OF $\phi\{n\}$. The exact number of DFA states $F(n)$ in this type is calculated as follows.

$$F(n) = |ES_1| + (n - |ES_2| + 1)(n - |ES_2| + 2)/2 - |ES_2|^2 \quad (16)$$

In regex syntaxes, the first caret indicates that the match of the regex must be at the start of the string the regex is applied to. Type 5 is also common in real-world regex sets [3,7]. For this type, the OF α^* makes the suffix unanchored, and thus the OF $\phi\{n\}$ can cause semantic overlapping coupled with ES_2. Accordingly, polynomial state inflation occurs in corresponding DFA (e.g. the DFA of the regex "^SEARCH\s+[^\n]{1024}" in Snort has 525,832 states).

3.2.6. Type 6: $\bigcup_{k=1}^{n}(ES_{k1} \cdot \phi^* \cdot ES_{k2})$

Consider n regexes with form $ES_{k1} \cdot \phi^* \cdot ES_{k2}$ ($1 \leqslant k \leqslant n$), where all characters of ES_{k1} and ES_{k2} are included by the ϕ. Suppose all first characters of ES_{k1} and ES_{k2} are distinct. Then, the exact number of DFA states $F(n)$ satisfies the following formula.

$$F(n) = \left[2 + \sum_{k=1}^{n}(|ES_{k1}| + |ES_{k2}| - 1)\right] \cdot 2^{n-1} \quad (17)$$

Although the regexes of Type 1 cannot lead to exponential state explosion individually, they can cause it when compiled into a composite DFA, because the OF in each regex can cause semantic overlapping together with the ESs in other regexes. In fact, similar to Type 6, regexes in previous types can also cause semantic overlapping with each other, which results in greater state increase.

3.3. Life conditions

Previous quantitative analysis shows that semantic overlapping is the origin of DFA state explosion, and an OF is a necessary condition to produce semantic overlapping. However, it is not the sufficient condition.

A common regex with form $ES_1 \cdot OF \cdot ES_2$ can be expressed in the equivalent form $ES_{11} \cdot ES_{12} \cdot OF_1 \cdot OF_2 \cdot ES_2$, where $ES_1 = ES_{11} \cdot ES_{12}$, $OF = OF_1 \cdot OF_2$, and $ES_1 > 0$, and OF_1 and OF_2 are not null. For OF ϕ^*, it can be denoted $\phi^* \cdot \phi^*$; for OF $\phi\{n\}$, it can be denoted $\phi\{k\} \cdot \phi\{n-k\}$ $(0 < k < n)$; for OF $\phi\{m,n\}$, it can be denoted $\phi\{k\} \cdot \phi\{m-k, n-k\}$ $(0 < k < m)$; for OF $\phi\{n,\}$, it can be denoted $\phi\{k\} \cdot \phi\{n-k,\}$ $(0 < k \leqslant n)$. Then, we provide Theorem 1.

Theorem 1. *For a regex $ES_1 \cdot OF \cdot ES_2$, if $\exists$ $ES_{11} \subset OF_1$, and $\exists$ $ES_\alpha \not\subset OF_2$, where $\alpha \in ES_1$, and ES_α consists of countless α, then the regex has semantic overlapping.*

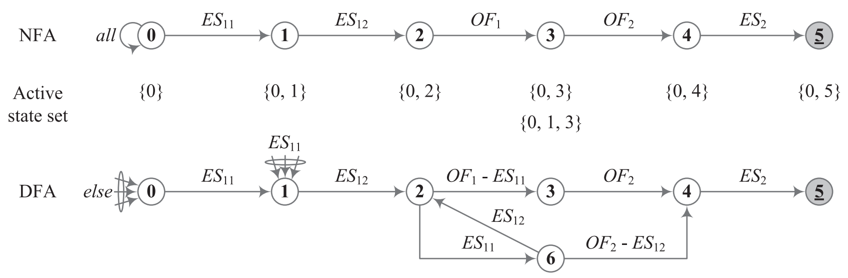
Proof. Construct the NFA for $ES_{11} \cdot ES_{12} \cdot OF_1 \cdot OF_2 \cdot ES_2$ (Fig. 2(a)). $ES_{11} \subset OF_1$ (i.e., ES_{11} can match OF_1); thus, in the NFA, states 0, 1, and 3 can be activated simultaneously when ES_{11} occurs in the string that matches OF_1. Another regular case is when only states 0 and 3 are active. In addition, due to $ES_\alpha \not\subset OF_2$, the next-hop state set of {0,1,3} must be different from that of {0,3}. This makes the two state sets {0,1,3} and {0,3} not equivalent; thus, they cannot be merged in the final minimized DFA. Consequently, the exhaustive result of all possible active state sets in NFA are {0}, {0,1}, {0,2}, {0,3}, {0,1,3}, {0,4}, and {0,5}, and corresponding DFA have the additional state (state 6) corresponding to {0,1,3}. In this situation, the OF_1 coupled with ES_{11} makes the regex fall into semantic overlapping, and thus corresponding DFA must use extra states to record the overlapping semantics in which ES_{11} occurs in the string that matches OF_1. □

For a pair of regexes, one containing an OF has the form $ES_{11} \cdot OF_1 \cdot ES_{12}$, where $ES_{11} > 0$, and the other containing no OF has the form ES_2, i.e., $ES_{21} \cdot ES_{22}$. Thus, we formulate Theorem 2.

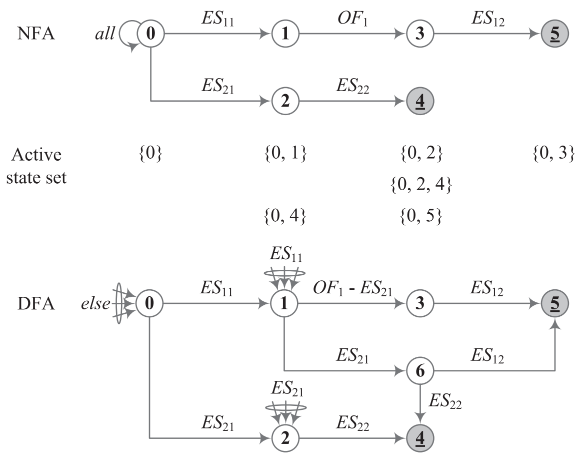
Theorem 2. *For two regexes $ES_{11} \cdot OF \cdot ES_{12}$ and ES_2, if $\exists$ $ES_{21} \subset OF$, then the two regexes have semantic overlapping.*

Proof. Construct the NFA for $ES_{11} \cdot OF \cdot ES_{12}$ and $ES_{21} \cdot ES_{22}$ (Fig. 2(b)). $ES_{21} \subset OF$; thus, in the NFA, states 0, 2, and 4 can be activated simultaneously when ES_{21} occurs in the string that matches OF. Another regular case is when only states 0 and 2 are active. In addition, the next-hop state set of {0,2,4} could correspond to the match of either of the two regexes; thus, it must be different from the next-hop state set of {0,2}, which only corresponds to one regex match. In other words, the two state sets {0,2,4} and {0,2} are not equivalent and cannot be merged in the final minimized DFA. Consequently, the exhaustive result of all possible active state

[3] In fact, the unminimized DFA of the Type 4 regex has more states than $F(n)$ in Type 2 when the value of n is equal.



(a) For a single regex



(b) For a pair of regexes

Fig. 2. Schematics of NFA vs. DFA corresponding to semantic overlapping.

sets are {0}, {0,1}, {0,2}, {0,2,4}, {0,3}, {0,4}, and {0,5}, and corresponding DFA have the additional state (state 6) corresponding to {0,2,4}. In this situation, the *OF* coupled with ES_{21} makes the two regexes produce semantic overlapping; thus, corresponding DFA must use extra states to record the overlapping semantics in which ES_{21} occurs in the string that matches *OF*. □

In fact, Theorem 2 equally applies to the situation when the regex containing OF is anchored, i.e.,$\hat{\ } ES_{11} \cdot OF_1 \cdot ES_{12}$. The proof is similar to the above.

3.4. Removal

Theorems 1 and 2 indicate that, if $\nexists\ ES \subset OF$, then semantic overlapping cannot occur in regexes. Thus, excluding the first character α of the *ES* from the character set ϕ of the *OF*, i.e., $\alpha \notin \phi$, can remove semantic overlapping. For example, for the regex "AUTH\s[ˆA\n]{100}", its DFA has the same number of states (i.e., 106) as NFA. In this manner, the overlapping semantics of the original regexes are sacrificed by rewriting the OF in exchange for non-explosive DFA.

Similarly, anchoring could also help a regex eliminate semantic overlapping at the cost of the loss of semantics because, for an anchored regex with form $\hat{\ } ES_1 \cdot OF \cdot ES_2$, its ES_1 and *OF* match respective strings only once. Thus, it is not necessary to recognize the semantics even if ES_1 occurs in the strings that match the *OF*. An intuitive example is the regex "ˆAUTH\s[ˆA\n]{100}" whose DFA only has 107 states.

Although these two methods are not acceptable for exact matching, we are motivated to consider that first removing semantic overlapping to defuse DFA, and after the non-explosive DFA is built, the complete semantics of original regexes are restored. For this purpose, we present PaCC, which first partitions regexes to separate overlapping semantics, compiles them into compact DFA, and then combines the separated semantics.

4. Partition procedure

Here, we introduce the partition procedure in PaCC. The goal of partitioning is to split the original regexes with semantic overlapping into overlapping-free regex segments so that the DFA for the resulting segments never suffer the state explosion problem. To achieve this goal, a universal partition mechanism is proposed. The partition procedure produces two segment sets and one relation set based on the original regex set.

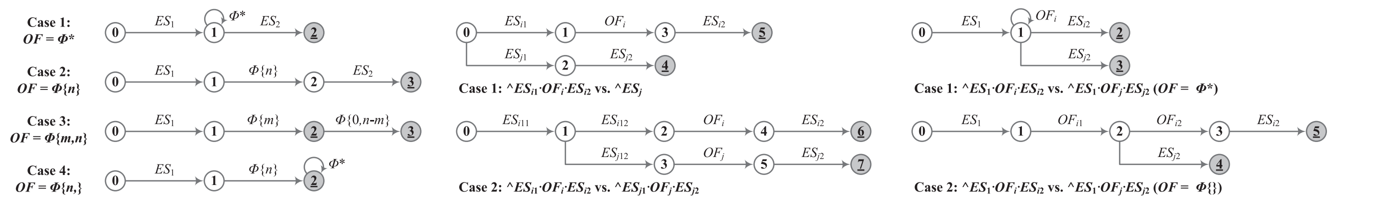
4.1. Definition

First, regex partition is formally defined as follows.

Definition 2. Regex partition is an operation that splits any complete regex into one or more literal segments in sequence, and guarantees that:

1. Each segment is a correct regex in legal syntaxes.
2. Each pair of neighbor segments is originally concatenated in the complete regex in terms of syntaxes and semantics.
3. Each segment is anchored by adding a caret at its beginning, with the exception of the first segment of the complete regex.

This definition determines the essential premise for performing segmentation of regexes, without respect to its effect against semantic overlapping. Above all, the first item ensures that the derived segments can be compiled validly as the original regexes.



Case 1: OF = Φ*
Case 2: OF = Φ{n}
Case 3: OF = Φ{m,n}
Case 4: OF = Φ{n,}
Case 1: ^ES_i1·OF_i·ES_i2 vs. ^ES_j
Case 2: ^ES_i1·OF_i·ES_i2 vs. ^ES_j1·OF_j·ES_j2
Case 1: ^ES_1·OF_i·ES_i2 vs. ^ES_1·OF_j·ES_j2 (OF = Φ*)
Case 2: ^ES_1·OF_i·ES_i2 vs. ^ES_1·OF_j·ES_j2 (OF = Φ{})

(a) For Principle 2 (b) For Principle 3 (c) For Principle 4

Fig. 3. Schematic NFA corresponding to overlapping-free anchored segments that obey the given principles.

For example, the regex "AUTH\s[^\n]{100}" cannot be divided into the segments "AUTH\s[^\n]" and "{100}" because "[^\n]{100}" corresponds to an atomic regex literally.

The second item limits that the partition must be at the positions of the concatenation operator, whose anteroposterior segments are concatenated in semantics. This facilitates restoration of the complete regex simply by uniformly concatenating the corresponding segments in sequence. According to this, the regex "AB.*CD|EF.*GH", i.e., "AB.*CD|EF.*GH", cannot be split into the segments "AB", ".*CD|EF", and ".*GH" because they are concatenated only in syntaxes, but not in semantics. When these segments are re-concatenated, they actually correspond to a different regex "AB.*(CD|EF).*GH". Thus, for such a regex with alternation (whose priority is lower than concatenation), the partition should first rewrite the regex into multiple alternated sub-regexes with identical regex ID, and then enforce partition for each sub-regex.

The last item introduces the anchors for the non-first segments to satisfy that the match of each of these segments must start exactly at the end of the match of its anterior segment. For example, "AUTH\s[^\n]{100}" can be split into "AUTH", "^\s", and "^[^\n]{100}". In particular, "AB.*CD" may be partitioned into "AB" and "CD", where "CD" is simplified from "^.*CD" because "anchored dot-star" is equivalent to "unanchored" in regex syntax.

Based on Definition 2, if a regex is split into only one segment, then the segment is just the regex itself. In other words, the partition is at the end of the regex. Hence, for regexes without semantic overlapping, partitioning each of them into one segment is acceptable. For those having semantic overlapping, it is necessary to partition them into at least two segments. Because the regexes with semantic overlapping must have OFs, two types of segments could result from the partition; one is just ES and has no OF, and the other still has OFs. It is known that ESs are not problematic; therefore, the partition is required to ensure that the segments with OFs are free of semantic overlapping.

4.2. Principles for overlapping-free partition

According to the analysis presented in Section 3.4, semantic overlapping of a regex could be removed if the regex 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. Note that even if anchored, the special OF $.^*$ can render its subsequent segment unanchored; hence, it should always be kept at the start of segments because "anchored dot-star" is equivalent to "unanchored". In addition, its corresponding segments should be uniformly regarded as the unanchored non-first segments and, accordingly, cannot have OFs. Similarly, the special OF $.\{n,\}$ should be changed into $.\{n\}\cdot.^*$ first, where the latter $.^*$ should be treated in the same manner.

However, the extended Theorem 2 indicates that, for arbitrary regexes having OFs, regardless of whether they are anchored or unanchored, they are very likely to cause semantic overlapping with the unanchored regexes. This means it is necessary to categorize the partition-derived segments into two sets. Consequently, the primacy principle is as follows.

Principle 1. *A given set of regexes should be partitioned into at most two sets of segments, where one contains all the first segments of regexes as well as the unanchored non-first segments derived from the OFs $.^*$ and $.\{n,\}$, and the other contains all the rest, i.e., the anchored non-first segments. Moreover, all OFs except for $.^*$ and $.\{n,\}$ are kept in the latter set of segments.*

For the former set of segments, because they have no OF, it is certain that there is no semantic overlapping. For the latter, although its segments are anchored, two more principles must be followed for the partition to guarantee that no semantic overlapping occurs.

4.2.1. Free of intra-segment overlapping

The principle for the intra-segment overlapping free partition is as follows.

Principle 2. *For each anchored segment that contains OFs, it should satisfy the form $\hat{}ES_1\cdot OF\cdot ES_2$, where $|ES_1|\geqslant 0$, and if OF is ϕ^* or $\phi\{n\}$, then $|ES_2|\geqslant 0$. Else, if OF is $\phi\{m,n\}$ or $\phi\{n,\}$, then $|ES_2|=0$.*

Principle 2 only prescribes the criteria for the anchored segments with OFs because those without OF, namely $\hat{}ES$, never cause overlapping.

To explain the validity of Principle 2, for each type of OF, corresponding NFA of $\hat{}ES_1\cdot OF\cdot ES_2$ is constructed (Fig. 3). Differing from Fig. 2(a), there is no self-loop transition in the initial state (i.e., state 0) of the NFA for an anchor regex. Therefore, regardless of the empty set, for OF ϕ^* (Case 1), the NFA can only activate state sets {0}, {1}, and {1,2} if $ES_2\subset OF$ (or {0}, {1}, and {2} if $ES_2\not\subset OF$). For OF $\phi\{n\}$ (Case 2), the NFA can only activate {0}, {1}, {2}, and {3}. For OF $\phi\{m,n\}$ (Case 3) or $\phi\{n,\}$ (Case 4), because there is no suffix ES, the number of possible active state sets in the NFA is the same as the number of NFA states.

In summary, when Principle 2 is obeyed, the anchored segments have no overlapping individually.

4.2.2. Free of inter-segment overlapping

Given the premise of Principle 2, the principle for the inter-segment overlapping free partition is as follows.

Principle 3. *For each pair of anchored segments that satisfy Principle 2 and respectively have the form $\hat{}ES_{i1}\cdot OF_i\cdot ES_{i2}$ and $\hat{}ES_{j1}\cdot OF_j\cdot ES_{j2}$, each pair should satisfy $|ES_{i1}|>0, |ES_{j1}|>0$, and $\exists\ k <= min\{|ES_{i1}|, |ES_{j1}|\}$, for the kth character α in ES_{i1} and the kth character β in $ES_{i1}, \alpha\neq\beta$.*

The anchored segments without OF have no intra-segment overlapping. In addition, they also do not have overlapping with the anchored segments satisfying Principle 2.

In the NFA of the segments $\hat{}ES_{i1} \cdot OF_i \cdot ES_{i2}$ and $\hat{}ES_j$ (Fig. 3(b), (Case 1), if $\nexists\ ES_{j1} = ES_{i1}$, then any of states 1, 3, and 5 cannot be activated simultaneously with any of states 2 and 4. In addition, Principle 2 is satisfied; thus, states 1, 3, or 5 cannot be active together at the same time (neither can states 2 or 4). Consequently, it is conclusively shown that no overlapping exists. Similarly, if $\exists ES_{j1} = ES_{i1}$, then the NFA can possibly activate {0}, {1,2}, {3}, {3,4}, and {5} when $ES_{j2} \subset OF_i$ (or {0}, {1,2}, {3}, {4}, and {5} when $ES_{j2} \not\subset OF_i$). Therefore, it is verified that the anchored segments without OF never lead to overlapping.

For the anchored segments with OFs, if Principle 3 is satisfied, then the corresponding NFA built from any two of them is shown in Fig. 3(b) (Case 2). Because the kth characters in ES_{i1} and ES_{j1} are different, $ES_{i12} \neq ES_{j12}$ (both start with the k characters), then similar to Case 1, any of states 2, 4, and 6 cannot be simultaneously activated with any of states 3, 5, and 7. Principle 2 is obeyed; thus, states 2, 4, or 6 cannot be active at the same time (neither can states 3, 5, or 7). Consequently, it is conclusively shown that no overlapping exists.

In summary, based on the partition following Principles 2 and 3, anchored non-first segments in the same set will have no internal overlapping, and will cause no external overlapping with each other. Thus, they are ensured to have no semantic overlapping as a whole.

4.3. Principle for compressive partition

The partition determined by Principles 2 and 3 can result in DFA having states equal to those of NFA for regexes. However, the following principle can relax the criteria of Principle 3 and further render DFA smaller than NFA in terms of the number of states.

Principle 4. *For each pair of anchored segments that satisfies Principles 2 and 3, and the segments have the form* $\hat{}ES_{i1} \cdot OF_i \cdot ES_{i2}$ *and* $\hat{}ES_{j1} \cdot OF_j \cdot ES_{j2}$*, they are permitted and required to satisfy* $ES_{i1} = ES_{j1}$ *if* OF_i *is equivalent to* OF_j*.*

Here, we consider that two OFs are equivalent if their character set ϕ is identical and, their type is the same (i.e., is either ϕ^* or is $\phi\{\}$). According to Principle 4, the NFA of the segments $\hat{}ES_{i1} \cdot OF_i \cdot ES_{i2}$ and $\hat{}ES_{j1} \cdot OF_j \cdot ES_{j2}$ is shown in Fig. 3(c). For $OF_i = OF_j = \phi^*$ (Case 1), the NFA can only activate the same number of state sets as the number of NFA states; thus, no extra states are required by corresponding DFA. Likewise, for equivalent OF_i and OF_j, which are not ϕ^* (Case 2), the segments also have no semantic overlapping (analogous to Case 1, Fig. 3(b)).

Three conclusions can be drawn about the effect of state deflation. First, in Cases 1 and 2, the states corresponding to the prefix ESs are reduced because they are identical and their states can be merged. Second, in Case 1, if the suffix ESs are the same, their repetitive states can also be merged. Third, in Case 2, for the OFs $\phi\{n_1\}, \phi\{n_2, n_3\}$, and $\phi\{n_4,\}$, the states will be $\max\{n_1, n_3, n_4\}$, rather than $n_1 + n_3 + n_4$ due to the reduction of repetitiveness. For example, for the set of segments “^AX[^\n]{64}ABCD”, “^BX[^\n]{48,72}”, “^CY[^\r]*ABCDEFGA”, and “^DY[^\r]*ABCDEFGB”, its DFA has 166 states. In contrast, when their prefix ESs are changed into “X”, “X”, “Y”, and “Y”, respectively, which satisfies Principle 4, then the DFA will only have 89 states. Thus, the reduced number of states is $2 + 7 + 64 + 4 = 77$.

4.4. Principle-driven universal mechanism

As a whole, based on Principles 1–4, the universal partition requires that each segment is either a segment without an OF or is an anchored segment with a prefix ES and only one OF.

Table 3
Universal partition for different types of OFs.

OF type	Partition positions (▲: add to set 1, ▼: add to set 2)
$.^*$	$\cdots ES_{pre}$ ▲$\hat{}\ .^* \cdot ES_{suf} \cdots$
$.\{n,\}$	$\cdots ES_{pre1}$▼$\hat{}ES_{pre2} \cdot .\{n\}$▲$\hat{}\ .^* \cdot ES_{suf} \cdots$
ϕ^*	$\cdots ES_{pre1}$▼$\hat{}ES_{pre2} \cdot \phi^* \cdot ES_{suf} \cdots$
$\phi\{n\}$	$\cdots ES_{pre1}$▼$\hat{}ES_{pre2} \cdot \phi\{n\} \cdot ES_{suf} \cdots$
$\phi\{m,n\}$	$\cdots ES_{pre1}$▼$\hat{}ES_{pre2} \cdot \phi\{m,n\}$▼$\hat{}ES_{suf} \cdots$
$\phi\{n,\}$	$\cdots ES_{pre1}$▼$\hat{}ES_{pre2} \cdot \phi\{n,\}$▼$\hat{}ES_{suf} \cdots$

In addition, the prefixes tend to be identical for equivalent OFs or are distinct. In fact, these principles introduce the guidelines for the writer of the signatures of intrusions to design partition-friendly regexes. In summary, Table 3 shows the partition positions for different types of OFs. In Table 3, the ▲ means the subsequent segment should be added into segment set 1, and the ▼ corresponds to segment set 2. Besides, the ES_{pre2} (the suffix of ES_{pre}) for each OF should obey Principles 3 and 4. To obtain the qualified ES_{pre2} for every requested OF, the trie tree is leveraged in the partition decision to find the best partition position.

As Fig. 4 shows, the OFs whose prefix ESs have already been found will be inserted into the trie tree. According to Principles 3 and 4, there cannot be more than one gray node in any path from the root to the leaf in the trie tree. In other words, each prefix ES must correspond to a leaf node. To determine the partition position for a new OF, its prefix ES will be traversed from the tail to the head (the first three trees in Fig. 4). If the end node (denoted as gray) for the tested ES_{pre2} is not a leaf or is the leaf of a gray node, then the tested ES_{pre2} is unqualified. Otherwise, if the end node is a present leaf node, then the new OF should be compared with the existing OF to determine whether they are equivalent. If they are equivalent, then the new OF is qualified and can be added into the trie tree. If the end node is the leaf of a non-gray node, the tested ES_{pre2} is different from all present prefix ESs and can be partitioned with certainty. The last tree in Fig. 4 shows the determined ES_{pre2} for the exemplified OF.

For the prefix ESs with at least k characters, the probability that they are identical is $(|\Sigma|^*(1/|\Sigma|)^2)^k = 1/|\Sigma|^k$. If $k = 2$ and $|\Sigma| = 256$, then $1/256^2 \approx 0.0015\%$. Therefore, it is easy to find a qualified prefix ES that obeys Principle 3 for each OF. Furthermore, in real-world regex sets, the OFs mainly concentrate on a small number of non-equivalent types. For example, in Snort [3], there are fewer than 50 non-equivalent OFs, even though the total number of these OFs exceeds 5000 and the percentage of the number of the maximum six OFs is over 95%. In addition, the prefixes of most of the equivalent OFs are identical, which allows the partition to benefit from Principle 4. For example, in Snort, the OF “[^\r\n]*” often has the prefix “\x3a”, and the OF “[^\n]{n}” commonly has the prefix “\s”. If neither of the two principles can be satisfied for some prefix ESs, the remaining state inflation, which maybe linear, is still insignificant compared with the exponential state explosion before regex partition.

In consideration of the final performance, the partition should generate a minimum number of segments to satisfy Principles 1–4. Therefore, in practice, there cannot be two adjacent segments both of which are ESs in their original regex. Further, for the OFs whose $|\phi|$ is small, such as “\s”, “\d”, their partition is ignored because they rarely lead to semantic overlapping according to Theorems 1 and 2 (at the expense of potential little state inflation). Hence, the common regex “^SEARCH\s+[^\n]{1024}” in Type 5 can be partitioned into “^SEARCH\s*” and “^\s[^\n]{1024}” to remove semantic overlapping.

4.5. Overall procedure

Algorithm 1. Partition Procedure

```
Input: Regex Set $S_{RE}$.
Output: Segment Sets $S_{SG1}$ and $S_{SG2}$, Relation Set $R_{SG}$.
Function:
1:  for all $RE \in S_{RE}$ do
2:    $RE$.rewrite_if_necessary($RE$.regex_id);
3:  end for
4:  for all $RE \in S_{RE}$ do
5:    for all $OF \in RE$ do
6:      if $OF == .^*$ || $OF == .\{n,\}$ then
7:        if $OF == .\{n,\}$ then
8:          $RE$.replace_of($OF, .\{n\}.^*$);
9:        end if
10:       $RE$.insert_tag($RE$.get_head_position($.^*$), ▲);
11:     else
12:       if $OF == \phi\{m,n\}$ || $OF == \phi\{n,\}$ then
13:         $RE$.insert_tag($RE$.get_tail_position($OF$), ▼);
14:       end if
15:       $S_{ES}$.insert($OF, RE$.get_entire_prefix_es($OF$), $RE$);
16:     end if
17:   end for
18:   $RE$.remove_tag_if_present($RE$.end());
19: end for
20: for all $ES \in S_{ES}$ do
21:   $S_{ES}$.count_for_equivalent_of($S_{ES}$.of_num, $ES.OF$);
22: end for
23: __sort($S_{ES}$.begin(), $S_{ES}$.end(), $S_{ES}$.of_num, descending);
24: for all $ES \in S_{ES}$ do
25:   $ES_{pre2}$ = __get_best_es_via_trie_tree($ES.OF, ES.ES_{pre}$);
26:   $ES.RE$.insert_tag($ES.RE$.get_head_position($ES_{pre2}$), ▼);
27: end for
28: for all $RE \in S_{RE}$ do
29:   $TAG_{begin} = RE$.begin();
30:   for all $TAG_{end} \in RE$ && $TAG_{end} \neq TAG_{begin}$ do
31:     $S_{SG}[0]$.push_back($RE$.partition($TAG_{begin}, TAG_{end}$), $RE$);
32:     $TAG_{begin} = TAG_{end}$;
33:   end for
34:   $S_{SG}[1]$.push_back($RE$.partition($TAG_{begin}, RE$.end()), $RE$);
35: end for
36: for all $0 \leqslant i \leqslant 1$ do
37:   for all $SG \in S_{SG}[i]$ do
38:     if $SG.TAG_{begin}$ == ▼ then
39:       segment_id = $S_{SG2}$.get_avail_segment_id($SG$);
40:       $S_{SG2}$.insert(segment_id, __anchor($SG$));
41:     else
42:       segment_id = $S_{SG1}$.get_avail_segment_id($SG$);
43:       if $SG.TAG_{begin}$ == ▲ then
44:         $S_{SG1}$.insert(segment_id, __anchor($SG$));
45:       else
46:         $S_{SG1}$.insert(segment_id, $SG$);
47:       end if
48:     end if
49:     $R_{SG}$.append($SG.RE$.regex_id, segment_id);
50:   end for
51: end for
```

The pseudo code of the partition procedure is shown in Algorithm 1. In this algorithm, step 2 aims to rewrite the special regex into the alternative sub-regexes with the same regex ID (i.e., *RE*.regex_id) to meet the prerequisites of partition. Steps 20–23 guarantees that most OFs that are equivalent are always prioritized in step 25 to satisfy both Principles 3 and 4. Steps 36–51 label each generated segment with a unique segment ID (i.e., segment_id), and based on the ID, the syntagmatic relations of these segments can be recorded as a relation set.

There are often many equivalent segments that, can be merged even though they are distinctive by default. For any two segments SG_{X_i} and SG_{Y_j}, where SG_{X_i} is the *i*th segment in regex *X* that has *I* segments, and SG_{Y_j} is the *j*th segment in regex *Y* that has *J* segments, we consider that SG_{X_i} and SG_{Y_j} are equivalent if they satisfy the following.

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

For example, for the two regexes “AB.*CD.*EF” and “AB.*CD.*GH”, their identical segments “CD” (and “AB”) are equivalent and can be merged into one segment because the remaining segments will correspond to a complete regex “AB.*CD.*(EF|GH)”, which is equal to the original regexes. Therefore, the segment merger based on the equivalency (step 39/42 in Algorithm 1) will greatly reduce the number of segments and improve the final performance.

The overall partition result for the exemplified regex set has been shown in Fig. 1. It is evident that the partition maintains the full semantics of the original regexes in the segment sets and relation set, but has already eliminated the semantic overlapping.

5. Compilation procedure

The foundation for scalability is built on the partition procedure described in Section 4. In this section, two sets of overlapping-free segments are constructed as non-explosive DFAs, respectively. The relationships of the segments are also transformed into a small index array, i.e., RMT. Although the state redundancy has been removed from the two resulting DFAs, the huge transition redundancy still exists due to the nature of the partition-derived segments, and thus are further reduced by an effective compression approach. Finally, a succinct matching engine can be obtained in PaCC by combining the compact DFAs and RMT.

5.1. Construction and transformation

In the construction step, two sets of segments are directly compiled into corresponding DFAs, i.e., Front-end DFA and Back-end DFA, using parse tree, Thompson construction, NFA reduction, subset construction and DFA minimization [6] consecutively. As Fig. 5 displays, each match state (denoted by underlined digit) in the two DFAs corresponds to the occurrence of segments rather than regexes. For example, the match of segment SG1 occurs when state 7 is activated. The state number of Back-end DFA starts from 10, rather than 0. This is used to distinguish Back-end DFA states from Front-end DFA states.

The total number of states in two DFAs is only 29, which is over an order of magnitude less than the inflated DFA (with 362 states) of the original regex set and even less than the linear NFA (with 35 states). The theoretical size of two DFAs is $(N_{FDFA} + N_{BDFA}) \cdot |\Sigma| \cdot log_2(N_{FDFA} + N_{BDFA})$ bits, where N_{FDFA} and N_{BDFA} are the number of states in Front-end DFA and Back-end DFA, respectively. As there is no state explosion and less state redundancy, $N_{FDFA} + N_{BDFA} \leqslant N_{RE} \cdot L_{RE}$, where N_{RE} is the number of regexes, and L_{RE} is the average length of regexes.

In the transformation step, the relation set is translated into a two-dimensional RMT indexed through the segment ID. Each row of entries in the RMT corresponds to the related information of

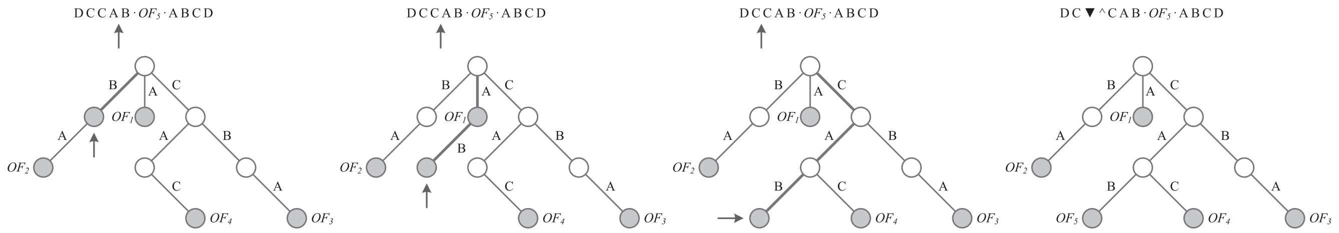


Fig. 4. Leveraging trie tree to search the best prefix ES for every OF in the given set of regexes.

the indexed segment. Here, SG_{prev} denotes the anterior segment of the inquired segment, RE_{origin} denotes the complete regex to which the inquired segment corresponds, and $STATE_{next}$ is the initial state of the DFA where the posterior segment of the inquired segment exists. For example, segment SG4 indexes the 4th line of the RMT; the anterior segment is denoted as SG3, and its complete corresponding regex is null because it is not a tail segment. The posterior segment (SG7) should be matched from state 10 of Back-end DFA.

The syntagmatic relations of segments are all maintained in the RMT. For example, according to the RMT, the anterior segments of segments SG4 and SG7 are SG3 and SG4, respectively (i.e., segment SG4/SG7 is at the back of segment SG3/SG4), and segment SG7 corresponds to the complete regex RE3. Therefore, RE3 is recoverable by concatenating SG3, SG4, and SG7 in sequence.

The theoretical size of the RMT is $N_{SG} \cdot (\log_2(N_{SG} - N_{RE}) + \log_2 N_{RE} + \log_2(N_{FDFA} + 1))$ bits, where N_{SG} is the number of segments, and $N_{SG} \ll N_{RE} \cdot L_{RE}$. It can be easily determined that the size of the RMT is much smaller than $3/|\Sigma|$ of the size of two DFAs. Although both DFAs and the RMT can be further compressed, their total size is primarily determined by $N_{RE} \cdot L_{RE}$.

5.2. Compression

For a large scale regex set whose $N_{RE} \cdot L_{RE} > 1,000,000$, even though the DFAs are succinct in terms of states, the number of states could be over 1,000,000. For this many states, at least 1 GB memory is required. Therefore, it is necessary to compress the DFAs further. According to the characters of the segments, there are a large number of redundancies in the state transitions of the DFAs that can be comprehensively removed by proper compression.

5.2.1. Transition redundancy

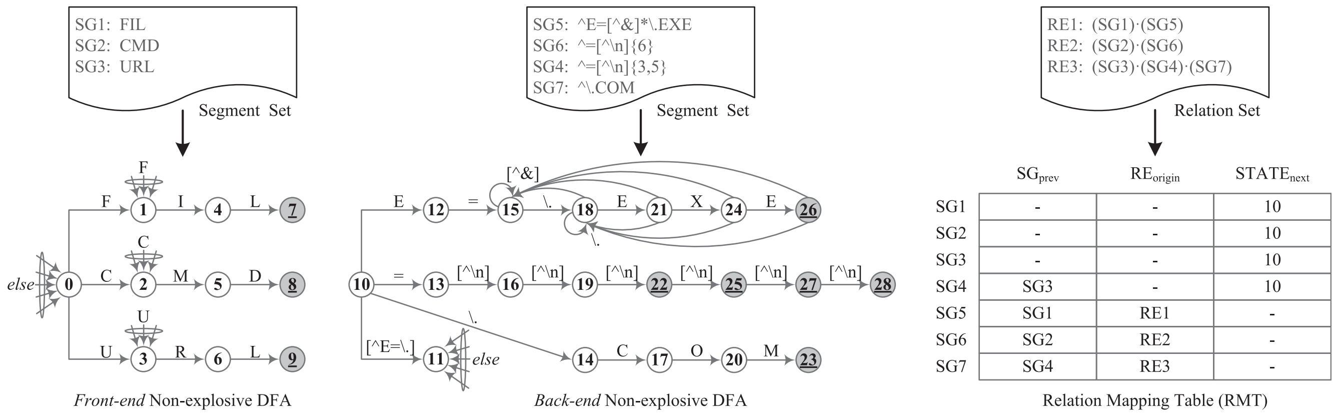
The partition procedure guarantees that all segments in set 1 have no OF, that all segments in set 2 are anchored, and each segment contains at most one OF. Therefore, Front-end DFA is construct from the strings, where the primary syntax is concatenation. In a typical case, a state must be added for each concatenated character. Except for the state transition corresponding to the significant character, the transitions of each state consistently fall back to the initial state or the neighbors of the initial state.

For example, in the Front-end DFA shown in Fig. 6, all states will transit to states 1, 2, and 3 (the neighbors of state 0) for the characters 'F', 'C', and 'U', respectively, and tend to transit to state 0 for any other character. Those distinct transitions, such as the transition to state 4 in state 1, correspond to the significant characters in the segments. Therefore, most transitions are identical for all states in Front-end DFA.

The Back-end DFA is construct from segments where the syntaxes contain anchoring and OF as well as concatenation. Anchoring makes almost all states transit to the dead state (which can only transit to itself for any character, e.g. state 11 in Fig. 5) by default. For OF ϕ^*, it can produce an intermediate state, where subsequent states (corresponding to the concatenated characters) transit for the characters in ϕ. For OF $\phi\{\}$, it can produce a chain of states, where the transitions corresponding to characters in ϕ will go forward to the next neighbor states. In addition, because the OFs often have a large ϕ, these transitions are primarily identical inside the corresponding state.

As Fig. 6 shows, the OF "[^\&]*" makes states 18, 21, 24, and 26 (including state 15) transit to the self-loop state 15 for any character except '&'. The OF "[^\n]{6}" makes states 13, 16, 19, 22, 25, and 27 transit to their next states, i.e., states 16, 19, 22, 25, 27, and 28, respectively, for any character except '\n'. Furthermore, anchoring makes all mismatched transitions transit to the dead state 11. Each OF will not intersect with the characters or other OFs for the segments; thus, the states are either very similar (e.g. states 15 and 18) or completely dissimilar (e.g. states 13 and 16).

Thus, in Front-end DFA, the single character in concatenation invariably brings about the intra-state identity of transitions back to the initial state, and the inter-state identity of transitions back to the initial state and the neighbors of the initial state. Furthermore, in Back-end DFA, the character in concatenation


Segment Set
Segment Set
Relation Set
F
I
L
C
M
D
U
R
else
E
=
[^&]
\.
X
[^\n]
[^E=\.]
O
Front-end Non-explosive DFA
Back-end Non-explosive DFA
Relation Mapping Table (RMT)

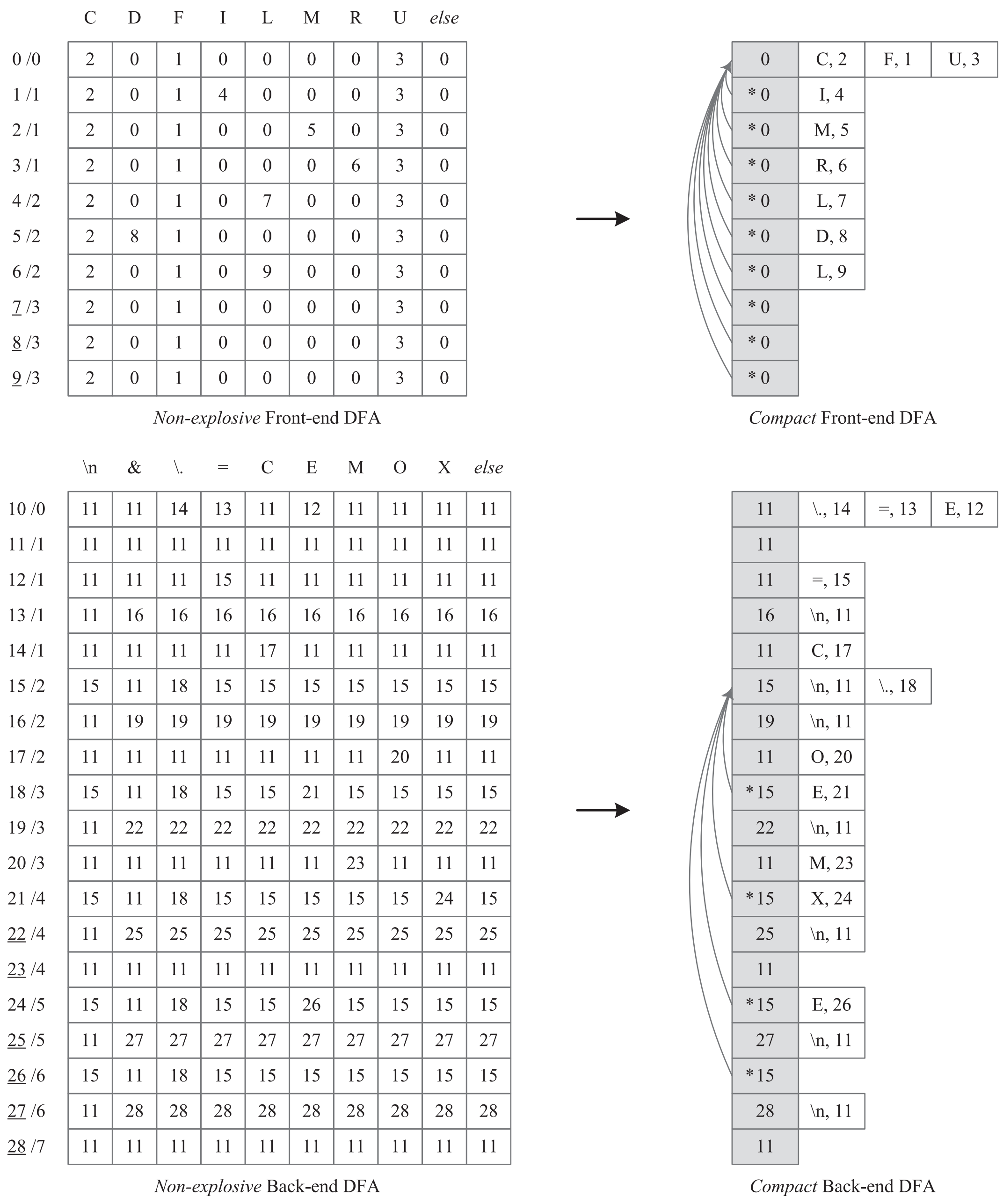
SG1: FIL
SG2: CMD
SG3: URL

SG5: ^E=[^&]*\.EXE
SG6: ^=[^\n]{6}
SG4: ^=[^\n]{3,5}
SG7: ^\.COM

RE1: (SG1)·(SG5)
RE2: (SG2)·(SG6)
RE3: (SG3)·(SG4)·(SG7)

	SG_{prev}	RE_{origin}	$STATE_{next}$
SG1	-	-	10
SG2	-	-	10
SG3	-	-	10
SG4	SG3	-	10
SG5	SG1	RE1	-
SG6	SG2	RE2	-
SG7	SG4	RE3	-

Fig. 5. Construction and transformation for the exemplified two segment sets and the relation set resulting from the partition procedure.



	C	D	F	I	L	M	R	U	*else*
0 /0	2	0	1	0	0	0	0	3	0
1 /1	2	0	1	4	0	0	0	3	0
2 /1	2	0	1	0	0	5	0	3	0
3 /1	2	0	1	0	0	0	6	3	0
4 /2	2	0	1	0	7	0	0	3	0
5 /2	2	8	1	0	0	0	0	3	0
6 /2	2	0	1	0	9	0	0	3	0
7 /3	2	0	1	0	0	0	0	3	0
8 /3	2	0	1	0	0	0	0	3	0
9 /3	2	0	1	0	0	0	0	3	0

Non-explosive Front-end DFA

0	C, 2	F, 1	U, 3
* 0	I, 4		
* 0	M, 5		
* 0	R, 6		
* 0	L, 7		
* 0	D, 8		
* 0	L, 9		
* 0			
* 0			
* 0			

Compact Front-end DFA

	\n	&	\.	=	C	E	M	O	X	*else*
10 /0	11	11	14	13	11	12	11	11	11	11
11 /1	11	11	11	11	11	11	11	11	11	11
12 /1	11	11	11	15	11	11	11	11	11	11
13 /1	11	16	16	16	16	16	16	16	16	16
14 /1	11	11	11	11	17	11	11	11	11	11
15 /2	15	11	18	15	15	15	15	15	15	15
16 /2	11	19	19	19	19	19	19	19	19	19
17 /2	11	11	11	11	11	11	11	20	11	11
18 /3	15	11	18	15	15	21	15	15	15	15
19 /3	11	22	22	22	22	22	22	22	22	22
20 /3	11	11	11	11	11	11	23	11	11	11
21 /4	15	11	18	15	15	15	15	15	24	15
22 /4	11	25	25	25	25	25	25	25	25	25
23 /4	11	11	11	11	11	11	11	11	11	11
24 /5	15	11	18	15	15	26	15	15	15	15
25 /5	11	27	27	27	27	27	27	27	27	27
26 /6	15	11	18	15	15	15	15	15	15	15
27 /6	11	28	28	28	28	28	28	28	28	28
28 /7	11	11	11	11	11	11	11	11	11	11

Non-explosive Back-end DFA

11	\., 14	=, 13	E, 12
11			
11	=, 15		
16	\n, 11		
11	C, 17		
15	\n, 11	\., 18	
19	\n, 11		
11	O, 20		
*15	E, 21		
22	\n, 11		
11	M, 23		
*15	X, 24		
25	\n, 11		
11			
*15	E, 26		
27	\n, 11		
*15			
28	\n, 11		
11			

Compact Back-end DFA

Fig. 6. Compression for the two non-explosive DFAs (denoted as matrix) resulting from compilation in Fig. 5. (The numbers on the left of the matrix stand for state number and state depth, e.g. 10/0 means state 10 with depth 0. The numbers, i.e., state transitions in the matrix means the next states the current states will transit to, e.g. for current state 10 and input character 'E', the next-hop state is state 12.)

(with no anterior OF) also results in the intra- and inter-state identity of transitions back to the dead state. The OF ϕ^* invariably introduces the intermediate self-loop state and results in the intra-state identity of transitions back to this state, as well as the inter-state identity of transitions back to this state, the neighbor of this state, and the dead state. In addition, the OF $\phi\{\}$ leads to the intra-state identity and the inter-state difference of transitions going forward to the neighbor in a chain of states.

In fact, all identical transitions caused by the characteristics of the segments are redundant; thus, a tailor-made compression algorithm is designed to reduce these transitions to compress the DFAs.

5.2.2. Algorithm design

To facilitate the textual description, it is necessary to define the following terms.

- The transition, which is identical for most characters inside a state, is the major transition (MT, e.g. the value 15 inside state 15 in the non-explosive Back-end DFA of Fig. 6).
- The states, which have the same MT and share a larger number of identical transitions than the number of MTs they have, are a state cluster (SC, e.g. states 15, 18, 21, 24, and 26).
- The state, which has the most MTs and the minimum state depth [21] in corresponding SC, is the default state (DS, e.g. state 15 with depth 2).
- The state, which is not the DS in the corresponding SC, is a labeled state (LS, e.g. state 18).
- The transition of LS, which directs to the DS in the same SC, is the default transition (DT, e.g. the value 15 inside state 18).
- The transition, which is neither the MT in DS nor the DT in LS, is a labeled transition (LT, e.g. the value 21 inside state 18).

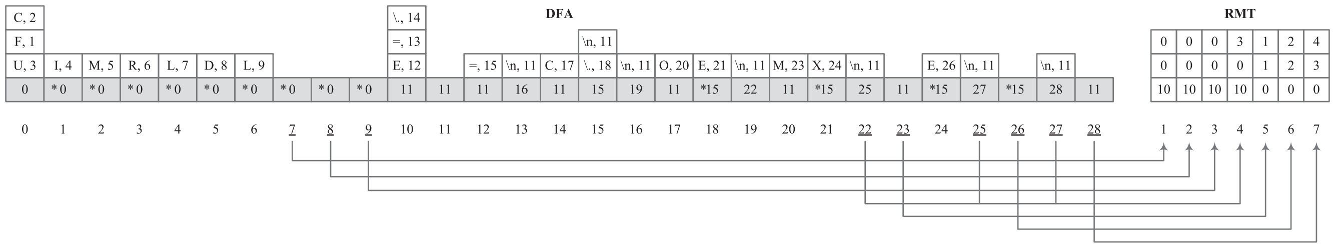


Fig. 7. Matching engine PaCCE for the exemplified regex set: “FILE=[ˆ&]*\.EXE”, “CMD=[ˆ\n]{6}”, and “URL=[ˆ\n]{3,5}\.COM”.

In the compression, the MT is only used to reduce the intra-state identical transitions of the DFAs, and the DT is employed to remove inter-state identical transitions. The algorithmic procedure is as follows.

Step 1: **For each state in the DFA:**
(*A*) Calculate the state depth based on breadth-first traversal.
(*B*) Count the number of identical transitions for each transition of the state, and compare the counts to determine the MT.
(*C*) Group the state into the state set where all the states have the same MT .
Step 2: **For each set of states:**
(*A*) Compare the number of MTs and the state depth for each state to determine the DS and group it into a new SC.
(*B*) Calculate the number of transitions that are identical to the DS for each of the remaining states; if the number is larger than the number of MTs, then group the state into the same SC as a LS, and set its DT to the DS.
(*C*) After a round of processing, iteratively execute (*A*)–(*C*) for the remaining states in the set until every state in the set has a corresponding SC.
Step 3: **For each SC:**
(*A*) Reserve only the transitions that are different from the corresponding transitions in the DS (as LTs), as well as the unique DT for each LS.
(*B*) Reserve only the transitions that are different from MT (as LTs), as well as a single MT for the DS.

The compact DFAs are shown in Fig. 6. The gray transitions with no asterisk are MTs, and the corresponding states are all DSs. The gray transitions with an asterisk are DTs, and the corresponding states are all LSs. All white transitions are LTs.

In compact Front-end DFA, all states are grouped into one SC because they have the same MT (e.g. the value 0), and for any pair of states, the identical transitions between them (the number is 8 or 9) are more than the MTs inside each of them (the number is 5 or 6). State 0 has the most MTs (i.e., 6) and the minimum state depth (i.e., 0, denoted by “/0”); thus, it is regarded as the DS. Except for the LTs (‘C’,2), (‘F’,1), and (‘U’,3), only one MT is retained in state 0. In other states, the DT is set to 0, and the LT is the transition that differs from the one in state 0 for the same character.

In compact Back-end DFA, with the exception of the SC of states 15, 18, 21, 24, and 26, each SC only contains a single state. For states 10, 11, 12, 14, 17, 20, 23, and 28, they are not grouped into one SC even though they have the same MT (e.g. the value 11), because the number of MTs inside each of them is no less than the number of identical transitions between them.

In both compact Front-end and Back-end DFAs, the average number of transitions per state is reduced to less than two, and the compression ratio $R_{compress}$ is $(9 \times 10 - 19)/(9 \times 10) \approx 79\%$ $(|\Sigma| = 9)$ and $(10 \times 19 - 37)/(10 \times 19) \approx 81\%$ $(|\Sigma| = 10)$, respectively. This compression effect only benefits from the characteristics of the segments. In practice, the algorithm can achieve a consistent 96–99% compression ratio ($|\Sigma| = 256$) for the real-world regex sets.

Because MT, DT, and LT have an extra label (LT has a ASCII character, and MT and DT have a mark of distinction) compared with the state transition in the uncompressed DFA, the total size of the Front-end and Back-end DFAs should be $(N_{FDFA} + N_{BDFA}) \cdot |\Sigma| \cdot (1 - R_{compress}) \cdot (log_2(N_{FDFA} + N_{BDFA}) + 8)$ bits. Suppose $N_{FDFA} + N_{BDFA} = 65,536$ and $R_{compress} = 99\%$, the memory usage of the two DFAs before and after the compression will be 32 MB and 492 KB respectively (i.e., reduced by almost two orders of magnitude).

5.3. Combination

The succinct DFAs (in terms of both states and transitions) are constructed to match the segments, and a small RMT is established to maintain the syntagmatic relations of the segments. To restore the complete regexes, the DFAs and RMT must be combined to obtain the final matching engine.

On one hand, the state numbers of Front-end and Back-end DFAs are totally different; thus, these two DFAs can be combined into a single DFA, as is shown by the DFA part of Fig. 7. For the combined DFA, its first 10 states can be activated with no doubt; however, for the remaining 19 states, they can be accessed only when state 10 is initialized.

On the other hand, according to Section 5.1, the RMT is indexed based on the segment ID; therefore, every match state in the DFA maintains the ID of the corresponding segment. Thus, using the segment IDs, the DFA and the RMT are combined. For example, when state 7 is matched in the DFA, ID 1 of corresponding segment SG1 will be used to index the entries in the 1st column of the RMT (note that the RMT is shown vertically in Fig. 7, and zero is used to denote the null value of RMT entries shown in Fig. 5 in practice). The arrows in Fig. 7 indicate the index relationship between the DFA and the RMT.

For the combined matching engine, referred to as PaCCE, the original regexes can be uniquely restored based on the segments in the DFA part and the relations in the RMT part. For example, segments 3 and 4 are the anterior segments of segments 4 and 7, respectively, and segment 3 is the first segment (due to no anterior segment). Therefore, segments 3, 4, and 7 correspond to a complete regex sequentially, i.e., regex 3. Furthermore, according to the match states of segments 3, 4, and 7 in the DFA, the literal content of the segments can be derived; thus, the regex 3 can be fully reassembled.

This reversible procedure verifies that the matching engine preserves the complete semantics of the original regexes, and we can further use the proper matching algorithm to guarantee no false positives or false negatives for runtime matching.

6. Matching procedure

Algorithm 2. Matching Procedure

```
Input: Matching Engine $PaCCE$, Input Content $INPUT$.
Output: Match Result.
Function:
1:  $FSU$.init(0, 0); $BSU$.init($BDFA\_INIT\_STATE$, 0); $Q_{SU}$.push_back($FSU$);
2:  for all $C \in INPUT$ do
3:    for all $SU \in Q_{SU}$ do
4:      $SU$.state = $PaCCE.DFA$.take_state_transition($SU$.state, $C$);
5:      if $SU$.state.is_dead_state() == $true$ then
6:        $Q_{SU}$.remove($SU$); continue;
7:      end if
8:      if $SU$.state.is_match_state() == $false$ then
9:        continue;
10:     end if
11:     for all segment_id $\in SU$.state.segment_ids do
12:       $SR$ = $PACCE.RMT$.get_segment_relation(segment_id);
13:       if __get_bit($SU$.bitset, $SR.SG_{prev}$) == 0 then
14:         continue;
15:       end if
16:       if $SR.RE_{origin} \neq 0$ then
17:         __output("Regex $SR.RE_{origin}$ is matched"); continue;
18:       end if
19:       if $SR.STATE_{next}$ == 0 then
20:         __set_bit($FSU$.bitset, segment_id);
21:       else
22:         __set_bit($BSU$.bitset, segment_id);
23:       end if
24:     end for
25:   end for
26:   if __test_bitset($FSU$.bitset) $\neq 0$ then
27:     $Q_{SU}$.push_back($FSU$); __clear_bitset($FSU$.bitset);
28:   end if
29:   if __test_bitset($BSU$.bitset) $\neq 0$ then
30:     $Q_{SU}$.push_back($BSU$); __clear_bitset($BSU$.bitset);
31:   end if
32:   if $Q_{SU}$.size() > $T$ then
33:     $T = Q_{SU}$.merge_state_unit();
34:   end if
35: end for
```

In this section, we introduce the runtime processing of the matching engine PaCCE resulting from the PaCC compilation procedure to guarantee both semantic equivalence and high speed for final regex matching.

6.1. Runtime processing mechanism

In the matching engine PaCCE, an original regex is matched only when all its segments are matched in sequence and in succession.

The matching of each segment is separate in the DFA part. For the compact DFA, suppose the current state is $STATE_{curr}$ and the current input character is C. The matching will first search the LTs of $STATE_{curr}$ to find the one corresponding to C. If the satisfied LT exists, then follow it and transit to the next state $STATE_{next}$. Else, if there is a MT (i.e., $STATE_{curr}$ is DS), then follow it as well and transit to the next state $STATE_{next}$. Else, follow the DT (i.e., $STATE_{curr}$ is LS, not DS) and first transit to the corresponding DS $STATE_{def}$, then continue to search the LTs of $STATE_{def}$ to find the one corresponding to C. If the satisfied LT exists, then follow it and finally transit to the next state $STATE_{next}$. Else, follow the MT (because $STATE_{def}$ is DS) and finally transit to the next state $STATE_{next}$. Whenever one input character C is processed, the compact DFA will transit from one state $STATE_{curr}$ to another state $STATE_{next}$. If $STATE_{next}$ is a match state, then the corresponding segments are matched.

The RMT is used to determine whether the matched segment is in the correct sequence. To be matched, alternated segments belonging to the same regex should follow the concatenation order in their original regex. Else, even though these segments are matched in the DFA part, they will be regarded invalid and ignored in the RMT part.

It should be noted that the definition of partition requires that the non-first segments are anchored. This ensures 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 a false positive could occur.

Based on the above conditions, correct regex matching is guaranteed by activating state units to keep track of the current state number (based on a $log_2(N_{FDFA} + N_{BDFA})$-bit value) and the IDs of previously matched segments (based on a $log_2(N_{SG} - N_{RE})$-bit bitset) during the matching process.

6.2. Overall procedure

Algorithm 2 describes the matching procedure. In this algorithm, only step 4 must be executed for each input character. Details of this step have been explained in Section 6.1. It is evident that, in the worst case, at most two state accesses must be executed per input character for the compact DFA.

If no new state units are generated at steps 26–31, then the matching engine is equal to the compact DFA. One can find that the premise of producing new state units is that none of steps 5, 8, 13, and 16 is satisfied (i.e., step 19–23 must be executed), which means a dead state cannot be met and a match state must be accessed where the match should be an eligible match of a non-tail segment.

6.2.1. Explanation

For step 5, the dead state is actually an endless loop state, which always exists in the DFA whose corresponding regexes are all anchored and have no OF $.^*$ or $.\{n,\}$ (i.e., Back-end DFA). Once an active state unit SU falls into the dead state (i.e., SU.state is equal to the dead state number), regardless of input character, no additional segment match can occur for SU, because the SU will never transit to a match state other than the dead state. Thus, such an active state unit SU can be removed from the queue Q_{SU} without affecting the matching procedure. This is the purpose of steps 5–7.

Steps 8–10 mean that, for a non-match state, there is nothing else to do. However, for a match state, it is necessary to determine whether the match is eligible for each of its corresponding segments (note that one match state may correspond to multiple segments). Hence, at step 12, the segment_id of each matched segment will be used to index the RMT and fetch the ID SG_{prev} of its anterior segment, the ID RE_{origin} of its corresponding regex, and the state $STATE_{next}$ of its posterior segment. If the expected anterior segment SG_{prev} has been matched previous to the just matched segment, then the bit corresponding to SG_{prev} must be nonzero in the current SU.bitset; thus, the match is eligible. This is shown at steps 13–15. Note that, for the first segments, no anterior segments are required.

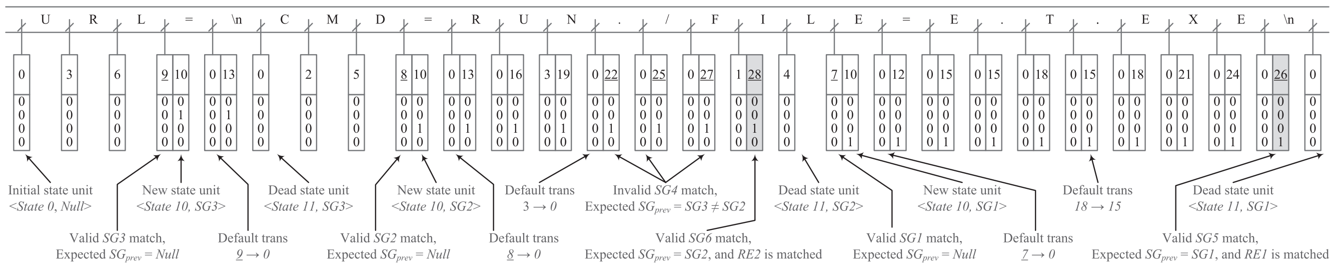

U R L = \n C M D = R U N . / F I L E = E . T . E X E \n
Initial state unit <State 0, Null>
New state unit <State 10, SG3>
Dead state unit <State 11, SG3>
New state unit <State 10, SG2>
Default trans 3 → 0
Invalid SG4 match, Expected SGprev = SG3 ≠ SG2
Dead state unit <State 11, SG2>
New state unit <State 10, SG1>
Default trans 18 → 15
Dead state unit <State 11, SG1>
Valid SG3 match, Expected SGprev = Null
Default trans 9 → 0
Valid SG2 match, Expected SGprev = Null
Default trans 8 → 0
Valid SG6 match, Expected SGprev = SG2, and RE2 is matched
Valid SG1 match, Expected SGprev = Null
Default trans 7 → 0
Valid SG5 match, Expected SGprev = SG1, and RE1 is matched

Fig. 8. Runtime processing for the matching engine PaCCE in Fig. 7 for the exemplified input: "URL = \nCMD = RUN./FILE = E.T.EXE\n".

For each valid matched segment, whether it corresponds to a match of a complete regex will be determined at step 16. If the valid matched segment is the tail segment, i.e., all its anterior segments have been matched in sequence and in succession, then the corresponding regex is matched correctly. Steps 16–18 will record these match results.

If none of steps 5, 8, 13, and 16 is satisfied, new state units will be activated and processed for the next input character. Note that, at most two fresh state units will be created for each input character even though there are multiple qualified segments. This is because the new state unit can only has the initial state of either Front-end DFA or Back-end DFA as its start state, and the difference in the bitset of the state units can be recorded by setting corresponding bit, as shown at step 19.

Therefore, when all current state units are processed for the input character *C*, given that new state unit *FSU* and/or *BSU* must be generated, *FSU* and/or *BSU* will be appended to the queue Q_{SU}. For the next input character *C*, all the old (excluding the dead) and new state units will be processed together.

6.2.2. Demonstration

To demonstrate Algorithm 2, we take Fig. 8 as example. The initial state unit *FSU* in Q_{SU} consists of state number 0 and bitset 0000 (i.e., 4-bit bitset, because there are 4 non-tail segments).

For input "URL", segment SG3 is matched in the DFA, then its ID 3 is used to index the 3rd entry in the RMT, where no anterior segment is required (anterior segment ID is 0). Thus, this match is valid, and a new state unit with state number 10 and bitset 0100 (the 3rd bit is set) is generated and then processed in parallel with the original state unit after then. Next, for input "=\n", one state unit enters the dead state 11; thus, it is removed. For the other state unit, one can find that it follows the path 9 → (0) → 0 → 0 to do state transition, where the first transition from state 9 to DS 0 is an extra jump aroused by the DT of state 9 because there is no LT corresponding to the input character '=' in state 9.

Next, for input "CMD", segment SG2 is correctly matched, an in a similar manner, a corresponding state unit with state number 10 and segment ID 2 is generated. Next, for input "=RUN", another match state is reached; however, it is an invalid match because the expected anterior segment SG3 is not matched previously for current state unit. Next, for the subsequent input "." and "/", the situation is identical. Next, for input "F", segment SG6 is matched in the DFA, the indexed 6th entry in the RMT shows that segment SG2 is expected to be previously matched, and in fact it had been matched because the 2nd bit is set in the bitset 0010. Thus, this is a valid match. In addition, the match of regex RE2 (i.e., "CMD=[^\n]{6}") is found (the background is grayed) because the original regex of segment SG6 is RE2.

For the following input "ILE = E.T.EXE\n", regex RE1 is also matched, and the number of state units is further reduced to one. Related details are provided in Fig. 8.

6.2.3. Dynamic merger of state units

According to the analysis in Section 5.2.1, most state transitions lead back to the initial states (e.g. state 0 or its neighbor states in Fig. 6) or the intermediate self-loop states (e.g. state 11 or state

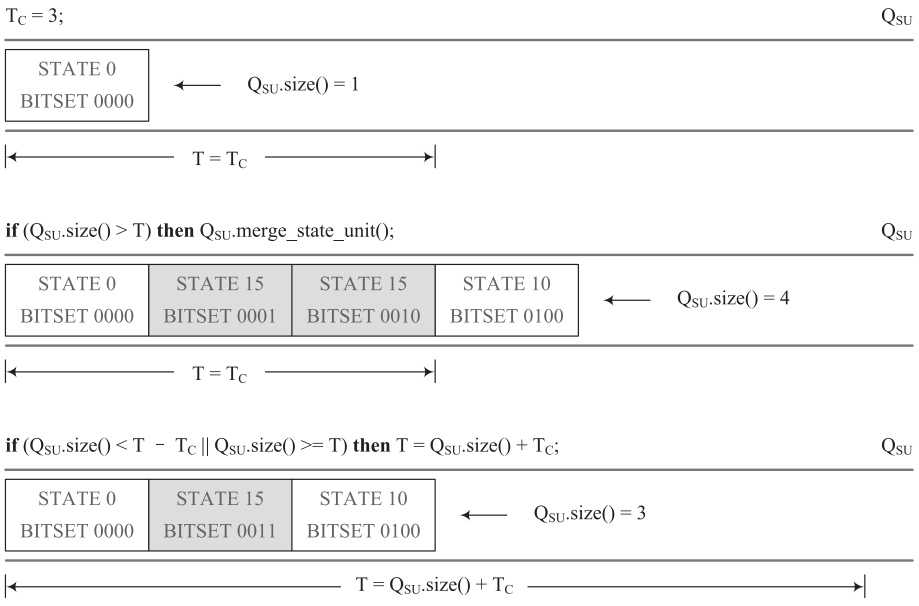


Fig. 9. Queue size adjustment for efficient state unit merge.

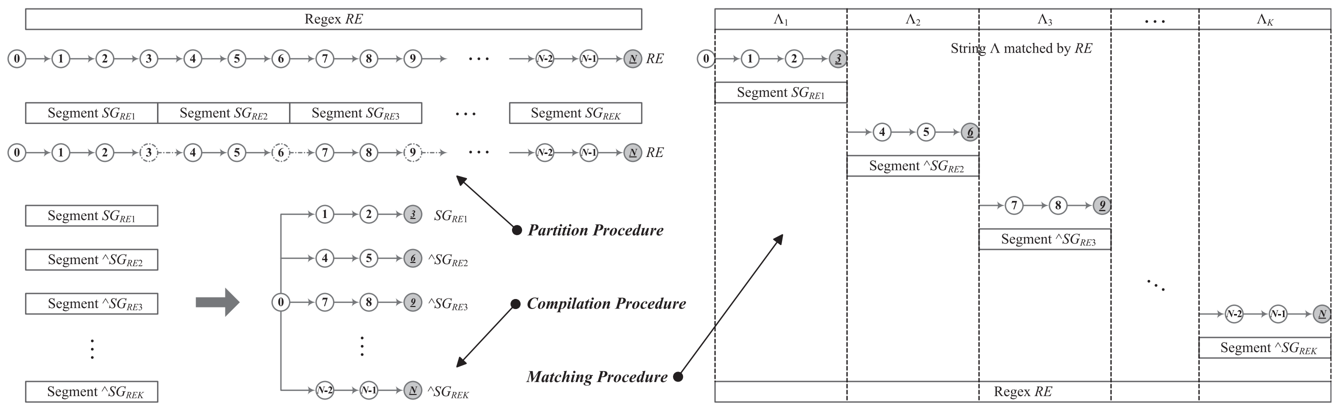


Fig. 10. The truth (behind PaCC) of the accurate regex matching based on the segments

15 in Fig. 6). Consequently, even if multiple state units are activated, they are likely to transit back to the same state in a short time. Then, for the state units having identical state number, they could be merged merely by the bitwise OR of their bitsets [30]. This means the number of state units in the matching engine can converge to one in any case.

In addition, as steps 32–34 of Algorithm 2 show, we can choose the proper time to merge the annexable state units as necessary, rather than for every input character because most of the time the number of state units in the queue is minimum (e.g. the number of active state units in Fig. 8 is minimum for each input character). Thus, the threshold of the number of active state units is used, as shown in Fig. 9. Initially, a default threshold $T = T_C$ is given. Once the size of the queue achieves the threshold, the merger is executed. After that, if the size of the queue is not less than the threshold, i.e., few states transit to the initial or self-loop states, then it is necessary to adjust the threshold to a value larger than the size of the queue. Similarly, if the size of the queue is much less than the threshold, then it is necessary to reduce the threshold accordingly. Otherwise, the threshold remains unchanged. By choosing the proper time to merge according to the dynamically optimized threshold, the matching engine can main matching at an optimal rate.

6.3. Proof of matching accuracy

6.3.1. For compact Front-end and Back-end DFAs

In the uncompressed DFA, the transition corresponding to the input character C can be directly found and executed for current state, however, there are five and only five possible and mutually exclusive cases for different states in the compact DFA.

Case 1: For current DS, LT corresponding to C is found and executed.
Case 2: For current DS, there is no corresponding LT (compressed due to MT), MT is found and executed.
Case 3: For current LS, LT corresponding to C is found and executed.
Case 4: For current LS, there is no corresponding LT (compressed due to DT), DT to the DS is executed first, then goto Case 1.
Case 5: For current LS, there is no corresponding LT (compressed due to DT), DT to the DS is executed first, then goto Case 2.

Because MT only compresses the transitions (in DS) the same with MT, and DT only compresses the transitions (in LS) identical with the ones in DS for the same C, the transition corresponding to C must be correctly found for any state in the compact DFA. In other words, the transition from $STATE_{curr}$ to $STATE_{next}$ for any input character C in the compact DFA is equivalent to the one in the uncompressed DFA. Consequently, the matching accuracy of the Front-end and Back-end DFAs is guaranteed.

6.3.2. For PaCCE

As Fig. 10 depicts, when original regexes 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 the definition (see the left of Fig. 10). Hence, in PaCCE, to simulate the accurate matching of a complete regex, the matching of the segments must be in sequence and in succession (see the right of Fig. 10).

According to the matching procedure described in Section 6.1 and 6.2, it can be proved that the matching engine constructed by PaCC does not sacrifice any matching correctness compared with the corresponding DFA.

Theorem 3. *Given regex set S_{RE}, its two segment sets resulting from the partition are S_{SG1} and S_{SG2}, then the PaCCE generated from the segment sets S_{SG1} and S_{SG2} is equal to the DFA built from the regex set S_{RE} in terms of matching accuracy.*

Proof. Support S_{RE} only has one regex RE, and RE has K segments $\{SG_{RE_k} | k \in Z^+, k \leqslant K\}$.

Assume the string Λ can be matched by the DFA of S_{RE} (i.e., RE), then there must exists K sections for Λ, where SG_{RE_k} matches Λ_k $(k \in Z^+, k \leqslant K)$. This is because each Λ_k is concatenated in Λ, and each SG_{RE_k} is originally concatenated in RE as well. Because Λ_k is processed in sequence and in succession, i.e., Λ_{k+1} will be immediately processed once the processing of Λ_k is finished, SG_{RE_k} will be successively and correctly matched in PaCCE, and at last the original RE is judged to be matched validly. Thus, PaCCE cause no false negative during the matching process.

Assume the string Λ can be matched by the PaCCE of S_{RE} (i.e., RE), but Λ cannot be matched by the DFA of S_{RE}, then for every possible $\{\Lambda_k | k \in Z^+, k \leqslant K\}$, there at least exists Λ_k, where SG_{RE_k} cannot match Λ_k, otherwise Λ can be matched by the DFA of RE. However, in PaCCE, RE has no chance to be matched unless all of the segments SG_{RE_k} are matched, and SG_{RE_k} can not be matched for Λ_i $(i \neq k)$, because the caret attached to the SG_{RE_k} $(k \in Z^+, 1 < k \leqslant K)$ guarantees that there should be a one-to-one correspondence between SG_{RE_k} and Λ_k. In other words, the initial assumption is contradictory, and Λ which is matched by the PaCCE of S_{RE} must be able to be matched by the corresponding DFA. Hence, PaCCE does not cause false positives during the matching process.

Consequently, PaCCE and DFA are equivalent for every RE in S_{RE}, Theorem 3 is proved. □

Table 4
Statistics for the regex sets used in our experiments.

Regex set	# Of regexes			# Of OFs				# Of states		# Of segments			
	Total	Explosive	W/overlapping	Total	Non-equal	W/ϕ^* form	W/$\phi\{\}$ form	NFA	DFA	Segment set 1		Segment set 2	
										Raw	Merged	Raw	Merged
backdoor	162	0	61	466	17	436	30	3534	$\gg$15M	176	164	72	72
blacklist	29	0	18	37	11	29	8	777	$\gg$15M	31	30	19	19
botnet	90	0	50	180	28	120	60	2015	$\gg$15M	98	97	76	76
deleted	1315	5	1257	9989	21	9449	540	86,288	$\gg$15M	1620	403	1561	1558
ftp	71	0	55	94	6	52	42	7262	54,009	71	68	55	55
imap	45	16	44	72	9	49	23	5488	$\gg$15M	47	35	46	46
pop3	16	0	14	19	3	7	12	1149	34,269	16	16	14	14
smtp	126	9	113	420	16	403	17	5612	$\gg$15M	128	38	117	114
spyware	343	0	276	620	12	587	33	6058	$\gg$15M	368	138	311	311
sql	26	0	16	76	13	69	7	1081	$\gg$15M	30	29	18	18
web-activex	1243	1	1237	16,073	8	15,488	585	59,109	$\gg$15M	2465	933	1858	1523
web-cgi	18	5	15	24	13	14	10	2621	$\gg$15M	22	20	15	15
web-iis	28	2	24	33	15	27	6	2279	$\gg$15M	28	28	25	25
web-misc	125	31	93	232	32	169	63	14,819	$\gg$15M	134	124	107	105
web-php	64	1	35	131	13	70	61	1648	$\gg$15M	64	62	50	50
bro	1104	42	80	96	2	48	48	22,860	$\gg$15M	1149	1144	47	47
syn.3	5000	601	1400	1500	8	750	750	166,526	$\gg$15M	5000	4266	1500	1499
syn.5	5000	993	2330	2500	8	1250	1250	183,602	$\gg$15M	5000	3595	2500	2497
syn.7	5000	1397	3244	3500	8	1750	1750	199,183	$\gg$15M	5000	2872	3500	3494

6.4. Performance advantages

It is important to note that, although two DFAs could be built for PaCCE, 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 regex in the Front-end DFA is matched correctly, can Back-end DFA be activated to process the subsequent segments belonging to the same regex. Moreover, 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 is likely to be trapped in the dead state or merged with others. Therefore, the average-case matching complexity of PaCCE is $O(1)$. The theoretical worst-case complexity is $O(N_{FDFA} + N_{BDFA})$ because, at most, all concurrent active state units correspond to the distinct state numbers in the DFA. However, this is not possible in practice. According to our analysis, the tighter worst-case complexity is proportional to the number of nonequivalent OFs because, the DFA can only be trapped in the states representing the OFs while the equivalent OFs are deflated due to Principle 4.

Furthermore, unlike a general data structure with a prefilter (e.g. Snort IDS [2,3]), where the back-end matching engines of the prefilter are built from the complete regexes and are required to process the input characters from the start when a match occurs in the prefilter (i.e., multi-pass matching), the Back-end DFA in our matching engine is also constructed from segments, and only needs to process the subsequent input characters if executed (i.e., one-pass matching).

7. Experimental evaluation

In this section, we use real and synthetic data and tests to verify the effectiveness of the PaCC solution comprehensively.

7.1. Test environment

All experiments were conducted on a HP Z220 SFF workstation, where the CPU is 3.40 GHz Intel Core i7-3770 (with 256 KB L1 cache, 1 MB L2 cache, and 8 MB L3 shared cache), the memory is 2×8 GB 1600 MHz DDR3 SDRAM, and the OS is 32 bit Ubuntu Server 12.04 LTS.

7.1.1. Regex sets

As Table 4 shows, the regex sets used include 16 real-world sets collected from the open source Snort [3] (the first 15 regex sets, i.e., from *backdoor* to *web-php*) and Bro [5] (the 16th regex set, i.e., *bro*), and 3 synthetic sets generated from the public regex processor [31] (the last 3 regex sets, i.e., *syn.3* with 30% OFs, *syn.5* with 50% OFs, *syn.7* with 70% OFs). Note that all the duplicated regexes have been eliminated from these sets to make the given scale (i.e., number of regexes) more meaningful.

According to the statistics shown in Table 4, most of these regex sets are very complex, due to a large number of OFs and significant semantic overlapping. Especially, the real-world regex sets *deleted* and *web-activex* both have about 10,000 OFs and over 1000 regexes with semantic overlapping. For every given regex set, Table 4 also lists the number of explosive regexes each of which cannot be individually compiled into a single DFA within 15,000,000 (i.e., 15 M) states.

It can be found that, in practice, except for *ftp* and *pop3*, none of the regex sets in Table 4 can be transformed into a single composite DFA, because the DFA with significantly more than 15 M states requires memory larger than the SDRAM size of our testbed. In contrast, their NFAs are all available and contain no more than 200 K states (the NFA of *syn.7* has the maximum number of states, i.e., 199,183).

7.1.2. Algorithms for comparison

The state-of-the-art algorithm used for performance comparison is Hybrid-FA [14], the source code of which is publicly available in [31]. This is because Hybrid-FA can be built for all types of regexes and thus has no limitation for practical usage. Besides, Hybrid-FA has the minimum and acceptable preprocessing complexity compared with other hybrid construction solutions. In comparison, for SFA [16], its algorithm used to judge state conflicts is so complex (i.e., $O(N_P^2 \cdot L_P^2 \cdot |\Sigma|^{L_P})$ preprocessing complexity) that using SFA to implement practical regex matching is infeasible.

In fact, previous state compression algorithms focus on finding a best tradeoff between NFA and DFA (where Hybrid-FA is one of the superior), therefore, few of them can create a data structure with states less than corresponding NFA, and keep the matching speed close to DFA. In consequence, NFA, DFA, and Hybrid-FA are mainly used to do the comparison in our experiments.

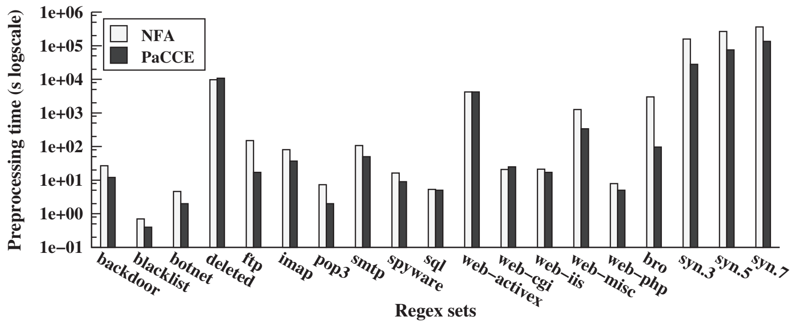


Fig. 11. Comparison of the preprocessing time for NFA and PaCCE.

Other well-known state compression algorithms, such as XFA [42,12], are not considered because they fail to handle the widely existing regexes with inherent semantic overlapping. In addition, XFA also does not support the common regex $ES_1 \cdot \phi^* \cdot ES_2$ where the suffix of ES_1 and the prefix of ES_2 are identical. Moreover, these algorithms do need a prohibitive preprocessing duration for large scale regex sets. For example, according to the experimental results of XFA [42], 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 of XFA. Although these algorithms will not be compared in this paper, there is already a preliminary result [30] of the comparison among Grouping [7], XFA and our solution for the supported types of regexes in Snort, which indicates that our solution is superior to them.

Furthermore, the algorithms used for the comparison with our tailor-made transition compression method is D^{2}FA [45], A-DFA [21], and RCDFA [29]. They are the most representative software algorithms to reduce the duplicated inter- and intra-state transitions of DFA. Note that, for D^{2}FA, the maximum default path length is limited to 1, to guarantee that the number of memory lookups per input character is 2 for D^{2}FA in the worst case (which is identical with A-DFA and our method). For RCDFA, it can compress the DFA without introducing additional memory lookups. Besides, D^{2}FA combined with RCDFA [29], A-DFA combined with RCDFA [29], and our method combined with RCDFA are compared as well.

7.2. Preprocessing statistics

7.2.1. Partition results

Table 4 displays OF statistics. Comparatively, *deleted*, *web-activex*, and *syn.** are more complex, because they have thousands of OFs (i.e., 750–15,488). Especially, they have hundreds of OFs with form $\phi\{\}$ (i.e., 540–1750), 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 previously in Section 4.4). For example, for *backdoor*, the number of nonequivalent OFs is only 17, although the total number of OFs is 446. This facilitates satisfying Principles 3 and 4.

For these regex sets with more OFs, such as *deleted* and *web-activex*, it is necessary to split their regexes into more segments to eliminate semantic overlapping completely. Because each regex is partitioned into at least one segment, the number of segments must be larger than the number of regexes. For example, for *deleted* in Table 4, the number of original regexes is 1315, and the number of segments is $1620 + 1561 = 3181$, larger than 1315. However, based on the merger of equivalent segments, the number of segments can be practically reduced. As Table 4 shows, for *deteled*, the number of segments in the raw segment sets is 3181, but the number of distinguishing segments for the merged segment sets is $403 + 1558 = 1961$. There is a large decrease (e.g. $3181 - 1961 = 1220$ for *deleted*) for the practical number of segments. Therefore, compared with the number of original regexes, the number of segments does not increase too significantly, e.g. no more than twice for all given regex sets in Table 4. Thus, the increase in the number of segments can be considered 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 regex set *deleted* is 9989, and the number of segments in the raw segment sets is 3981, much less than 9989. In fact, according to the analysis presented in Section 4.4, OFs with a small character set ϕ are unlikely to cause semantic overlapping; therefore, they are ignored for partition.

7.2.2. Preprocessing time

PaCCE preprocessing involves two steps: regex partition and segment compilation. The total preprocessing time for PaCCE is primarily determined by the number of states in the DFA part of PaCCE (i.e., $N_{FDFA} + N_{BDFA}$), because the time required for the compilation process is proportional to $N_{FDFA} + N_{BDFA}$, while the time required for the partition process is proportional to N_{SG} and is relatively insignificant ($N_{SG} < N_{RE} \cdot L_{RE} \leqslant N_{FDFA} + N_{BDFA}$).

Fig. 11 shows that the preprocessing time of PaCCE is less than that for NFA in general. For example, for *bro*, the construction of PaCCE only requires 97 s, but the construction of NFA costs over 50 min. This is because the NFA state reduction process [31] is more time-consuming for the original regexes than the overlapping-free segments. However, for *deleted* and *web-cgi* in Fig. 11, the NFA is slightly more efficient than PaCCE. This is because both PaCCE and NFA spend almost the same time in the process of NFA state reduction, but PaCCE requires more time for the processes of subset construction and DFA minimization [6]. For the most complex real-world regex set *deleted*, constructing PaCCE only requires less than 3 h. Besides, for *syn.7*, the preprocessing time of PaCCE is about 1.5 days. In comparison, the construction of NFA requires over 4 days for *syn.7*.

Because the existing state compression algorithms are mainly based on NFA, their preprocessing time will be not better (often be far worse) than NFA in practice. According to our simple tests, for Grouping, Hybrid-FA, and XFA, they cost at least one order of magnitude (above two orders of magnitude in most cases) more preprocessing time than PaCCE for the real-world regex sets of the supported regexes (e.g. *ftp*). The shorter preprocessing time can make PaCCE practically available within tolerable update deadlines.

7.3. Spatial performance

According to Fig. 12, the spatial performance of PaCCE is generally superior to NFA in terms of fewer number of states. This is because Principle 4 causes massive repetitiveness among the generated segments according to Section 4.3, and thus makes the built

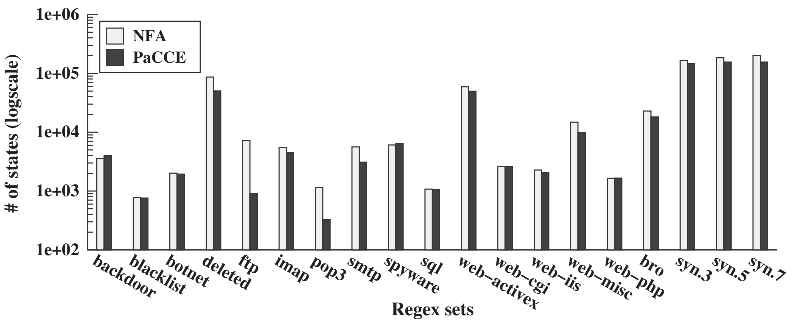


Fig. 12. Comparison of the number of states for NFA and PaCCE.

Table 5
Comparison of compression ratios for D^2FA, A-DFA, RCDFA, and our method (%).

Regex set	D^2FA			A-DFA			RCDFA			Ours		
	for FDFA	*for* BDFA	*for* PaCCE	*for* FDFA	*for* BDFA	*for* PaCCE	*for* FDFA	*for* BDFA	*for* PaCCE	*for* FDFA	*for* BDFA	*for* PaCCE
backdoor	84.51	75.53	82.32	97.76	84.63	94.56	79.49	93.06	82.78	96.84	97.08	96.90
blacklist	87.28	79.66	85.35	98.57	90.00	96.40	90.84	98.00	92.56	98.45	96.65	98.03
botnet	84.26	83.72	83.95	98.16	93.49	95.78	79.32	96.95	88.10	98.09	96.45	97.28
deleted	90.58	N/A	N/A	98.65	93.37	93.60	79.49	98.45	97.63	98.43	98.19	98.21
ftp	73.74	16.46	31.39	85.32	18.48	35.90	98.60	98.37	98.43	98.78	99.15	99.06
imap	87.33	25.95	28.22	98.25	3.21	6.73	95.90	98.40	98.31	98.73	99.15	99.14
pop3	80.22	2.93	16.67	96.67	2.93	19.60	98.77	98.80	98.80	98.89	99.18	99.13
smtp	92.48	11.02	20.57	98.34	12.23	22.32	91.33	98.03	97.24	98.76	98.71	98.72
spyware	86.60	86.92	86.79	98.70	98.33	98.45	78.30	78.17	78.22	96.59	98.12	97.58
sql	86.69	18.09	43.64	98.76	22.82	51.11	86.31	98.04	93.68	98.70	98.27	98.43
web-activex	92.23	N/A	N/A	99.03	98.00	98.11	87.92	98.35	97.14	97.86	98.66	98.57
web-cgi	84.21	21.68	28.52	98.22	2.42	12.92	86.17	97.16	95.95	98.39	96.49	96.70
web-iis	88.88	9.47	18.66	98.42	10.17	20.39	88.47	97.44	96.40	99.02	98.56	98.62
web-misc	85.96	8.12	20.11	98.52	8.56	22.42	75.02	96.90	93.53	98.23	98.53	98.49
web-php	86.46	42.59	64.35	98.60	45.90	72.05	82.07	94.83	88.49	98.54	97.68	98.11
bro	N/A	0.38	N/A	99.09	0.41	80.44	49.55	98.83	58.86	98.04	99.22	98.27
syn.3	N/A	N/A	N/A	99.22	9.24	67.73	59.36	84.00	67.99	97.90	98.10	97.97
syn.5	N/A	N/A	N/A	99.22	9.56	48.93	59.37	84.95	73.72	97.93	98.34	98.16
syn.7	N/A	N/A	N/A	99.22	9.86	33.10	59.75	86.72	79.71	98.05	98.55	98.42

Regex set	D^2FA with RCDFA			A-DFA with RCDFA			Ours with RCDFA		
	for FDFA	*for* BDFA	*for* PaCCE	*for* FDFA	*for* BDFA	*for* PaCCE	*for* FDFA	*for* BDFA	*for* PaCCE
backdoor	96.22	97.96	96.64	98.74	98.78	98.75	98.95	99.08	98.99
blacklist	98.15	98.80	98.31	99.07	98.81	99.01	99.09	99.07	99.09
botnet	96.19	98.74	97.46	98.82	98.84	98.83	99.01	99.10	99.06
deleted	97.39	N/A	N/A	98.86	99.17	99.16	98.95	99.23	99.22
ftp	98.92	98.78	98.82	98.79	98.79	98.79	99.16	99.51	99.42
imap	98.75	98.73	98.73	99.12	98.46	98.49	99.17	99.41	99.40
pop3	99.01	98.82	98.86	98.72	98.80	98.79	99.22	99.59	99.53
smtp	98.54	98.11	98.16	98.81	98.27	98.34	99.08	99.25	99.23
spyware	96.47	96.48	96.48	99.01	99.13	99.09	99.09	99.17	99.15
sql	97.46	98.49	98.11	99.14	98.55	98.77	99.17	99.49	99.37
web-activex	98.36	N/A	N/A	99.18	99.19	99.19	99.19	99.20	99.20
web-cgi	97.25	97.56	97.53	99.12	97.54	97.72	99.15	99.14	99.14
web-iis	98.06	98.21	98.20	98.89	98.21	98.29	99.16	99.56	99.52
web-misc	95.94	98.00	97.69	98.84	98.06	98.18	98.97	99.41	99.35
web-php	97.01	97.90	97.46	99.13	98.07	98.60	99.20	99.37	99.29
bro	N/A	98.83	N/A	99.10	98.83	99.05	99.11	99.61	99.21
syn.3	N/A	N/A	N/A	99.22	92.07	96.72	99.22	98.22	98.87
syn.5	N/A	N/A	N/A	99.22	92.95	95.71	99.22	98.39	98.76
syn.7	N/A	N/A	N/A	99.22	93.18	94.76	99.22	98.58	98.75

FDFA and BDFA more succinct. However, for *backdoor* and *spyware* in Fig. 12, the number of states in PaCCE is slightly more than NFA because PaCCE 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.

For the most complex real-world regex set *deleted*, PaCCE has only 50,326 states, and for the most complex synthetic regex set *syn.7*, PaCCE has only 155,536 states. In comparison, NFA has 86,288 and 199,183 states respectively for *deleted* and *syn.7*, and Hybrid-FA has 199,115 and 200,303 states respectively for *deleted* and *syn.7*. Note that the results for Hybrid-FA are not shown in Fig. 12 because Hybrid-FA must have larger number of states than NFA according to its design principle.

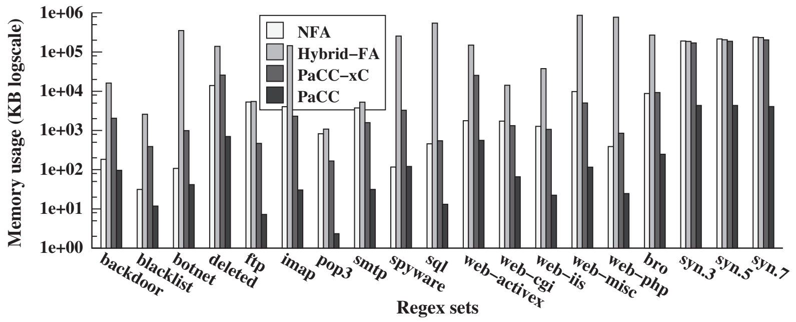


Fig. 13. Comparison of the memory usage for NFA, Hybrid-FA, PaCCE-xC, and PaCCE.

7.3.1. Scalability

Because state explosion is defused by the partition, like NFA (see Fig. 11 and Fig. 12), the number of states of the DFA in PaCCE increases proportionally with the scale of the regex sets. Besides, as indicated in Table 4 and Section 7.2, the number of segments grows linearly with the number of original regexes. Thus, the size of the RMT in PaCCE is also proportional to the number of regexes. Because PaCCE consists of the DFA and RMT parts, the size of PaCCE is determined by the sizes of both DFA and RMT (although the size of the RMT part is ignorable compared with the DFA part). Consequently, PaCCE scales linearly in terms of space consumption.

7.3.2. Deflation

Due to the large repetitiveness of segments, PaCCE can have a more succinct engine than NFA according to Section 4.3. For example, for the regex set *ftp*, there are a considerable number of OFs “[^\n]{*n*}” (i.e., the OFs are equivalent) that have identical prefix ESs “\s” (i.e., the OFs can be merged). Thus, the DFA can be deflated according to Principle 4. It is evident from Fig. 12 that almost 90% of the states are deflated (PaCCE only has 911 states but the NFA has 7262 states) for the DFA in PaCCE for the regex set *ftp*.

7.3.3. Compression ratio

Table 5 shows that our transition compression method maintains stability with a compression ratio greater than 96% (for both FDFA and BDFA) for all the given regex sets. This is determined by the features of the partition because the partition enables the DFAs to be constructed from the prescribed segments and have a large number of regular duplicate transitions.

In contrast, the compression ratio of D^{2}FA and A-DFA is very unstable for the BDFA of PaCCE. For example, for the BDFA corresponding to *bro*, both D^{2}FA and A-DFA only achieve less than 1% compression ratio. This is because for the BDFA, there are few states among which most transitions are identical for the same characters, according to the features of the segments in the segment set 2. On the contrary, for FDFA, the A-DFA algorithm almost achieves stable and over 96% compression ratio except the 85.32% compression ratio (for *ftp*), and the D^{2}FA algorithm has a relatively stable compression ratio between 80% and 95%. Like A-DFA, D^{2}FA also has a bad compression ratio (i.e., 73.74%) for the FDFA corresponding to *ftp*. Besides, due to the high spatial complexity of construction, D^{2}FA is not applicable for the FDFA and/or BDFA corresponding to the complex regex sets such as *deleted*, *web-activex*, *bro*, and *syn.**.

For RCDFA, its compression ratio also fluctuates with different regex sets. Especially, RCDFA achieves only about 49% compression ratio for the FDFA corresponding to *bro*. This is because there are many states whose transitions are not consecutively the same, according to the characteristics of the segments in the segment set 1, and RCDFA can only compress the consecutively identical transitions inside states. However, for the FDFA corresponding to the regex sets such as *ftp* and *pop3*, RCDFA can achieve over 98% compression ratio. For the BDFA, the compression efficiency of RCDFA is stable (over 93%) for most of the regex sets, but only 78.17% for *spyware* and around 85% for *syn.**.

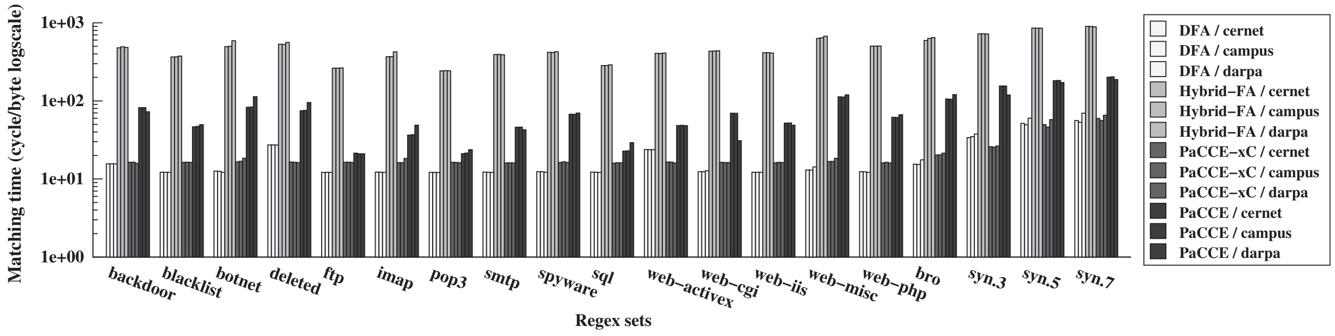
In Table 5, the total compression ratios for PaCCE show that, our tailor-made method is indeed superior to other algorithms and very suitable for the PaCC solution. However, for the applications where higher matching speed is demanded, RCDFA can be employed instead of our method for PaCCE, because RCDFA can achieve a good compression ratio without additional memory lookups (i.e., its worst-case processing complexity is equal to DFA in terms of memory access).

In fact, RCDFA can be further used to enhance other algorithms in practice [29]. Table 5 also shows the comparison among D^{2}FA combined with RCDFA [29], A-DFA combined with RCDFA [29], and our method combined with RCDFA. One can find that, the combination of our method and RCDFA can make the compression ratio of PaCCE more than 98.75% for all the regex sets, which is better than other two schemes. Over 98.75% compression ratio means that there are no more than 3.2 transitions in average for each state of the compressed DFA. A-DFA combined with RCDFA can achieve stable and over 97.54% compression ratio for all real-world regex sets, but slightly lower compression ratio (i.e., over 94.76%) for the synthetic regex sets. In fact, the major transition can compress the identical transitions which are not consecutive inside states (and thus cannot be compressed by RCDFA), therefore, the combination of our method and RCDFA is the best.

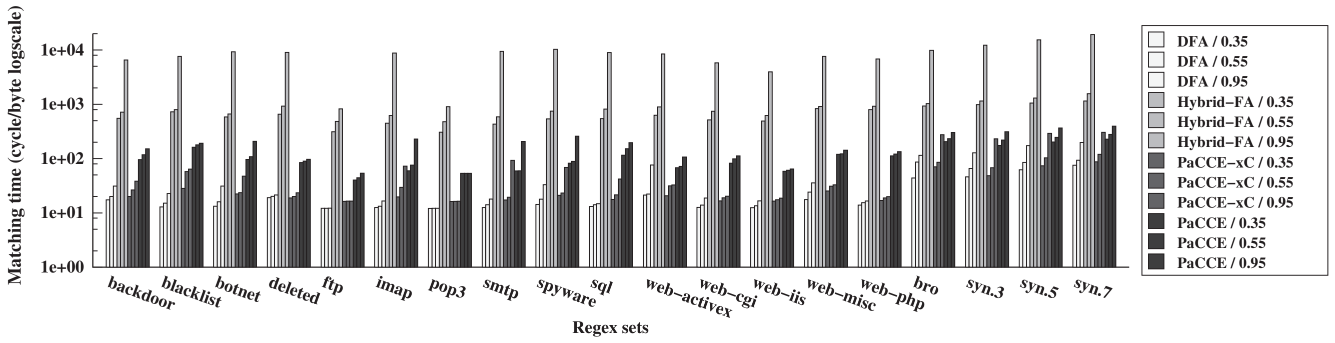
7.3.4. Memory usage

The memory footprints of NFA, Hybrid-FA, PaCCE-xC (PaCCE without transition compression), and PaCCE are compared in Fig. 13. According to Section 5.1, the size of PaCCE-xC can be calculated by the formula $(N_{FDFA}+N_{BDFA})\cdot|\Sigma|\cdot log_2(N_{FDFA}+N_{BDFA})+N_{SG}\cdot(log_2(N_{SG}-N_{RE})+log_2N_{RE}+log_2(N_{FDFA}+1))$. For example, the practical size of PaCCE-xC for *imap* is $(168+4358)\times256\times2\text{B}+81\times(1\text{B}+1\text{B}+1\text{B})\approx2.3$ MB. Fig. 13 shows that the size of PaCCE-xC is one to three orders of magnitude less than the size of Hybrid-FA for real-world regex sets. For example, the size of Hybrid-FA for *imap* is up to 145 MB, much larger than the size of PaCCE-xC. However, for the synthetic regex sets *syn.**, the size of PaCCE-xC is only a little less than the size of Hybrid-FA, because most of the states in corresponding Hybrid-FA are NFA states. Note that, the size of PaCCE-xC is not always smaller than NFA although the number of states in PaCCE-xC is less than NFA (e.g. for *backdoor*), because all DFA states of PaCCE-xC has 256 transitions, but the NFA states often have less than 256 transitions.

For PaCCE, its size can be calculated as $(N_{FDFA}+N_{BDFA})\cdot|\Sigma|\cdot(1-R_{compress})\cdot(log_2(N_{FDFA}+N_{BDFA})+8)+N_{SG}\cdot(log_2(N_{SG}-N_{RE})+log_2N_{RE}+log_2(N_{FDFA}+1))$. For example, the size of PaCCE for *imap* is $(168+4358)\times256\times(1-99.14\%)\times(2\text{B}+1\text{B})+81\times(1\text{B}+1\text{B}+1\text{B})\approx30$ KB. According to Fig. 13, PaCCE is much smaller than



(a) For the cernet trace from the CERNET network, the campus trace from Tsinghua University, and the darpa trace from the MIT Lincoln Lab



(b) For the synthetic traces that have matching probabilities of 0.35, 0.55, and 0.95

Fig. 14. Comparison of the matching time for DFA, Hybrid-FA, PaCCE-xC, and PaCCE.

NFA; close to the size of the actual regex sets. For example, for the most complex real-world regex set *deleted*, the memory footprint of PaCCE is approximately 700 KB, which is far smaller than the 1.4 MB size of NFA and closer to the 420 KB size of the regex set. In comparison, Hybrid-FA is over 10 MB size, which is very inefficient compared with both PaCCE and PaCCE-xC.

7.4. Temporal performance

The network traffic used for temporal performance evaluation includes the 925 MB published darpa trace from the MIT Lincoln Lab [46], the 1.86 GB real-world campus trace captured from Tsinghua University, the 668 MB real-world cernet trace captured from the CERNET network, and the 280 KB synthetic traces generated based on the given regex sets. Note that the synthetic traces can match the regexes in the regex sets with the given probabilities, i.e., 0.35, 0.55, and 0.95. The greater the probability is, the more likely it is that regexes will be matched.

All the experiments use only one core of the 3.40 GHz CPU in the HP Z220 SFF workstation.

7.4.1. Runtime speed

Fig. 14 shows the matching time of the rewritten DFA, Hybrid-FA, PaCCE-xC, and PaCCE. Note that, to make DFA feasible for comparison, all the complex regexes beyond DFA processing capacity are rewritten by narrowing the OFs character sets before evaluation. In other words, the actual performance of DFA is worse than the evaluated performance (ideal value). According to our simulated analysis, the performance of practical DFA should be over one order of magnitude slower than the rewritten DFA, whose size is comparatively very small.

From Fig. 14(a), it can be observed that the speed of the PaCCE-xC engine is quite close to and even better than DFA in terms of lower matching time. For example, for the most complex real-world regex set *deleted*, the PaCCE achieves average 16 cycle/byte (i.e., 1.7 Gbps for 3.40 GHz CPU) performance to process the cernet, campus, and darpa traces, but the rewritten DFA costs average 27 CPU cycles per byte. Besides, the matching speed of PaCCE-xC is relatively stable (i.e., 16–22 cycle/byte for the real-world regex sets, and 25–65 cycle/byte for synthetic regex sets). Besides, the matching speed of PaCCE-xC is a little lower for darpa trace in most cases, because the darpa trace contains the attack flows whose content can be matched by a few real-world regexes and thus the BDFA of PaCCE-xC can be activated sometimes.

Compared to PaCCE-xC, PaCCE is approximately 1–6 times slower (i.e., 21–105 cycle/byte for the real-world regex sets, and 155–203 cycle/byte for the synthetic regex sets). The reason for this can be deduced from two aspects. On one hand, the transition compression may introduce one more state access (at most) for each input character. On the other hand, for each state, all labeled transitions must be compared for the input character in the worst case due to the heterogeneous data structure of the compact DFA. According to the compression ratio, the average number of transitions in each state is between 2 (99% compression ratio) and 10 (96% compression ratio); therefore, the matching speed of PaCCE is slightly worse. However, PaCCE is still much faster (about one order of magnitude faster) than Hybrid-FA, which suffers from very low performance of its NFA part.

Furthermore, Fig. 14(b) shows the performance of these algorithms in the worse situations when more regexes are likely to be matched. This means that more states are frequently traversed in the matching engines. Due to the cache miss, the DFA speed decreases slightly when the matching probability increases. For Hybrid-FA, speed becomes very slow when the probability is 0.95 because, in this case, matching is fully trapped in its NFA part. Although there are also large performance gaps between the synthetic traces with different probabilities (due to the frequent activation of BDFA), the speed of PaCCE-xC (i.e., 16–276 cycle/byte for the real-world regex sets, and 48–304 cycle/byte for the synthetic regex sets) is still close to DFA (i.e., 12–114 cycle/byte for the

Table 6
Statistics of per-flow state units for PaCCE.

Regex set	# Of per-flow state units											
	Cernet trace		Campus trace		Darpa trace		p = 0.35 trace		p = 0.55 trace		p = 0.95 trace	
	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.
backdoor	2	1.000	2	1.000	2	1.000	5	1.127	7	1.566	8	2.294
blacklist	2	1.000	2	1.000	2	1.000	5	1.157	8	1.567	8	4.454
botnet	2	1.001	2	1.000	4	1.005	4	1.080	5	1.282	13	2.886
deleted	2	1.000	2	1.000	2	1.000	2	1.005	3	1.038	4	1.201
ftp	1	1.000	1	1.000	1	1.000	1	1.000	1	1.000	2	1.001
imap	2	1.000	2	1.000	2	1.000	5	1.046	8	1.392	23	5.794
pop3	1	1.000	1	1.000	1	1.000	1	1.000	1	1.000	2	1.001
smtp	2	1.000	2	1.000	3	1.008	3	1.016	4	1.081	31	6.491
spyware	3	1.001	3	1.001	2	1.157	6	1.139	6	1.191	13	5.337
sql	2	1.000	2	1.000	2	1.000	8	1.099	7	1.516	8	4.065
web-activex	2	1.000	2	1.000	2	1.000	6	1.067	5	1.128	13	2.161
web-cgi	2	1.000	2	1.000	2	1.000	4	1.010	4	1.051	3	1.108
web-iis	2	1.000	2	1.000	2	1.009	3	1.003	4	1.072	3	1.093
web-misc	3	1.001	3	1.001	2	1.011	5	1.039	4	1.082	6	1.278
web-php	2	1.000	3	1.000	4	1.009	3	1.014	5	1.156	8	1.082
bro	3	1.000	3	1.000	2	1.006	5	1.206	10	1.927	42	13.035
syn.3	6	1.062	9	1.068	13	1.163	12	2.916	24	8.457	43	22.977
syn.5	12	1.243	17	1.227	23	1.825	15	3.664	32	9.822	53	25.814
syn.7	16	1.410	21	1.534	28	1.908	17	4.429	39	10.565	61	27.712

real-world regex sets, and 46–197 cycle/byte for the synthetic regex sets).

In contrast to the data presented in Fig. 14(a), the PaCCE performance does not reduce significantly (i.e., 40–301 cycle/byte for the real-world regex sets, and 174–395 cycle/byte for the synthetic regex sets), because the PaCCE states can be fully cached in the CPU LLC. Besides, the speed of PaCCE is up to two orders of magnitude faster than Hybrid-FA.

7.4.2. Per-flow statistics

Table 6 shows the statistics of per-flow state units for PaCCE during the matching procedure. The results verify that the average number of per-flow state units in PaCCE is indeed convergent and close to one in a typical case. Due to the convergence of state units, PaCCE-xC and PaCCE match faster than Hybrid-FA.

For the synthetic trace with probability equal to 0.95, the average number of per-flow state units is often large than 2, which means there are always more than 2 state units in the processing queue of state units. This is why the matching speed of PaCCE-xC and PaCCE decreases when the probability increases.

In addition, the maximum number of per-flow state units is often small (i.e., less than 4) for the real-world regex sets and traces, however, for the synthetic regex sets and traces, both the maximum and average number of per-flow state units increase. This is the reason of performance decrease for corresponding PaCCE-xC and PaCCE in Fig. 14. Despite the increased number of state units, PaCCE-xC and PaCCE are much more efficient than Hybrid-FA in the same situation.

7.5. Matching accuracy

In the experiments, the matching accuracy of PaCCE is also verified by comparing PaCCE with Hybrid-FA in the aspect to the matched regexes, the times of occurrences of each matched regex, and the positions where each matched regex occurs. All of the experiments show that the matching of PaCCE is indeed correct.

8. Conclusion

In this paper, we have described PaCC, a distinctive solution designed to build a scalable matching engine for increasingly large sets of complex regexes. PaCC leverages partition to disarm the intrinsic semantic overlapping for regexes. On this basis, PaCC is able to compile the overlapping-free regex segments into non-explosive and super-compact DFA and further compress the DFA by a tail-made transition compression approach. A simple RMT is used to preserve the complete semantics of the original regexes. Finally, PaCC combines the DFA and the RMT together to obtain a succinct matching engine PaCCE. Benefiting from partition, PaCCE achieves high matching performance based on the compact DFA for large scale regex sets, and the small RMT guarantees the semantic equivalence between PaCCE and the original regexes. Experiments and assessments using real-world and synthetic data verified that PaCCE is scalable (like NFA) and sufficiently fast for practical application (like DFA).

References

[1] R. Sommer, V. Paxson, Enhancing byte-level network intrusion detection signatures with context, in: Proceedings of ACM Conference on Computer and Communications Security (CCS), 2003, pp. 262–271.
[2] M. Roesch, Snort-lightweight intrusion detection for networks, in: Proceedings of USENIX Conference on System Administration (LISA), 1999, pp. 229–238.
[3] Snort <http://www.snort.org/>.
[4] V. Paxson, Bro: a system for detecting network intruders in real-time, Comp. Netw. 31 (23) (1999) 2435–2463.
[5] The Bro Network Security Monitor <http://www.bro.org/>.
[6] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, third ed., Addison-Wesley, 2007.
[7] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman, R.H. Katz, Fast and memory-efficient regular expression matching for deep packet inspection, in: Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2006, pp. 93–102.
[8] A. Majumder, R. Rastogi, S. Vanama, Scalable regular expression matching on data streams, in: Proceedings of ACM SIGMOD, 2008, pp. 161–172.
[9] J. Rohrer, K. Atasu, J. van Lunteren, C. Hagleitner, Memory-efficient distribution of regular expressions for fast deep packet inspection, in: Proceedings of IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2009, pp. 147–154.
[10] S. Kumar, B. Chandrasekaran, J. Turner, G. Varghese, Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia, in: Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2007, pp. 155–164.
[11] M. Becchi, P. Crowley, Extending finite automata to efficiently match perl-compatible regular expressions, in: Proceedings of ACM CoNEXT, 2008, pp. 1–12.
[12] R. Smith, C. Estan, S. Jha, S. Kong, Deflating the big bang: fast and scalable deep packet inspection with extended finite automata, in: Proceedings of ACM SIGCOMM, 2008, pp. 207–218.
[13] D. Pasetto, F. Petrini, V. Agarwal, Tools for very fast regular expression matching, Computer 43 (3) (2010) 50–58.
[14] M. Becchi, P. Crowley, A hybrid finite automaton for practical deep packet inspection, in: Proceedings of ACM CoNEXT, 2007, pp. 1–12.

[15] Y. Xu, J. Jiang, Y. Song, T. Jiang, H.J. Chao, i-dfa: A Novel Deterministic Finite Automaton Without State Explosion, Tech. rep., Technical report, Polytechnic Institute of New York University, Brooklyn, NY, 2010.

[16] Y.E. Yang, V.K. Prasanna, Space-time tradeoff in regular expression matching with semi-deterministic finite automata, in: Proceedings of IEEE INFOCOM, 2011, pp. 1853–1861.

[17] C. Liu, J. Wu, Fast deep packet inspection with a dual finite automata, IEEE Trans. Comp. 62 (2) (2013) 310–321.

[18] B.C. Brodie, D.E. Taylor, R.K. Cytron, A scalable architecture for high-throughput regular-expression pattern matching, in: Proceedings of International Symposium on Computer Architecture (ISCA), 2006, pp. 191–202.

[19] S. Kumar, J. Turner, J. Williams, Advanced algorithms for fast and scalable deep packet inspection, in: Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2006, pp. 81–92.

[20] M. Becchi, S. Cadambi, Memory-efficient regular expression search using state merging, in: Proceedings of IEEE INFOCOM, 2007, pp. 1064–1072.

[21] M. Becchi, P. Crowley, An improved algorithm to accelerate regular expression evaluation, in: Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2007, pp. 145–154.

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

[23] C.R. Meiners, J. Patel, E. Norige, E. Torng, A.X. Liu, Fast regular expression matching using small tcams for network intrusion detection and prevention systems, in: Proceedings of USENIX Conference on Security, 2010, pp. 1–16.

[24] Y. Qi, K. Wang, J. Fong, Y. Xue, J. Li, W. Jiang, V. Prasanna, Feacan: Front-end acceleration for content-aware network processing, in: Proceedings of IEEE INFOCOM, 2011, pp. 2114–2122.

[25] T. Liu, Y. Yang, Y. Liu, Y. Sun, L. Guo, An efficient regular expressions compression algorithm from a new perspective, in: Proceedings of IEEE INFOCOM, 2011, pp. 2129–2137.

[26] K. Peng, S. Tang, M. Chen, Q. Dong, Chain-based dfa deflation for fast and scalable regular expression matching using tcam, in: Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2011, pp. 24–35.

[27] J. van Lunteren, A. Guanella, Hardware-accelerated regular expression matching at multiple tens of gb/s, in: Proceedings of IEEE INFOCOM, 2012, pp. 1737–1745.

[28] J. Patel, A.X. Liu, E. Torng, Bypassing space explosion in regular expression matching for network intrusion detection and prevention systems, in: Proceedings of Annual Network and Distributed System Security Symposium (NDSS), 2012, pp. 1–15.

[29] R. Antonello, S. Fernandes, D. Sadok, J. Kelner, G. Szabo, Deterministic finite automaton for scalable traffic identification: the power of compressing by range, in: Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS), 2012, pp. 155–162.

[30] K. Wang, J. Li, Towards fast regular expression matching in practice, in: Proceedings of ACM SIGCOMM, 2013, pp. 531–532.

[31] Regular Expression Processor <http://regex.wustl.edu/>.

[32] R. Sidhu, V.K. Prasanna, Fast regular expression matching using fpgas, in: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2001, pp. 227–238.

[33] B.L. Hutchings, R. Franklin, D. Carver, Assisting network intrusion detection with reconfigurable hardware, in: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2002, pp. 111–120.

[34] C.R. Clark, D.E. Schimmel, Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns, in: Proceedings of International Conference on Field Programmable Logic and Applications (FPL), 2003, pp. 956–959.

[35] J. Bispo, I. Sourdis, J.M.P. Cardoso, S. Vassiliadis, Regular expression matching for reconfigurable packet inspection, in: Proceedings of IEEE International Conference on Field Programmable Technology (FPT), 2006, pp. 119–126.

[36] I. Sourdis, J. Bispo, J.M.P. Cardoso, S. Vassiliadis, Regular expression matching in reconfigurable hardware, J. Sig. Process. Syst. 51 (1) (2008) 99–121.

[37] N. Yamagaki, R. Sidhu, S. Kamiya, High-speed regular expression matching engine using multi-character nfa, in: Proceedings of International Conference on Field Programmable Logic and Applications (FPL), 2008, pp. 131–136.

[38] Y.E. Yang, W. Jiang, V.K. Prasanna, Compact architecture for high-throughput regular expression matching on fpga, in: Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2008, pp. 30–39.

[39] H. Wang, S. Pu, G. Knezek, J. Liu, A modular nfa architecture for regular expression matching, in: Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), 2010, pp. 209–218.

[40] N. Cascarano, P. Rolando, F. Risso, R. Sisto, infant: Nfa pattern matching on gpgpu devices, ACM SIGCOMM Comp. Commun. Rev. (CCR) 40 (5) (2010) 20–26.

[41] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, Q. Dong, Gpu-based nfa implementation for memory efficient high speed regular expression matching, in: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2012, pp. 129–140.

[42] R. Smith, C. Estan, S. Jha, Xfa: Faster signature matching with extended automata, in: Proceedings of IEEE Symposium on Security and Privacy (SSP), 2008, pp. 187–201.

[43] J. Hopcroft, An n log n Algorithm for Minimizing States in a Finite Automaton, Tech. rep., Technical report, Stanford University, Stanford, CA, 1971.

[44] S. Zhang, H. Luo, B. Fang, C. Yun, An efficient regular expression matching algorithm for network security inspection, Chin. J. Comp. 33 (10) (2010) 1976–1986 (in Chinese).

[45] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, J. Turner, Algorithms to accelerate multiple regular expressions matching for deep packet inspection, in: Proceedings of ACM SIGCOMM, 2006, pp. 339–350.

[46] Darpa Intrusion Detection Data Sets <http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data/>.