A Prefiltering Approach to Regular Expression Matching for Network Security Systems

Tingwen Liu[1,2], Yong Sun[3], Alex X. Liu[4], Li Guo[3], and Binxing Fang[1,3]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] Graduate University of Chinese Academy of Sciences, Beijing, China
[3] National Engineering Laboratory for Information Security Technologies, Beijing
[4] Dept. of Computer Science and Engineering, Michigan State University
{liutingwen,suny}@software.ict.ac.cn alexliu@cse.msu.edu

Abstract. Regular expression (RegEx) matching has been widely used in various networking and security applications. Despite much effort on this important problem, it remains a fundamentally difficult problem. DFA-based solutions can achieve high throughput, but require too much memory to be executed in high speed SRAM. NFA-based solutions require small memory, but are too slow. In this paper, we propose RegexFilter, a prefiltering approach. The basic idea is to generate the RegEx print of RegEx set and use it to prefilter out most unmatched items. There are two key technical challenges: the generation of RegEx print and the matching process of RegEx print. The generation of RegEx is tricky as we need to tradeoff between two conflicting goals: filtering effectiveness, which means that we want the RegEx print to filter out as many unmatched items as possible, and matching speed, which means that we want the matching speed of the RegEx print as high as possible. To address the first challenge, we propose some measurement tools for RegEx complexity and filtering effectiveness, and use it to guide the generation of RegEx print. To address the second challenge, we propose a fast RegEx print matching solution using Ternary Content Addressable Memory. We implemented our approach and conducted experiments on real world data sets. Our experimental results show that RegexFilter can speedup the potential throughput of RegEx matching by 21.5 times and 20.3 times for RegEx sets of Snort and L7-Filter systems, at the cost of less than 0.2 Mb TCAM chip.

Keywords: regular expression, prefilter, RegEx print, TCAM

1 Introduction

Regular expressions (RegExes) have been widely used in a variety of network and security applications, such as anti-virus scanners [1], network intrusion detection and prevention systems [2], firewalls, traffic classification and monitoring [3]. In intrusion detection and prevention systems, RegExes are used to specify attack signatures. In traffic classification and monitoring, RegExes are used to specify the signature of application protocols, thus allowing the classification and monitoring of network traffic based on application protocols. The widespread usage is because of the expressive power, simplicity and flexibility of RegExes in specifying signatures.

The RegEx matching problem can be defined as follows: given a set R of RegExes, at run time, for each incoming item i (*e.g.*, packets), we want to get RegEx set $O(R, i)$ whose members are all matched by the item. RegEx matching, as the core operation of many applications, needs to be done at high speed with small memory. However, despite much work that has been done, this remains a fundamentally difficult problem. As a set of RegExes can be formally represented as a Deterministic Finite Automata (DFA) or Nondeterministic Finite Automata (NFA), prior RegEx matching solutions often fall into two categories: DFA-based and NFA-based. First, DFA-based solutions may achieve high speed because at any time there is only one active state, but may require too much memory. For applications running on networking devices such as intrusion detection and prevention systems and application firewalls, RegEx matching needs to be done in high speed SRAM, which has small capacity in terms of a few megabytes. Second, NFA-based solutions require small memory, but cannot achieve high speed because at any time there may be many active states [4].

In this paper, we propose RegexFilter, a prefiltering approach to RegEx matching for network security systems. Given a RegEx set R, we want to construct another RegEx set R' so that any unmatched item of R' is also an unmatched item of R. An *unmatched item of a RegEx set* is an item that does not match any RegEx in the set. Moreover, we want the matching efficiency of R' to be much higher than that of R; thus, we can use R' as a prefilter procedure of R: Given an item i, we first match it against R' and get set $O(R', i)$, if $O(R', i)$ is empty, then it for sure does not match any member in R and therefore we can skip this item safely; otherwise $O(R', i)$ is not empty, then we continue to match it against $T(R, O(R', i))$, where $O(R, i) \subseteq T(R, O(R', i)) \subseteq R$, and $T(R, O(R', i))$ can be obtained reversely from $O(R', i)$. Because most items are unmatched items for network security systems [5] and the matching cost of R' is much less than that of R, the overall throughput of this prefiltering approach is much higher than directly matching against R. We call R' the RegEx print of R. According to the main idea, RegexFilter divides the RegEx matching process into two stages: filtering stage and verifying stage. The filtering stage performs high-speed RegEx print matching on each arriving item. If one RegEx print is matched, the corresponding RegEx will be checked in the verifying stage.

There are two main technical challenges to implementing RegexFilter. The first challenge is the construction of the RegEx print for a given RegEx set. On one hand, we want the RegEx print to filter out as many unmatched items as possible. On the other hand, we want the matching efficiency of the RegEx print to be as high as possible. These two goals are unfortunately conflicting. With the RegEx print being the original RegEx set, the RegEx print can filter out all unmatched items, but the matching efficiency is the lowest. With the RegEx print being empty, the matching efficiency of zero cost is the highest, but it cannot filter out any unmatched item. We need to carefully tradeoff between these two conflicting goals. The second challenge is the matching of items against RegEx prints. We want this process to be as fast as possible to achieve high overall RegEx matching throughput.

To address the first challenge, we first propose a method to generate all possible RegEx prints for each RegEx in the given RegEx set. Second, we propose an estimation method to quantitatively measure the filtering effectiveness and complexity of RegEx prints. Third, we reduce the problem of selecting the RegEx prints with high filtering effectiveness and low complexity among all candidate RegEx prints of a RegEx to the classical 0-1 knapsack problem. In this paper, we use dynamic programming to choose RegEx prints among all candidates with the goal of maximizing filtering effectiveness while keep the complexity of a RegEx print less than a predefined threshold.

To address the second challenge, we propose a Ternary Content Addressable Memory (TCAM) based solution for RegEx print matching. As larger TCAMs have lower lookup frequency, require more power, generate more heat, and also have high hardware cost, we want to minimize the TCAM space required to encode the RegEx print DFA. Unfortunately, as minimizing TCAM space is NP-hard, we propose a heuristic method that uses the Quine-McCluskey algorithm to reduce TCAM space.

We make three key contributions in this paper. First, we propose an efficient method to generate RegEx prints. In particular, we propose some measurement tools for RegEx complexity and filtering effectiveness, and then use the tools to guide the generation of RegEx print. Second, we propose an efficient method of implementing RegEx print matching based on TCAMs. Third, we implemented our approach and conducted experiments on real-world RegEx sets and traffic traces. Our experimental results show that RegexFilter can speedup the throughput of RegEx matching by 21.5 times and 20.3 times for RegEx sets of Snort and L7-Filter systems, at the cost of less than 0.2 Mb TCAM chip.

The rest of the paper is organized as follows. We review related work in Section 2. In Sections 3 and 4 we explain the generation of RegEx prints and the implementation of high-speed RegEx print matching in TCAM for RegexFilter respectively. In Section 5, we present experimental results. Finally, We give conclusions and future work in Section 6.

2 Related Work

As DFA is the preferred representation of RegEx matching, recent work has focused on reducing the huge memory usage of DFA-based RegEx matching [4, 6–11]. However, they achieve memory reduction only for signature sets of simple or specific RegExes. None of them can achieve high-speed RegEx matching for real-world signature sets that contain thousands of complex RegExes. However, these solutions are orthogonal to our work as they focus on improving RegEx matching in our verifying stage. Meiners et al. [12] propose a well-designed TCAM-based RegEx matching solution that introduces three novel techniques to reduce TCAM space and improve matching speed. The solution cannot work on real-world signature sets directly as the composite DFAs are too big to be encoded in a TCAM chip. However, it can be used in our RegEx print matching as the RegEx print DFA is small enough even for real-world RegEx sets. In fact, we design a more effective solution, that goes a step further and tackles the prefix problem in [12].

Several string-based prefiltering techniques have been developed to improve the performance of RegEx matching [5, 13–16]. To the best of our knowledge, the most outstanding one is sigMatch [5]. The sigMatch technique organizes a signature set into a (processor) cache-efficient q-gram index structure, called the sigTree. For a given signature set, sigMatch requires that each signature has at least one string of length b to construct its sigTree. For these signatures that do not satisfy the requirement, sigMatch rewrites them into multiple signatures that have at least one string of length b in enumerating idea. Then, for each signature, sigMatch picks exactly one discriminative substring (without any meta-characters of RegExes) of length $b + \beta$ as its fingerprint. The first b bytes of the substring map the signature to a sigTree node, and the next β bytes following the b bytes in the substring are used to hash into the Bloom Filter at that node using a "set" of hash functions. Linked lists are used in sigTree nodes for short signatures that do not have a substring of length $b + \beta$.

These string-based prefiltering techniques have three major drawbacks. First, they suffer from the problem of member set explosion when they are applied to RegExes with character subclasses. An example is RegEx `^[a-z][a-z0-9]{5, 15}` that matches user ID starting with an English alphabet followed by some alphanumeric characters. These techniques have to enumerate all possible strings represented by the RegEx, the size of which is more than 26^{16}. Obviously this step is time-consuming and impracticable. Second, the fingerprints generated by these techniques are strings, which do not include the positioning of RegExes. For example anchor `^` in the above RegEx, which indicates that successful matchings must start from the beginning position of items. This inability leads to the problem that they may have poor filtering effectiveness. Third, there needs to be one Bloom Filter for each possible length of fingerprints [17]. The hardware cost can be prohibitive if fingerprints have a large number of distinct lengths. RegexFilter addresses these problems by generating short RegExes as fingerprints for each original RegEx without enumerating its all possible strings, and performing RegEx print matching with DFA representation in TCAM.

Ficara et al. [18] propose the first RegEx-based prefiltering solution that uses sampling technique to accelerate RegEx matching. The main idea is to sample a byte every θ bytes over packet payloads (θ is the sampling period). The sampled payloads are then used to match with a proper sampled DFA constructed from sampled RegExes. The method can process normal packets θ times faster at the cost of false-positive alarms. However, sampling RegExes correctly sometimes is very hard. Moreover, the sampled DFA may still experience state explosion . For example, RegEx `ab.{1024}cd` is sampled into two RegExes `a.{512}c` and `b.{512}d` given sampling period $\theta = 2$, the sampled DFA constructed from the two sampled RegExes has billions of states.

3 RegEx Print Generation

For a given RegEx, we first present a method to find all of its possible RegEx prints in this section, and then we introduce an algorithm to selectively generate good RegEx prints that satisfy our goal.

3.1 RegEx Print

Before presenting our work, we give some definitions to be used first. A RegEx r is a string over symbol set $\Sigma \cup \{\epsilon, |, \cdot, {}^*, (,)\}$, which is recursively defined as the empty character ϵ; a character $\alpha \in \Sigma$; and (r_1), $r_1 \cdot r_2$, $r_1|r_2$, and $r_1{}^*$, where r_1 and r_2 are RegExes. It represents a set of strings without enumerating them explicitly over alphabet Σ, which is defined recursively on the structure of r as follows:

- if r=ϵ, $S(r)$=$\{\epsilon\}$, the empty string
- if r=α ($\alpha \in \Sigma$), $S(r)$=$\{\alpha\}$, a single string of one character
- if r=(r_1), $S(r)$=$S(r_1)$
- if r=$r_1 \cdot r_2$, $S(r)$=$S(r_1) \cdot S(r_2)$, where $S(r_1) \cdot S(r_2)$ is the set of strings w such that w=w_1w_2, with $w_1 \in S(r_1)$ and $w_2 \in S(r_2)$. The operator '$\cdot$' represents the classical concatenation of strings
- if r=$r_1|r_2$, $S(r)$=$S(r_1) \cup S(r_2)$, the union of the two sets. The operator '$|$' is called union operator.
- if r=$r_1{}^*$, $S(r)$=$S(r_1)^*$=$\bigcup_{i=0}^{\infty} S(r_1)^i$, where S^0=$\{\epsilon\}$ and S^i=$S \cdot S^{i-1}$ for any string set S. That is, the result is the set of strings formed by a concatenation of zero or more strings represented by r_1. The operator '*' is called star operator.

In order to construct an automata (NFA or DFA) for RegExes, most of the constructions use a binary tree representation as an intermediate form. The leaves of the tree are labeled with the characters of alphabet Σ or the symbol ϵ, and the internal nodes are labeled with the operators. The nodes that are labeled with '$|$' or '$\cdot$' have two children, while nodes labeled with '*' have only one child. Prior work describe how to parse a RegEx to obtain its parse tree recursively, in fact this conversion is reversible. Given a parse tree, its original RegEx can be obtained recursively just like the parsing process. In our work, we perform the generation of RegEx prints over the parse-tree representation for a given RegEx.

Definition 1. *Given a RegEx r, its Expression Size, denoted by $\mathbb{ES}(r)$, is the number of strings represented by r, namely $\mathbb{ES}(r) = |S(r)|$.*

For the given RegEx r, how to calculate its $\mathbb{ES}$ value is an open question. One simple method is to enumerate all the strings represented by r according to the definition, and then count the size. However, it is very inefficient as mentioned above. In this paper, we propose a novel method that can calculate $\mathbb{ES}(r)$ approximately, as shown in the following recursive way:

- if $r{=}\epsilon$, $\mathbb{ES}(r) = 1$
- if $r{=}\alpha$ $(\alpha \in \Sigma)$, $\mathbb{ES}(r) = 1$
- if $r{=}(r_1)$, $\mathbb{ES}(r) = \mathbb{ES}(r_1)$
- if $r{=}r_1 \cdot r_2$, $\mathbb{ES}(r) = \mathbb{ES}(r_1) \times \mathbb{ES}(r_2)$ (We can infer that $\mathbb{ES}(r) = \mathbb{ES}(r_1)^n$ if $r{=}r_1\{n\}$)
- if $r{=}r_1|r_2$, $\mathbb{ES}(r) = \mathbb{ES}(r_1) + \mathbb{ES}(r_2)$
- if $r{=}r_1{}^*$, $\mathbb{ES}(r) = \sum_{t=0}^{\infty} \mathbb{ES}(r_1)^t = \infty$ ($\mathbb{ES}(r_1)$ is an integer no less than 1)

The easiest cases are single characters and ϵ. For operators '$\cdot$', '|' and '*', the equations do not hold strictly, because there may be the same strings among all the combinations. For example, RegEx `a(b|`ϵ`)(b|`ϵ`)c` represents a set of strings {`ac, abc, abbc`} (because of $\epsilon \cdot r = r \cdot \epsilon = r$), its real expression size should be 3. According to our calculating method, we get $\mathbb{ES}(\texttt{a(b|}\epsilon\texttt{)(b|}\epsilon\texttt{)c}) = \mathbb{ES}(\texttt{a}) \times \mathbb{ES}(\texttt{b|}\epsilon) \times \mathbb{ES}(\texttt{b|}\epsilon) \times \mathbb{ES}(\texttt{c}) = 1 \times 2 \times 2 \times 1 = 4$. Fortunately, our method produces approximate $\mathbb{ES}$ values that are very close to the real values.

Similarly, we define the Expression Size of a RegEx set $R = \{r_1, \cdots, r_n\}$, denoted by $\mathbb{ES}(R)$, as the number of strings represented by $r_1|\cdots|r_n$. We can infer that $\mathbb{ES}(R) = \sum_{i=1}^{n} \mathbb{ES}(r_i)$ according to our calculating method of $\mathbb{ES}$.

Obviously, a RegEx set has larger or equal $\mathbb{ES}$ value than any of its RegEx prints. Meanwhile, a RegEx set has higher complexity than any of its RegEx prints, where the complexity is regarded as state size for DFA-based matching solutions in this paper. Inspired by the insight, we argue that $\mathbb{ES}$ can be used as a measurement tool to compare the complexity even for two totally different RegEx sets. This speculation is reasonable intuitively: a RegEx set with larger $\mathbb{ES}$ value represents more strings, and more strings consume more states when constructing Aho-Corasick complete automata (a special DFA for string matching). A persuasive example is a string signature, such as r_1=`ACNS`, and a RegEx signature that contains constrained repetitions of wildcards, such as r_2=`AC.{10}NS`. After calculating we know $\mathbb{ES}(r_1) = 1$ and $\mathbb{ES}(r_2) = 256^{10}$, meanwhile the DFA of r_1 has 5 states and the DFA of r_2 has more than one thousand states. An important application of our $\mathbb{ES}$ tool is that it can be used to deal with state explosion pertinently by combining with previous work [11, 10, 19, 9]. Because we can locate the accurate positions that will lead to state explosion quantitatively by calculating $\mathbb{ES}$, while previous work solve the problem qualitatively and empirically.

A signature with better filtering effectiveness is matched with a lower probability. A string, which is a RegEx too, only represents itself. Moreover, the length of a string is fixed. It is easy to prove that the matching probability of a string is inverse to the size of alphabet Σ to the power of its length over random inputs of infinite length. Some prefiltering work tend to generate longer strings as fingerprints because they will be matched much less frequently than shorter ones. However, it is hard to measure the matching probability of a RegEx, because a RegEx usually represents many strings of different lengths. Motivated by the insight that shorter strings have much higher matching probability, we speculate that the matching probability of a RegEx mainly depends on its Minimum Expression Length and its Shortest Expression Size, which are defined as follows.

Definition 2. *Given a RegEx r, the Minimum Expression Length of r, denoted by $\underline{\mathbb{L}}(r)$, is the number of characters in s, where s is the shortest string in set $S(r)$.*

Unlike in the calculation of $\mathbb{ES}$ value, the same strings in $S(r)$ do not change $\underline{\mathbb{L}}$ value for the given RegEx r. Thus, we can calculate $\underline{\mathbb{L}}(r)$ accurately in the following recursive way:

- if r=ϵ, $\underline{\mathbb{L}}(r) = 0$
- if r=α ($\alpha \in \Sigma$), $\underline{\mathbb{L}}(r) = 1$
- if r=(r_1), $\underline{\mathbb{L}}(r) = \underline{\mathbb{L}}(r_1)$
- if r=$r_1 \cdot r_2$, $\underline{\mathbb{L}}(r) = \underline{\mathbb{L}}(r_1) + \underline{\mathbb{L}}(r_2)$
- if r=$r_1|r_2$, $\underline{\mathbb{L}}(r) = min(\underline{\mathbb{L}}(r_1), \underline{\mathbb{L}}(r_2))$
- if r=$r_1{}^*$, $\underline{\mathbb{L}}(r) = 0$

Definition 3. *Given a RegEx r, the Shortest Expression Size of r, denoted by $\mathbb{SES}(r)$, is the number of strings of length $\underline{\mathbb{L}}(r)$ in set $S(r)$.*

Similar to the calculation of $\mathbb{ES}$, we can calculate $\mathbb{SES}$ approximately with the following method:

- if r=ϵ, $\mathbb{SES}(r) = 1$
- if r=α ($\alpha \in \Sigma$), $\mathbb{SES}(r) = 1$
- if r=(r_1), $\mathbb{SES}(r) = \mathbb{SES}(r_1)$
- if r=$r_1 \cdot r_2$, $\mathbb{SES}(r) = \mathbb{SES}(r_1) \times \mathbb{SES}(r_2)$
- if r=$r_1|r_2$:
  - if $\underline{\mathbb{L}}(r_1)$ is bigger than $\underline{\mathbb{L}}(r_2)$, $\mathbb{SES}(r) = \mathbb{SES}(r_2)$
  - if $\underline{\mathbb{L}}(r_1)$ is smaller than $\underline{\mathbb{L}}(r_2)$, $\mathbb{SES}(r) = \mathbb{SES}(r_1)$
  - if $\underline{\mathbb{L}}(r_1)$ equals to $\underline{\mathbb{L}}(r_2)$, $\mathbb{SES}(r) = \mathbb{SES}(r_1) + \mathbb{SES}(r_2)$
- if r=$r_1{}^*$, $\mathbb{SES}(r) = 1$

Definition 4. *The matching probability of a RegEx r, denoted by $\mathbb{MP}(r)$, is defined as the ratio of $\mathbb{SES}(r)$ to $\mathbb{SCS}(r)$, where $\mathbb{SCS}(r)$ represents the total number of strings of length $\underline{\mathbb{L}}(r)$ over alphabet Σ.*

Definition 5. *A RegEx r is dividable if and only if it can be rewritten into the form of $r_1 \cdot r_2$, where r_1 and r_2 are RegExes, and $S(r_1) \neq \{\epsilon\}$, $S(r_2) \neq \{\epsilon\}$. RegEx r is atomic if it is not dividable.*

Lemma 1. *Given a RegEx $r = r_1 \cdots r_n$, where r_t is atomic ($1 \leq t \leq n$). We abbreviate RegEx r of this type as $r = r_{[1,n]}$ later. Then 1) $r_{[i,j]}$ ($1 \leq i \leq j \leq n$) is a RegEx print of r; 2) RegEx r has at most $\frac{n(n+1)}{2}$ RegEx prints.*

Proof. 1) First, $r_{[i,j]}$ is obviously a RegEx. Second, it is a fingerprint, because any input T matched by r will be matched by $r_{[i,j]}$: T must contains one string s in set $S(r)$, while s is the concatenation of a string in set $S(r_{[1,i-1]})$, a string in set $S(r_{[i,j]})$ and a string in set $S(r_{[j+1,n]})$.

2) Since 1) is right, the proof of 2) is simple and trivial.

3.2 RegEx Print Generation Algorithm

For a RegEx set R, we want to generate a set of RegEx prints that produces a DFA with as few states as possible and prefilters as many items as possible. However, comparing among all possible RegEx print sets is high-cost. In this section, we present a novel algorithm that can achieve the goal with low cost by pruning uncompetitive RegEx print sets. Our algorithm involves three stages: selecting stage, refining stage and deciding stage, which are described below.

Selecting Stage. As compiling the RegEx print set into a composite DFA within limited memory is a hard requirement, we try to limit the DFA state size, at the time we want to bypass the time-consuming process of DFA construction. The main idea is to select those RegEx prints whose $\mathbb{ES}$ values are no more than a predefined expression size threshold β for each RegEx. Detailedly speaking, for a given dividable RegEx $r_{[1,n]}$, RegEx print $r_{[i,j]}$ is selected in this stage if it satisfies the following conditions: 1) $\mathbb{ES}(r_{[i,j]}) \leq \beta$; 2) $\mathbb{ES}(r_{[s,t]}) > \beta$ for $s \leq i$ and $t > j$, or $s < i$ and $t \geq j$. We can easily prove that $r_{[i,j]}$ is also a RegEx print of $r_{[s,t]}$.

Before describing our algorithm, we introduce a theorem first.

Theorem 1. *For any dividable RegEx $r = r_1 \cdot r_2$, where r_1, r_2 are RegExes, the following two conditions hold: 1) $\mathbb{ES}(r) \geq \mathbb{ES}(r_1)$, meanwhile $\mathbb{MP}(r) \leq \mathbb{MP}(r_1)$.*

Proof. According to the above calculation methods, we know that $\mathbb{ES}$ value is not less than one meanwhile $\mathbb{MP}$ value is not more than one for any RegEx. Thus $\mathbb{ES}(r_2) \geq 1$, $\mathbb{MP}(r_2) \leq 1$. Then we can infer that: 1) $\mathbb{ES}(r) = \mathbb{ES}(r_1) \times \mathbb{ES}(r_2) \geq \mathbb{ES}(r_1)$, moreover $\mathbb{ES}(r) = \mathbb{ES}(r_1)$ only when r_2 represents the string set $\{\epsilon\}$ or $\{\alpha\}$ ($\{\alpha \in \Sigma\}$); 2) $\mathbb{MP}(r) = \frac{\mathbb{SES}(r)}{|\Sigma|^{\mathbb{L}(r)}} = \frac{\mathbb{SES}(r_1)\times\mathbb{SES}(r_2)}{|\Sigma|^{\mathbb{L}(r_1)+\mathbb{L}(r_2)}} = \mathbb{MP}(r_1) \times \mathbb{MP}(r_2) \leq \mathbb{MP}(r_1)$, moreover $\mathbb{MP}(r) = \mathbb{MP}(r_1)$ only when the shortest expressing string set of r_2 is its shortest complete string set.

Theorem 1 accords with our intuition: a RegEx requires more resource but has better filtering effectiveness than any of its RegEx print. Our algorithm works recursively in post-order traversal from the root node of the parse tree of $r_{[1,n]}$ to select RegEx prints. Figure 1 shows the selecting process of RegEx "`a[bc]d.[bc]`" that has five atoms. Given $\beta = 256$, we begin the selecting stage from the first atom in step 1, *curr* pointer keeps moving to the next atom if $\mathbb{ES}$ value of the RegEx print between *begin* pointer and *curr* pointer is less than or equal to β. When *curr* pointer arrives at the fourth atom "`.`", condition 1 does not hold, thus RegEx print `a[bc]d` is selected. Because we can infer that $\mathbb{ES}$ value of any RegEx print that contains "`a[bc]d.`" is longer than 256 according to Theorem 1, we begin step 2 from the second atom. Although RegEx print `[bc]d` satisfies the two conditions at the same time, it is included in the already selected RegEx print "`a[bc]d`". According to Theorem 1 we know that `a[bc]d` has higher $\mathbb{MP}$ value than `[bc]d`, thus `[bc]d` is not selected. Step 3, 4 and 5 follow the same idea to select RegEx prints.

For RegEx $r_{[1,n]}$, the $\mathbb{ES}$ value of a atom, supposing r_i, may be larger than β. Obviously, any RegEx print containing r_i will not be selected. However, these

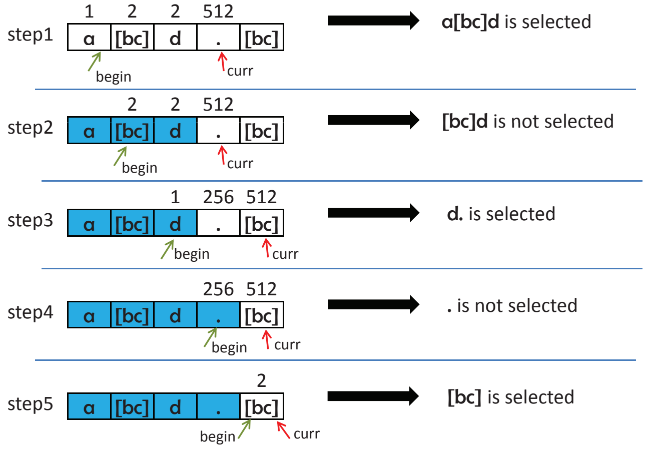


Fig. 1. Selecting RegEx prints for RegEx `a[bc]d.[bc]` with $\beta = 256$. In each step atoms in blue indicate that they are covered by already selected RegEx prints, the next selected RegEx print must contain at least one atom that is not in blue.

RegEx prints may have low $\mathbb{MP}$ value while all the remaining RegEx prints have high $\mathbb{MP}$ value. To address this problem, we rewrite r into multiple RegExes until no atom has $\mathbb{ES}$ value larger than β or no atom is in the form of the union of RegExes. For instance, given $\beta = 256$, RegEx $r =$`(tele|phone|AC.*NS)[^a]` has only one RegEx print `[^a]` whose $\mathbb{MP}$ value is close to 1. We can rewrite it into two RegExes: $r_1 =$`(tele|phone)[^a]`, $r_2 =$`AC.*NS[^a]`, then we can select RegEx prints with the same $\mathbb{ES}$ value and lower $\mathbb{MP}$ value for rewritten RegExes.

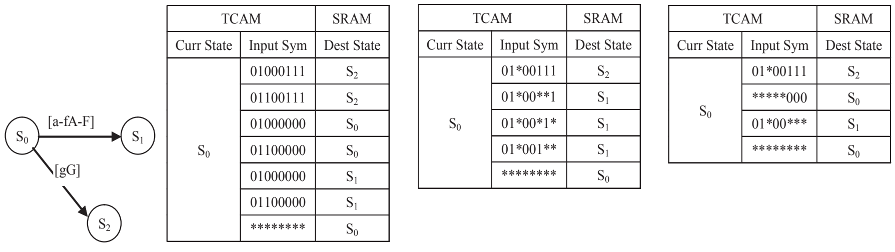
Refining Stage. In this stage, we refine the RegEx prints selected in the last stage in the following steps. Step 1, for each selected RegEx prints, the first atom (the last atom) whose $\mathbb{MP}$ value equals 1 should be removed repeatedly. For the example in Figure 1, the wildcard in "`d.`" introduces very limited filtering effectiveness by requiring one random symbol after "`d`". However it will introduce much state in the RegEx print DFA. Thus, we need to remove the wildcard. RegEx prints of the example become `a[bc]d`, `d` and `[bc]` now. Step 2, these RegEx prints, whose atoms are included by other RegEx prints, should be deleted according to Theorem 1. For the above example, RegEx print `d` has only one atom that is included by RegEx print `a[bc]d`. We should delete RegEx print `d` because it is meaningless: at any time RegEx `a[bc]d` is matched means `d` must be matched. One thing to notice that, RegEx print `[bc]` should not be deleted if allowing one RegEx has more than one RegEx print (see in deciding stage). Because its atoms are not included by `a[bc]d`. Step 3, RegEx prints with positioning (or anchors) should be kept as many as possible. Because positioning add the limitation that these RegEx prints must be matched at the special positions of items. As a result, they will not be matched frequently even for these RegEx prints with high $\mathbb{MP}$ value. In our implementation, we regard the filtering effectiveness of `^` as a normal character to decrease $\mathbb{MP}$ value while not increase $\mathbb{ES}$ value.

Deciding Stage. In this stage, we decide the final RegEx print for each RegEx by limiting its $\mathbb{MP}$ value no larger than a predefined matching probability threshold η. Given a RegEx $r = r_{[1,n]}$, assuming it has k RegEx prints $(p_1, \cdots, p_k)$ left after refining stage: if $\mathbb{MP}$ value of one RegEx print is smaller than η, we removing the first atom (or the last atom) repeatedly in the RegEx print to make its $\mathbb{MP}$ value larger than η, at the same time close to η as much as possible; if $\mathbb{MP}$ value of any one RegEx print is larger than η, then we use multiple RegEx prints together for RegEx r to reduce the number of items that need to be verified. Because if a RegEx has multiple RegEx prints, only when an item is matched by these RegEx prints sequentially (the order depends on the position of the first atom of these RegEx prints), the item will be verified. For RegEx prints `a[bc]d` and `[bc]` in Figure 1, the former one matched does not imply that the latter one must be matched in sequential matching order. One vivid data item is `abdea`. How to choose the final RegEx prints is an open problem at present. We want the sum of $\mathbb{ES}$ value of these final RegEx prints is smaller than or equal to β and the product of $\mathbb{MP}$ value of these final RegEx prints is as large as possible. Obviously this is a typical 0-1 knapsack problem. In this paper, we achieve the goal using the classical dynamic programming solution for 0-1 knapsack problem.

4 Regex Print Matching

As the RegEx print set in RegexFilter can be compiled into a composite DFA within limited memory, prior TCAM-based DFA matching solutions can be used here directly for high-speed RegEx print matching. TCAM is a special type of memory which takes input of data as key to look-up address. It has the following three capacities: i) ternary states encoding: 0's, 1's, and *'s where *'s stand for either 0 or 1, enabling one TCAM entry to encode multiple DFA transitions; ii) parallel content lookup, enabling TCAM to complete lookups in a single operation no matter the number of occupied TCAM entries; iii) first-match semantic, making TCAM to return the index of the first address for the content that the key matches.

Meiners et al. [12] propose a well-designed TCAM-based RegEx matching solution, which uses three novel techniques to reduce TCAM space and improve RegEx matching speed: transition sharing, table consolidation, and variable striding. The main idea is to encode multiple DFA transitions into a TCAM entry by the help of TCAM capacities. However, the character bundling algorithm used to encode transitions inside each state in the work is designed for TCAM-based packet classification applications, which produce prefix TCAM entries: the predicate of each entry is a prefix bit string (*e.g.*, 01**) where no 0 and 1 behind *. In fact, a ternary TCAM entry allows * to appear at any positions (*e.g.*, 0**1), which means it misses the opportunity of encoding transitions created by non-prefix entries.



(a) Transitions of State S_0

TCAM		SRAM
Curr State	Input Sym	Dest State
S_0	01000111	S_2
	01100111	S_2
	01000000	S_0
	01100000	S_0
	01000000	S_1
	01100000	S_1
	********	S_0

(b) Character Bundling

TCAM		SRAM
Curr State	Input Sym	Dest State
S_0	01*00111	S_2
	01*00**1	S_1
	01*00*1*	S_1
	01*001**	S_1
	********	S_0

(c) Transition Logic Simplification

TCAM		SRAM
Curr State	Input Sym	Dest State
S_0	01*00111	S_2
	*****000	S_0
	01*00***	S_1
	********	S_0

(d) Optimal Encoding

Fig. 2. Outgoing transitions of state S_0 and their different encodings in TCAM

In this section we go a step further and tackle the prefix problem. For any DFA state S_i, it has 256 transitions (assuming alphabet Σ is ASCII), which have the same current state and completely different 8-bit input symbols. Obviously encoding all the 256 transitions into TCAM entries can be regard as the simplification of a logic function with 8-input 1-output, hereinafter referred to as a transition logic function. However. the outputs of the transition logic function have $|S|$ possible values, which is different from the classical logic function that always have a logic value of either "0" or "1". Using exhaustive searching method can get the optimal solution, but its cost is higher than that of the classical Quine-McCluskey algorithm.

To reduce the complexity we add a limitation of encoding the transitions with the same output together: treating a destination state as value "1" of classical logic functions, and simplifying the transitions to the state with Quine-McCluskey algorithm; then setting their destination states to irrelevant term (any state is allowed), and encoding the remaining transitions. We give a detailed description with the example of encoding the transitions in Figure 2 (a). Current state S_0 moves to state S_1 if the input symbol is in character range `[a-fA-F]`, moves to state S_2 along g and G, and moves to itself for the remaining input symbols. As long as the occupied TCAM entries are arranged according to the encoding order of destination states, the capacity of first-match semantic ensures the correctness of lookup results. Figure 2 (c) shows the result with the encoding order $S_2 \rightarrow S_1 \rightarrow S_0$, which occupies 5 TCAM entries, while character bundling algorithm occupies 7 entries, as shown in Figure 2 (b). One thing to notice is that the occupied TCAM entries of our encoding is relevant to the encoding order.

In this paper, we do not address the encoding order problem by testing all possible orders. We propose a near-optimal solution to the problem based on the distribution of transitions with the same destination state: for each DFA state, its 256 outgoing transitions moves to few destination states, furthermore the distribution of these transitions is very uneven. This observation can be verified by the statistical results over the whole state set averagely. Given a state S_i in state set S, assuming its 256 outgoing transitions move to M_i different

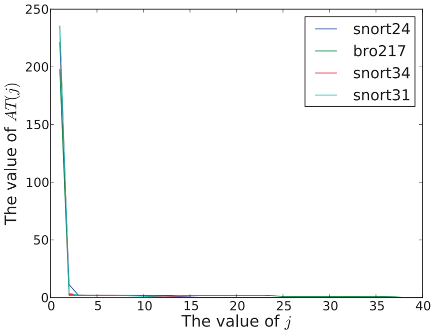


Fig. 3. The distribution of $AT(j)$ for Bro217, Snort24, Snort31 and Snort34

destination states, thereinto N_{ij} transitions move to the j-th destination state ($1 \leq j \leq M_i$, and N_{i*} is in descending order. If not, sorting them). Obviously, $\sum_{j=1}^{j\leq M_i} N_{ij} = 256$. For state set S, the number of different destination states per state ADS is defined as $\frac{\sum_{i=0}^{i<|S|} M_i}{|S|}$, its average number of j-th most transitions $AT(j)$ is defined as $\frac{\sum_{i=0}^{i<|S|} N_{ij}}{|S|}$.

Taking RegEx sets bro217, snort24, snort31 and snort34 (widely used in prior experiments) as examples, the corresponding DFAs has 36.8, 14.4, 11.6 and 12.3 different destination states in average. The distribution of $AT(j)$ for the DFAs is shown in Figure 3: the transitions to the most destination state ($AT(0)$) account for 90% proportions; the transitions to other destination states roughly equal, and the value is no more than 2 in most case. Therefore, a preferable encoding order of is determined heuristically by the number of the transitions to the same destination state. Because when the number is one or two, encoding them will consume the same TCAM entries regardless of the order. Nevertheless, encoding them first can improve subsequent results. In Figure 2 (c), encoding the transitions to state S_0 only consumes one TCAM entries.

An important thing to note is that the heuristic order does not guarantee to minimize the transition logic function. The optimal encoding is shown in Figure 2 (d), which encodes partial transitions to state S_0 first. This change makes one TCAM entry enough to encode all the transitions to state S_1.

5 Experimental Results

In this section, we first give a brief description of our experimental setup. Then we evaluate RegexFilter on the metrics of memory consumption and matching performance. At last we show how RegexFilter changes as the change of expression size threshold β and matching probability threshold η.

Table 1. Comparison of state size among SinstrFilter, MulstrFilter and RegexFilter.

RegEx	# of RegExes		# of DFA states / # of RegExes unable to handle		
sets	original	rewritten	SinstrFilter	MulstrFilter	RegexFilter
backdoor	158	161	1452 / 0	2128 / 0	2302 / 0
l7filter	107	166	1317 / 8	1541 / 8	2847 / 0

5.1 Experimental Setup

In this paper, we evaluate RegexFilter on RegEx sets extracted from two real-world systems, namely L7-Filter [3] and Snort [2]. L7-Filter is a popular open-source application layer traffic classifier for Linux. It re-assembles the payload content of a flow and identifies its application level protocol through RegEx matching. The latest version of L7-Filter has 112 RegExes for traffic classification. In this paper, we remove five RegExes that are overmatched, and select the remaining ones to constitute our experimental RegEx set. Snort is a famous open-source intrusion detection system, which can be configured to perform protocol analysis, content inspecting over online traffic to detect a variety of worms, attacks and probes. We consider all the RegExes in `backdoor.rules` file of Snort systems. In Both L7-Filter and Snort systems, each RegEx is compiled into one automaton; at run time all automatons are used to match each incoming item sequentially.

In this paper, we compare RegexFilter with SinstrFilter and MulstrFilter. SinstrFilter chooses a single string that is longest as the fingerprint for each RegEx while MulstrFilter uses all the strings as fingerprints. All the three filters can work in two modes. One is item-filter mode, which matches an item with all RegExes of R in verifying stage, if the item passes through filtering stage. The other is pair-filter mode that matches an item with RegEx set $T(R, O(R', i))$ in verifying stage, $T(R, O(R', i))$ contains all the RegExes that match the item successfully as described in section 1.

5.2 Experimental Evaluation

In our evaluation, we use threshold $\log_2(\beta) = 16$ and $-\log_{256}(\eta) = 6$ to generate RegEx print for RegexFilter. As shown in Table 1, RegexFilter can construct a small Regex print DFA with less than three thousand states for each RegEx set. On the contrary, both backdoor set and l7filter set produce a composite DFA with more than one million states. One thing to notice is that it is impossible to extract any strings for 8 RegExes of l7filter set. One example is RegEx `^[a-z][a-z0-9 -_]+`, which is used to classify Finger traffic. Both SinstrFilter and MulstrFilter have to experience the step of enumerating all possible strings represented by these RegExes. In our experiment, SinstrFilter and MulstrFilter do not generate fingerprints for these eight RegExes because of the high cost of enumeration.

Table 2. TCAM size and throughput for RegEx print DFAs

Filters	backdoor set			l7filter set		
	TCAM size	TPS	Throughput	TCAM size	TPS	Throughput
SinstrFilter	0.050 Mb	1.007	7.27 Gbps	0.046 Mb	1.018	7.27 Gbps
MulstrFilter	0.073 Mb	1.003	7.27 Gbps	0.054 Mb	1.015	7.27 Gbps
RegexFilter	0.15 Mb	1.03	5.44 Gbps	0.17 Mb	1.61	5.44 Gbps

In this paper, we estimate the throughput of TCAM-based fingerprint matching using Agrawal and Sherwood's TCAM model, which makes the assumption that each TCAM chip is manufactured with a 0.18 μm process. Table 2 shows the results of TCAM-based fingerprint matching of SinstrFilter, MulstrFilter and RegexFilter on TCAM size and throughput for backdoor and l7filter sets [1]. TCAM size is the TCAM memory required to encode the corresponding DFAs. We calculate its value by multiplying the number of entries by the TCAM width. For all the fingerprint DFAs, we need at most 15 state ID bits, thus TCAM width 36 is enough to store the lookup key. TPS means TCAM entries Per State, which is calculated by dividing the number of TCAM entries required by the number of states. DFA engine takes fixed stride over inputs with one character each transition, the throughput is estimated by the number of TCAM lookups that can be performed in a second for a given number of TCAM entries by 8 bits.

We can draw the following conclusions from Table 2. First, our encoding can reduce TCAM memory required sharply, whose maximum value is less than 0.2 Mb for all the fingerprint DFAs. Second, the TPS value is far less than 256, precisely close to 1, which means that our encoding is possible to encode a DFA with more than one millon states into a 72 Mb TCAM. Consequently, it makes RegexFilter to work on a set of thousands of RegExes. Third, TCAM-based matching can achieve high throughput, the value is over 5 Gbps for all the fingerprint DFAs. Fourth, SinstrFilter and MulstrFilter are superior to RegexFilter on the performance of fingerprint matching. The primary reason is that their fingerprint DFAs have less states and occupies less TCAM entries than that of RegexFilter.

A key criterion to measure filtering effectiveness is verifying rate, which is defined as the percentage of matches performed in verifying stage when with filtering stage among the total number of matches performed when without filtering stage. We make a comparison over a real traffic trace captured in 2010 from a backbone network. The result is shown in Figure 4. The percentages of items (each item is a flow here) that are matched by backdoor set and l7filter set are 10.4% and 91.4%. The malicious ratio on the normal traffic for backdoor set is higher than that in real Snort system, the primary reason may be we skip the packet classification step before RegEx matching in Snort. Although more than

[1] Our implementation first removes transitions redundancy among state using shadow encoding technology in [12], and then encoding each state's remaining labeled transitions with transition logic simplification method instead of character bundling.

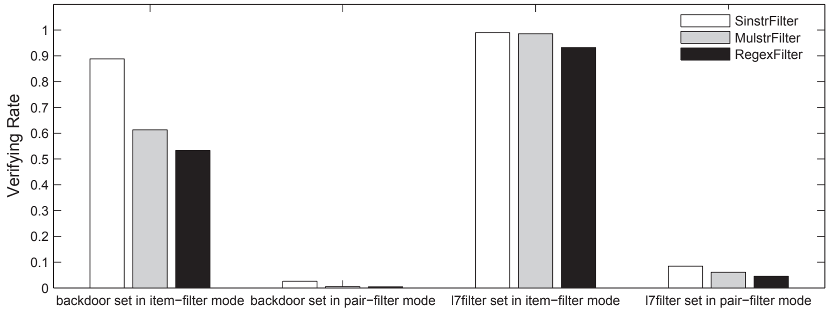


Fig. 4. Comparison among SinstrFilter, MulstrFilter and RegexFilter on verifying rate over a real traffic .

90% items pass through filtering stage for l7filter, verifying rate is still very small in pair-filter mode. The result confirms our assumption that an item is usually matched by limited RegExes. Our experiment presents RegexFilter shows the minimum verifying rate, and recalls all matched pairs of items and RegExes.

After knowing the performance of TCAM-based fingerprint matching and verifying rate, we can estimate the potential throughput by determining the time required to process a byte as the sum of the time required by TCAM-based fingerprint matching and the expected time required by verifying stage to process a byte under the verifying rate. In pair-filter mode, RegexFilter can improve the throughput by 21.5 times for backdoor set and by 20.3 times for l7filter set. Our RegexFilter achieves the potential throughput as 1.2 times and 1.7 times high on backdoor and l7filter set comparing with TCAM-based SinstrFilter, and achieves the potential throughput as 0.79 times and 1.3 times high comparing with TCAM-based MulstrFilter. One thing to notice is that MulstrFilter introduces additional time to validate whether all fingerprints of the same RegEx are matched in sequence, which is not included in the above estimation.

5.3 Effect of Expression Size Threshold

In this section we evaluate the effect of threshold β for RegexFilter. As β is used to bound the number of strings represented by a RegEx print, state size of RegEx print DFA is expected to grow along with the increase of β.

Figure 5 shows the change of RegEx print DFA state size for different β on our experimental RegEx sets. All RegEx prints are generated under threshold $-\log_{256}(\eta) = 6$. From Figure 5 we can find that backdoor set experiences almost the same state size of for different β. Because each RegEx in backdoor set has string fingerprints. As for l7filter set, state size of RegEx print DFA initially decreases rapidly as the increase of β, and then increases after a certain limit. This behavior is because some RegExes do not have any RegEx print for small β, *i.e.* RegEx `^[\x14\x1c\$].{6,15}[\xc6-\xff]` used to classify "network time protocol" traffic in L7-Filter. We add these RegExes into the set of RegEx prints

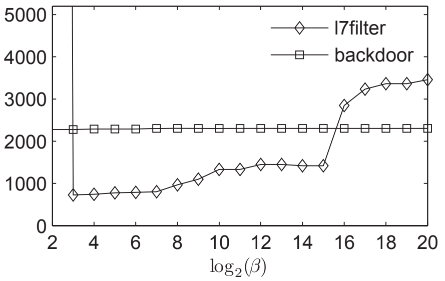


Fig. 5. State size of fingerprint DFAs as a function of $\log_2(\beta)$.

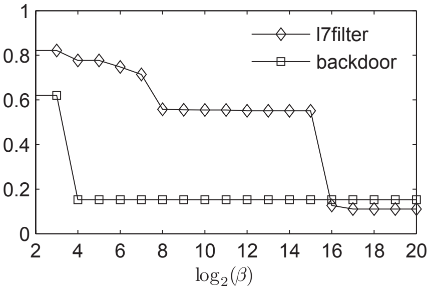


Fig. 6. Verifying rate in item-filter mode as a function of $\log_2(\beta)$.

to ensure that no false-negative matches occur, as a result the fingerprint DFAs experience state explosion for small β. As β increases further, RegexFilter can generate RegEx prints for these RegExes. Therefore RegEx print DFA becomes compact suddenly, and then increases in the number of states stably.

Figure 6 shows that how verifying rate of RegexFilter in item-filter mode varies along with the increase of β. The input trace is a synthetic file generated by regex-tool [20] with $p_m = 0.15$. Owing to the same reason as described in the last paragraph, RegexFilter presents fluctuant curve on verifying rate for l7filter set. In one word, verifying rate decreases as the increase of β in the whole.

5.4 Effect of Matching Probability Threshold

Threshold η, which is used to bound the maximum matching probability of each RegEx print, is the other parameter that can impact the generation of RegEx prints in RegexFilter. In our evaluation, we keep $\log_2(\beta)$ fixed at 16 because backdoor set and l7filter set presents normal behavior on state size and filtering rate for $\log_2(\beta) \geq 14$. It gives a good trade-off when $\log_2(\beta)$ is 16.

Figure 7 and Figure 8 show the effect of threshold η on state size of RegEx print DFA and verifying rate respectively. As can be seen in Figure 7, decreasing η, namely increasing $-\log_{256}(\eta)$, will increase the state size of RegEx print DFAs, as more symbols will be included for higher η. The state size of RegEx print DFA experience sublinear growth for backdoor set, while it grows slowly and flatly for l7filter set. The primary reason is that little RegExes in l7filter107 set have the RegEx print whose $\mathbb{MP}$ value is lower than 256^{-8}, on the contrary some RegExes of backdoor set have long string fingerprints. We expect that the growth rate of RegEx print DFA state size will slow down gradually and become zero after a certain value.

From Figure 8, we find that increasing $-\log_{256}(\eta)$ does not improve verifying rate any more after a certain value, which are 5 and 3 for the two RegEx sets respectively. This indicates that RegExes in real-world systems are distinct enough with $-\log_{256}(\eta) = 6$.

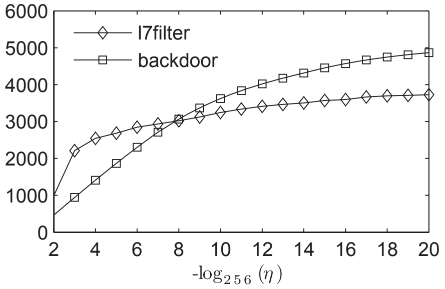


Fig. 7. State size of fingerprint DFAs as a function of $-\log_{256}(\eta)$.

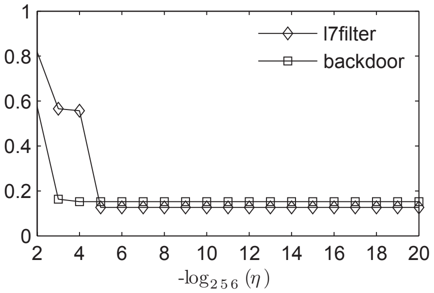


Fig. 8. Verifying rate in item-filter mode as a function of $-\log_{256}(\eta)$.

6 Conclusions

In this paper, we present RegexFilter, a high-speed and memory-efficient technique to improve the throughput of RegEx matching for network and security applications. Our solution leverage the insights that an item is usually matched by limited RegExes, and most items do not match any member in real-word RegEx sets. Thus we try to speedup RegEx matching by quickly finding these RegExes that may match each arriving item as little as possible. First, we develop a novel method that generate RegEx prints to filter a large number of unmatched items with little memory requirement. The method utilizes some new tools to guide the generation of RegEx prints without constructing DFAs. Second, we propose an non-prefix encoding algorithm to minimize the TCAM entries required for TCAM-based RegEx matching. As a result, RegexFilter can perform RegEx print matching quickly.

We evaluate our work on some reasonable metrics and compare it with other two solutions. The preliminary experimental results show that our TCAM-based RegexFilter is suitable to accomplish the filtering task in high-speed for sets of large-scale and complex RegExes. As part of future work, we will beef our work and explore its extension for multi-cores.

7 Acknowledgments

This work was supported by the National High-Tech Research and Development Plan of China under Grant No. 2011AA010703; the National Natural Science Foundation of China under Grant No. 61070026 and No. 61003295.

References

1. Kojm, T.: Clam Anti-virus Signature Database. `http://www.clamav.net`
2. Roesch, Martin: Snort - Lightweight Intrusion Detection for Networks. In: Proc. USENIX LISA. (1999) 229–238

3. Levandoski, J., Sommer, E., Strait, M.: Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/
4. Yu, F., Chen, Z., Diao, Y., Lakshman, T., Katz, R.: Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In: Proc. ACM/IEEE ANCS. (2006) 93–102
5. Kandhan, R., Teletia, N., Patel, J.: SigMatch: Fast and Scalable Multi-Pattern Matching. Proceedings of the VLDB Endowment **3**(1–12) (2010) 1173–1184
6. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In: Proc. ACM SIGCOMM. (2006) 339–350
7. Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G., Di Pietro, A.: An Improved DFA for Fast Regular Expression Matching. ACM SIGCOMM Computer Communication Review **38**(5) (2008) 29–40
8. Liu, T., Yang, Y., Liu, Y., Sun, Y., Guo, L.: An Efficient Regular Expressions Compression Algorithm From A New Perspective. In: Proc. IEEE INFOCOM. (2011) 2129–2137
9. Becchi, M., Crowley, P.: A Hybrid Finite Automaton for Practical Deep Packet Inspection. In: Proc. ACM CoNEXT conference. (2007) 1–12
10. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. ACM SIGCOMM Computer Communication Review **38**(4) (2008) 207–218
11. Kumar, S., Chandrasekaran, B., Turner, J., Varghese, G.: Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia. In: Proc. ACM/IEEE ANCS. (2007) 155–164
12. Meiners, C., Patel, J., Norige, E., Torng, E., Liu, A.: Fast Regular Expression Matching using Small TCAMs for Network Intrusion Detection and Prevention Systems. In: Proc. USENIX Security Symposium. (2010) 8–8
13. Watson, B.: A New Regular Grammar Pattern Matching Algorithm. Algorithms-ESA'96 (1996) 364–377
14. Cho, J., Rajagopalan, S.: A Fast Regular Expression Indexing Engine. In: Data Engineering, 2002. Proceedings. 18th International Conference on. (2002) 419–430
15. Yang, C., CHENG, C., WANG, S.: Two-phase Pattern Matching for Regular Expressions in Intrusion Detection Systems. Journal of information science and engineering **26** (2010) 1563–1582
16. Ramaswamy, R., Kencl, L., Iannaccone, G.: Approximate Fingerprinting to Accelerate Pattern Matching. In: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. (2006) 301–306
17. Broder, A., Mitzenmacher, M., Mitzenmacher, A.: Network Applications of Bloom Filters: A Survey. In: Internet Mathematics, Citeseer (2002)
18. Ficara, D., Antichi, G., Di Pietro, A., Giordano, S., Procissi, G., Vitucci, F.: Sampling Techniques to Accelerate Pattern Matching in Network Intrusion Detection Systems. In: Proc. IEEE ICC, IEEE (2010) 1–5
19. Tang, Y., Jiang, J., Hu, C., Liu, B.: Managing DFA History with Queue for Deflation DFA. Journal of Network and Systems Management (2011) 1–26
20. Becchi, M.: Regular Expression Processor. http://regex.wustl.edu/