Towards Fast and Optimal Grouping of Regular Expressions via DFA Size Estimation

Tingwen Liu, Alex X. Liu, Jinqiao Shi, Yong Sun, and Li Guo

***Abstract*—Regular Expression (RegEx) matching, as a core operation in many network and security applications, is typically performed on Deterministic Finite Automata (DFA) to process packets at wire speed; however, DFA size is often exponential in the number of RegExes. RegEx grouping is the practical way to address DFA state explosion. Prior RegEx grouping algorithms are extremely slow and memory intensive. In this paper, we first propose DFAestimator, an algorithm that can quickly estimate DFA size for a given RegEx set without building the actual DFA. Second, we propose RegexGrouper, a RegEx grouping algorithm based on DFA size estimation. In terms of speed and memory consumption, our work is orders of magnitude more efficient than prior art because DFA size estimation is much faster and memory efficient than DFA construction. In terms of the resulting size sum of DFAs, our work is significantly more effective than prior art because we use a much finer grained quantification of the degree of interaction between two RegExes. For example, to divide the RegEx set of the L7-filter system into 7 groups, prior art uses 279.3 minutes and the resulting 7 DFAs have a total of 29047 states, whereas RegexGrouper uses 3.2 minutes and the resulting 7 DFAs have a total of 15578 states.**

***Index Terms*—Deep packet inspection, regular expression matching, intrusion prevention systems.**

I. Introduction

A. Background and Motivation

DEEP PACKET INSPECTION (DPI) is the core operation of many network and security applications such as network intrusion detection and prevention systems (NIPS), traffic classification and monitoring systems, and data leakage prevention systems. In the past, DPI was often accomplished by string matching, i.e., finding which strings in a set of predefined strings match the payload of a packet. Nowadays, DPI is typically accomplished by Regular Expression (RegEx) matching, i.e., finding which RegExes in a set of predefined RegExes match the payload of a packet, because RegExes are fundamentally more expressive and flexible than strings for specifying attack or malware signatures. Most open source and commercial NIPSes, such as Snort [1], Bro [2], and HP TippingPoint, use RegEx matching to implement DPI. Modern operating systems (such as Cisco IOS and Linux) have built-in RegEx matching modules as well.

To process packets at wire speed, RegEx matching is typically performed on Deterministic Finite Automata (DFA), where each input character only requires one table lookup. However, the DFA constructed from a given RegEx set may require a huge amount of memory because of the well known state explosion problem—the number of DFA states can be exponential in the size of RegExes. When RegExes contain ".$\star$"s, or other Kleene closure of the some characters, the NFA states that correspond to each RegEx can be replicated an exponential number of times. For example, in the DFA in Fig. 1 built from RegEx set $\{\texttt{abc}, \texttt{d.}\star\texttt{e}, \texttt{fgh}, \texttt{i.}\star\texttt{j}\}$, the NFA states that correspond to RegEx `abc` are replicated 2^2 times where the power of 2 came from the two ".$\star$" RegExes $\texttt{d.}\star\texttt{e}, \texttt{i.}\star\texttt{j}$. DFA state explosion is a key issue in RegEx matching for two main reasons. First, for packet processing algorithms in general, not just RegEx matching, more memory often means slower speed because with less memory, an algorithm can run in SRAM, but with large memory, an algorithm has to run in DRAM, and SRAM is 1–2 orders of magnitude faster than DRAM [3], [4]. Second, given a set of many RegExes with ".$\star$"s and/or counters, the DFA may be too large to built.

An effective approach to address DFA state explosion in practice is to partition the given RegEx set into subsets and then build one DFA from each subset. The gain of this approach is great memory reduction. For example, in the extreme end, if we build one DFA from each individual RegEx, then the multiplicative effect among RegExes completely turns into additive. The cost of this approach is that for each input character, we need to perform one lookup per DFA. Although several variations of the DFA model, such as HFA [5], [6], XFA [7], [8], and Overlay Automata [9], have been proposed to alleviate the DFA state explosion issue, they are complementary to the RegEx grouping approach. These automata models need RegEx grouping when the resulting automata is too big. Regex grouping needs these advanced automata models because they help to reduce the sizes of individual automata constructed from RegEx subsets.

Manuscript received December 31, 2013; revised May 16, 2014; accepted June 3, 2014. Date of publication September 18, 2014; date of current version November 26, 2014. This work was supported in part by the National High-Tech Research and Development Program of China under Grants 2011AA010703 and 2011AA010705; the National Natural Science Foundation of China under Grants 61303260, 61472184, 61321491, and 61272546.; the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant XDA06030200. This work was also supported in part by the National Science Foundation under Grant Numbers CNS-1017588, CNS-1017598, CCF-1347953, and CNS-1318563. *(Corresponding authors: Alex X. Liu and Jinqiao Shi.)*

T. Liu, J. Shi, Y. Sun and L. Guo are with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing-100093, China (e-mail: liutingwen@iie.ac.cn; shijinqiao@iie.ac.cn; sunyong@iie.ac.cn; guoli@iie.ac.cn).

A. X. Liu is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA, and also with National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: alexliu@cse.msu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2014.2358839

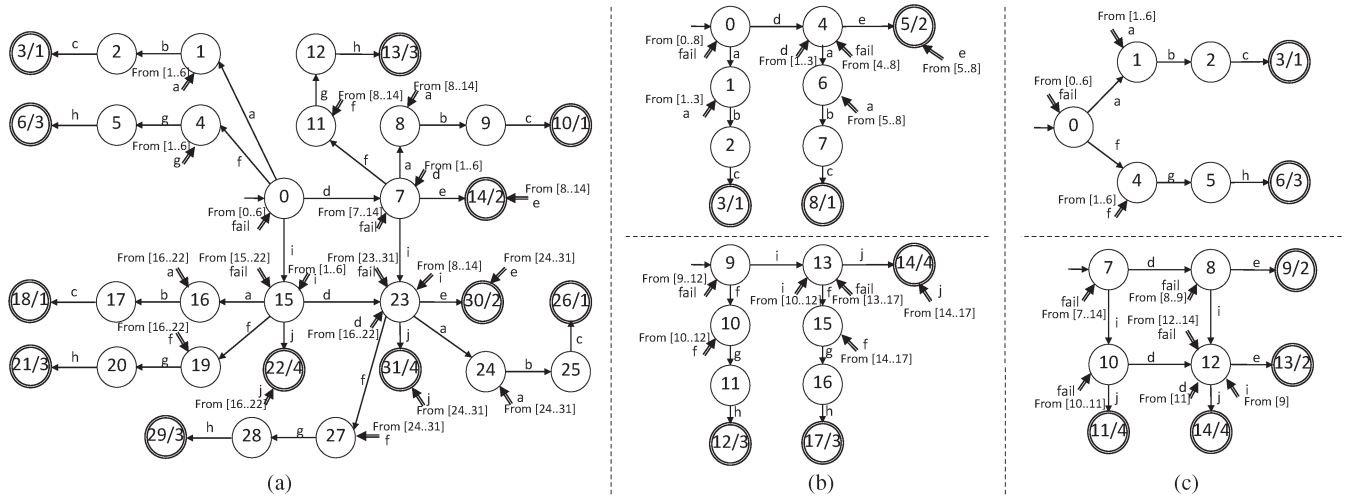

Fig. 1. Example of state explosion and effectiveness of RegEx grouping. (a) DFA for $\{\texttt{abc}, \texttt{d.}, \star\texttt{e}, \texttt{fgh}, \texttt{i.}\star\texttt{j}\}$. (b) DFA for $\{\texttt{abc}, \texttt{d.}, \star\texttt{e}\}$ and DFA for $\{\texttt{fgh}, \texttt{i.}, \star\texttt{j}\}$. (c) DFA for $\{\texttt{abc}, \texttt{fgh}\}$ and DFA for $\{\texttt{d.}\star\texttt{e}, \texttt{i.}\star\texttt{j}\}$.

The key technical chanllenge in RegEx grouping is that there are a huge number of ways of partioning of a RegEx set and different partitioning results in different number of total DFA states. Given a set of n Regeexes and the nubmer of groups k, there are a total of $\frac{1}{k!}\sum_{j=1}^{k}(-1)^{k-j}\frac{k!}{(k-j)!j!}j^n$ ways of partitioning. Considering the two example groupings shown in Fig. 1, one grouping (Fig. 1(b)) results in a total of 18 states and another (Fig. 1(c)) results in 15 states, noting that the DFA without RegEx grouping (Fig. 1(a)) has a total of 32 states.

B. Limitations of Prior Art

Although RegEx grouping is a fundamental and practical strategy in RegEx matching, not much work has been done. Yu *et al.* proposed a RegEx grouping algorithm based on pairwise interaction of RegExes [10]. Two RegExes *interact* if and only if the number of states in the DFA constructed from the two RegExes is larger than the sum of the number of states in the two individual DFAs constructed from the two RegExes respectively. Given a RegEx set and a threshold, Yu's algorithm works as follows. First, they select a RegEx that *interacts* with the least number of other RegExes in the set, and put this RegEx in a new group. Then, they keep choosing RegExes from the original set to put into this new group, each time choosing the RegEx that interacts with the least number of RegExes in the new group. Once a new RegEx is inserted into the new group, they construct a DFA from the new group and check whether the number of states in the DFA exceeds a threshold. If exceeds, they stop inserting new RegExes into the group, start to spawn a new group, and repeat the same above process for the new group. The algorithm terminates until all RegExes have been placed into groups. Yu's algorithm has two fundamental limitations. First, the grouping process is very slow because it needs to construct many DFAs. Slow grouping causes slow updating because when the given Regex set changes, we may need to perform the grouping again. Second, as the interaction relationship between two RegExes is quantified as 0 or 1, the degree of interaction between two RegExes is not captured in Yu's algorithm. However, the interaction relationship can be quantified in a much finer way and thus allow better grouping of RegExes.

C. Proposed Approach

In this paper, we first propose DFAestimator, an algorithm that can quickly estimate DFA size (i.e., the number of states in a DFA) for a given RegEx set without building the actual DFA. To the best of our knowledge, this is the first DFA size estimation algorithm. The key insight is that given a RegEx set, the size of its DFA depends on its NFA size (i.e., the number of states in the NFA) and the complexity of the interaction among NFA states, and therefore can be estimated from the NFA by exploring the interaction among its states. In this paper, we categorize the relationship between two NFA states into four categories: *exclusive, equivalent, inclusive, and independent*. Given an NFA, we first calculate the relationship between any two states and then estimate the DFA size based on the caculated relationship. This light-weight DFA size estimation is many orders of magnitude faster and more memory efficient than DFA construction. Second, we propose RegexGrouper, a RegEx grouping algorithm based on DFA size estimation. The key insight is that our fast DFA size estimation algorithm allows us to extensively, yet quickly, explore the complex interaction among RegExes. Given a set with n RegExes, we construct a weighted complete graph where each node represent a RegEx and the weight of each edge is the "overhead" of combining the two RegExes into one DFA. This overhead, which we call *combining cost*, of combining two RegExes into one DFA, is the estimated difference between the size in the DFA constructed from two RegExes and the sum of the sizes of the two individual DFAs constructed from the two RegExes respectively, without even estimating the size of these DFAs. Then, we partition this graph into k subgraphs with the goal of maximizing the weight sum of the edges being cut. This is the maximum k-cut problem in graph theory, which is NP-complete. In this paper, we use a heuristic algorithm based on local search to solve this graph partitioning problem in polynomial time.

Our RegexGrouper algorithm fundamentally addressed the two limitations of Yu's algorithm. First, as our DFA size

estimation is way faster and more memory efficient than DFA construction, our algorithm is orders of magnitude faster and more memory efficient than Yu's algorithm. Second, as our RegexGrouper algorithm quantifies the degree of interaction between two RegExes as the "overhead" of combining the two RegExes into one DFA, instead of a binary value, our algorithm achieves significant better grouping results than Yu's algorithm in terms of the size sum of resulting DFAs. For example, to divide the RegEx set of the L7-filter system [11] in Linux into 7 groups, prior art uses 279.3 minutes and the resulting 7 DFAs have a total of 29047 states, whereas RegexGrouper uses 3.2 minutes and the the resulting 7 DFAs have a total of 15578 states.

II. Related Work

Several DFA-like automata models have been proposed to address the DFA state explosion problem. These automata models are complementary and orthogonal to RegEx grouping because they help to reduce the automaton size of each group and grouping helps to bring down the automaton size. (1) History-based Finite Automata (HFA) [5], [6]: An HFA is a DFA where each transition is augmented with a program, which is executed each time the transition is followed; more precisely, an HFA is a DFA with an auxiliary vector of bits (called history bits) where each transition is augmented with a Boolean condition specified in history bits and some actions of setting/clearing history bits, allowing us to query and update history bits, respectively. (2) Extended Finite Automata (XFA) [7], [8]: An XFA is a DFA where each state is augmented with a program, which is executed each time the state is reached. Both HFA and XFA use augumented programs to avoid the exponential replication of NFA states caused by ".$\star$"s. In HFA, the programs are only allowed to test, set, or clear hisotry bits whereas in XFA, the programs can be arbitrary code, which in essence is a Turing machine. Becuase of the difference in the expessive power of augumented programs, HFA can be automatically constructed but XFA cannot. (3) Delayed-input DFA ($\mathrm{D^2FA}$) [12]: A $\mathrm{D^2FA}$ is a DFA where some states have incomplete transition tables and for a character that the transition table of a state have no corresponding entry, empty transitons are followed to reach a state that may have an entry for that character. $\mathrm{D^2FA}$ significantly saves the memory for storing transition tables at the cost of possible multiple transition table lookups per input character. Delta FA (δFA and δ^NFA) [13], [14] are variations of $\mathrm{D^2FA}$. Several follow-up models such as $\mathrm{CD^2FA}$ [15] and A-DFA [16] have been proposed to address the shortcomings of $\mathrm{D^2FA}$. For example, in A-DFA, the default transition is limited to transfer from an high-depth state from a low-depth state. (4) Overlay DFA (ODFA) and Overlay $\mathrm{D^2FA}(\mathrm{OD^2FA})$ [9]: OverlayDFA addresses the closure-based replication of automata components in DFAs. The basic idea is to overlay all the DFA states that are replicas of the same NFA state vertically together into what is called a *super-state*. If we view a DFA as a 2-D object, then an ODFA can be viewed as a 3-D object. The ODFA model gives us two key benefits: first, we can compactly reference all replicas of the same NFA state using super-states; second, we can compactly represent replicas of the same NFA transition by one super-state transition. The Overlay$\mathrm{D^2FA}$ model combines the benefits of both OverlayDFA and $\mathrm{D^2FA}$. (5) Hybrid DFA-NFA Finite Automaton (Hybrid-FA) [17]: A Hybrid-FA is a mix of DFA and NFA where the "problematic" part of a DFA is replaced by an NFA. This model combines the benefits of both DFA and NFA as the nodes that contribute to state explosion retain as NFA. (6) Tunable Finite Automata (TFA) [18], [19]: A TFA is a general infinite automaton model in which DFA and NFA are two special cases. TFA achieves significant in automata size as it enforces a tunable bound on the number of concurrent active states. (7) Lookahead Finite Automata (LaFA) [20], [21]: A LaFA is a DFA where certain RegEx components, such as strings, are identified and used to provide new possibilities for memory optimization. Thus, it can perform scalable RegEx matching using a small amount of memory.

One approach to speed up RegEx matching is RegEx prefiltering [22], [23], which is complementary and orthogonal to the above automata models and RegEx grouping. Given a RegEx set S, the basic idea of RegEx prefiltering is to construct another RegEx set S' so that the following two conditions hold: (1) S' is faster and more memory efficient to match traffic against than S, and (2) if input traffic does not match any RegEx in S', then it for sure does not match any RegEx in S. Thus, input traffic is first matched again S'; only when a RegEx in S' is matched, the input traffic is then further matched against S. The benefit of this approach is that maching traffic against S' can be efficiently done. The cost of this approach is that it requires the buffering of input traffic and when matches happen on S', the buffered traffic needs to be matched against S starting from the begining. For traffic that rarely matches the given RegEx set, the benefit overweight the cost for this approach in general.

Another approach to speed up RegEx matching is to implement RegEx matching on special hardware platforms such as TCAM [9], [24]–[27], FPGA [28], [29], and GPU [30]. This approach is complementary and orthogonal to RegEx grouping. Recently a fast and scalable automata construction algorithm is proposed [31], [32]. This algorithm is complementary and orthogonal to the RegEx grouping approach as it can be used to build DFAs from RegEx groups.

III. DFA Size Estimation

In this section, we present our DFA size estimation algorithm. First, we introduce four types of relationship between two NFA states. Second, we build a theoretical foundation for calculating the upper and lower bounds of DFA sizes based on NFA and the relationship among NFA states. Third, we present algorithms to efficiently compute the relationship among NFA states. Third, we present our DFA size estimation algorithm. Table I lists the symbols used throughout this paper.

A. Convolvement Relation

An NFA is a 5-tuple $(\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{A})$, where $\mathcal{Q}$ is a set of states, Σ is an alphabet, $\delta : \mathcal{Q} \times \Sigma \rightarrow 2^{\mathcal{Q}}$ is the transition function, q_0 is the starting state, and $\mathcal{A} \subseteq \mathcal{Q}$ is a set of accepting states. For any state $q \in \mathcal{Q}$, its *language* denoted as $L(q)$, is defined as the set of strings that can traverse from state q_0 to q in the NFA. For example, in the NFA in Fig. 2, for state 1, its language $L(1)$ is the set of all strings matching $.*$; and for

TABLE I
SYMBOLS USED IN THIS PAPER

Symbols	Description
$\mathcal{N}$	an NFA
$\mathbb{D}$	a DFA
$\mathcal{Q}$	state set of an NFA
$\mathbb{Q}$	state set of a DFA
Σ	alphabet
δ	transition relation
q_0	starting state in NFA/DFA
$\mathcal{A}$	set of accepting states in NFA
$\mathbb{A}$	set of accepting states in DFA
$L(q)$	language of state q
$p \parallel q$	state p and state q are exclusive
$p \equiv q$	state p and state q are equivalent
$p \triangleleft q$	state p *includes* state q
$q \triangleright p$	state p *includes* state q
$p \lozenge q$	states p and q are independent
α	a character in Σ
G	independent graph
E	edge set of independent graph G
$p \bowtie q$	convolvement relation between states p and q

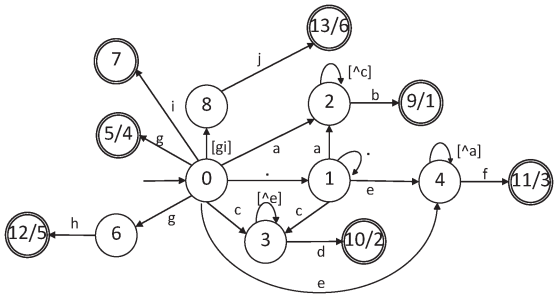

Fig. 2. NFA for $\{\mathtt{a}[\hat{}\,\mathtt{c}]\star\mathtt{b}, \mathtt{c}[\hat{}\,e]\star\mathtt{d}, \mathtt{e}[\hat{}\,a]\star\mathtt{f}, \hat{}\,\mathtt{g}, \hat{}\,\mathtt{gh}, \hat{}\,\mathtt{i}, \hat{}\,[\mathtt{gi}]\mathtt{j}\}$.

state 2, its language $L(2)$ is the set of all strings that match $.\star\mathtt{a}[\hat{}\,\mathtt{c}]\star$.

We extend the convolvement relationship between two NFA states defined by Yang *et al.* [33] to the following four types: *exclusive, equivalent, inclusive, and independent*.

1) Exclusive: Two NFA states p and q are *exclusive*, denoted as $p \| q$, if and only if $L(p) \cap L(q) = \emptyset$, which means that states p and q can never be active at the same time for any input string. For example, in the NFA in Fig. 2, states 5 and 7 are exclusive as $L(5)$ contains only the string g and $L(7)$ contains only the string i, and they can never be both active.
2) Equivalent: Two NFA states p and q are *equivalent*, denoted as $p \equiv q$, if and only if $L(p) = L(q) \neq \emptyset$, which means that states p and q are always active at the same time for any input string. For example, in the NFA in Fig. 2, states 5 and 6 are equivalent as both $L(5)$ and $L(6)$ contain only the string g, and they are always both active.
3) Inclusive: Two NFA states *pincludesq*, denoted as $p \triangleleft q$ or $q \triangleright p$, if and only if $L(p) \supset L(q) \neq \emptyset$, which means that whenever q is active, p is active, for any input string. For example, in the NFA in Fig. 2, state 1 includes state 2 as $L(1)$ is the set of all strings matching $.\star$, $L(2)$ is the set of all strings that match $.\star\mathtt{a}[\hat{}\,\mathtt{c}]\star$, and any string that matches $.\star\mathtt{a}[\hat{}\,\mathtt{c}]\star$ also matches $.\star$. In other words, whenever 2 is active, 1 is active.
4) Independent: Two NFA states p and q are *independent*, denoted as $p \lozenge q$, if and only if $L(p) \cap L(q) \neq \emptyset$, $L(p) \not\subseteq L(q)$, and $L(q) \not\subseteq L(p)$, which means that p and q are none of the above relations. For example, in the NFA in Fig. 2, states 2 and 3 are independent as there are strings such as a that can reach state 2 but not 3, strings such as c that can reach state 3 but not 2, and strings such as ca that can reach both states 2 and 3.

B. Bounding DFA Size

Next, we present our theoretical results on bounding the size of the DFA constructed from an NFA. These results are the foundation of our DFA size estimation algorithm.

To understand the relationship between an NFA and the size of its DFA, let us first understand the relationship between NFA states and DFA states. The standard algorithm for constructing a DFA from NFA is subset construction. Given an NFA $(\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{A})$, we first create a starting DFA state that is labeled with set $\{q_0\}$ and add this DFA state $\{q_0\}$ to $\mathbb{Q}$, which was initialized to $\emptyset$; second, for any character $c \in \Sigma$ and any DFA state represented as a set of NFA states $\{q_1, \ldots, q_k\}$ in $\mathbb{Q}$, if the NFA state set resulted from transitioning from each NFA state in this set, denoted $\{\delta(q_1, \alpha) \cup \cdots \cup \delta(q_k, \alpha)\}$ is not in $\mathbb{Q}$, then add this new NFA state set to $\mathbb{Q}$. For an NFA state q and a character α, $\delta(q, \alpha)$ denotes the set of NFA states that can reach from state q on character α. We repeat the above second step until no new NFA state set can be added to $\mathbb{Q}$. For simplicity, in this paper, we refer to a DFA state and the set of NFA states labeled for the DFA state interchangeably.

We now define the concept of the *independent representation* of a DFA state as a set of NFA states. Consider an NFA and its DFA with state set $\mathbb{Q}$ constructed as above. First, for any two NFA states p and q that are equivalent (which means that p and q always show up together in DFA states) or p includes q (which means that for any DFA state containing q, it also contains p), if we replace p and q together by q (i.e., delete p) in every DFA state that contains both p and q, then the resulting DFA state set $\mathbb{Q}'$ has the same number of states as $\mathbb{Q}$, i.e., $|\mathbb{Q}'| = |\mathbb{Q}|$. Otherwise, there must be two states in $\mathbb{Q}$ where they both contain q and they only differ in that one state contains p and the other does not, which will become the same set after deleting p from that state containing p; however, this contradicts with the fact that p and q are equivalent or p contains q. *For a DFA stateSrepresented as a set of NFA state, for each statep $\in S$, we delete fromSall states that are equivalent topand all states that includesp*; *the resulting NFA state set is called the independent representation ofS*. Note that the independent representation of a DFA state S may be not unique. For example, if S contains only two equivalent NFA states p and q, then both $\{p\}$ and $\{q\}$ are the independent representations of S. We use $U(S)$ to denote one independent representation of S.

For a DFA state S, it is reachable from the starting state for exactly the language of $\bigcap_{p \in S} L(p)$ because only for such a langue, all NFA states in S are activated. The following Lemma 1 shows that a DFA state and its unique representation define the same language.

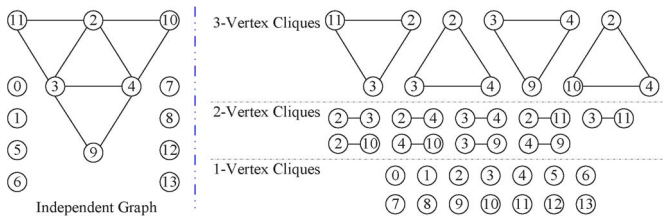

Fig. 3. Independent graph for NFA in Fig. 2, and its all cliques.

Lemma 1: For any DFA state S and its unique representation $U(S)$, we have $\bigcap_{p\in S} L(p) = \bigcap_{p\in U(S)} L(p)$.

Proof: Recall that for any $p \in S$ and $q \in S$, if p and q are equivalent, then $L(p) = L(q)$; if p includes q, then $L(p) \supset L(q)$. This lemma follows directly from this fact and the definition of independent representations. □

The following Lemma 2 shows that if we convert each state in a DFA to its independent representation, the number of DFA states remains the same.

Lemma 2: For any two DFA states S_1 and S_2, there exists no $U(S_1)$ and $U(S_2)$ such that $U(S_1) = U(S_2)$.

Proof: If $U(S_1) = U(S_2)$, then according to Lemma 1, we have $\bigcap_{p\in S_1} L(p) = \bigcap_{p\in U(S_1)} L(p) = \bigcap_{p\in U(S_2)} L(p) = \bigcap_{p\in S_2} L(p)$, which means that the two DFA states S_1 and S_2 are activated for the same input. This controdicts with the fact that at any time, there is only one DFA is activated. □

The following Theorem 1 shows that among the four types of NFA state relationship, namely exclusive, equivalent, inclusive, and independent, it is the independent relationship that causes DFA size explosion. It also shows that when the NFA states have no independent relations, the upper bound for the DFA size is simply the NFA size.

Theorem 1: For an NFA with state set $\mathcal{Q}$ and its DFA with state set $\mathbb{Q}$, if no two NFA states in $\mathcal{Q}$ are independent, then $|\mathbb{Q}| \leq |\mathcal{Q}|$.

Proof: No two NFA states in $\mathcal{Q}$ being independent means that for any two NFA states in $\mathcal{Q}$, they are exclusive, equivalent, or inclusive. If two NFA states are exclusive, then they cannot both show up in any DFA state. Thus, here, for any DFA state, according to the definition of independent representation of DFA states, its independent representation contains only one NFA state. Thus, according to Lemma 2 and the pigeon hole theory, we have $|\mathbb{Q}| \leq |\mathcal{Q}|$. □

Next, we develop an upper bound for the DFA size when the NFA states have independent relations. First, we need to introduce the concept of independent graph. Given an NFA with state set $\mathcal{Q}$, we construct an undirected graph $G = (V, E)$ where each vertex in V corresponds to a unique state in $\mathcal{Q}$ and for any two states $p, q \in \mathcal{Q}$, p and q are independent if and only if their corresponding vertices in V are connected by an edge. We call this graph G the *independent graph* of the NFA. Fig. 3 shows the independent graph for the NFA in Fig. 2. This independent graph has four 3-vertex cliques, nine 2-vertex cliques, and fourteen 1-vertex cliques. A clique is a set of vertices where there is an edge between any two vertices in the set. A *maximum clique* of a graph is a clique with the largest size.

The following Theorem 2 shows that the upper bound for the DFA size can be obtained based on the number of cliques in the independent graph of the NFA.

Lemma 3: For any DFA state S as a set of NFA states and its independent representation $U(S)$, for any two NFA states $p, q \in U(S)$, p and q are independent.

Proof: This lemma follows directly from the definition of independent representation. □

Theorem 2: Given an NFA with state set $\mathcal{Q}$ and its independent graph $G = (V, E)$ where the size of the maximum clique is m, denoting the number of k-vertex cliques by c_k, for the DFA with state set $\mathbb{Q}$, we have $|\mathbb{Q}| \leq |\mathcal{Q}| + |E| + \sum_{k=3}^{m} c_k$.

Proof: As $|\mathcal{Q}| = c_1$ and $|E| = c_2$, $|\mathcal{Q}| + |E| + \sum_{k=3}^{m} c_k$ denotes the total number of all cliques in the independent graph. According to Lemmas 2 and 3, the maximum number of DFA states is the total number of the different combinations of independent states, which equals to the total number of all cliques in the independent graph. Note that any non-empty subset of a clique is also a clique. □

Often $|\mathbb{Q}| < |\mathcal{Q}| + |E| + \sum_{k=3}^{m} c_k$ holds because of the following two reasons. First, a DFA state may have multiple independent representations due to the equivalent relation between two NFA states. Second, a clique may be not the independent representation of any DFA state.

The following Lemma 4 and Theorem 3 gives the lower bound for DFA size. Here, for any NFA state, p we use $I(p)$ to denote the set of NFA states that are included by p or are independent with p.

Lemma 4: For any DFA state S as a set of NFA states, for any NFA state $p \in \mathcal{Q}$, if $L(p) - \bigcup_{q\in I(p)} L(q) \neq \emptyset$, then there exists one DFA state where $\{p\}$ is its independent representation.

Proof: If $L(p) - \bigcup_{q\in I(p)} L(q) \neq \emptyset$, then there exists an input string that activates state p but not any state that p includes or p is independent with. Thus, there exists one DFA state where $\{p\}$ is its independent representation. □

Theorem 3: For an NFA with state set $\mathcal{Q}$ with no two equivalent states and its DFA with state set $\mathbb{Q}$, for any NFA state $p \in \mathcal{Q}$, if $L(p) - \bigcup_{q\in I(p)} L(q) \neq \emptyset$, then $|\mathbb{Q}| \geq |\mathcal{Q}|$.

Proof: According to Lemma 4 and the fact that $\mathcal{Q}$ with no two equivalent states, for any NFA state $p \in \mathcal{Q}$, there is a unique DFA state where $\{p\}$ is its independent representation. Thus, we have $|\mathbb{Q}| \geq |\mathcal{Q}|$. □

C. DFA Size Estimation

Next, we present our algorithm for estimating the size of a DFA based on its NFA and the convolvement relationship among NFA states. According to Theorems 2 and 3, in general, DFA size is between $|\mathcal{Q}|$ and $|\mathcal{Q}| + |E| + \sum_{k=3}^{m} c_k$. These two bounds are not accurate enough to directly estimate DFA size because the lower bound is too small and the upper bound is too big. Furthermore, the upper bound is too time consuming to calculate. It is NP-hard to find all cliques from an independent graph.

In this paper, we propose to simply estimate the DFA size using $|\mathcal{Q}| + |E|$ because of three reasons. First, $|\mathcal{Q}| + |E|$ lies between the lower bound and the upper bound. Second, intuitively, an NFA with more states ($|\mathcal{Q}|$) and more independent pairs ($|E|$) tends to result in a larger DFA. Third, $|\mathcal{Q}| + |E|$ can be quickly calculated. Fig. 4 shows the lower bound ($|\mathcal{Q}|$), the accurate DFA size (i.e., $|\mathbb{Q}|$), our estimated DFA size (i.e.,

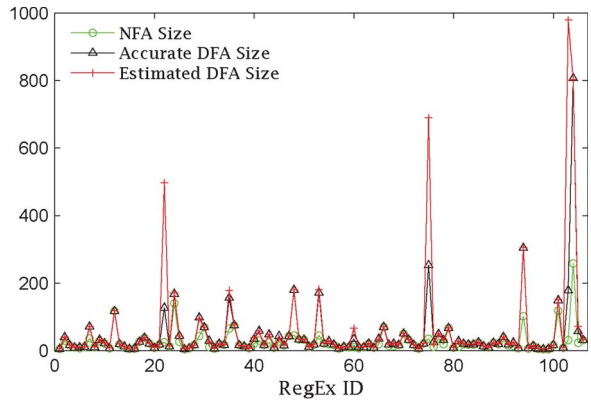

Fig. 4. NFA size, accurate DFA size, and estimated DFA size.

$|\mathcal{Q}| + |E|$), for each RegEx in the RegEx set of L7-filter system [11]. The upper bound is not caculated as the problem is NP-hard. We observe that the accurate DFA size is always bigger than its lower bound. This shows that Theorem 3 often holds in practice. We also observe that our estimated DFA size is very close to its accurate size. We admit that our algorithm may give not arbiturily accurate DFA sizes. However, we argue that it is practically good enough for RegEx grouping as we will show our experimental results.

D. Calculating Convolvement Relationship

Our DFA size estimation algorithm requires to calculate the number of edges in the independent graph, which further requires to calculate the convolvement relation between any two NFA states. The straightforward way to compute convolvement relationship based on definitions. However, this requires us to build the DFA first. Y.H.E. Yang *et al.* proposed to compute the convolvement relationship for two given NFA states p and q by comparing the suffix sets of their languages [33]. Initially, the current suffix length is set to one. Second, they find all possible suffixes of the current suffix length in the language, and use them to construct the corresponding suffix set; then, the current suffix length is increased by one. The above second step repeats until a common suffix that can simutaneously activate both p and q from the starting state. The key limitation of this method is too time-consuming as too many suffix sets need to be created repeatedly to obtain the convolvement relationship between any two states. In our experiments, sometimes it runs even slower than building the DFA directly. Note that $L(p)$ and $L(q)$ may be both infinite and their intersection is empty at the same time. In such case, this algorithm will not terminate as there is no common suffix that can simutaneously activate both p and q.

In this paper, we propose an algorithm to quickly calculate the convolvement relationship for any two NFA states.

According to the definition of convolvement relationship, the convolvement relation between two NFA states p and q, denoted by $x \bowtie y$, depends on which of the following conditions holds: (1) $L(p) \cap L(q) \neq \emptyset$, (2) $L(p) \not\subseteq L(q)$, and (3) $L(q) \not\subseteq L(p)$. Thus, $p \bowtie q$ can be represented by three bits: the first bit is used to indicate whether $L(p) \cap L(q) \neq \emptyset$ is true, the second bit is to indicate whether $L(p) \not\subseteq L(q)$ is true, and the last bit is to indicate whether $L(q) \not\subseteq L(p)$ is true. When a predicate is true or false, we set the corresponding bit as 1 or 0. As a result, $p \bowtie q$ becomes a three-bit value, as shown in Fig. 5. For example, $p \bowtie q = 7$ implies that p and q are independent because all the three predicates are true. Thus, we convert the problem of computing the convolvement relationship between two NFA states to the problem of computing the values of these predicates.

$L(p) \cap L(q) \neq \emptyset$	$L(p) \not\subseteq L(q)$	$L(p) \not\subseteq L(q)$	$p \bowtie q$
0	0	0	‖
0	0	1	‖
0	1	0	‖
0	1	1	‖
1	0	0	≡
1	0	1	▷
1	1	0	◁
1	1	1	⋈

Fig. 5. Mapping between $p \bowtie q$ and three predicates: $L(p) \cap L(q) \neq \emptyset$, $L(p) \not\subseteq L(q)$, and $L(q) \not\subseteq L(p)$.

Our convolvement relationship computing algorithm is based on Theorem 4. For an NFA state p, let $T(\alpha, p)$ denote the set of all NFA states who transit to state p on character α. For any state r, let $L(r)|\alpha$ denote the resulting language by appending character α to each string in language $L(r)$. Thus, we have $L(p) = \bigcup_{r \in T(\alpha,p)} (L(r)|\alpha)$.

Theorem 4: For any two NFA states p and q,

1) if there exists $\alpha \in \Sigma$ so that $\bigcup_{r \in T(\alpha,p)} (L(r)|\alpha) \cap \bigcup_{r \in T(\alpha,q)} (L(r)|\alpha) \neq \emptyset$, then $L(p) \cap L(q) \neq \emptyset$;
2) if there exists $\alpha \in \Sigma$ so that $\bigcup_{r \in T(\alpha,p)} (L(r)|\alpha) \not\subseteq \bigcup_{r \in T(\alpha,q)} (L(r)|\alpha) \neq \emptyset$, then $L(p) \not\subseteq L(q)$;
3) if there exists $\alpha \in \Sigma$ so that $\bigcup_{r \in T(\alpha,q)} (L(r)|\alpha) \not\subseteq \bigcup_{r \in T(\alpha,p)} (L(r)|\alpha) \neq \emptyset$, then $L(q) \not\subseteq L(p)$.

Proof: This theorem directly follows from the above discussion. □

Based on Theorem 4, to compute the convolvement relationship between two NFA states p and q, we can first compute the convolvement relationship between each pair of two states where one is from $T(\alpha, p)$ and the other one is from $T(\alpha, q)$; then based on these convolvement relations, we can compute the convolvement relationship between p and q using set theory. Furthermore, for any of these NFA state pairs $x \in T(\alpha, p)$ and $y \in T(\alpha, q)$, to compute the convolvement relationship between x and y, we first compute the convolvement relationship between each pair of two states where one is from $T(\alpha, x)$ and the other one is from $T(\alpha, y)$; then based on these convolvement relations, we can compute the convolvement relationship between x and y using set theory. As we want to compute the convolvement relationship between any two NFA states, we start the computing process from computing the convolvement relationship between the starting state and each of other state, in a breadth-first fashion.

Note that for faster speed, our convolvement relationship calculation algorithm does not guarantee that the computed result is 100% accurate. However, this tradeoff accuracy for speed is worthwhile for two reasons. First, most of the time the computed results are correct in practice, as we show in our experimental results in Section V. Second, for our DFA size estimation, we do not need to require that each

convolvement relation is calculated correctly. We next explain why our convolvement relationship calculation algorithm does not guarantee that the computed result is 100% accurate. First, the set operations used in our algorithm may have more than one values, although they result in one value most of the time. For example, for three NFA states p, q, s that $p \bowtie q = 6$ and $p \bowtie s = 6$, which means that $L(q)$ and $L(s)$ are two non-empty proper subsets of $L(p)$, thus the value of $\{p\} \bowtie \{q, s\}$ may be 4 when $L(p) = L(q) \cup L(s)$, otherwise the value is 6. Note that the possibility that $L(p) = L(q) \cup L(s)$ is very low for NFAs constructed from real RegExes. Our algorithm ignores these small probability cases. Second, when calculating $p \bowtie q$, for state $x \in T(\alpha, p)$ and state $y \in T(\alpha, q)$, $x \bowtie y$ may be still unknown, even if we calculate the convolvement relationship between any two NFA states in a breadth-first fashion. One example is the existence of loop in an NFA, which happens rarely in practice.

IV. RegEx Grouping Algorithm

In this section, we present our RegEx grouping algorithm without building DFAs.

A. Combining Cost Estimation

We estimate combining cost based on our DFA size estimation algorithm. For a RegEx set $R = \{r_1, \ldots, r_n\}$, $\mathcal{N}(\{r_1, \ldots, r_n\})$ denotes the NFA constructed from $\{r_1, \ldots, r_n\}$; $E(\{r_1, \ldots, r_n\})$ denotes the total number of independent edges in $\mathcal{N}(\{r_1, \ldots, r_n\})$; $C(\{r_1, \ldots, r_n\})$, which is called the *estimated combining cost*, denotes the additional independent edges introduced by combining $\mathcal{N}(r_1), \ldots, \mathcal{N}(r_n)$ together; $|\mathcal{Q}(\{r_1, \ldots, r_n\})|$ denotes the number of states in $\mathcal{N}(\{r_1, \ldots, r_n\})$; and $\tilde{\mathbb{Q}}(\{r_1, \ldots, r_n\})$ to denote the estimated size of the DFA constructed from $\{r_1, \ldots, r_n\}$. Next, Lemma 5 shows how to calculate combining cost, Lemma 6 shows how to calculate the estimated DFA size for a set of RegExes, and finally Theorem 5 shows how to calculate the estimated DFA size for multiple groups of RegExes. Our RegEx grouping algorithm is based on Theorem 5.

Lemma 5: For a RegEx set $R = \{r_1, \ldots, r_n\}$, its combining cost is

$$C(\{r_1, \ldots, r_n\}) = \sum_{1 \le i < j \le n} C(\{r_i, r_j\})$$

Proof: This can be simply proved using induction. □

Lemma 6: For a RegEx set $R = \{r_1, \ldots, r_n\}$, its estimated DFA size is

$$\tilde{\mathbb{Q}}(\{r_1, \ldots, r_n\}) = \sum_{i=1}^{n} |\mathcal{Q}(r_i)| + \sum_{i=1}^{n} E(r_i) + \sum_{1 \le i < j \le n} C(\{r_i, r_j\})$$

Proof: By definition and Lemma 5, we have

$$E(\{r_1, \ldots, r_n\}) = \sum_{i=1}^{n} E(r_i) + C(\{r_1, \ldots, r_n\}) = \sum_{i=1}^{n} E(r_i) + \sum_{1 \le i < j \le n} C(\{r_i, r_j\}) \quad (1)$$

$$|\mathcal{Q}(\{r_1, \ldots, r_n\})| = \sum_{i=1}^{n} |\mathcal{Q}(r_i)| \quad (2)$$

According to our DFA size estimation algorithm, we have

$$\tilde{\mathbb{Q}}(\{r_1, \ldots, r_n\}) = |\mathcal{Q}(\{r_1, \ldots, r_n\})| + E(\{r_1, \ldots, r_n\}) = \sum_{i=1}^{n} |\mathcal{Q}(r_i)| + \sum_{i=1}^{n} E(r_i) + \sum_{1 \le i < j \le n} C(\{r_i, r_j\}) \quad (3)$$

□

Theorem 5: Partitioning RegEx set $\{r_1, \ldots, r_n\}$ into k non-overlapping RegEx groups $R_1, \ldots, R_k$, the total estimated DFA sizes for these k groups is

$$\tilde{\mathbb{Q}}\left(\bigcup_{m=1}^{k} R_m\right) = \sum_{i=1}^{n} |\mathcal{Q}(r_i)| + \sum_{i=1}^{n} E(r_i) + \sum_{m=1}^{k} \sum_{1 \le i < j \le |R_m|} C(\{r_i, r_j\})$$

Proof: This can be simply proved based on the following fact and Lemma 6.

$$\tilde{\mathbb{Q}}\left(\bigcup_{m=1}^{k} R_m\right) = \sum_{m=1}^{k} \tilde{\mathbb{Q}}(R_m) \quad (4)$$

□

Algorithm 1: RegEx Grouping Algorithm

Input: A RegEx set R of n RegExes $r_1, \cdots, r_n$, an integer k, estimated combining cost $C(\{r_i, r_j\})$ for any two RegExes r_i and r_j

Output: k non-overlapping RegEx groups $R_1, \cdots, R_k$ that $\bigcup_{m=1}^{k} R_m = R$

```
foreach i ∈ [1, n] do
    if rand() mod k == m then
        R_m := R_m ∪ {r_i};
while true do
    bool nochange := true;
    foreach i ∈ [1, n] do
        foreach m ∈ [1, k] do
            F[m] := Σ_{j∈R_m, j≠i} C({r_i, r_j});
        t := {x ∈ [1, k] | ∀y ∈ [1, k], x ≠ y, F[x] < F[y]};
        s := {z ∈ [1, k] | v_i ∈ R_s};
        if s != t then
            R_s := R_s − {r_i};
            R_t := R_t ∪ {r_i};
            nochange := false;
    if i > n and nochange == true then
        break;
```

B. RegEx Grouping Algorithm

Given a RegEx set $R = \{r_1, \ldots, r_n\}$, we want to partition it into k RegEx groups $R_1, \ldots, R_k$ so that $\sum_{m=1}^{k} \sum_{1 \le i < j \le |R_m|} C(\{r_i, r_j\})$ is minimized, which means that $\tilde{\mathbb{Q}}(\bigcup_{m=1}^{k} R_m)$ is also minimized according to Theorem 5 because $\sum_{i=1}^{n} |\mathcal{Q}(r_i)| + \sum_{i=1}^{n} E(r_i)$ are fixed for a given RegEx set. We call this problem the optimal RegEx k-grouping problem. Theorem 6 shows that this problem is NP-complete.

We prove by mapping it to the maximum k-cut problem in graph theory as follows.

Theorem 6: The optimal RegEx k-grouping problem is NP-complete.

Proof: We first prove that an instance of this problem can be mapped to an instance of the maximum k-cut problem, which is NP-complete. Given a RegEx set $\{r_1, \ldots, r_n\}$, we construct an weighted undirected complete graph $G = (V, E)$, where each vertex v_i represents a RegEx r_i, every pair of vertices v_i and v_j has an edge and the weight $w(v_i, v_j)$ of this edge is the combining cost $C(\{r_i, r_j\})$. Note that $C(\{r_i, r_j\}) = E(\{r_i, r_j\}) - E(r_i) - E(r_j)$. Thus, the problem of partitioning n RegExes into k groups $R_1, \ldots, R_k$ with the optimization goal of minimizing $\sum_{m=1}^{k} \sum_{1 \le i < j \le |R_m|} C(\{r_i, r_j\})$ becomes the problem of partitioning V into k groups $\{V_1, \ldots, V_k\}$ so that the total weights of the edges between groups is maximized, namely $\text{Max} \sum_{1 \le i < j \le k} \sum_{u \in v_i, v \in v_j} w(u, v)$. Next, we prove that an instance of the maximum k-cut problem can be mapped to an instance of the optimal RegEx k-grouping problem. Given a graph weighted undirected graph $G = (V, E)$, we map each vertex v_i to a RegEx r_i, and map the weight of an edge between v_i and v_j to the combining cost $C(\{r_i, r_j\})$; if there is no edge between vertices v_i and v_j, then the combining cost $C(\{r_i, r_j\})$ is assigned 0. □

Our randomized RegEx grouping algorithm works as follows. Given a RegEx set $R = \{r_1, \ldots, r_n\}$, we first randomly partition them into k groups and calculate $\sum_{m=1}^{k} \sum_{1 \le i < j \le |R_m|} C(\{r_i, r_j\})$. Then, our algorithm repeats the following step: randomly picks an RegEx r in a group say R_m, for each other group, we calculate the benefit of moving r to that group in terms of reducing $\sum_{m=1}^{k} \sum_{1 \le i < j \le |R_m|} C(\{r_i, r_j\})$, then if the benefit is positive, we move r to the group with the maximum benefit. The algorithm terminates when there does not exist any RegEx so that moving that RegEx to another group has a positive benefit. The pseudocode of our algorithm is in Algorithm 1. Theorem 7 shows its approximation ratio.

Theorem 7: Our RegEx grouping is a $(1 - k^{-1})$-optimal.

Proof: Let $\{R_1, R_2, \ldots, R_k\}$ be the final k groups of our RegEx grouping algorithm for partitioning RegEx set R, and $\{R_1^*, R_2^*, \ldots, R_k^*\}$ be the optimal grouping result. Obviously $\{R_1^*, R_2^*, \ldots, R_k^*\}$ is also the optimal partition for the corresponding maximum k-cut problem. Note that $\{R_1, R_2, \ldots, R_k\}$ is local optimal, thus $^{\forall} i \in R_m$, $\sum_{1 \le j \le |R_m|} C(\{r_i, r_j\}) \le \sum_{1 \le j \le |R_t|, t \ne m} C(\{r_i, r_j\})$. Then we can infer that $\sum_{1 \le i, j \le |R_m|} C(\{r_i, r_j\}) \le \sum_{1 \le i \le |R_m| 1 \le j \le |R_t| t \ne m} C(\{r_i, r_j\})$, and further infer that $(k\text{–}1) \sum_{1 \le i, j \le |R_m|} C(\{r_i, r_j\}) \le \sum_{t=1}^{k} \sum_{1 \le i \le |R_m| 1 \le j \le |R_t| t \ne m} C(\{r_i, r_j\})$. Therefore, for the optimal grouping result $\{R_1^*, R_2^*, \ldots, R_k^*\}$, its total weights of the edges between disjoint groups is

$$OPT^* = \sum_{m=1}^{k-1} \sum_{t=m+1}^{k} \sum_{\substack{1 \le i \le |R_m^*| \\ 1 \le j \le |R_t^*|}} C(\{r_i, r_j\})$$

$$\le \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} C(\{r_i, r_j\})$$

$$= \sum_{m=1}^{k} \sum_{1 \le i, j \le |R_m|} \frac{C(\{r_i, r_j\})}{2} + \sum_{m=1}^{k-1} \sum_{t=m+1}^{k} \sum_{\substack{1 \le i \le |R_m| \\ 1 \le j \le |R_t|}} C(\{r_i, r_j\})$$

$$\le \frac{1}{2} \sum_{m=1}^{k} \frac{1}{k-1} \sum_{t=1}^{k} \sum_{\substack{1 \le i \le |R_m| \\ 1 \le j \le |R_t| \\ t \ne m}} C(\{r_i, r_j\}) + OPT$$

$$= \frac{1}{k-1} \sum_{m=1}^{k-1} \sum_{t=m+1}^{k} \sum_{\substack{1 \le i \le |R_m| \\ 1 \le j \le |R_t|}} C(\{r_i, r_j\}) + OPT$$

$$= \frac{k}{k-1} OPT$$

where $OPT = \sum_{m=1}^{k-1} \sum_{t=m+1}^{k} \sum_{\substack{1 \le i \le |R_m| \\ 1 \le j \le |R_t|}} C(\{r_i, r_j\})$ is the total weights of the edges between disjoint groups for the final grouping result of our grouping algorithm.

Thus $OPT \ge (1 - k^{-1}) OPT^*$. □

According to Theorem 7, our grouping algorithm exports a grouping result with no less than $1 - k^{-1}$ times of the optimal partition for maximum k-cut problem. To the best of our knowledge, the best approximation is $1 - k^{-1} - 2k^{-2} \ln k$ for maximum k-cut problem, which is obtained via a semi-definite programming relaxation [34]. However, it is not suitable to be used to address the RegEx grouping problem because of its high complexity.

C. RegEx Updating

RegExes need to be updated sometime in practice. When the RegEx set changes, the multiple DFAs need to be changed as well. For our RegEx grouping algorithm, this updating is simple and fast to perform. When a new RegEx is added, we can insert it to the smallest group and then rebuild the DFA for that group. When a RegEx is deleted or modified, we only need to rebuild the DFA for the group that the RegEx belongs to. As time goes by, if the DFA for one RegEx group becomes too big, we can rerun our RegEx grouping algorithm to recompute the groups.

D. Parallel Grouping and Matching

Our RegEx grouping algorithm can leverage parallel processing architecture such as the widely used multi-core architecture to achieve better performance. For our RegEx grouping algorithm, the time is mainly spent on computing the convolvement relationship and on finding the optimal move for each RegEx. Computing the convolvement relationship can be paralleled because we can let each core run the algorithm for different character α. Finding the optimal move for each RegEx because we can let each core to calculate the benefit of different moves.

TABLE II
COMPARISON OF *Con-Incremental* WITH *Col-ViaDFA*

RegEx set	NFA size	DFA size	# of convolvement relations	accuracy	seconds used		speedup times
					Col-ViaDFA	*Con-Incremental*	
snort24	575	13886	165025	0.9992	119.23	1.48	80.5
snort31	917	20068	680361	0.9998	155.36	3.25	47.8
snort34	891	13825	396495	0.9997	128.49	2.81	45.7
bro217	1999	6533	1997001	0.9999	66.20	10.34	6.4

After RegEx grouping, we can easily parallel packet processing by assigning the DFAs to different cores and let multiple cores perform the matching process for each input character at the same time.

V. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness and efficiency of our DFA size estimation and RegEx grouping algorithms.

A. Experimental Setup

We obtained one proprietary RegEx set, namely *C758*, from a major networking vendor, and three real-world RegEx sets, namely *l7filter*, *backdoor* and *bro831*, from three open-source systems: L7-filter [11], Snort [1], and Bro [2]. Set C758 contains 758 real RegExes. L7-filter is a popular application-layer traffic classifier for Linux's Netfilter. Snort and Bro are two open-source intrusion detection systems, which can be configured to perform protocol analysis and content inspection over online traffic to detect a variety of worms, attacks and probes. Set l7filter contains 107 RegExes in the latest version of L7-Filter. Set backdoor contains all the 158 RegExes in the "backdoor.rules" file of Snort systems. Set bro831 contains 831 RegExes from payload rules of Bro. For each of these RegEx sets, the size of the corresponding DFA has at least ten million states as our DFA construction process terminates with this many DFA states.

For our DFA size estimation algorithm, as it is the first of its kind, we compare it with DFA construction. For our RegEx grouping algorithm, we compared it Yu's algorithm [10] and Becchi's algorithm [35] as well as exhaustive grouping. Yu's algorithm has been explained in Section I. Becchi's algorithm performs binary cutting of the input RegEx set until the DFA of each group has a size less than a predefined threshold. Note that the order of RegExes in the given set affects the effectiveness of Becchi's algorithm. The exhaustive algorithm enumerates all the possible grouping in a brutal-force fashion to find the best grouping. We implemented our algorithms in C++ and performed the experiments on an Intel Xeon CPU E5410 with 2.33 GHz, 16 GB RAM, 64 KB L1 Cache, 6 MB L2 Cache and RedHat AS5 64 bit.

B. Results on Convolvement Relationship Calculation

We compare our convolvement relationship calculation algorithm, denoted *Con-Incremental* as we incrementally compute all convolvement relationship between every pair of states in a NFA, with the algorithm of computing convolvement relationship by constructing the DFA from a given NFA, denoted *Col-ViaDFA*. We implemented Yang's algorithm for computing convolvement relationship and observed that it often failed to terminate for the reason that we explained in Section III-D and it is often slower than the *Col-ViaDFA* algorithm; thus, we do not include Yang's algorithm in the results. We conducted the comparison using four public RegEx sets, namely snort24, snort31, snort34, and bro217, from the authors of [16], where the digits in the set name represents the number of RegExes in the set. We cannot conduct this experiment using the above mentioned large RegEx sets because their DFAs are too big to be practically built. Table II shows our experimental results. From this table, we observe that our *Con-Incremental* algorithm grows quadratically with NFA size in CPU time whereas the *Col-ViaDFA* algorithm grows exponentially in the worst case. As shown in Table II, *Con-Incremental* is on average 45.1 times faster than *Col-ViaDFA*. Among the four RegEx

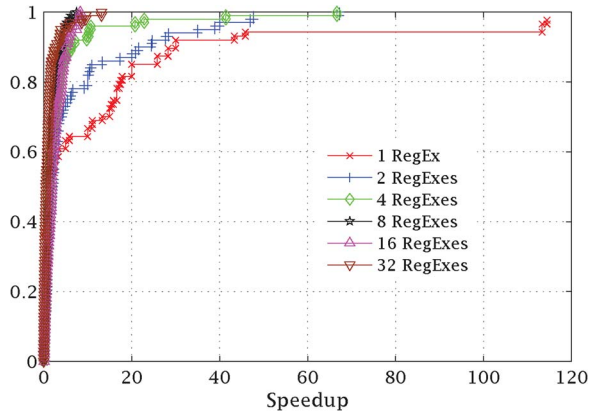

Fig. 6. Cumulative distribution function of speedup.

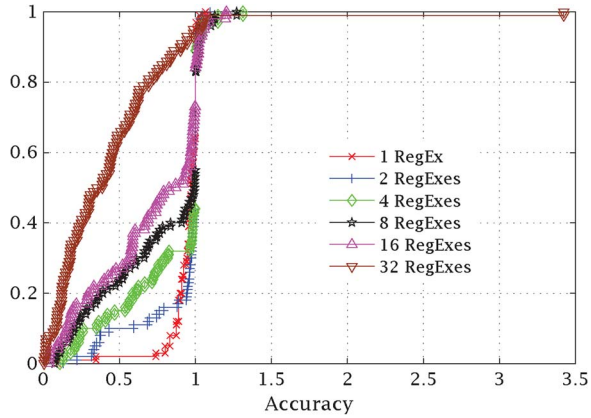

Fig. 7. Cumulative distribution function of accuracy.

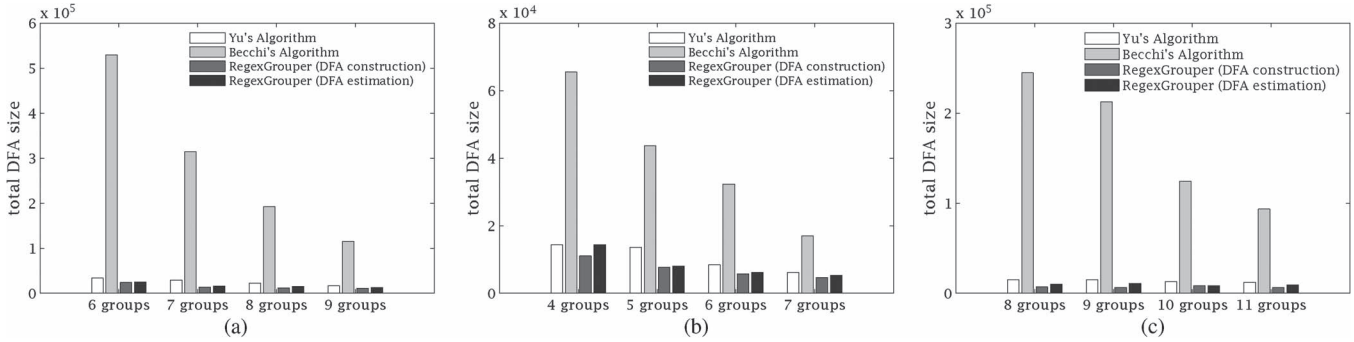

Fig. 8. Comparison of total DFA sizes on l7fitler, backdoor and bro831. (a) l7filter set; (b) backdoor set; (c) bro831 set.

sets, the speedup of *Con-Incremental* over *Col-ViaDFA* is the smallest for bro217 because most RegExes in it are strings, which experiences the least DFA state explosion compared with other RegEx sets. Our *Con-Incremental* algorithm achieves an accuracy of more than 99.9%, which is practically good enough to be used in applications such as DFA size estimation.

C. Results on DFA Size Estimation

We compared our DFA size estimation algorithm with DFA construction in terms of *accuracy* and *speedup*. The accuracy of DFA size estimation is defined as the ratio between estimated DFA size and actual DFA size. The speedup is the defined as the ratio between DFA construction time and DFA size estimation time. We randomly choose RegExes from C758, l7filter, backdoor, and bro831 to form RegEx sets of sizes 1, 2, 4, 8, 16, and 32, respectively, for this evaluation. For each size, we generate 100 RegEx sets of that size. The primary reason that we do not choose more RegExes to constitute the temporary RegEx set is that it is time-consuming to construct a composite DFA for large RegEx sets. What is worse that we need to perform 100 times. Fig. 6 shows the cumulative distribution function of speedup. From this figure, we observe that our DFA size estimation algorithm is orders of magnitude faster than DFA construction. The larger the RegEx set is, the higher the speedup is. Fig. 7 shows the cumulative distribution function of accuracy. From this figure, we observe that the smaller a RegEx set is, the more accurate that our DFA size estimation algorithm is.

D. Results on RegEx Grouping

We compare our RegEx grouping algorithm with three grouping algorithms: Yu's algorithm [10], Becchi's algorithm [35]. We evaluated two versions of our RegEx grouping algorithm: *RegexGrouper with DFA estimation*, which uses DFA size estimation procedure as we have explained in this paper, and *RegexGrouper with DFA construction*, which uses DFA construction procedure instead of the DFA size estimation procedure. As Yu's and Becchi's algorithms use thresholds to control the grouping process, the number of final groups is unknown in advance. In our comparisons, we constantly adjust the thresholds until we obtain k groups.

For effectiveness, we compare them on the total size of the DFAs constructed from groups. Fig. 8 shows the effectiveness results. First, the results show that our RegEx grouping algorithms, both RegexGrouper with DFA estimation and RegexGrouper with DFA construction achieve significantly smaller the total DFA size than all other algorithms. For example, on l7filter, RegexGrouper with DFA estimation achieves 1.5 times and 15.8 times smaller total DFA size than Yu's algorithm and Becchi's algorithm, respectively; on backdoor, RegexGrouper with DFA estimation achieves 1.3 times and 4.6 times smaller total DFA size than Yu's algorithm and Becchi's algorithm, respectively; on bro831, RegexGrouper with DFA estimation achieves 1.4 times and 17.1 times smaller total DFA size than Yu's algorithm and Becchi's algorithm, respectively. Second, the results show that RegexGrouper with DFA estimation and RegexGrouper with DFA construction result in very similar total DFA size. This shows that our DFA size estimation is effective in RegEx grouping.

For efficiency, we compare them on the total running time, which include both the preprocessing time spent on calculating interaction relationship or combining cost between RegExes and the RegEx grouping time. Note that Becchi's algorithm works without calculating anything in advance, thus its total running time is only the RegEx grouping time. Fig. 9 shows the efficiency results. First, the results show that our RegexGrouper with DFA estimation runs significantly faster than all other algorithms. For example, on l7filter, RegexGrouper with DFA estimation runs 87.9 times and 50.4 times faster than Yu's algorithm and Becchi's algorithm, respectively; on backdoor, RegexGrouper with DFA estimation runs 72.7 times and 19 times faster than Yu's algorithm and Becchi's algorithm, respectively; on bro831, RegexGrouper with DFA estimation runs 25.7 times and 7.5 times faster than Yu's algorithm and Becchi's algorithm. Second, the results show that RegexGrouper with DFA estimation and RegexGrouper with DFA construction have stable total running time for different k. Because RegexGrouper only needs to perform a limited number of moving steps to partition a RegEx set, as a result the grouping time is less 0.1 seconds, which is negligible. That is to say for RegexGrouper the total running time is the preprocessing time.

Table III shows both effectiveness and efficiency for RegEx set C758. From this table, we can draw similar conclusions. For effectiveness, RegexGrouper with DFA estimation achieves 2.4 times and 7.11 times smaller total DFA size than Yu's algorithm and Becchi's algorithm, respectively. For efficiency, on C758, RegexGrouper with DFA estimation runs 41.7 times and

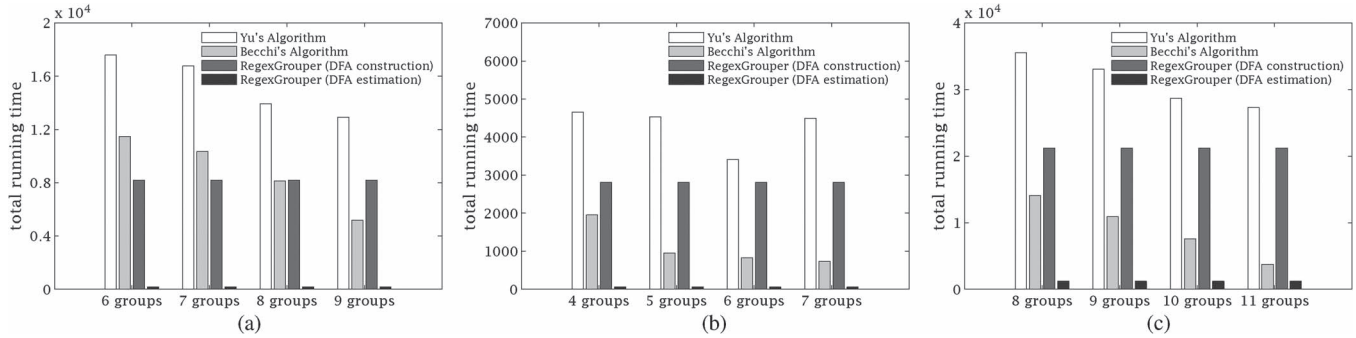

Fig. 9. Comparison of grouping times on l7fitler, backdoor and bro831. (a) l7filter set; (b) backdoor set; (c) bro831 set.

TABLE III
COMPARISON ON TOTAL RUNNING TIME AND TOTAL DFA SIZE FOR C758 SET

grouping approach	k=12		k=13		k=14		k=15	
	total size	total time	total size	total time	total size	total time	total size	total time
Yu's Algorithm	166449	46699	113391	36052	86199	34042	47615	29553
Becchi's Algorithm	478083	28392	340926	26202	232208	29349	166718	18869
RegexGrouper (DFA construction)	45852	17512	38072	17512	35344	17512	25019	17512
RegexGrouper (DFA estimation)	60117	877	42169	877	34396	877	29428	877

TABLE IV
VALUES OF $S(n,k)$ FOR $10 \leq n \leq 18$ AND $0 \leq k \leq 5$

n \ k	0	1	2	3	4	5
10	0	1	511	9330	34105	42525
11	0	1	1023	28501	145750	246730
12	0	1	2047	86526	611501	1379400
13	0	1	4095	261625	2532530	7508501
14	0	1	8191	788970	10391745	40075035
15	0	1	16383	2375101	42355950	210766920
16	0	1	32767	7141686	171798901	1096190550
17	0	1	65535	21457825	694337290	5652751651
18	0	1	131071	64439010	2798806985	28958095545

29.3 times faster than Yu's algorithm and Becchi's algorithm, respectively.

We also compared our algorithms with the exhaustive algorithm. The key limitation of this algorithm is too slow. First, the number of ways to divide n entities (e.g., RegExes) into k non-empty groups is given by the Stirling number of the second kind $S(n,k) = \frac{1}{k!}\sum_{j=1}^{k}(-1)^{k-j}\frac{k!}{(k-j)!j!}j^n$ [36]. Thus, to get the best grouping result, the total number of DFAs we need to construct is $k \times S(n,k)$. Table IV shows the values of $S(n,k)$ for some specific n and k. Note that $S(n,k)$ is bigger than ten thousands when $n > 14$ or $k > 3$. On the other hand, the time complexity of finding all possible grouping results is $O(k^n)$, which is far larger than $O(S(n,k))$.

In our experiments, we randomly choose $n = 14$ RegExes from C758, l7filter, backdoor and bro831 sets and distribute them into $k = 2$ groups. We repeat this process for 100 times. Fig. 10 shows the cumulative distribution function of the ratio between the total DFA size of RegexGrouper and that of Exhaustive approach. We observe that, when $k = 2$, the total DFA size of RegexGrouper is no more than 4 times as that of exhaustive algorithm, and the average ratio of total DFA size is 1.56. Fig. 11 shows the cumulative distribution function of the

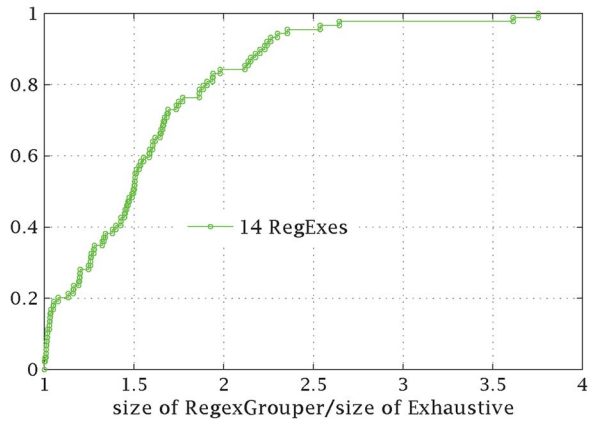

Fig. 10. Comparison of Exhaustive and RegexGrouper on total DFA size.

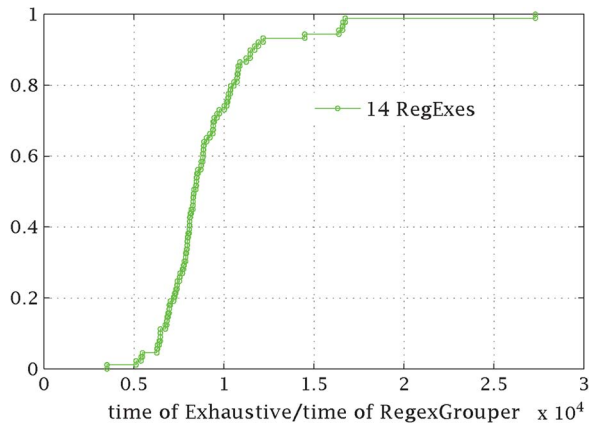

Fig. 11. Comparison of Exhaustive and RegexGrouper on total running time.

ratio between the total running time of the exhaustive algorithm and that of RegexGrouper. We observe that the total running time of Exhaustive approach is usually more than 5000 times

as that of RegexGrouper, and the average ratio of total running time is 9,099.

VI. Conclusion

We make three key contributions in this paper. First, we propose a DFA size estimation algorithm, which is the first of its kind to the best of our knowledge. Second, we propose a RegEx grouping algorithm with no DFA construction, which achieves orders of magnitude faster and more memory efficient than Yu's algorithm. Third, we also propose a convolvement relationship calculation algorithm, which is much faster than using DFA construction. Fourth, we implemented our algorithm in C++ and conducted experiments on real-world RegEx sets. Our experimental results show that our algorithm achieves not only much faster speed than Yu's algorithm but also much better grouping results.

Acknowledgment

We are extremely grateful for the valuable suggestions and comments from the reviewers. We would like to thank Zhilu Zhang for helping with the experiments when we conduct the revision.

References

[1] M. Roesch, "Snort—Lightweight intrusion detection for networks," in *Proc. USENIX LISA*, 1999, pp. 229–238.
[2] P. Vern, "Bro: A system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, no. 23, pp. 2435–2463, Dec. 1999.
[3] S. Iyer, R. R. Kompella, and N. McKeown, "Designing packet buffers for router linecards," *IEEE/ACM Trans. Netw.*, vol. 16, no. 3, pp. 705–717, Jun. 2008.
[4] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 24–33, Jun. 2009.
[5] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, acalculia," in *Proc ACM/IEEE ANCS*, 2007, pp. 155–164.
[6] A. X. Liu, E. Norige, and S. Kumar, "A few bits are enough—ASIC friendly regular expression matching for high speed network security systems," in *Proc IEEE ICNP*, Gottingen, Germany, Oct. 2013, pp. 1–10.
[7] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *Proc IEEE Symp. Security Privacy*, 2008, pp. 187–201.
[8] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 207–218, 2008.
[9] A. X. Liu and E. Torng, "An overlay automata approach to regular expression matching," in *Proc IEEE INFOCOM*, 2014, pp. 952–960.
[10] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. ACM/IEEE ANCS*, 2006, pp. 93–102.
[11] J. Levandoski, E. Sommer, and M. Strait, *Application Layer Packet Classifier for Linux*. [Online]. Available: http://l7-filter.sourceforge.net/
[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 339–350, Oct. 2006.
[13] D. Ficara *et al.*, "An improved DFA for fast regular expression matching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 29–40, Oct. 2008.
[14] D. Ficara *et al.*, "Differential encoding of DFAs for fast regular expression matching," *IEEE/ACM Trans. Netw.*, vol. 19, no. 3, pp. 683–694, Jun. 2011.
[15] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. ACM/IEEE ANCS*, 2006, pp. 81–92.
[16] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. ACM/IEEE ANCS*, 2007, pp. 145–154.
[17] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. ACM CoNEXT Conf.*, 2007, pp. 1–12.
[18] Y. Xu, J. Jiang, Y. Song, T. Jiang, and H. J. Chao, "i-DFA: A novel deterministic finite automaton without state explosion," Polytech. Inst. New York Univ. (NYU), Brooklyn, NY, USA, Tech. Rep. Polytechnic Institute of NYU 2010, 2010.
[19] Y. Xu, J. Jiang, R. Wei, Y. Song, and H. J. Chao, "TFA: A tunable finite automaton for regular expression matching," Polytech. Inst. New York Univ. (NYU), Brooklyn, NY, USA, Tech. Rep., 2013.
[20] M. Bando, N. S. Artan, and H. J. Chao, "LaFA: Lookahead finite automata for scalable regular expression detection," in *Proc. ACM/IEEE ANCS*, 2009, pp. 40–49.
[21] M. Bando, N. S. Artan, and H. J. Chao, "Scalable lookahead regular expression detection system for deep packet inspection," *IEEE/ACM Trans. Netw.*, vol. 20, no. 3, pp. 699–714, Jun. 2012.
[22] T. Liu, Y. Sun, A. X. Liu, L. Guo, and B. Fang, "A prefiltering approach to regular expression matching for network security systems," in *Proc. ACNS*, 2012, pp. 363–380.
[23] X. Wang, J. Jiang, Y. Tang, B. Liu, and X. Wang, " StriD^{2}FA: Scalable regular expression matching for deep packet inspection," in *Proc. IEEE ICC*, 2011, pp. 1–5.
[24] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems," in *Proc. 19th USENIX Security*, Washington DC, USA, Aug. 2010, pp. 111–126.
[25] C. R. Meiners, J. Patel, E. Norige, A. X. Liu, and E. Torng, "Fast regular expression matching using small TCAM," *IEEE/ACM Trans. Netw.*, vol. 22, no. 1, pp. 94–109, Feb. 2014.
[26] K. Peng, S. Tang, M. Chen, and Q. Dong, "Chain-based DFA deflation for fast and scalable regular expression matching using TCAM," in *Proc. ACM/IEEE ANCS*, 2011, pp. 24–35.
[27] K. Huang *et al.*, "Scalable TCAM-based regular expression matching with compressed finite automata," in *Proc. ACM/IEEE ANCS*, San Jose, CA, USA, Oct. 2013, pp. 83–93.
[28] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. IEEE FCCM*, 2001, pp. 227–238.
[29] P.-C. Lin, Y.-D. Lin, Y.-C. Lai, Y.-J. Zheng, and T.-H. Lee, "Realizing a sub-linear time string-matching algorithm with a hardware accelerator using bloom filters," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 8, pp. 1008–1020, Aug. 2009.
[30] V. C. Valgenti *et al.*, "GPP-Grep: High-speed regular expression processing engine on general purpose processors," in *Proc. RAID*, 2012, pp. 334–353.
[31] J. Patel, A. X. Liu, and E. Torng, "Bypassing space explosion in regular expression matching for network intrusion detection and prevention systems," in *Proc. 19th Annu. NDSS*, San Diego, CA, USA, Feb. 2012, pp. 1–15.
[32] J. Patel, A. X. Liu, and E. Torng, "Bypassing space explosion in high-speed regular expression matching," in *Proc. 19th Annu. NDSS*, San Diego, CA, CA, Feb. 2012, to be published.
[33] Y.-H. Yang and V. K. Prasanna, "Space-time tradeoff in regular expression matching with semi-deterministic finite automata," in *Proc. IEEE INFOCOM*, 2011, pp. 1853–1861.
[34] A. Frieze and M. Jerrum, "Improved approximation algorithms for MAX k-CUT and MAX BISECTION," *Algorithmica*, vol. 18, no. 1, pp. 67–81, May 1997.
[35] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Proc. IEEE IISWC*, 2008, pp. 79–89.
[36] Wikipedia, *Stirling Numbers of The Second Kind*. [Online]. Available: http://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind

Tingwen Liu received the Ph.D. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Shanghai, China, in 2013. He is currently an Assistant Professor at the Institute of Information Engineering, Chinese Academy of Sciences, Shanghai. His research interests include networking, algorithms, and security.

Alex X. Liu received the Ph.D. degree in computer science from The University of Texas at Austin, Austin, TX, USA, in 2006. His research interests focus on networking and security. He is an Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING and the Journal of Computer Communications. He is the TPC Co-Chair of ICNP 2014. Dr. Liu received the IEEE & IFIP William C. Carter Award in 2004, an NSF CAREER Award in 2009, and the Michigan State University Withrow Distinguished Scholar Award in 2011. He received Best Paper Awards from ICNP 2012, SRDS 2012, LISA 2010, and TSP 2009.

Jinqiao Shi received the Ph.D. degree in computer science from Harbin Institute of Technology, Harbin, China, in 2007. He is currently an Associate Professor at the Institute of Information Engineering, Chinese Academy of Sciences. His research interests focus on network security and data leakage prevention.

Yong Sun received the Ph.D. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Shanghai, China, in 2008. He is currently an Associate Professor at the Institute of Information Engineering, Chinese Academy of Sciences, Shanghai. His research interests include stream mining and information security.

Li Guo is the director of the Laboratory of Intelligent Information Processing Technologies, Institute of Information Engineering, Chinese Academy of Sciences, Shanghai, China. She is also the standing vice director of the National Engineering Laboratory for Information Security Technologies, Beijing, China. Her research interests include data stream management and information security.