TFA: A Tunable Finite Automaton for Regular Expression Matching

Yang Xu[†], Junchen Jiang[§], Rihua Wei[†], Yang Song[†] and H. Jonathan Chao[†]
[†]Polytechnic Institute of New York University, USA
[§]Carnegie Mellon University, USA

***Abstract*—Deterministic Finite Automatons (DFAs) and Nondeterministic Finite Automatons (NFAs) are two typical automatons used in the Network Intrusion Detection System (NIDS). Although they both perform regular expression matching, they have quite different performance and memory usage properties. DFAs provide fast and deterministic matching performance but suffer from the well-known state explosion problem. NFAs are compact, but their matching performance is unpredictable and with no worst case guarantee. In this paper, we propose a new automaton representation of regular expressions, called Tunable Finite Automaton (TFA), to resolve the DFAs' state explosion problem and the NFAs' unpredictable performance problem. Different from a DFA, which has only one active state, a TFA allows multiple concurrent active states. Thus, the total number of states required by the TFA to track the matching status is much smaller than that required by the DFA. Different from an NFA, a TFA guarantees that the number of concurrent active states is bounded by a bound factor b that can be tuned during the construction of the TFA according to the needs of the application for speed and storage. Simulation results based on regular expression rule sets from Snort and Bro show that with only two concurrent active states, a TFA can achieve significant reductions in the number of states and memory usage, e.g., a 98% reduction in the number of states and a 95% reduction in memory space.**

I. Introduction

Deep Packet Inspection (DPI) is a crucial technique in today's Network Intrusion Detection System (NIDS), where it compares incoming packets byte-by-byte against patterns stored in a database to identify specific viruses, attacks and protocols. Early DPI methods rely on exact string matching [1] [2] [3] [4] for attack detection, whereas recent DPI methods use regular expression matching [5] [6] [7] [8] because the latter provides better flexibility in representing the ever-evolving attacks [9]. Regular expression matching has been widely used in many NIDSes such as Snort [10], Bro [11], and several network security appliances from Cisco systems [12] and has become the de facto standard for content inspection.

Despite its flexible attack representation, regular expression matching introduces significant computational and storage challenges. Deterministic Finite Automatons (DFAs) and Nondeterministic Finite Automatons (NFAs) are two typical representations of regular expressions. Given a set of regular expressions, we can easily construct the corresponding NFA, from which the DFA can be further constructed using subset construction scheme [13]. DFAs and NFAs have quite different performance and memory usage characteristics. A DFA has at most one active state during the entire matching and, therefore, requires only one state traversal for each character processing, resulting in a deterministic memory bandwidth requirement. The main problem of using a DFA to represent regular expressions is the DFA's severe state explosion problem [5], which often leads to a prohibitively large memory requirement. In contrast, an NFA represents regular expressions with much less memory storage. However, this memory reduction comes with the price of a high and unpredictable memory bandwidth requirement. This is because the number of concurrent active states in an NFA is unpredictable during the matching. Processing a single character in a packet with an NFA may induce a large number of state traversals, which translate into a large number of memory accesses and limit the matching speed.

Recently, many research works have been proposed in literature pursuing a tradeoff between the computational complexity and storage complexity for the regular expression matching [5] [6] [7] [8] [9] [14]. Among these proposed solutions, some [8] [9] have a motivation similar to ours, i.e., to design a hybrid finite automaton fitting between DFAs and NFAs. These automatons, though compact and fast when processing common traffic, suffer from poor performance in the worst cases. This is because none of them can guarantee an upper bound on the number of active states during the matching processing. This weakness can potentially be exploited by attackers to construct a worst-case traffic that can slow down the NIDS and cause malicious traffic to escape from the inspection.

In fact, the design of a finite automaton with a small (larger than one) but bounded number of active states remains an open and challenging problem. In this paper, we propose Tunable Finite Automaton (TFA), a new automaton representation, for regular expression matching to resolve the DFAs' state explosion problem and NFAs' unpredictable performance problem. The main idea of TFA is to use a few TFA states to remember the matching status traditionally tracked by a single DFA state. As a result, the number of TFA states required to represent the information stored on the counterpart DFA is much smaller than that of DFA states. Unlike an NFA, a TFA has the number of concurrent active states strictly bounded by a bound factor b, which is a parameter that can be tuned during the construction of the TFA according to the needs for speed and storage.

Our main contributions in this paper are summarized below.

(1) We introduce TFA, which to the best of our knowledge, is the first finite automaton model with a clear and tunable bound on the number of concurrent active states (more than

one) independent of the number and patterns of regular expressions. TFA is a general finite automaton model, which covers DFA and NFA as two special cases. It becomes DFA when the bound factor b is set to 1 and NFA when b is set to infinite. In addition, a TFA can be equivalently constructed from any NFAs and therefore supports all regular expressions.

(2) A mathematical model is built to analyze the set split problem (SSP), the most critical step in the TFA construction. We prove that the SSP problem is NP-hard and propose a heuristic algorithm to approximately solve it.

(3) We develop a novel state encoding scheme to facilitate the implementation of a TFA. With the state encoding, a TFA can be stored in a compacted memory, and the run-time overheads of TFA operations are significantly reduced.

(4) The proposed TFA is evaluated using regular expression sets from Snort and Bro. Simulation results show that a TFA can achieve significant reductions in the number of states and memory usage even with the simplest TFA (only two concurrent active states).

The rest of the paper is organized as follows. Section II reviews the related work. Section III presents the motivation and main idea of the paper. Section IV explains the technical details of the TFA, including its mathematical model, construction scheme, and operation procedure. In section V, we formalize an important problem in the TFA design: set split, and propose a heuristic algorithm to solve it. Section VI presents the state encoding scheme. Section VII presents the simulation results. Finally, section VIII concludes the paper.

II. Related Work

Most of the research today in regular expression matching focuses on reducing the memory usage of DFAs and can be classified into the following categories:

(1) Transition reduction

Schemes in this category reduce the memory usage of a DFA by eliminating redundant transitions. The D^{2}FA [6] proposed by Kumar et al. is a representative method in this category. It eliminates redundant transitions in a DFA by introducing default transitions, and saves memory usage at the cost of increasing the memory access times for each input character. After the D^{2}FA, many other schemes, such as the CD^{2}FA [14] and [15] were proposed to improve the D^{2}FA's worst-case run-time performance and construction complexity.

(2) State reduction

Schemes in this category reduce the memory usage of a DFA by alleviating its state explosion. Due to the fact that many regular expressions interact with others, the composite DFA for multiple regular expressions could possibly be extremely large (i.e., state explosion). Yu et al. [5] and Jiang et al. [16] proposed to combine regular expressions into multiple DFAs instead of one to eliminate the state explosion. This scheme reduces the memory usage but usually requires much more DFAs, which increases the memory bandwidth demand linearly with the number of DFAs used. The XFA [7] [17] uses auxiliary memory to achieve a significant memory reduction. Unfortunately, the creation of XFA involves a lot of manual work which is error-prone and inefficient and its performance is non-deterministic.

In [18], Becchi proposed an algorithm to merge DFA states by introducing labels on their input and output transitions. In [19], Kumar et al. proposed history-based finite automatons to record history information in matching which capture one of the major reasons for DFA state explosion and reduce the memory cost. However, to record history will increase the worst case complexity and thus compromise scalability.

(3) Hybrid Finite Automaton

Schemes in this category aim at designing automatons fitted into the middle ground between NFAs and DFAs so that the strengths of both NFAs and DFAs can be obtained. Becchi et al. proposed a hybrid finite automaton called Hybrid-FA [8], which consists of a head DFA and multiple tail-NFAs/tail-DFAs. Although a Hybrid-FA can achieve an average case memory bandwidth requirement similar to that of a single DFA with significantly reduced memory usage, its worst case memory bandwidth requirement is unpredictable and varies when the regular expression rule set is updated. Lazy DFA [9] is another automaton used to leverage the advantages of both NFAs and DFAs. Its main function is to store only frequently used DFA states in memory, while leaving others in NFA representation. In case an uncommon DFA state is required, lazy DFA has to be extended at run-time from the NFA. So it is no surprise that this automaton is fast and memory-efficient in common cases, but in the worst case, the whole DFA needs to be expanded, making it vulnerable to malicious traffic.

III. Motivation

In this section, we first review the time-space tradeoff between an NFA and its counterpart DFA by way of example, and then demonstrate how a TFA combines both of their strengths. In Figure 1(a) and (b), we show an NFA and a DFA representing the same set of regular expressions, i.e., `.*a.*b[^a]*c`, `.*d.*e[^d]*f`, `.*g.*h[^g]*i`, with the alphabet $\Sigma = \{a, b, ..., i\}$. We can see that although the NFA and DFA have the same functionality, the state number in the DFA (54) is 5.4 times that in the NFA (10).

Although the NFA requires much less memory, its memory bandwidth requirement is four times that of the DFA. This is because the NFA may have up to four concurrent active states while the DFA only has one. Consider an input string of "$adegf$". The initial active state combination of NFA is {O}. The active state combinations of the NFA after the scanning of each of these characters are {O,A}, {O,A,D}, {O,A,E}, {O,A,E,G}, and {O,A,F,G}. We can see that after character "g" is read, there are four states: i.e., "O", "A", "E", and "G", active simultaneously in the NFA. Unlike the NFA, the DFA has only one state activated during the entire matching. Consider the same input string "$adegf$". The initial state of DFA is "O". The states visited by the DFA after each character is scanned are "OA", "OAD", "OAE", "OAEG", and "OAFG"[1].

[1]In this paper, lowercase letters are used to denote input characters; single capital letters denote NFA states, while the strings of capital letters denote DFA and TFA states.

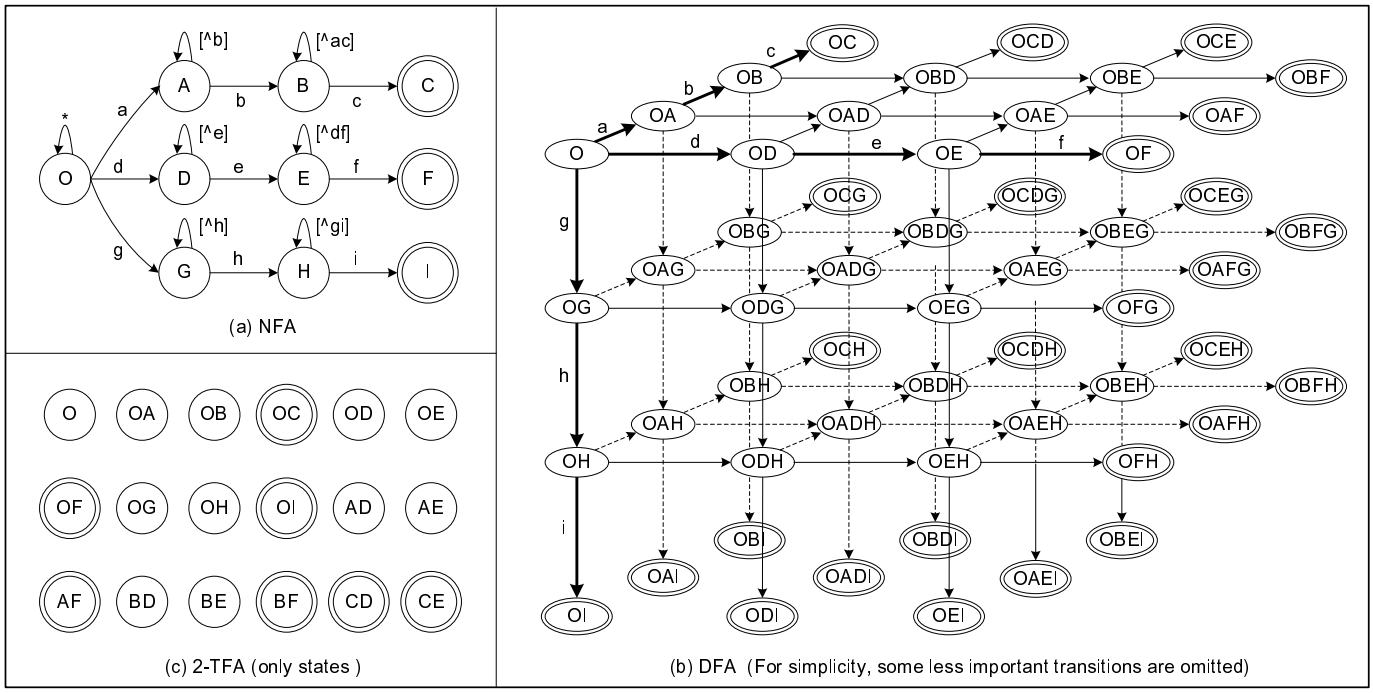


Fig. 1. Example of an NFA, a DFA and a 2-TFA associated with three regular expressions: `.*a.*b[^a]*c`, `.*d.*e[^d]*f` and `.*g.*h[^g]*i`.

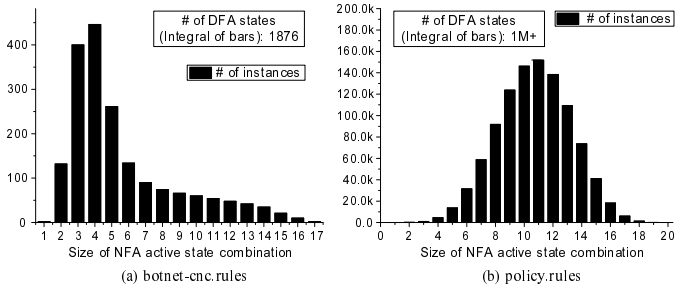


Fig. 2. Distribution of sizes of NFA active state combinations.

In Figure 2, we show the distribution of sizes of NFA active state combinations associated with two small regular expression rule sets from Snort. The integral of the distribution over the X-axis represents the size of the corresponding DFA. From the figure, we have three observations that (1) the NFA has many concurrent active states, even though its counterpart DFA is not that big (as in Figure 2 (a), the NFA has up to 17 states activated simultaneously, even though the corresponding DFA is very small with only 1876 states); (2) the integral of the distribution over the X-axis is much larger than the NFA size, resulting in the DFA state explosion (the sizes of NFA and DFA for Figure 2 (b) are 398 and 1M+, respectively. The rate of state explosion is more than 2.5K); (3) a larger state explosion rate implies a slower NFA and a bigger DFA.

We have seen the main reason for the DFA having far more states than the corresponding NFA is that the DFA needs one state for each NFA active state combination. If we want to reduce the DFA size (denoted by N_D), one possible solution is to *allow multiple automaton states (bounded by a given bound factor b) to represent each combination of NFA active states*.In other words, we allow up to b active states in the new automaton, and name it Tunable Finite Automaton (TFA). For simplicity, we use b-TFA to denote a TFA with up to b active states.

To see the potential of a TFA, let N_T be the number of TFA states; the number of all possible statuses (denoted by P) that can be represented by at most b active states of the b-TFA is (normally, $b \ll N_T/2$)

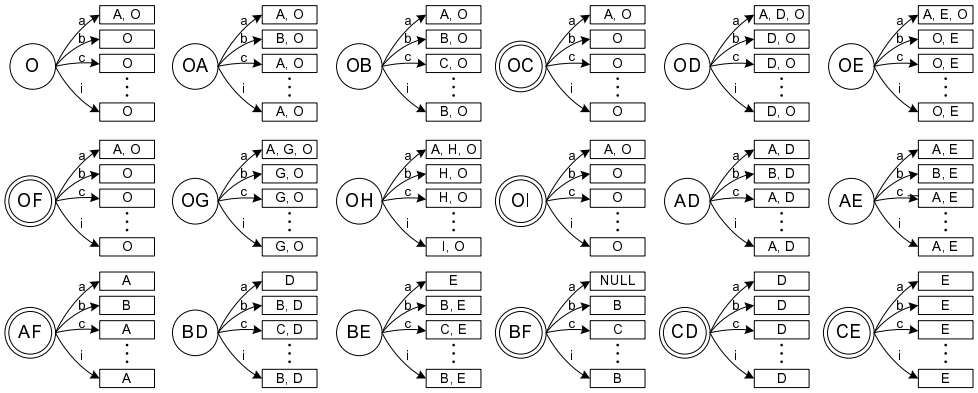
$$P = \sum_{i=1}^{b} \left(\begin{array}{c} N_T \\ i \end{array} \right) = O\left({N_T}^b\right) \quad (1)$$

Thus, a TFA with $N_T = O\left(\log_b\left(N_D\right)\right)$ states can represent a DFA with N_D states.

A. TFA States

The main idea of TFA is illustrated with an example. Suppose we want to design a 2-TFA based on the NFA in Figure 1(a). We first generate the corresponding DFA, which provides us with all valid combinations of NFA active states (Figure 1(b)). Then, we split each combination of NFA active states into two subsets, with the aim of minimizing the number of distinct subsets, and generate one 2-TFA state for each distinct subset. For instance, the NFA active state combination {`O,A,D,G`} can be split into {`O,G`} and {`A,D`}, and represented by two 2-TFA states "`OG`" and "`AD`".

Figure 1(c) shows a 2-TFA with only 18 states (only the 2-TFA states are given. The details of how it works will be provided shortly). It is easy to see that any valid combination of NFA active states can always be exactly covered by (at most) two 2-TFA states. The 2-TFA (18 states, at most 2 active) achieves a significant reduction in the number of



(a) A TFA structure with b=2.

{O}	&O	NULL
{A,O}	&OA	NULL
{B,O}	&OB	NULL
…	…	…
{A,D,O}	&O	&AD
{A,E,O}	&OA	&AE
…	…	…
{A,E,G,O}	&OG	&AE
{A,F,G,O}	&OG	&AF
…	…	…

(b) Set Split Table (SST)

Fig. 3. The architecture of a sample TFA ($b = 2$) associated with the NFA in Figure 1(a).

states compared to the corresponding DFA (54 states) and a significant reduction in the memory bandwidth requirement compared to the NFA (4 active states in the worst case).

B. TFA Transitions

The most challenging part of designing a TFA is to connect TFA states with the proper transitions and let the TFA emulate the corresponding NFA and DFA. Recall that when an NFA scans an input string, the real-time matching status is tracked by its concurrent active states. To let a TFA emulate an NFA, we just need to guarantee that the active states of the NFA can be recovered from the active states of the TFA after each character is processed.

Consider the NFA in Figure 1(a) again and suppose that we have a 2-TFA as in Figure 1(c) that can emulate the NFA until time slot[2] t. At the end of time slot t, suppose there are two active states in the 2-TFA ("OD" and "OG") and the active state combination in the corresponding NFA is {O, D, G}. Now assume that the character read in time slot $t+1$ is "a". It is easy to see that the active state combination of NFA at the end of time slot $t+1$ would be {O, A, D, G}. The problem is how we could operate the 2-TFA to make it work exactly as the NFA does in time slot $t+1$? If we run the two active states "OD" and "OG" separately, their next active states should be "OAD" and "OAG", respectively, which clearly are not in the state set given in Figure 1(c). Adding new state "OAD" and "OAG" into the state set of 2-TFA will cause a bigger automaton than we expected.

In this paper, we propose a novel way to operate a TFA. Rather than running TFA active states individually in each time slot to get the next TFA active states, we first recover the active state combination of the NFA by combining the results obtained by the individual TFA states. Then we access a table called *Set Split Table (SST)* to find out the next TFA active states whose combination is equal to this NFA active state combination. This way, no extra TFA states need to be added.

[2]One time slot is defined as the time period required to process a character. It is a constant (or variable) if the automaton has a deterministic (or non-deterministic) performance.

IV. Tunable Finite Automaton(TFA)

A TFA can be generated from any NFA. In this section, we first give the formal definitions of NFAs, DFAs, and TFAs, and then present the procedure to generate the TFA based on an NFA. After that, we show how to operate the TFA to emulate the operation of the NFA.

A. Definition of TFAs

Remember that an NFA can be represented formally by a 5-tuple $<Q_N, \Sigma, \delta_N, q_N, F_N>$, consisting of

- A finite set of NFA states Q_N;
- A finite set of input symbols Σ;
- A transition function δ_N : $Q_N \times \Sigma \mapsto P(Q_N)$;
- An initial state q_N;
- A set of accept states $F_N \subseteq Q_N$;

where $P(Q_N)$ denotes the power set of Q_N.

To be deterministic, a DFA consists of the similar 5-tuple $< Q_D, \Sigma, \delta_D, q_D, F_D >$ but with a transition function δ_D : $Q_D \times \Sigma \mapsto Q_D$ that transfers the current state to only one next state if any symbol is read.

A b-TFA extends the 5-tuple definition of DFA/NFA, by introducing the set split function SS. Formally, a b-TFA is a 6-tuple $< Q_T, \Sigma, \delta_T, I, F_T, SS >$, consisting of

- A finite set of TFA states Q_T;
- A finite set of input symbols Σ;
- A transition function $\delta_T : Q_T \times \Sigma \mapsto P(Q_N)$;
- A set of initial states $I \subseteq Q_T$, $|I| \leq b$;
- A set of accept states $F_T \subseteq Q_T$;
- A set split function $SS : Q_D \mapsto (Q_T)^b \bigcup \ldots \bigcup (Q_T)^1$.

B. Constructing A TFA

The implementation of a TFA, based on its definition, logically consists of two components: a TFA structure that implements $Q_T, \Sigma, \delta_T, I, F_T$; and a *Set Split Table (SST)* that implements *SS*. Figure 3(a) and (b) show the TFA structure (with 18 isolated states) and SST table associated with the NFA in Figure 1(a). Each entry of the SST table corresponds to one combination of NFA active states (i.e., a DFA state)

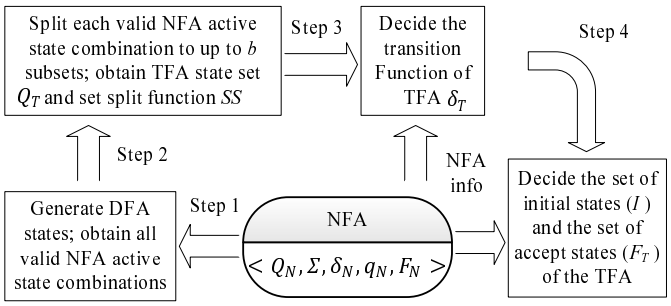


Fig. 4. The flowchart for generating a TFA.

recording how to split the combination into multiple TFA states (the memory addresses of the TFA states are stored).

The main flowchart for generating a equivalent b-TFA from an NFA is given in Figure 4, which consists of four steps:

(1) Generate the DFA states using the subset construction scheme [13]. The obtained DFA states provide us with all valid NFA active state combinations;

(2) Split each NFA active state combination into up to b subsets, with the objective of minimizing the number of distinct subsets, and generate one TFA state for each distinct subset. After this step, we obtain the TFA state set Q_T and the set split function SS;

(3) Decide the transition function δ_T. Different from traditional automatons, outgoing transitions of TFA states do not point to other TFA states. Instead, they point to a data structure called *state label*, which contains a set of NFA state IDs. Given a TFA state s, its state label associated with character "c" includes all NFA states that can be reached via character "c" from the NFA states associated with TFA state s. For instance, consider TFA state "AD" in Figure 3(a); its state label associated with character "b" is {B,D}, which can be obtained by running state "A" and "D" using "b" in the NFA.

(4) Decide the set of initial states (I) and the set of accept states (F_T). Set I includes TFA states split from the initial state of the counterpart DFA (i.e., q_D). Set F_T includes TFA states associated with at least one NFA accept state.

Note that although the construction of a TFA requires obtaining all DFA states via subset construction, it does not require the generation of a complete DFA since no DFA transition is computed or stored. In our experiments, the total memory usage during this procedure is only 1% of that for compiling the complete DFA.

C. Operating A TFA

Algorithm 1 describes the operations of a b-TFA in each time slot. Consider the TFA in Figure 3 and assume the input string is "*adegf*". After reading in the first character "*a*", the initial active state of TFA "O" returns a state label {A,O}. We use {A,O} to run a set query in the SST table, which returns the memory address of the next active TFA state (i.e., &OA in this case). With the next character "*d*", active state "OA" will return state label {A,D,O}. After searching the SST table using {A,D,O}, we get two active states (&O and &AD). With the third input character "*e*", active state "O" returns state label {O} and active state "AD" returns state label {A,E}. The union of the two labels (i.e., {A,E,O}) is sent to the SST table to find the next active states (&OA and &AE). The above procedure repeats in every time slot until the entire input string is scanned.

It should be noted that the scheme of TFA is essentially different from the DFA grouping scheme proposed in [5]. The DFA grouping scheme cannot be applied or performs badly under certain circumstances, such as the situations in which the rule set has only one regular expression, or has multiple regular expressions but one of them is extremely complex. Consider the NFA and DFA shown in Figure 5(a) and 5(b), which represent a single regular expression `.*ab.{3}cd` used in [8]. Apparently the DFA grouping scheme cannot be used in this single-rule case; however, the TFA can still be adopted to reduce the memory cost. Consider the 2-TFA with only 9 states shown in Figure 5(c); we can always use (at most) two 2-TFA states to exactly cover a valid combination of NFA active states. This example also shows the efficiency of TFAs when handling regular expressions with repetitions (i.e., counting).

In Section V and VI, we will address two critical issues in the implementation of a TFA: (1) How to split the NFA active state combinations to obtain a small TFA; (2) How to store TFA and SST table efficiently to facilitate the TFA operation.

V. Splitting NFA Active State Combinations

A. Set Split Problem (SSP)

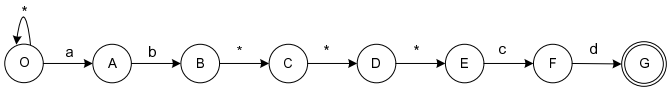
The SSP problem is to split each NFA active state combination into up to b non-empty subsets (overlaps among the subsets are allowed). To get a small TFA, the number of distinct subsets after the set split should be minimized. The SSP problem can also be rephrased to a special set covering problem; i.e., to find a minimal number of subsets from the NFA state set, so that for any valid NFA active state combination, we can always find up to b subsets to exactly cover it. We denote the SSP problem with a bound factor of b as b-SSP problem, and formalize it in Table II based on notations in Table I.

We have proved that the b-SSP problem is an NP-hard problem for any $b > 1$ (please refer to the proof in the

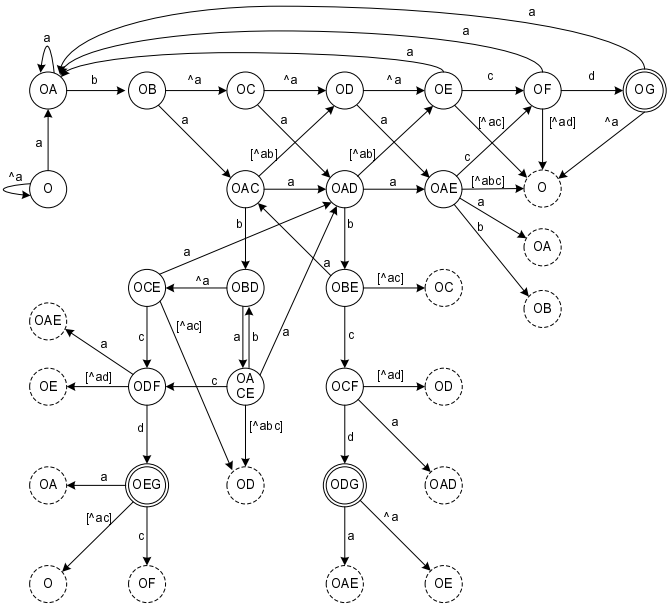
Algorithm 1 Operations of b-TFA in each time slot

Input:
  s $(s \leq b)$; ▷ no. of active states in current time slot
  $A[j](j = 1, \ldots, s)$; ▷ current active states
  c; ▷ input character
Output:
  s' $(s' \leq b)$; ▷ no. of active states in next time slot
  $A'[j]\,(j = 1, ..., s')$; ▷ active states in next time slot

$T = NULL$;
for $(j = 1, ..., s)$ **do**
  $T = T \cup$ state label on state $A[j]$ labeled with c;
end for
use T to access SST table, returning s' and $A'[j]\,(j = 1, ..., s')$



(a) NFA (8 states; 4 active states in the worst case)



(b) DFA (19 states; some less important transitions are omitted)

(c) 2-TFA (9 states; at most 2 active states)

Fig. 5. Automatons representing a single regular expression `.*ab.{3}cd`.

Appendix). Thus, no optimal solution can be found to solve it in polynomial time. We present here a heuristic algorithm to solve the b-SSP problem.

B. A Heuristic Algorithm for SSP Problem

To simplify the problem, we add another constraint (called *isolation constraint*) on the model of the b-SSP problem, which is shown in (5).

TABLE I
NOTATIONS USED IN b-SSP PROBLEM

Notations	Descriptions
Q_N	The set of all NFA states
N_D	The number of different combinations of NFA active states (i.e., the number of states in the corresponding DFA, $N_D = \|Q_D\|$)
S_i	The i-th combination of NFA active states ($i = 1, ..., N_D$)
$S_{i,j}$	The j-th subset split from S_i ($j = 1, ..., b$)
Q_T	The union of $S_{i,j}$ ($i = 1, ..., N_D; j = 1, ..., b$)

TABLE II
SET SPLIT PROBLEM

Subject to.

$$\bigcup_j S_{i,j} = S_i; (i = 1, ..., N_D; j = 1, ..., b) \quad (2)$$

$$Q_T = \{S_{i,j} \mid i = 1, ..., N_D; j = 1, ..., b\} - \{\emptyset\} \quad (3)$$

Objective.

$$\text{Minimize: } |Q_T| \quad (4)$$

$$S_{i,j} \bigcap S_{i,k} = \emptyset \quad (\forall j \neq k;\ i = 1, ..., N_D) \quad (5)$$

The isolation constraint requires that there be no overlap between the subsets split from the same NFA active state combination.

B.1 2-SSP Problem

We first consider the b-SSP problem with $b = 2$. Later we consider more general situations with $b > 2$.

Let v_i be the number of states in the i-th NFA active state combination. The number of different ways to split the combination (denoted as F_i) under the 2-SSP problem can be expressed as follows.

$$F_i = 2^{v_i - 1} \quad (6)$$

Since there are N_D different NFA active state combinations, the number of possibilities to split these state combinations is $\prod_{i=1}^{N_D} F_i$. Obviously, the problem space is too large if we go through every possibility of the split. To design a practical algorithm, we need to reduce the problem space.

Given an NFA active state combination with v states, we consider only two kinds of special splits:

(1) No split at all (i.e., one subset is empty);
(2) Splits that divide the combination into two subsets whose sizes are 1 and $v - 1$, respectively.

This way, we can reduce the value of F_i from that given in (6) to $v_i + 1$. The reason to use the second special split is that, after analyzing the NFA active state combinations of many rule sets, we find *many combinations of NFA active states differ from each other in only one NFA state*. For instance, combination {`A,B,C,D`}, {`A,B,C,E`}, {`A,B,C,F`}, and {`A,B,C,G`} differ from each other only in the last state. Splitting {`A,B,C`} out from these combinations gives us five subsets, i.e., {`A,B,C`}, {`D`}, {`E`}, {`F`}, {`G`}. It is very likely that the four single-element subsets are already (or will be) used in other splits, so the four splits produce only one distinct subset {`A,B,C`}, resulting in a high reusability of subsets.

The main data structure used in the heuristic algorithm is a tripartite graph, as shown in Figure 6. Each vertex in the left partition (called a *state-combination vertex*) corresponds to an NFA active state combination. Each vertex in the middle partition (called a *split-decision vertex*) denotes a valid split decision for one of the combinations. Each vertex in the right partition (called a *subset vertex*) corresponds to a unique subset obtained via the splits in the middle partition. Each split-decision vertex is connected with its associated state-combination vertex, as well as the subset vertices that can be obtained from the split decision.

The heuristic algorithm runs in multiple iterations to find a set of subsets (Q_T) which satisfies the constraints in the 2-SSP problem. Initially, Q_T is empty. In each iteration, the algorithm starts with the subset vertices in the right partition, from which we select the largest-degree subset (the number of connected edges) among the subsets whose sizes and degrees

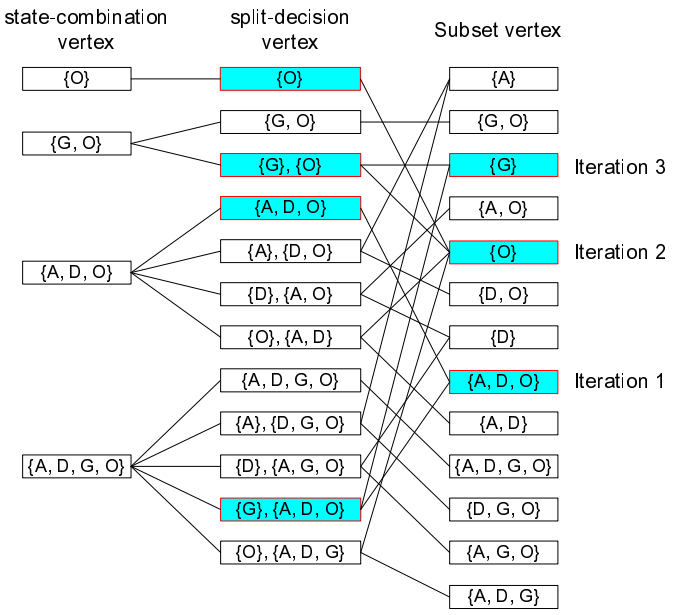


Fig. 6. Tripartite graph used in the heuristic algorithm.

are both larger than 1 [3]. If no such subset is found, we select the largest-degree subset from among all the subsets. The selected subset will be put into Q_T. The split-decision vertices connected with the selected subset vertices will also be selected and stored in the SST table. Then we remove those vertices that will not be used from the tripartite graph. We remove (1) the state-combination vertices (including their edges) connected with the selected split-decision vertices; (2) the split-decision vertices (including their left and right edges) previously connected with the just-removed state-combination vertices; (3) the subset vertices whose degrees are zero. Now one iteration is finished. The iteration repeats until all subset vertices are removed or selected.

Consider the example in Figure 6. In the first iteration, we select subset vertex $\{A, D, O\}$, and split-decision vertices $\{A, D, O\}$ and $\{G\}$, $\{A, D, O\}$. In the second iteration, we select the subset vertex $\{O\}$, and split-decision vertices $\{O\}$ and $\{G\}$, $\{O\}$. In the third iteration, we select the subset vertex $\{G\}$. After running the heuristic algorithm, four NFA active combinations are split into three TFA states.

B.2 b-SSP Problem

Given an arbitrary algorithm solving the 2-SSP problem, we can easily expand it to support the b-SSP problem if b is equal to the power of 2. We just need to run the algorithm of the 2-SSP problem recursively for $\log_2 b$ times, each time using the output of the previous run as the input.

To solve the b-SSP problem when b is an arbitrary integer, we can run the proposed heuristic algorithm for $b-1$ times, each time using the output of the previous run as the input.

[3]The reason that we first consider subsets with sizes larger than 1 is that the second special split tends to generate many large-degree single-element subset candidates (such as $\{O\}$ in the figure). Selecting these subsets in the very beginning will cause many unwanted splits.

VI. State Encoding

A challenging issue involved in the TFA implementation is the storage of state labels, since different state labels include different numbers of NFA state IDs. A simple scheme is to implement each state label as an array, including all associated NFA state IDs. However, this scheme causes two problems: (1) high storage cost and (2) TFA operation overhead.

Let us recall the operations of a TFA in one time slot (Algorithm 1). After examining the outgoing transitions of current active states, the TFA returns up to b state labels, each containing a set of NFA state IDs. A union operation is required on these ID sets, and the result is sent to the SST table to search for the next active states. To achieve a constant lookup performance, we implement the SST table as a perfect hash table as proposed in [20] and [21]. However, the perfect hashing implementation requires the set union operation to return a deterministic and unique representation (i.e., hash key) for each valid combination of NFA active states. If we were to implement each state label as an array, two complicated operations would be required after the set union operation:

- Redundancy elimination. Consider two state labels $\{D, O\}$ and $\{G, O\}$. To get their union, we have to identify and remove one redundant state "O".
- Sorting. The unions of different state labels could result in different representations for the same NFA active state combination (for example, $\{O, D, G\}$ and $\{D, G, O\}$ are logically the same). We have to sort state IDs in the result set before performing the table lookup.

To overcome the problems above, we propose an efficient state encoding scheme.

A. State Encoding Problem

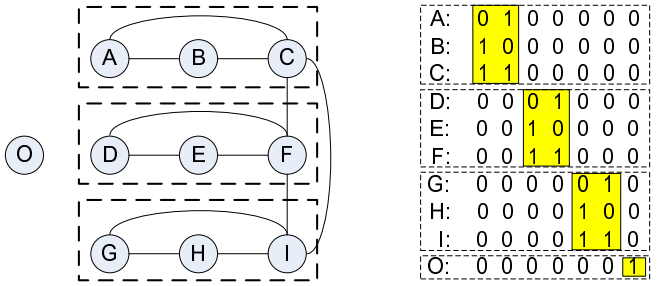
The main idea of state encoding is to assign a bit vector to each NFA state (as its ID), so that the union operation on multiple NFA states can be replaced by a simple bitwise OR operation. As a result, the redundancy elimination and sorting operations are no longer needed to get a deterministic representation for each NFA active state combination. Furthermore, with state encoding, each state label in the TFA structure no longer needs to store the IDs of all associated NFA states (refer to Figure 3(a)). Instead, only the result of the bitwise OR operation on these NFA state IDs needs to be stored. Therefore, all state labels have the same length in bits.

In order to operate the TFA correctly, only one constraint needs to be satisfied in the state encoding; that is, *the bit vector associated with each valid combination of NFA active states (i.e., each DFA state) must be unique*. We call it *uniqueness constraint*.

The state encoding problem (SEP) can be formally described as follows: *find a way to assign each NFA state a bit vector, so that (1) the uniqueness constraint is satisfied; (2) the number of bits used in the bit vector is minimized.*

B. State Encoding Algorithm

The state encoding problem is an NP-hard problem. Here we propose a practical algorithm to solve it.



(a) An independent graph corresponding to the NFA in Figure 1 (a).

(b) State encoding map based on cliques in (a).

Fig. 7. Independent graph and state encoding map.

Given an NFA, we first construct an independent graph, where each node corresponds to one NFA state. Two nodes in the independent graph are connected by an edge, if and only if their associated NFA states are never active together (i.e., never together in one NFA active state combination). An example of the independent graph is shown in Figure 7(a), which corresponds to the NFA in Figure 1(a). We then divide nodes of the independent graph into a small number of maximal cliques, and denote these cliques by C_j $(j = 1, ..., n)$, where n is the number of cliques. Let m_j be the size of clique C_j. The state encoding scheme requires $\sum_{j=1}^{n} \lceil \log_2(m_j + 1) \rceil$ bits in total for the encoding. Nodes in clique C_1 are encoded contiguously using bit position No.1 to No. $\lceil \log_2 (m_1 + 1) \rceil$, with other bit positions left as "0". Nodes in clique C_k $(k > 1)$ are encoded using bit position No. $\sum_{j=1}^{k-1} \lceil \log_2 (m_j + 1) \rceil + 1$ to No. $\sum_{j=1}^{k} \lceil \log_2 (m_j + 1) \rceil$, with other bit positions left as "0". Consider the independent graph in Figure 7(a), from which we can get four cliques. The first clique includes "A", "B", and "C", which are encoded consecutively using the first two bit positions (as shown in the state encoding map in Figure 7(b)).

It is easy to prove that with the state encoding scheme, each valid combination of NFA active states will have a unique bit vector, because the NFA states in each valid combination are always from different cliques and use different bit positions.

VII. Performance Evaluation

A. Evaluation Settings

The evaluation is made based on regular expression rule sets from Snort and Bro. Each rule in Snort includes header pattern, exact strings, and regular expressions. Since header pattern matching and exact string matching have been well-studied and are not the focus of this paper, we consider only regular expressions in the evaluation process. We cluster regular expressions with common header patterns together and evaluate them using traffic traces collected from our campus network at different time points. The regular expression rule sets selected include all major characteristics appearing in Snort and Bro rules, such as infinite repetitions, finite repetitions, and character ranges. Table III gives the statistics of these regular expression sets.

TABLE III
Basic statistics of regular expression rule sets

Rule set	# of RegEx	Avg. ASCII length of RegEx	% RegEx having infinite repetitions, e.g., "*", "+"	% RegEx having finite repetitions, e.g., {n,m},"?"	% RegEx having char ranges, e.g., \s,".", [a-z]
Snort 1	17	38.6	88.2%	41.2%	94.1%
Snort 2	12	16.7	16.7%	83.3%	100.0%
Snort 3	1	19.0	100.0%	100.0%	100.0%
Snort 4	34	29.7	32.4%	58.8%	82.4%
Bro	63	14.6	11.1%	12.7%	23.8%

We compare the TFA against the NFA, DFA and two other state-of-the-art DFA-based solutions: DFA grouping [5] and Hybrid-FA [8]. The latter two schemes are selected because they have a similar motivation to ours, i.e., to trade off the storage with the number of active states.

For a given regular expression set, the DFA grouping scheme [5] minimizes the number of DFAs subject to either of the two storage constraints: (1) the size of each DFA is limited and (2) the total size of DFAs is limited. Since our objective is different, which is to minimize the storage with a bounded number of active states, we need to modify the DFA grouping scheme for a straight comparison. A bound factor b implies that we can use b DFAs in the DFA grouping scheme. The modified DFA grouping scheme still uses the *interaction* concept proposed in [5]. It first creates b empty groups, and then recursively inserts each regular expression into the group that has the least interactions with the regular expression undergoing. Hybrid-FA is another effective solution to achieve a tradeoff between storage complexity and run-time complexity. To get a better worst-case run-time performance, we generate Hybrid-FA with tail-DFAs. To make fair comparisons, we use states rather than counters to implement the finite repetitions in all automatons in the evaluation.

B. Storage Complexity

The storage of a TFA consists of two parts: a TFA structure and an SST table.

B.1 Storage of SST Table

The storage of the SST table is a fixed expense for a TFA. Consider that (1) the number of entries in the SST table is equal to the DFA state number, and (2) each entry in the SST table stores the addresses of b states. The SST table for a b-TFA requires $b/256$ times the memory required by the corresponding DFA. Normally, a practical TFA takes b between 2 and 4, so the storage cost of the SST table is about $0.78\% \sim 1.5\%$ that of the DFA. Perfect hashing implementation of the SST table provides $O(1)$ run-time performance but requires extra entries to facilitate the table construction. In this paper, we use cuckoo hashing [20] to build the SST table. In our experiment, an SST table with millions of entries can be constructed with a load factor of 0.7 in 5 minutes. The storage cost of the SST table including the extra entries for perfect hashing implementation is about $1.11\% \sim 2.14\%$ of that required by a DFA.

B.2 Storage of TFA Structure

TABLE IV
STATE NUMBERS OF AUTOMATONS UNDER DIFFERENT REGULAR EXPRESSION SETS

Rule set	NFA	DFA	Hybrid-FA	DFA Grouping			TFA					
				b=2	b=3	b=4	b=2	% reduction	b=3	% reduction	b=4	% reduction
Snort 1	385	369870	7167	63637	10166	1857	6563	98.23%	1461	99.60%	871	99.76%
Snort 2	191	100697	7955	16851	8396	3816	5986	94.06%	805	99.20%	284	99.72%
Snort 3	38	363154	63443	363154	363154	363154	62133	82.89%	22418	93.83%	14668	95.96%
Snort 4	417	158060	11714	11459	6181	2434	10846	93.14%	3005	98.10%	1597	98.99%
Bro	567	407677	22305	148665	6890	2316	16576	95.93%	3685	99.10%	2000	99.51%

TABLE V
STATISTICS OF STATE ENCODINGS IN THE TFA

rule set	# of bits in state encoding	b=2		b=3		b=4	
		# of state lables	% of distinct state lables	# of state lables	% of distinct state lables	# of state lables	% of distinct state lables
Snort 1	40	1680128	0.87%	374016	0.98%	222976	0.88%
Snort 2	28	1532416	0.52%	206080	0.56%	72704	0.68%
Snort 3	38	15906048	0.48%	5739008	0.42%	3755008	0.46%
Snort 4	61	2776576	0.69%	769280	0.60%	408832	0.63%
Bro	55	4243456	0.64%	943360	0.77%	512000	0.90%

TABLE VI
MEMORY COSTS OF AUTOMATONS

rule set	DFA (MB)	Hybrid-FA (MB)	DFA Grouping (b=2) (MB)	TFA (b=2)	
				MB	% reduction
Snort 1	379	7.34	65.16	9.68	97.44%
Snort 2	103	8.15	17.26	6.94	93.27%
Snort 3	372	64.97	371.87	66.53	82.11%
Snort 4	162	12.00	11.73	9.59	94.07%
Bro	417	22.84	152.23	20.24	95.15%

The memory cost of a TFA structure depends on two factors: (1) TFA state number and (2) the number of bits used in state encoding.

Table IV compares the state numbers of finite automatons under different rule sets. We can see that with only two active states, a TFA can significantly reduce the number of states required by a DFA (the reduction rates are more than 93% under all tested rule sets except "Snort 3", which will be discussed shortly.) The reduction rates are above 98% when three active states are used. Allowing more active states leads to an even higher reduction. The DFA grouping scheme is inferior to the TFA scheme when b is small. When b is large enough, the difference between them becomes smaller. "Snort 3" is a single-rule rule set including nested character ranges and finite repetitions. From Table IV we can see that the DFA grouping scheme cannot handle those cases with single complex regular expression, whereas the TFA scheme is still in effect and can achieve a 96% reduction when $b = 4$.

To make a straight comparison, the head-DFA and tail-DFAs of the Hybrid-FA are adjusted so that their total state number is close to the state number of the TFA with $b = 2$. (A smaller Hybrid-FA can always be achieved by moving more states from the head-DFA to tail-DFAs, a process, however, that results in an even worse run-time performance.) Later we will compare the run-time performance between the Hybrid-FA and the TFA under this condition.

Table V shows the number of bits required by state encoding of the TFA under different rule sets. It also shows the number of state labels and the fraction of distinct state labels under each tested rule set. It can be seen that the fraction of unique state labels is very low (lower than 1% under all tested rule sets). This characteristic can be potentially exploited to further compress the memory usage.

Table VI compares the overall memory usages of different finite automatons. When calculating the memory usage of the TFA, we consider the cost of both the SST table and TFA structure. The TFA scheme with $b = 2$ can achieve a memory reduction between 93.27% and 97.44% in all tested rule sets, except the single-rule set "Snort 3", under which a reduction rate of 82.11% is achieved.

C. Memory Bandwidth Requirement

The memory bandwidth requirement (or the run-time speed) of an automaton can be expressed by the number of states which are activated during the processing of a character. Unlike the NFA and Hybrid-FA, a TFA can have the number of active states strictly bounded by the bound factor b; therefore it has a deterministic matching speed independent of the regular expression rule sets and traffic patterns.

Table VII shows the numbers of active states of the NFA, Hybrid-FA and TFA. The third and sixth columns give the theoretically worst case scenarios for the NFA and Hybrid-FA respectively. Note that when a tail-DFA of the Hybrid-FA includes a finite repetition, it might be triggered multiple times, resulting in a large number of active states in the tail-DFA [8]. It is discussed in [8] that with the use of a counter, the number of active states in such a tail-DFA can be limited to 2. In Table VII, we use a conservative way to calculate the worst-case active state number for the Hybrid-FA. We count only one active state for each tail-DFA. So the number of active state in the Hybrid-FA in the worst case is equal to the number of tail-DFAs plus one. The second and fourth columns of Table VII give the maximal numbers of active states we observed in the inspection of two million packets using the NFA and Hybrid-FA.

VIII. CONCLUSION

Given the fact that the rule sets of many NIDSes (such as Snort and Bro) are available online, cyber criminals can

TABLE VII
NUMBER OF ACTIVE STATES

rule set	NFA		Hybrid-FA			TFA (b=2)
	Max	Worst case	Max	# tail-FA	Worst case	Worst case
Snort 1	10	16	3	12	13+	2
Snort 2	3	8	3	5	6+	2
Snort 3	3	33	3	1	2+	2
Snort 4	8	18	5	15	16+	2
Bro	21	25	10	22	23+	2

easily construct a worst-case traffic pattern after carefully studying the rule sets for ways to slow down the NIDS and let malicious traffic escape from the inspection. Therefore, the worst-case run-time performance is critical to an NIDS if it is to survive under the network attack. In this paper, we have proposed a new finite automaton representation, TFA, for regular expression matching. It combines the strengths of both NFAs and DFAs by allowing a small bounded number of active states during the matching processing, so that the worst-case performance of a TFA is guaranteed. The construction of the TFA relies on an efficient solution to the Set Split Problem (SSP). Although an efficient heuristic algorithm has been proposed for the SSP problem in this paper, we believe that better solutions for the problem are still available, which may further bring down the storage cost of a TFA. Moreover, other related ideas, such as using counters to represent finite repetitions in regular expressions, can also be applied to the TFA to achieve even more compact finite automatons.

REFERENCES

[1] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Dept. of Computer Science, University of Arizona, Tech. Rep., 1994.

[2] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search." *Commun. of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[3] S. Dharmapurikar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems." *IEEE J SEL AREA COMM*, vol. 24, no. 10, pp. 1781–1792, 2006.

[4] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection." in *Proc. of IEEE INFOCOM*, 2004.

[5] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. of ACM/IEEE ANCS*, 2006.

[6] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. of ACM SIGCOMM*, 2007.

[7] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *IEEE Symposium on Security and Privacy*, 2008.

[8] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. of ACM CoNEXT*, 2007.

[9] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2003.

[10] A free lightweight network intrusion detection system for UNIX and Windows. [Online]. Available: http://www.snort.org

[11] Bro Intrusion Detection System. [Online]. Available: http://www.bro-ids.org

[12] Cisco IPS Deployment Guide. [Online]. Available: http://www.cisco.com

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation - international edition (2. ed).* Addison-Wesley, 2003.

[14] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. of ACM/IEEE ANCS*, 2006.

[15] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. of ACM/IEEE ANCS*, 2007.

[16] J. Jiang, Y. Xu, T. Pan, Y. Tang, and B. Liu, "Pattern-based dfa for memory-efficient and scalable multiple regular expression matching," in *Proc. of IEEE ICC*, may 2010, pp. 1 –5.

[17] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *Proc. of ACM SIGCOMM*, 2008.

[18] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proc. of IEEE INFOCOM*, may 2007, pp. 1064 –1072.

[19] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. of ACM/IEEE ANCS*, 2007.

[20] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Journal of Algorithms*, 2001, p. 2004.

[21] F. J. Enbody and H. C. Du, "Dynamic hashing schemes," *ACM Computing Surveys*, vol. 20, pp. 85–113, 1988.

APPENDIX

A. Complexity of b-SSP Problem

We have proved that the b-SSP problem is an NP-hard problem for any $b > 1$. Due to space limitations, we briefly describe the reductions that prove b-SSP problem is in NP-hard. The reductions are done to the decision problem of b-SSP called b-SSPD, and prove that b-SSPD is in NP-complete. Given a threshold r of the objective (i.e., $|Q_T|$), b-SSPD asks if $|Q_T|$ can be less than r. We only provide the construction and reductions. The correctness proof is ignored due to space limitations.

First of all, b-SSPD is in NP, because any solution of b-SSPD which is the partition on each S_i can be verified whether $|Q_T| < r$ in polynomial time.

Second, we prove that b-SSPD is in NP-complete when b=2 by reducing 3SAT to 2-SSPD. For a 3SAT instance of n variables and m clauses, we create $3n + 7m + 2$ sets as follows. For any variablex_j, create three sets $\{x_j, T, F\}$, $\{\overline{x_j}, T, F\}, \{x_j, \overline{x_j}, T, F\}$. For any clause $C_k = l_{k,1} \vee l_{k,2} \vee l_{k,3}$, create seven sets: $\{u_{k,1}\}$,$\{u_{k,2}\}$,$\{u_{k,3}\}$,$\{C_k, l_{k,1}, l_{k,2}, u_{k,3}\}$, $\{C_k, l_{k,1}, l_{k,3}, u_{k,2}\}$,$\{C_k, l_{k,2}, l_{k,3}, u_{k,1}\}$, $\{C_k, l_{k,1}, l_{k,2}, l_{k,3}, T\}$. The reduction statement is following. *There is a solution (i.e., a satisfying assignment) to the 3SAT instance, if and only if there is a solution to the corresponding 2-SSPD instance with* $2 + 2n + 6m$ *subsets.*

With 2-SSPD in NP-complete, we only need to recursively reduce b-SSPD to (b-1)-SSPD and finally to 2-SSPD, proving that b-SSPD is in NP-complete for any $b > 1$. Consider a (b-1)-SSPD instance of n sets $\{S_1\}$, ... ,$\{S_n\}$ and decide whether any partition can produce no more than m distinct subsets. We create the b-SSPD instance by appending n different elements (which do not appear in any set of (b-1)-SSPD instance) to each $\{S_i\}$, creating n sets. The n elements appended to $\{S_i\}$ will be used to create $\binom{n}{2}$ 2-element sets. In sum, a b-SSPD instance of $n^2 + n\binom{n}{2}$ sets is created for (b-1)-SS instance of n sets. The reduction is following. *There is a solution to the (b-1)-SSPD instance with* m *distinct subsets if and only if there is a solution to the corresponding b-SSPD instance with* $m + n^2$ *subsets, where* $m \leq n$.