High Speed Regular Expression Matching Engine with Fast Pre-Processing

Zhe Fu[1,2], **Jun Li**[2,3]

[1] Department of Automation, Tsinghua University, Beijing 100084, China
[2] Research Institute of Information Technology, Tsinghua University, Beijing 100084, China
[3] Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing 100084, China

Abstract: Regular expression matching is playing an important role in deep inspection. The rapid development of SDN and NFV makes the network more dynamic, bringing serious challenges to traditional deep inspection matching engines. However, state-of-the-art matching methods often require a significant amount of pre-processing time and hence are not suitable for this fast updating scenario. In this paper, a novel matching engine called BFA is proposed to achieve high-speed regular expression matching with fast pre-processing. Experiments demonstrate that BFA obtains 5 to 20 times more update abilities compared to existing regular expression matching methods, and scales well on multi-core platforms.

Keywords: deep inspection; finite automaton; regular expression matching; pre-processing

I. INTRODUCTION

Deep inspection is one of the most fundamental techniques in various network functionalities, including traffic identification, intrusion detection, content-based charge, data loss prevention, etc. Nowadays, regular expressions are playing an important role in the implementation of deep inspection for their powerful expressiveness, and almost all rules are written using regular expressions. To perform matching, regular expressions are compiled into either NFA (Nondeterministic Finite Automaton) or DFA (Deterministic Finite Automaton) [1]. Though NFA has a more compact data structure than DFA, the nondeterministic state transitions would significantly slow down the performance of NFA in the worst case. On the contrary, DFA only needs one state transition for each input character. As a result, DFA is very fast and becomes the preferred choice for deep inspection.

However, the outstanding performance of DFA comes at the price of huge memory consumption and long pre-processing time. With the rapid growth of the Internet, both of the network bandwidth and the number of deep inspection rules have been increasing dramatically in the past decades. When handling with a large number of regular expressions, the size of DFA usually explodes exponentially, leading to huge memory consumption. Worse still, the enforcement of deep inspection rules is becoming more frequent than before because the network is getting more and more dynamic along with the fast development of SDN (Software Defined Network) and NFV (Network Function Virtualization). In Cisco's email security appliance, the rules are updated every

Received: Jan. 21, 2018
Revised: May 3, 2018
Editor: Luoyi Fu

In this paper, a novel matching engine called BFA is proposed to achieve high-speed regular expression matching with fast pre-processing.

3 to 5 minutes to provide an up-to-date threat defence capability [2]. In [3], the authors also demonstrate that the update performance is one of the most crucial challenges for regular expression matching because the deep inspection system must react quickly to the attack characteristics that are changed rapidly. The significant amount of pre-processing time makes DFA and DFA based matching engines hard to meet the pace of rule update in practice.

In this paper, we propose BFA (Bit-based Finite Automaton), an update-friendly regular expression matching engine for deep inspection. Instead of conventional state transitions in NFA or DFA, BFA executes Boolean matrix multiplications to perform regular expression matching. In addition, some optimization methods are put forward for BFA to further accelerate the matching speed and reduce the memory consumption. We also propose a new criterion to measure the update abilities of different regular expression matching engines. Evaluations demonstrate that BFA achieves fast matching speed in a very short period of pre-processing time, about 5 to 20 times improvement on average compared to state-of-the-art matching methods. BFA also scales well and could get 3 to 4 times further improvement on a commodity multi-core platform.

The rest of the paper is organized as follows. In Section II, we introduce the background of regular expression matching. In Section III, the design and implementation of BFA are proposed. Further optimizations are described in Section IV. In Section V, several experiments are conducted to evaluate the performance of BFA and other matching engines. A conclusion is drawn in Section VI.

II. Background

Deep inspection, as its name indicates, is to inspect the packet deeply, analyse the payload and make the appropriate decisions according to the inspection rules. Deep inspection is widely used in network security (anti-virus, spam email filtering, etc.), bandwidth management (QoS guarantees, bandwidth pricing, etc.), user profiling (differentiation charge, ad injection, etc.) and so forth. The key technology of deep inspection is pattern matching, which scans the payload of network packets against a set of predefined rules.

In early days, the deep inspection rules were written in exact strings. Many string matching algorithms have been proposed, including Knuth-Morris-Pratt (KMP) [4], Boyer-Moore (BM) [5] for single pattern matching and Aho-Corasick (AC) [6], Wu-Manber (WM) [7] for multiple pattern matching. Over time, the patterns are getting more and more complicated, therefore exact strings cannot conveniently characterize the patterns for deep inspection anymore. Regular expression was first proposed in [8], and has become the first choice to describe the patterns for deep inspection due to its powerful ability of expression. The famous open-source network intrusion detection system Snort [9] has already employed more than 27,000 signatures defined by regular expressions.

NFA and DFA are two of the most popular methods to perform regular expression matching, and they are both 5-tuples $\{Q, \Sigma, \delta, q_0, q_f\}$. Q and Σ represent a finite set of states and a finite set of input characters respectively. q_0 denotes the start states, while q_f denotes the accept states. The only difference is the transition function δ. In DFA, δ only takes one state and returns a single state for an input symbol, while in NFA, δ may return empty, one state or multiple states. The non-deterministic transitions slow down the speed of NFA seriously in worst cases and restrict its applications in deep inspection. Therefore, almost all matching engines rely on DFA to match regular expressions for high-speed deep inspection.

However, the high matching speed of DFA brings the following two drawbacks. First, with the increase of number and complexity of regular expressions, the size of corresponding DFA expands exponentially, which is called *state explosion* in DFA. Second, the construction of DFA usually takes a significant amount of time since the construction from an NFA to an equivalent DFA needs to explore all the possible active states in the NFA through the subset construction algorithm [10].

Until now, many of the research efforts are focusing only on reducing the memory consumption of DFA. D^{2}FA [11] omits the equivalent transitions among different states and adds a default transition in order to compress the redundant state transitions. Hybrid-FA [12] constructs a hybrid structure with a head DFA followed by multiple tail NFAs. OD^{2}FA [13] merges the DFA states deriving from the same NFA state into a super state to further decrease the number of DFA states. TFA [14] puts forward tunable finite automata to achieve a better balance between memory bandwidth and space. All these researches have to further analyse the structure of the built DFA, thus they need more pre-processing time and cannot handle large or complex regular expression rulesets.

In [15-16], the authors proposed regular expression grouping methods that group the rules into several subsets and construct the DFA for each group individually to avoid state explosion when handling with a large scale of regular expressions. The shortcoming of this kind of work is that in deep inspection, a stream have to be matched with all of the DFAs to obtain the final matching result. This will decrease the matching performance linearly as the number of groups increasing. What's more, the grouping methods cannot solve the DFA state explosion caused by single regular expression.

Hardware platforms are also widely used to accelerate regular expression matching. FPGA based methods [17-18] have advantages in pipeline and parallelism; however, the small size of on-chip memories limits the practical deployment of large-scale rulesets. Besides, the synthesis, implementation, and place and route procedures of FPGA design are also quite time-consuming. GPU based methods [19-20] make full use of the high memory bandwidth and massive execution units, but GPU memory also limits the size and complexity of regular expression rulesets.

There are also some proposals that parallelize regular expression matching on multicore platform. Speculation based methods [21] need to guess the start DFA state of each data block, but the frequent rematch will degrade the matching performance. Enumeration based methods [22] compute all the state transitions from every possible DFA state, whose computation overhead is quite huge when DFA is large. ParaRegex [23] takes advantage of the *state aggregation* phenomenon during DFA traversals and reduces the overhead of states enumeration. Nevertheless, this type of work also relies on the construction of DFA.

None of the previous work described above addresses the problem of regular expression pre-processing in deep inspection. Most studies require even more pre-processing time compared to the original NFA or DFA methods. As the network is becoming more dynamic and flexible, so does the need for a high-speed regular matching engine with fast pre-processing.

III. Design and Implementation

3.1 Overview

NFA has two advantages that are suitable for the frequently updating scenarios: fast pre-processing time and relatively low memory consumption. However, the poor matching performance of NFA, which is attributed to uncertain active states and memory accesses, severely limits its applications in deep inspection. To

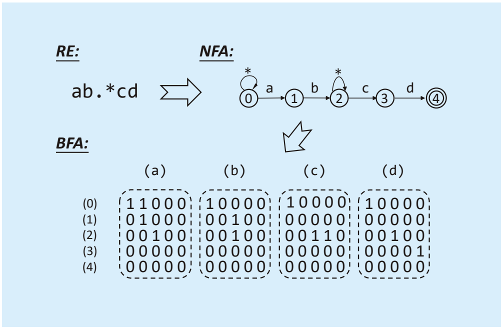


Fig. 1. *Example of BFA of regular expression "ab.*cd"*

overcome the shortcomings of NFA, BFA is designed to accelerate the state transitions of NFA. To implement BFA, firstly an NFA is generated from the regular expression ruleset via the Thompson's construction algorithm [1]. Then each state and each transition in the NFA are encoded into a bit vector. For every input character, the state traversal procedure turns into a Boolean multiplication between a vector and a matrix, which could be speeded up by CPU instruction level optimizations. Besides, due to the sparsity in bit vectors and bit matrices, several bitmap compression methods can be leveraged to efficiently reduce the memory consumption of BFA. Next, more details about BFA design and implementation will be given.

3.2 BFA construction

BFA extends the 5-tuple definition of NFA and DFA, by introducing bit vectors to encode the states in automata. Formally, a BFA is defined as a 5-tuple $\{Q, \Sigma, \mathcal{B}, \boldsymbol{v_0}, \boldsymbol{v_f}\}$. The first two terms are the same as those in traditional NFA. The transition table $\mathcal{B}$ is defined as a function to calculate the currently active states in BFA. Finally, the bit vector $\boldsymbol{v_0}$ denotes the set of initial states and $\boldsymbol{v_f}$ denotes a set of accept states.

Algorithm 1. BFA construction.

```
    Input: NFA N ={$Q$, $\Delta$, $\Delta$, $q_0$, $q_f$}
    Output: BFA B ={$Q$, $\Sigma$, $\mathcal{B}$, $\boldsymbol{v}_0$, $\boldsymbol{v}_f$}
1   // transition table initialization
2   $\mathcal{B} \leftarrow 0$
3   foreach character $k \in \Sigma$ do
4       foreach $i, j \in$ state $Q$ do
5           if state $i$ and $j$ has a transition with label $k$ then
6               $\mathcal{B}(i,k)[j] \leftarrow 1$
7           end
8       end
9   end
10  // start state vector initialization
11  $\boldsymbol{v}_0 \leftarrow 0$
12  foreach state $j \in$ state $q_0$ do
13      $\boldsymbol{v}_0[j] \leftarrow 1$
14  end
15  // accept state vector initialization
16  $\boldsymbol{v}_f \leftarrow 0$
17  foreach state $j \in$ state $q_f$ do
18      $\boldsymbol{v}_f[j] \leftarrow 1$
19  end
```

Given a previously built NFA, the corresponding BFA is constructed according to Algorithm 1. Assuming that there are N states in an NFA and the size of the alphabet is $|\Sigma|$, the transition table $\mathcal{B}$ of BFA is made up of $N \times |\Sigma|$ bit vectors with the length of N. Each bit in the vector indicates the status of the corresponding state, *i.e.*, whether this state is active or not. More precisely, in $\mathcal{B}(i,k)$, a $1 \times N$ bit vector which stands for the state i and input character k, if the j^{th} bit of this vector is 1, it denotes that if the input character is k and meanwhile the i^{th} state of an NFA is active, then state j will be activated.

Figure 1 shows an intuitive example of the BFA constructed from a simple regular expression rule “ab.*cd”. In this instance, the corresponding NFA has 5 states, and the size of the alphabet is 4, so there are $5 \times 4 = 20$ bit vectors in the transition table $\mathcal{B}$, and each vector has 5 bits. Taking $\mathcal{B}(2,c)$ (*i.e.*, [0 0 1 1 0]) as an example, if the input character is c, and meanwhile state 2 is active, then state 2 and 3 will be activated after reading character c. Following the steps in Algorithm 1, it is also easy to obtain that the start state vector $\boldsymbol{v_0}$= [1 0 0 0 0] and the accept state vector $\boldsymbol{v_f}$= [0 0 0 0 1] for this example.

The procedure of BFA construction only requires a one-time full traversal of the NFA that is previously built, so the pre-processing procedure is much easier and faster than DFA and other DFA based methods.

3.3 State transitions in BFA

After constructing BFA in the form of bits, the state transitions could be further transformed into a Boolean Matrix Multiplication (BMM) between a bit vector and a bit matrix. Additionally, a bit vector $\boldsymbol{v}_i$ with the length of N is used to record the active states after reading i input characters. If the next input character is k, then the k^{th} column of the transition table $\mathcal{B}$ is picked out, denoted as $\mathcal{B}(:,k)$. $\mathcal{B}(:,k)$ is an $N \times N$ bit matrix, which will be multiplied by the $1 \times N$ bit vector $\boldsymbol{v}_i$. We iteratively compute state transitions in BFA as below:

$$\boldsymbol{v}_{i+1}^{(1\times N)}[j] = \bigvee_m (\boldsymbol{v}_i^{(1\times N)}[m] \wedge \mathcal{B}(:,k)^{(N\times N)}[m,j]) \quad (1)$$

In formula (1), $m \in [1,N]$. $\boldsymbol{v}_i[m]$ denotes the mth bit in the vector $\boldsymbol{v}_i$, and $\mathcal{B}(:,k)[m,j]$ denotes the mth row and jth column of the bit matrix $\mathcal{B}(:,k)$. In figure 1, suppose $\boldsymbol{v}_i$= [1 0 1 0 0], which means the first and third state are active. Assuming that the input character is c, the state transitions in BFA are obtained by formula (2). The result $\boldsymbol{v}_{i+1}$= [1 0 1 1 0] indicates that after reading c, the first state, the third state, and the fourth state will be activated.

$$\begin{bmatrix}1 & 0 & 1 & 0 & 0\end{bmatrix} \times \begin{bmatrix}1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 1 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\end{bmatrix} \quad (2)$$
$$= \begin{bmatrix}1 & 0 & 1 & 1 & 0\end{bmatrix}$$

To determine whether a match happens or not, $\boldsymbol{v}_i$ is intersected by the accept state vector $\boldsymbol{v}_f$, *i.e.* $\boldsymbol{v}_i \wedge \boldsymbol{v}_f$. In the example of figure 1, the intersection of $\boldsymbol{v}_{i+1}$ and $\boldsymbol{v}_f$ equals [0 0 0 0 0]. The all-zero result means no match is found at this stage. Otherwise, the non-zero result indicates at least one regular expression is matched.

3.4 Multi-core scaling

The data-parallelization of NFA or DFA on multi-core platform is extremely hard because of the strong dependencies and unpredictable memory accesses during state transitions. In recent years, several methods have been proposed to parallelize DFA or DFA based methods and try to decrease the overhead as much as possible [21-23]. However, all these approaches have to build the DFA first, which needs plenty of pre-processing time and hence not suitable for the frequently update of rule-sets in deep inspection.

In BFA, the state transitions are actually a series of BMM between bit vectors and bit matrices, which can be executed on different cores separately. Algorithm 2 presents how to execute BFA in parallel with n cores. Scaling BFA on multi cores does not require extra pre-processing time, but would add some additional computational overhead since the multiplication of two bit matrices is more complex than that between a bit vector and a bit matrix. After all cores complete the bit matrix multiplication, the results of each core will be joined with the start state vector $\boldsymbol{v}_0$ in order to get the final state vector $\boldsymbol{v}_n$. Line 9 to Line 12 in Algorithm 2 illustrate the related steps.

IV. Optimizations

Although BMM is simpler than traditional matrix multiplication [24], there is still a great potential to accelerate it using instruction level optimizations. The multiplication between the bit vector $\boldsymbol{v}_i$ and the bit matrix $\mathcal{B}(:,k)$ can be simplified as bitwise *OR* operations between $\boldsymbol{v}_i$ and multiple bit vectors from $\mathcal{B}(:,k)$. Besides, the process to decide whether the currently active states are accept states or not is actually a bitwise *AND* operation between $\boldsymbol{v}_i$ and the accept state vector $\boldsymbol{v}_f$.

Algorithm 3 shows the procedure of optimized state transitions in BFA. In Line 1, the function *FindBitPosition*() returns the positions of 1 in the bit vector . A brute force way is to check whether each bit is 1 or 0 from the

Algorithm 2. Parallelize bfa on multi-core platform.

```
   Input: BFA B = {$Q$, $\Sigma$, $\mathcal{B}$, $\boldsymbol{v}_0$, $\boldsymbol{v}_f$}, input data $C = c_0c_{11}c_{12}\ldots c_{1m1}\ c_{21}c_{22}\ldots c_{2m2}\ \ldots$
          $c_{n1}c_{n2}\ldots c_{nmn}$, number of threads $n$
   Output: bit vector $\boldsymbol{v}_n$ that $\boldsymbol{v}_0 \xrightarrow[C]{\mathcal{B}} \boldsymbol{v}_n$
1  // perform optimized BMM on each core
2  foreach $i \in [1\ldots n]$ parallel do
3      $M_i \leftarrow I_{N\times N}$ // initialize $M_i$ as an identity matrix
4      for $k = 0 \rightarrow m_i$ do
5          $M_i \rightarrow M_i \times \mathcal{B}(:,c_{ik})$
6      end
7  end
8  // get the final $\boldsymbol{v}_n$
9  $\boldsymbol{v}_n \leftarrow \boldsymbol{v}_0 \times \mathcal{B}(:,c_0)$
10 for $i = 1 \rightarrow n$ do
11     $\boldsymbol{v}_n \leftarrow \boldsymbol{v}_n \times M_i$
12 end
```

first to the last, whose complexity is $O(W)$ if the length of the bit vector is W. In our optimization, we leverage the built-in instructions provided by recent popular processers to reduce the complexity of the *FindBitPosition*() function. *LZCNT*, which is first introduced in Intel Haswell microarchitecture, provides the capability to count the number of leading zero bits in a bit vector. Similarly, *POPCNT*, which is first introduced in Intel Nehalem microarchitecture, can count the number of bits set to 1 in a bit vector. These instructions have a high throughput that is up to one operation per CPU cycle. In BFA, we combine the *LZCNT* and *POPCNT* instructions with the shift operation to accelerate the execution of *FindBitPosition*() function. The union of multiple bit vectors (Line 3 to Line 5) is also parallelized by CPU *SIMD* operations. The BMM operation of BFA on multi cores (Line 4 to Line 6 in Algorithm 2) is speeded up by similar optimizations as shown in Algorithm 3.

In addition, because of the low density of bits that are set to 1 in the transition table $\mathcal{B}$, many bitmap compression techniques can be used to efficiently compress the memory consumption of BFA. What is more, several latest bitmap methods (including Roaring Bitmaps [25], WAH [26], *etc*.) have already provided the capabilities of direct bitwise operations over bit vectors in compressed format, even without decompression in advance. These operations are actually faster than those over uncompressed bitmaps. In BFA, the bit vectors in the transition table $\mathcal{B}$ are compressed by Roaring Bitmaps. These optimizations, together with the instruction level acceleration, decrease both computation overhead and memory usage of BFA.

Algorithm 3. Optimized state transitions in BFA.

```
Input: bit vector $\boldsymbol{v}_i$, input character $k$, BFA B = $\{Q, \Sigma, \mathcal{B}, \boldsymbol{v}_0, \boldsymbol{v}_f\}$
Output: bit vector $\boldsymbol{v}_{i+1}$
1 $S \leftarrow FindBitPosition(\boldsymbol{v}_i)$
2 $\boldsymbol{v}_{i+1} \leftarrow 0$
3 for $pos \in S$ do
4     $\boldsymbol{v}_{i+1} \leftarrow \boldsymbol{v}_{i+1} \vee \mathcal{B}(pos,k)$
5 end
```

Table I. *Rulesets information.*

name	*snort1*	*snort2*	*bro*	*tcp*	*dotstar*	*range*
number of REs	24	34	217	733	300	300
complexity	low	low	moderate	high	high	high

V. Evaluations

To evaluate the performance of BFA, we conducted a series of experiments on an HP Z228 workstation with Intel Core i7-4790 Processor (BMI1 and SSE4 supported) and 8GB memory. NFA, DFA, mDFA [15], Hybrid-FA [12], and D^2FA [11] are implemented based on Regular Expression Processor [27] as comparisons.

Six rulesets are tested in the experiments, as shown in Table I. Four rulesets are from the real world: two small rulesets (*snort1* and *snort2*) from Snort, one ruleset (*bro*) from Bro [28], and one large ruleset (*tcp*) from the Regular Expression Processor. Two synthetic rulesets are also used in the experiments. The *dotstar* ruleset consists of 300 regular expressions with ".*" characters, and the *range* ruleset consists of 300 regular expressions with range semantics. These synthetic rulesets are complex and challenging for deep inspection matching engines.

5.1 Total Running Time

First, we capture network traffic from our campus network and dump an 8MB PCAP file treated as the input data for our experiments. We measure the compiling time of different engines and the matching time for the input data on all the rulesets in Table I.

Pre-processing time

In Table II, we can see that the pre-processing procedure of NFA is the fastest among all these matching engines, since the data structure construction of all other previous work depends on a pre-built NFA and hence require more pre-processing time. DFA requires much more time than NFA because the subset construction algorithm is very time-consuming, especially when regular expressions ruleset is large or complex. Table II shows that DFA consumes 100 times more pre-processing time to build its data structure compared to NFA. Hybrid-FA is between NFA and DFA since it builds a hybrid structure composed of a head DFA and several tail NFAs. D^2FA requires even more pre-processing time than DFA as it is designed to reduce the memory usage of DFA by analysing the redundant state transitions in DFA and compressing them into default states. For large or complex rulesets (*tcp*, *dotstar* and *range*), DFA, Hybrid-FA and D^2FA all fail to generate their data structures because the memory consumption exceeds the limit of our platform after hours' compiling.

To handle with these large rulesets, we use regular expression grouping (mDFA) methods to group the rules into several subsets so that the finite automata could be constructed on our experimental platform. We implement the grouping method in [15] and generate as fewer regular expression subsets as possible in order to reduce the matching performance degradation caused by multiple DFAs. For rulesets *tcp*, *dotstar* and *range*, the number of subsets is 13, 11 and 2 respectively. As can be seen in Table II, the grouping method requires enormous pre-processing time, which can be ascribed to two parts. First, the grouping method needs to calculate the DFA state of every pair of regular expression rules to build the *convolvement relationship* matrix. Second, even though the large rulesets have been divided into smaller subsets, it is still complex and time-consuming to construct the DFA for the subsets. As a result, it takes 4 to 10 hours for mDFA to accomplish the regular expressions pre-processing.

Our method, BFA, only needs around 20% to 50% more pre-processing time than NFA since the construction of BFA is only to encode the NFA in forms of bits, and could handle all large and complex rulesets. In comparison with the remaining approaches, the pre-processing procedure of BFA is about 20 times faster than Hybrid-FA, more than 100 times faster than DFA and D^2FA and 500 to 8,000 times faster than mDFA.

Matching Time

In terms of matching speed, DFA is always the fastest since it performs only one state lookup per input character. D^2FA needs to visit the default state first and then find the correct transition, so its matching speed is slower than DFA. NFA has the worst matching speed owing to the non-deterministic memory accesses. Hybrid-FA achieves a higher matching speed than NFA, because most non-malicious traffic is filtered by the head DFA in Hybrid-FA. The matching speed of BFA is of the same order of magnitude as Hybrid-FA and D^2FA, which is about 10 to 100 times faster than NFA. For large or complex rulesets (*tcp*, *dotstar* and *range*), the matching time of mDFA is in pro-

Table II. *Pre-processing time and matching time of different engines on different rulesets (unit: second).*

rulesets		*snort1*	*snort2*	*bro*	*tcp*	*dotstar*	*range*
NFA	pre-processing	0.127	0.153	0.320	52.244	3.052	1.827
	matching	16.897	16.95	122.406	239.075	148.933	71.589
DFA	pre-processing	18.535	26.357	38.985	\	\	\
	matching	0.072	0.074	0.109	\	\	\
mDFA	pre-processing	\	\	\	34,649.657	15,859.416	20,965.463
	matching	\	\	\	1.577	1.194	0.361
Hybrid-FA	pre-processing	2.494	3.318	18.024	\	\	\
	matching	0.699	0.849	2.044	\	\	\
D^2FA	pre-processing	20.398	30.247	39.917	\	\	\
	matching	1.163	1.172	0.956	\	\	\
BFA	pre-processing	0.168	0.216	0.493	60.719	4.131	2.593
	matching (×1)	1.521	1.512	3.620	2.080	2.077	2.053
	matching (×8)	0.539	0.463	1.482	1.164	0.946	0.745

portion to the number of subsets generated by the grouping algorithm, which doesn't show significant advantage than Hybrid-FA, D^{2}FA and BFA. Besides, BFA running with 8 threads achieves around 3 to 4 times faster matching speed compared to the single thread BFA.

When taking both pre-processing time and matching time into consideration, BFA consumes the least time to complete regular expression matching on this PCAP file, obtaining 3 to 5 times improvement than Hybrid-FA, more than 10 times improvement than NFA, DFA and D^{2}FA, and around 500 to 4,500 times improvement than mDFA for large or complex rulesets. For all rulesets which contains no more than 300 regular expression rules, NFA and BFA are the only two approaches that meet the 5 minutes update time requirement for some reaction-sensitive security appliance such as [2], while BFA achieves significantly better matching performance than NFA.

5.2 Memory usage

We also measure the memory usage of different matching methods on these rulesets. As shown in Table III, NFA has the most compact data structure and consumes the least memory, except for the *bro* ruleset. DFA requires the largest memory space to store its data structure, which is around 35 times more than NFA. D^{2}FA efficiently reduces the size of DFA by compressing redundant transitions, and the memory usage of *bro* ruleset is even smaller in comparison with NFA. The memory consumption of Hybrid-FA and BFA lies between that of NFA and DFA. More importantly, BFA only requires about 20MB to 30MB memory for large and complex rulesets, while the memory usage of DFA and DFA-based engines all exceed the maximum size of our experimental platform. mDFA consumes aournd 500 MB to 600 MB memory, which is 15 to 25 times more than BFA.

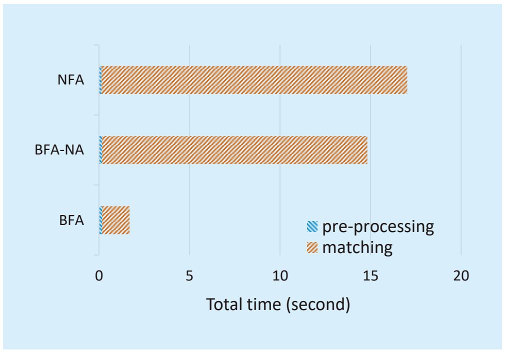


Fig. 2. *Evaluation of NFA, BFA without acceleration (BFA-NA) and BFA on ruleset snort1*

5.3 Optimizations

In this subsection, we evaluate BFA without instruction level accelerations to demonstrate the benefit brought by our optimizations introduced in Section IV. Figure 2 shows the pre-processing time, matching time and total time of NFA, BFA without acceleration (BFA-NA) and BFA on ruleset *snort1*. The pre-processing time of BFA-NA and BFA is almost the same, which is a little more than that of NFA. For the matching speed, BFA-NA only achieves about 15% improvement compared to NFA, while BFA is more than 10 times faster than NFA.

In NFA, the time complexity of state traversal is O(n^2) [29]. For a Boolean multiplication between a vector and a matrix, the time complexity decreases to O(n^2/log(n)) [30]. As a result, the Boolean multiplication would be faster in theory than the states traversals in NFA without hardware acceleration. However, the conducted evaluation doesn't present sig-

Table III. *Memory usage of different engines on different rulesets (unit: kb).*

rulesets	*snort1*	*snort2*	*bro*	*tcp*	*dotstar*	*range*
NFA	162.520	250.882	605.470	4,035.996	3,165.721	3,314.318
DFA	8,835.040	9,988.096	6,689.792	\	\	\
mDFA	\	\	\	485,840.786	609,587.440	620,601.645
Hybrid-FA	1,213.421	1,261.441	1,713.247	\	\	\
D^{2}FA	264.122	310.149	172.956	\	\	\
BFA	1,203.646	1,843.654	4,413.894	30,069.300	23,212.102	24,041.348

nificant improvement of BFA-NA over NFA. The O(n^2) time complexity only occurs in the worst case for NFA where every state in NFA is in active status, and every state needs to access all the other states for a input character. In average cases especially when most of the network traffic is non-malicious traffic, only a small part of NFA states are active. Therefore, the advantage of BFA-NA is not notable.

It can also be concluded that the optimizations actually contribute to the major boost of BFA. On one hand, BFA is a software/hardware co-design to achieve high-speed regular expression matching with fast pre-processing, On the other hand, without encoding to the types of bit vectors and bit matrices, state traversals in NFA or DFA cannot use *POPCNT*, *LZCNT*, and other bit manipulation methods directly in order to improve the performance. Therefore, the two ideas of BFA – Boolean matrix multiplication and CPU instruction acceleration – work together as a whole to obtain a better trade-off among pre-processing time, matching time and memory usage.

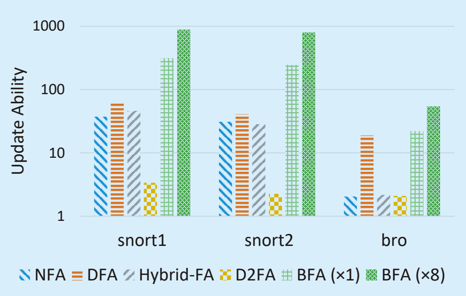
5.4 Update ability

As far as we know, there is no universal standard to measure the update abilities of regular expression matching engines. It is not enough to focus only on the construction time for a given ruleset or the matching time for specific network traffic. In fact, both of the throughput provided by a matching engine and the pre-processing time before it takes into effect need to be considered.

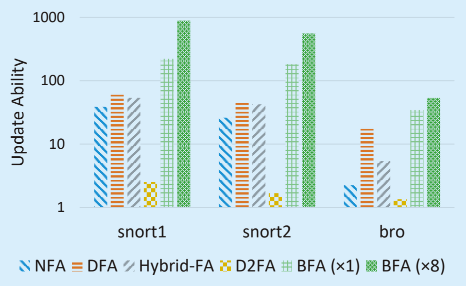
In this paper, a new criterion is proposed to quantify how a matching engine is suitable for a scenario where regular expression rules are frequently updated: the throughput a matching engine achieves per unit pre-processing time. Formally, for a matching engine E_i, if it could obtain T_i throughput in t_i pre-processing time, then the update ability (UA) of E_i is defined as:

$$UA(E_i) = T_i / t_i \tag{3}$$

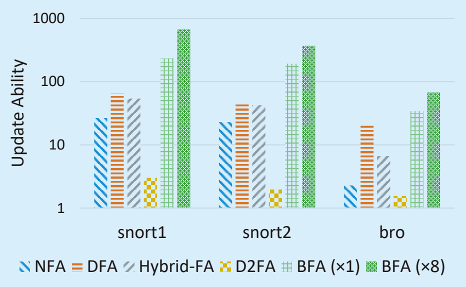
In order to measure the update abilities of different matching engines, we also use two different intrusion detection evaluation traffic



(a) Campus Network Traffic



(b) Darpa Traffic 1



(c) Darpa Traffic 2

Fig. 3. *Update abilities of different matching engines on different rulesets and traffic.*

from Darpa [31] as the input data in addition to the campus network traffic illustrated in Section 5.1. These data contain various malicious patterns, leading to more rule matches when performing deep inspection. Figure 3 presents the update abilities of NFA, DFA, Hybrid-FA, D^{2}FA and BFA on different rulesets and different traffic. Experiments on these three types of network traffic (figure 3a, figure3b and figure 3c) show the similar results. It is not surprising that D^{2}FA gets the worst update ability since it aims at compressing the memory consumption of traditional DFA at

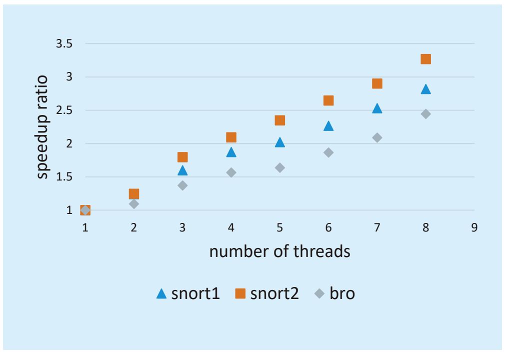


Fig. 4. *Speedup ratio of bfa on rulesets snort1, snort2 and bro with 1 to 8 threads.*

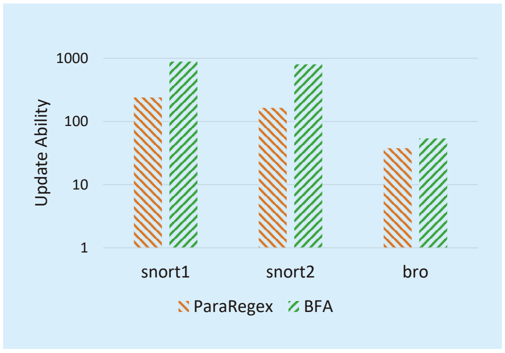


Fig. 5. *Update abilities of pararegex and bfa on the same multi-core platform.*

the cost of slower matching speed and longer pre-processing time. DFA method has a slight improvement over NFA and Hybrid-FA because of the leading throughput although DFA needs more pre-processing time. Our method BFA achieves about 5 times more update ability compared to DFA, Hybrid-FA and NFA and 20 times more update ability compared to D^2FA with just a single thread. When it scales to a multi-core platform, BFA can obtain around 3 times more update ability than the single thread. These experiments prove that BFA has a far better update ability, and thus BFA is more suitable for the frequently updating scenarios in comparison with state-of-the-art methods.

5.5 Scalability

To test the scalability of BFA, we run BFA with 1 to 8 threads on our experimental platform. Figure 4 shows the speedup ratio of BFA on rulesets *snort1*, *snort2* and *bro*, treating the single thread BFA as a baseline. As the number of threads increases, the matching speed of BFA grows almost linearly. We can also observe that the speedup ratios of BFA differ for different rulesets. This is because the transition tables in BFA for ruleset *snort1* and *bro* contain more bits that are set to 1, so the state transitions would take more time than *snort2*. Experiments on other rulesets obtain the similar results. In summary, BFA achieves an almost linear speedup ratio with multi threads and could obtain about 2 to 4 times speedup on an 8 core platform.

We also compare BFA to ParaRegex, which efficiently parallelizes DFA and DFA-based methods on multi-core platform. ParaRegex improves the matching speed by leveraging all CPU cores, but it does not do anything about the pre-processing procedure. Figure 5 shows the evaluations of the update abilities of both BFA and ParaRegex calculated from formula (3). Obviously, BFA still achieves about 2 to 8 times more update abilities compared to ParaRegex on the same multi-core platform.

VI. Conclusion

Regular expression matching is a fundamental component of deep inspection. While many researches only focus on improving the matching speed or reducing the memory consumption, the pre-processing of regular expression matching engines does not arouse much concern. As the network is becoming more dynamic than before, the state-of-the-art matching engines cannot meet the requirement of high-speed matching and fast pre-processing requirement simultaneously. In this paper, we propose BFA, a novel matching engine for deep inspection, which achieves high-speed regular expression matching with fast pre-processing.

The core idea of BFA is to transform the state transitions in traditional NFA or DFA into a multiplication between a bit vector and a bit matrix, which is in turn accelerated using fast bit manipulation and instruction level parallelism provided by modern commodity processors. Bitmap compression techniques are also used to reduce the memory usage of BFA. BFA is also designed to scale well on multi-core platform. Besides, a new criterion to measure the update ability of different matching engines is proposed in this paper. Our evaluations demonstrate that BFA achieves 5 to 20 times improvement compared to existing algorithms, and could get 3 to 4 times additional gains on a multi-core platform. By obtaining a much better trade-off among pre-processing, matching and memory consumption, BFA is able to meet the strict update time requirement for reaction-sensitive security appliances and perform high matching performance for large and complex regular expression rules in the meantime.

Our future work includes more efficient state encoding and accelerating multiplications on hardware (such as GPU and FPGA) to further improve the matching speed of BFA.

ACKNOWLEDGEMENT

This work was supported by the National Key Technology R&D Program of China under Grant No. 2015BAK34B00 and the National Key Research and Development Program of China under Grant No. 2016YFB1000102.

References

[1] Thompson K. "Programming techniques: Regular expression search algorithm." *Communications of the ACM* 11.6 (1968): 419-422.

[2] Cisco Email Security Appliance Data Sheet, https://www.cisco.com/c/en/us/products/collateral/security/email-security-appliance/data-sheet-c78-729751.html/.

[3] Xu C, Chen S, Su J, Yiu SM, Hui LC. "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms." *IEEE Communications Surveys & Tutorials* 18.4 (2016): 2991-3029.

[4] Knuth D E, Morris, Jr J H, Pratt V R. "Fast pattern matching in strings." *SIAM journal on computing* 6.2 (1977): 323-350.

[5] Boyer R S, Moore J S. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

[6] Aho A V, Corasick M J. "Efficient string matching: an aid to bibliographic search." *Communications of the ACM* 18.6 (1975): 333-340.

[7] Wu S, Manber U. "A fast algorithm for multi-pattern searching." (1994).

[8] Kleene S C. *Representation of events in nerve nets and finite automata*. No. RAND-RM-704. RAND PROJECT AIR FORCE SANTA MONICA CA, 1951

[9] Snort, https://www.snort.org/.

[10] Aho A V, Sethi R, Ullman J D. *Compilers: Principles, Techniques, and Tools* (1979).

[11] Kumar S, Dharmapurikar S, Yu F, et al. "Algorithms to accelerate multiple regular expressions matching for deep packet inspection." *ACM SIGCOMM Computer Communication Review*. Vol. 36. No. 4. ACM, 2006.

[12] Becchi M, Crowley P. "A hybrid finite automaton for practical deep packet inspection." *Proceedings of the 2007 ACM CoNEXT conference*. ACM, 2007.

[13] Liu A X, Torng E. "An overlay automata approach to regular expression matching." *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014.

[14] Xu Y, Jiang J, Wei R, et al. "TFA: A Tunable Finite Automaton for Pattern Matching in Network Intrusion Detection Systems." *IEEE journal on selected areas in communications* 32.10 (2014): 1810-1821.

[15] Liu T, Liu A X, Shi J, et al. "Towards fast and optimal grouping of regular expressions via DFA size estimation." *IEEE Journal on Selected Areas in Communications* 32.10 (2014): 1797-1809.

[16] Fu Z, Wang K, Cai L, et al. "Intelligent grouping algorithms for regular expressions in deep inspection." *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*. IEEE, 2014.

[17] Sidhu R, Prasanna V K. "Fast regular expression matching using FPGAs." *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 2001.

[18] Yang Y H E, Jiang W, Prasanna V K. "Compact architecture for high-throughput regular expression matching on FPGA." *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2008.

[19] Yu X, Becchi M. "GPU acceleration of regular expression matching for large datasets: exploring the implementation space." *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2013.

[20] Cascarano N, Rolando P, Risso F, et al. "iNFAnt: NFA pattern matching on GPGPU devices." *ACM SIGCOMM Computer Communication Review* 40.5 (2010): 20-26.

[21] Luchaup D, Smith R, Estan C, et al. "Multi-byte regular expression matching with speculation." *International Workshop on Recent Advances in Intrusion Detection*. Springer, Berlin, Heidelberg, 2009.

[23] Fu Z, Liu Z, Li J. "Efficient Parallelization of Regular Expression Matching for Deep Inspection." *Computer Communication and Networks (IC-*

CCN), 2017 26th International Conference on. IEEE, 2017.

[24] Yu H. "An improved combinatorial algorithm for boolean matrix multiplication." *International Colloquium on Automata, Languages, and Programming*. Springer, Berlin, Heidelberg, 2015.

[25] Lemire D, Ssi-Yan-Kai G, Kaser O. "Consistently faster and smaller compressed bitmaps with roaring." *Software: Practice and Experience* 46.11 (2016): 1547-1569.

[26] Wu K, Otoo E J, Shoshani A. "Optimizing bitmap indices with efficient compression." *ACM Transactions on Database Systems (TODS)* 31.1 (2006): 1-38.

[27] M Becchi. Regular expression processor, http://regex.wustl.edu/.

[28] Bro, https://www.bro.org/.

[29] Yu F, Chen Z, Diao Y, et al. "Fast and memory-efficient regular expression matching for deep packet inspection." *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. ACM, 2006.

[30] Arlazaro Vl, Dinits E A, Kronrod M A, et al. "On economical construction of the transitive closure of an oriented graph." *Doklady Akademii Nauk*. Vol. 194. No. 3. Russian Academy of Sciences, 1970.

[31] Darpa intrusion detection evaluation dataset, http://www.ll.mit.edu/ideval/data/1999data.html/.

Biographies

Zhe Fu, is currently a Ph.D. student at Tsinghua University, Beijing, China. He received the B.S. degree in the Department of Automation from Tsinghua University, Beijing, China, in 2013. He has been an IEEE student member since 2015. His research interests focus on security issues of network especially on pattern matching and traffic management. Email: fu-z13@mails.tsinghua.edu.cn

Jun Li, is currently Professor of Research Institute of Information Technology (RIIT), Tsinghua University. He is also Executive Deputy Director of the Tsinghua National Lab for Information Science and Technology. He holds a PhD degree in CS from New Jersey Institute of Technology (NJIT), and MS and BS degrees in Automation from Tsinghua University. He is a member of IEEE since 1996, and his research interest is in network security and Software Defined Network (SDN). Email: junl@tsinghua.edu.cn