Deflation DFA: Remembering History is Adequate

Yi Tang [1], Tianfan Xue [2], Junchen Jiang [1] and Bin Liu[1]
[1] National Laboratory for Information Science and Technology,
[1] Department of Computer Science and Technology, Tsinghua University, Beijing, PRC
[2] Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong
Email: tangy05@mails.tsinghua.edu, jimmyxuetianfan@gmail.com, livejc@gmail.com, lmyujie@gmail.com

ABSTRACT—There is an increasing demand for network devices to perform deep packet inspection (DPI) to enhance network security. In DPI the packet payload is compared against a set of predefined patterns which can be specified using regular expressions (regexes). It is well-known that mapping regexes to deterministic finite automata (DFA) will suffer from the state explosion problem. Through observation, we attribute DFA explosion to the necessity of remembering matching history. In this paper, we investigate how to record the matching history efficiently and propose an extended DFA approach for regex matching called fcq-FA, which can make a memory size reduction of about 1000 times with a fully automated approach. In fcq-FA, we use pipeline queues and counters to help recording the matching history. Hence, state explosion caused by Kleene closure and repetitions can be definitely avoided. Further, it achieves a fully automated signature compilation with polynomial running time and space.

Key Words—Deterministic Finite Automata (DFA), Deep Packet Inspection (DPI), Regular Expression (Regex)

I. INTRODUCTION

Nowadays, security has become one of the most serious concerns on current Internet. Network Intrusion Detection Systems (NIDSes) have been adopted by enterprises to defend against all kinds of exploits towards them. In the commercial world, regular expression (regex) has already been the de facto signature standard for signature-based NIDSes [7][10], because of its expressiveness and matching efficiency [14]. The famous open-source NIDS, Snort [7], has more than 11000 signatures. To make the signature harder to be evaded by attackers, people tend to write more and more complex regexes as signatures. It cause the memory size of naïve DFA-based regex approaches unacceptable, which increases exponentially in worst case, e.g., the Snort HTTP signature set will result in a DFA larger than 15GB [4]. Various approaches [1][3][4][5][8][9][13] have been proposed to solve this issue.

As is well known regex DFA explosion is a result of some functions such as Kleene closure (‘.*’, ‘[^a]*’) or length restrictions (‘a{100}’). However, few works investigates that different closures and restrictions have different impact on the DFA state explosion. Table I is an example of exponential inflation caused by length restriction. Though the regexes are very simple, the states number for these regexes changed dramatically.

In this paper, we attribute the regex DFA explosion to the necessity of remembering matching history. For regex ‘a[a|b]{13}’, the prefix ‘a’ overlaps with length restriction ‘[a|b]{13}’. To cover all cases, the DFA should remember the counter status when prefix ‘a’ is matched each time. For regex 'ab[^b]{1000}', the prefix ‘ab’ does not overlap with ‘[^b]{1000}’. So, it is not necessary to remember any history information. For real word cases, it is more complicated, we will discuss intensively in section III.E.

TABLE I. INFLATION CAUSED BY LENGTH RESTRICTION

Regex	# of states	Regex	# of states
A	2	ab	3
a[a|b]{11}	4096	ab[^b]{10}	33
a[a|b]{12}	8192	ab[^b]{100}	303
a[a|b]{13}	16384	ab[^b]{1000}	3003

On the purpose of providing an alternative regex implementation rather than DFA or numerous DFA compression methods, we investigate how to record the regex matching history efficiently. In this paper, we introduce bit-queue to remember matching history and further propose a new scheme called fcq-FA. Fcq-FA achieves fully automated signature compilation with polynomial running time and space. In general, our major contributions and advantages are highlighted below:

1. By introducing bit-queues we can remember the history on which parts of the regular expressions have been matched easily. This enables us to convert a regex to fcq-FA in a fully automated manner. As far as we know, our research is the first effort to achieve memory reduction and matching efficiency in a fully automatic generated regex matching system in polynomial time and space
2. Different closures require different amount of auxiliary information and we devise a general rule to decide whether a closure is friendly (requiring little auxiliary information) or not. Therefore, we can distribute less auxiliary information to those friendly rules to achieve a better compression effect.
3. Our method supports incremental updating for regex rules. It is easy to insert or delete any regex rule at running time without requiring the matching system to recompile all the other rules. Current automaton based methods can hardly support incremental updating regex rules.

The evaluation is based on 71 randomly chosen Snort HTTP rules. They require more than 2GB memory to combine together, however, in our approach only need 503KB DFA plus 0.3KB register memory for keeping the historic matching information. Further, we achieve such performance in a fully automated manner.

This paper is organized as follows. After related work presented in section II, we give the idea of fcq-FA and its optimization in section III. The experimental is provided in section IV. We will draw the conclusion in section V.

II. RELATED WORK

Currently, most approaches focus on how to modify DFA structure to save memory without much matching overhead. There are two major directions in the research of DFA's deflation. One is to eliminate transition redundancy; the other is to reduce state explosion. For transition eliminating, Kumar et al. proposed D2FA [9], which compresses DFA through the introduction of default transition, and it trades off storage requirement with processing time. CD2FA [8] following D2FA is proposed by using recursive content labels to reduce the diameter bounds of D2FA to 2 with only one 64-bit wide memory access. In [13], Becchi proposed to limit the bound of the number of default paths in D2FA, and improved the worst case of it. Essentially these methods are targeting at reduce the number of transition, which does not touch the kernel of regex DFA explosion and fail to obtain good performance.

For deflation state explosion, in [6], Becchi proposed an algorithm to merge states, which reduced partial states into labeled transitions and can achieve memory reduction one order of magnitude. However, to build the combined DFA is very hard given its size may be 10GB or even 100GB.

Currently, the state of art work XFA [2][4]. It uses auxiliary information to reduce space requirement. To build a DFA with auxiliary information, first, a NXFA (A NXFA is a NFA with nondeterministic auxiliary data domain) is constructed from the parsing tree of regular expression; Second, an original XFA is built from NXFA by ε-elimination and determinization; Third, an efficient implementable data domain (EIDD) is found and mapped to the original XFA, in order reduce the size of auxiliary information.

However, the actual performance of this algorithm remains arguable. First of all, the original XFA has little effect on reducing the space requirement. In XFA, it first uses nondeterministic auxiliary information (data domain) to represent rules and then converts it to a deterministic one. However, this will cause a possible auxiliary memory bloating up, as n possible values of auxiliary memory in NXFA may lead to 2^n possible ones in XFA, just like n possible state in NFA may lead to 2^n possible states in DFA. This data domain explosion in XFA can also be explained in a formal way. If the original DFA has A states, and the XFA has B states and C possible values of auxiliary information, then it must satisfy the following inequation: A<BC, otherwise XFA cannot demonstrate all cases of original DFA. Consequently, the total memory requirement B+C is no less than $O(\sqrt{A})$. For the original DFA with $O(2^n)$ states, XFA will have $O(2^n/2)$ states, which is also an exponential size. Notice, this is a best case bound. Actually, for most rules, the original XFA has similar size of the traditional DFA.

To solve this problem, XFA introduces EIDD to reduce auxiliary information requirement. Although EIDD can greatly reduce the size of auxiliary memory, it brings another problem: finding a mapping from EIDD to XFA is rather a time consuming job. As no systematic way to find a good EIDD, the only way is to carefully examine the rules and design EIDD manually. This may be feasible for a simple rule, but for those complex rules with large auxiliary information, there is no guarantee that a most efficient EIDD can be found by manual-effort, or even if it is found, it may require much time and effort. According to [4], these kinds of rules occupy the 13.5% in the rule set they choose. The major problem of XFA is the use of nondeterministic auxiliary information as gangway to build XFA. This not only fails to make the problems simpler, but actually introduce a great time and manual cost in constructing XFA. In the next section, we will give an automata directly constructed from regex, which can avoid the problems in XFA.

III. THE IDEA OF FCQ-FA

As its name, fcq-FA contains three parts: DFA, flag/counter and queue. DFA part is just a normal DFA built from sub-rules resulted from breaking rules by eliminating Kleene closure and length restriction. There are some actions taken on states, usually on final states. Flag/counter is used to mark an arrival of closure or length restriction respectively. The queue is used to record the matching history of certain parts of regex. To clearly state our idea, we start with a simple example with closures using fq-FA (without counter), then followed by another example with length restriction adopting cq-FA (without flag).

A. The Basic Idea of fq-FA

For the sample with two rules: r1='.*ab[^a]*ac', r2= '.*def'. We first eliminate closures of the regexes, break them into three sub-rules: s1='ab', s2= 'ac', s3='def'. Figure 1 shows the corresponding fq-FA structure (some of the transitions have been omitted to keep it readable). In this example the corresponding regex of the DFA is (.*s1|.*s2|.*s3)=(.*ab|.*ac|.*def).

Since there is only one closure, we use a single flag. The flag represents whether the sub-regex has matched up to the closure, 'ab[^a]*' in this example. Each time when a new character is read in, the flag can be set '0' or '1', or remain unchanged. Here we set the flag whenever the DFA reaches state 2 and clear it as long as the input character is 'a', otherwise the flag remains unchanged.

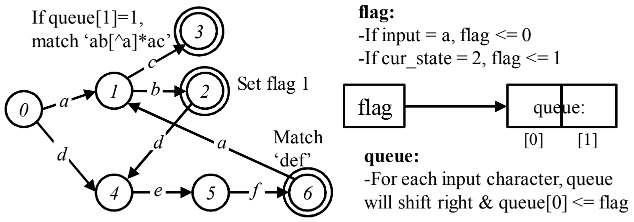
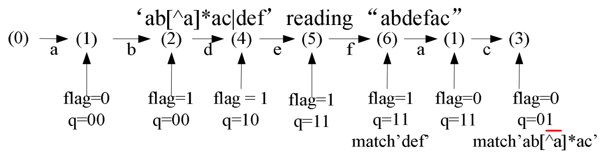


Figure 1. Example of fq-FA for Regex (r1='.*ab[^a]*ac', r2= '.*def')

The system maintains a queue to store most recent n flag values. When input character comes, we shift queue to the right for one step and set queue[0] the flag value of last cycle. The queue length is decided by the length of sub-rule just after closure. Here queue length is 2 as the length of s2.

The following is a sample illustrating the execution of fq-FA, using 'abdefac' as an input string. When parsing to state 2, it will set flag '1', which means 'ab[^a]*' is matched. When arriving state 3, queue[1] will be checked. As the value is '1', it will report that rule 'ab[^a]*ac' has been matched. Notice that although flag is '0' at that time, fq-FA still remembers that flag was '1' two characters ago using queue, and therefore it correctly reports a match.

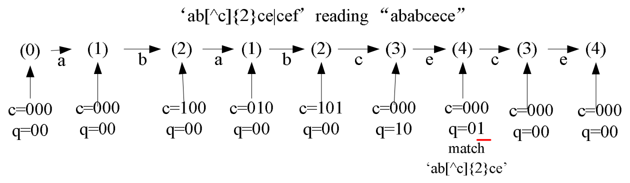

'ab[^a]*ac|def' reading "abdefac"
(0) a (1) b (2) d (4) e (5) f (6) a (1) c (3)
flag=0 q=00
flag=1 q=00
flag = 1 q=10
flag=1 q=11
flag=1 q=11 match'def'
flag=0 q=11
flag=0 q=01 match'ab[^a]*ac'

The intuition behind this example is that fq-FA remember the history when 'ab[^a]*' is matched. After the 'ac' is matched we use the bit-queue to check whether the 'ab[^a]*' has already been matched before 'ac' arriving. If yes, it is a valid match of whole expression. The key is to use queue structure to record the history information.

B. The basic idea of cq-FA

We take rule set: r1='ab[^c]{2}ce', r2= 'cef' as an illustration. For eliminate length restriction, we break rules into sub-rules: r1='ab', r2= 'ce', r3='cef' and then construct the corresponding cq-FA, as Figure 2 shows.

In cq-FA we move the rightmost bit of the counter, rather than the previous flag value, to the leftmost bit of queue. A counter is an 'm+1'-bit-queue as stated above. It can be set and reset, and shifted right on every input. When counter is set, count[0] will be set to '1'; When the counter is reset, all bits in it will be set to '0'. This occurs when the input character is conflicting with character set of closure (in this case, {'c'}). Besides, all bits will shift one step right each turn, and the leftmost bit is put into the queue accordingly.


'ab[^c]{2}ce|cef' reading "ababcece"
(0) a (1) b (2) a (1) b (2) c (3) e (4) c (3) e (4)
c=000 q=00
c=000 q=00
c=100 q=00
c=010 q=00
c=101 q=00
c=000 q=10
c=000 q=01 match 'ab[^c]{2}ce'
c=000 q=00
c=000 q=00

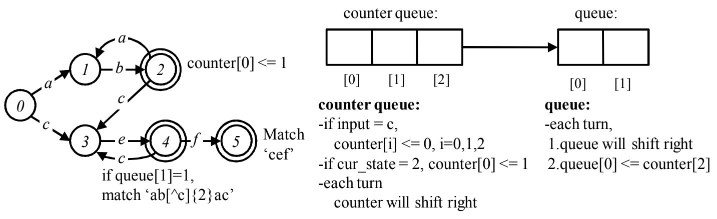
For input string 'ababcece', notice that the second time sub-rule 'ce' is matched, it will not report a match, because at that time queue[1] is already '0' since the first input of 'c' has reset the whole value of counter queue.

By combining fq-FA and cq-FA, we can get the fcq-FA. The following subsections will show the steps in building a matching system using fcq-FA.

C. Rewrite rules and build DFA Engine

The first step of constructing fcq-FA system is to rewrite rules. One original rule will be broken into several sub-rules to satisfy following requirements: First, no closure or length restriction appears at sub-rules; second, sub-rule should have a fixed length, otherwise it should be further split.

Table II shows the result of rewriting rule 'ab[^a]*(b|ab)c[^b]*de'. First two closures '[^a]*' and '[^b]*' have been eliminated, then we get three sub-rules



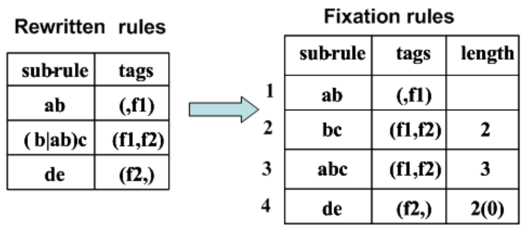
counter queue:
-if input = c,
counter[i] <= 0, i=0,1,2
-if cur_state = 2, counter[0] <= 1
-each turn
counter will shift right

queue:
-each turn,
1.queue will shift right
2.queue[0] <= counter[2]

Figure 2. Example of cq-FA for Regex(r1='ab[^c]{2}ce', r2= 'cef')

'ab', '(b|ab)c' and 'de'. Since '(b|ab)c' does not have a fixed length, it is further split into 'abc' and 'bc'. Besides, some tags are assigned to show positions of sub-rules in the original rule. In this case, 'f1' is used to represent the closure '[^a]*' and 'f2' is used to represent '[^b]*', so (f1,f2) means that sub-rule 'bc' is located between '[^b]*' and '[^a]*'. For each closure, we use a queue and a flag; for each length restriction, we use a queue and a counter.

TABLE II. REWRITTEN RULES



Rewritten rules

sub-rule	tags
ab	(,f1)
(b|ab)c	(f1,f2)
de	(f2,)

Fixation rules

	sub-rule	tags	length
1	ab	(,f1)	
2	bc	(f1,f2)	2
3	abc	(f1,f2)	3
4	de	(f2,)	2(0)

Then we build a DFA using these sub-rules. We combine all the sub-rules via '|' operator, and then build a DFA to represent this regex. The DFA of '.*ab|.*(b|ab)c|.*de' is shown in Figure 3 (Still, some less important transitions are ignored to make the figure readable. We also mark actions need to take and tags we got above). Besides, the index of sub-rules being matched need to store on each final state.

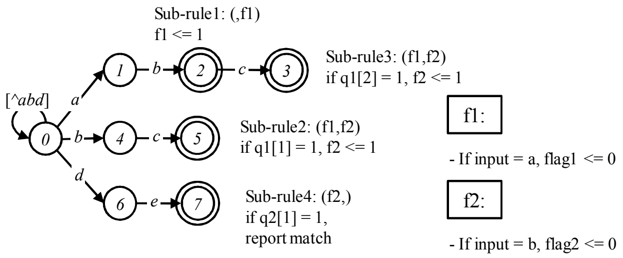


Figure 3. fcq-FA of 'ab[^a]*(b|ab)c[^b]*de'

D. Building a matching system

Here, we use two tables (action table and invalidation table) to store the actions to take on each state. Action table tells us when to set flag, counter or report a match, while invalidation table preserve when to reset a flag or a counter.

TABLE III. ACTION TABLE OF 'AB[^A]*(B|AB)C[^B]*DE'

request	length	action	While Reading (sub-rule)
null	2	set f1	ab
f1	2	set f2	bc
f1	3	set f2	abc
f2	2	report match r1	de

In action table, each row stands for a sub-rule. And for each sub-rule, there are three entries: 'request', 'length' and 'action'. The 'request' entry indicated which queue we need to check when this sub-rule is matched. The 'length' entry shows the length of this sub-rule, and further implies which bit of that queue needs to be checked. And 'action' entry illustrates what we need to do if the sub-rule is matched and 'request' is fit. We still use the example mentioned above. Since the 'requested' entry of sub-rule 'bc' is 'f1' and the 'length' entry is 2 (shown in Table III), when sub-rule 'bc' is matched, we need to check whether q1[2-1] is '1'. If yes, we will set flag2. (Here, flag k and q[*k*] represent *k*th flag and *k*th queue respectively, and we suppose that *k*th flag is correspond with *k*th queue)

TABLE IV. INVALIDATION TABLE OF 'AB[^A]*(B|AB)C[^B]*DE'

input	f1	f2
'a'	0	1
'b'	1	0
others	1	1

In the invalidation table, each row represents a possible input character and each column represents a flag or a counter. The value of the table entry may be either '0' or '1'. If a certain entry is '0', for example, t['a', f1] = '0'(shown in table IV), it means that when the input character is 'a', it will reset queue f1; Otherwise the f1 will remain unchanged.

E. *Reduce auxiliary information*

The fcq-FA has significantly reduced the states number of DFA, but the auxiliary memory also incurs high cost. In this subsection, we introduce a method to eliminate some of redundant auxiliary information for fcq-FA. Different closures and length restrictions actually play different roles in regex, through distinguishing different properties of closures and treating them diversely, we achieve a reduction memory about 50% for bit-queues.

First, for a rule like r1[^c]*r2, if it satisfies the following two qualities, only one bit is required to represent the closure without introducing any false negative or false positive (Here c denotes for any character or a character set):

1. For any string $s_1 \in L(r_1)$ and $s_2 \in L(r_2)$, suffix of s1 doesn't equal to prefix of s2. Also, s1 itself doesn't appear in s2.
2. For any string $s_2 \in L(r_2)$, any character $c' \in s_2$ should not appear at c (that is $c' \notin c$).

The condition 1 ensures that when matching r2, the flag will not be set to '1'; the condition 2 guarantees that when matching the sub-rule r2, flag will not be set to '0'. In this way, we can eliminate the queue from fcq-FA, and when r2 is matched, we need only check the corresponding flag. If the regex just satisfies condition 2 but not condition 1, the flag queue can be replaced by an integer counter of strlen(r2), not just 1 bit.

Taking regex 'ab[^a]*bc' as an example, the regex satisfies condition 2 but not condition 1. Hence, the bit-queue for closure '[^a]*' cannot be eliminated. Otherwise, when system is reading 'abc', the flag is set '1', meanwhile DFA for 'ab|bc' matches string 'abc'. If using 1-bit flag, rather than bit-queue, the system will report false positive results. Another example, for regex 'ab[^c]*ecf', it satisfies condition 1 but not condition 2. So, while matching 'ecf', the flag for '[^c]*' will be set to '0'. If using just 1-bit flag, system would report false negative results.

As for counter queue, we can use an integer counter rather than a n-bit-queue in some cases. For a rule like r1[^c]{n}r2, if the following property is met, the length restriction can be replaced by a simple integer counter:

1. For any string $s_1 \in L(r_1)$, there is at least one character $c' \in s_1$ appearing at c (that is $c' \in c$)

This condition ensures that when matching '[^c]{n}', r1 will not be matched and consequently the counter will not be set again. Hence, an integer counter is enough in this case. This can reduce the space requirement, because for length restriction '[^c]{n}', we need (n+1)bits if bit-queue is used, while only log(n) bits are is needed if an integer is used.

F. *A more Complicated Example*

Taking regex of protocol "imesh" in L7-filter [12] as example, using original way to build DFA, it will have 3416 states and 132383 transitions. Such a large memory requirement will make the hardware implementation impossible if we combine such rules together. The regex is:

```
^(post[\x09-\x0d\x20-~]*<PasswordHash>................................</Password
Hash><ClientVer>|\x34\x80?\x0d?\xfc\xff\x04|get[\x09-\x0d\x20-~]*Host
:\x20imsh\.download-prod\.musicnet\.com|\x02(\x01|\x02)\x83.?.?.?.?.?.?.
?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?\x02(\x01|\x02)\x83)
```

First, we divide the big rule into small ones according to the operator '|'. The above regex is split into 4:

```
1.^(post[\x09-\x0d\x20-~]*<PasswordHash>................................</Passwo
rdHash><ClientVer>)
2.^(\x34\x80?\x0d?\xfc\xff\x04)
3.^(get[\x09-\x0d -~]*Host: imsh\.download-prod\.musicnet\.com)
4.^(\x02(\x01|\x02)\x83.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?.?\x
02(\x01|\x02)\x83)
```

An action table is produced by going through the regex.

TABLE V. ACTION TABLE OF IMESH REGEX

Sub-rule	Request	Length	Action
^post	null		set f1
<PasswordHash>	f1	14(0)	set cq1
</PasswordHash><ClientVer>	cq1	32+26	matched
^(\x34\x80?\x0d?\xfc\xff\x04)	null		matched
^get	null		set f2
Host: imsh\.download-prod\.musicnet\.com	f2	37(0)	matched
^\x02(\x01|\x02)\x83	null		set cq2
\x02(\x01|\x02)\x83	cq2	0-24	match

Meanwhile, we combine new generated sub-rules. The DFA for combined rules has 100 states and 3800 transitions. Comparing with the traditional DFA which has 3416 states and 132383 transitions, fcq-FA deflates the DFA greatly by only introducing 4 bit-queues, while 2 of them (f1 and f2) can be further eliminated as optimization scheme discussed in section III.E. Moreover, the whole process is accomplished automatically without any manual effort.

IV. Experimental Evaluation

A. *Experiment on sample rules*

We randomly select 71 rules from Snort's web pcre ruleset. The result is shown in table VI.

TABLE VI. bit-queues's Consumption

Rule num	Sub rules num	Flag queue num	Flag queue bits	Optimized flag queue num	Optimized flag queue bits
		65	335	43	103
71	174	count queue num	count queue bits	Optimized counter queue num	Optimized counter queue bits
		28	7445	15	2287

The rule set contains of 71 regexes, which produces 174 sub-rules as table VI shows. The length restriction appears 28 times in 71 rules. The average length of them is 266 bits (7445/28), and the longest is 1024. fcq-FA can be constructed regardless the length n. In evaluation, 13 of 28 counter bit-queues can be replaced. And the total length of counter queue reduces from 7445 to 2287. In fcq-FA, the DFA for the combination of 174 sub-rules requires 2862 states. For total memory requirement of fcq-FA, we must consider memory used by DFA as well as bit-queue. As the above Table VI, less than 0.6M bytes of static memory plus 0.3k bytes dynamic register are needed to represent the combination of these 71 Snort rules.

B. *Experiment on Snort rule*

We calculate the memory consumption of our system for Snort-web, Snort-ftp, Snort-smtp and Snort-voip. Notice that bitmap compression does not use to optimize bit-queue here.

TABLE VII. Snort Rule's memory infomation

Rule set	Rule num	Flag queue bits	Counter queue bits	Optimized flag queue bits	Optimized counter queue bits
web	733	36932	21455	12056	3540
ftp	60	259	6378	120	1458
smtp	42	192	8888	110	1514
voip	51	318	1929	150	410

Rule set	Mem. for DFA	Sub rule num	Combined DFA	Combined in fcq-FA	
			States	States	Transition
web	9000KB	4095	>3.1M	40457	4604294
ftp	300KB	114	>3.1M	349	17943
smtp	700kB	141	>3.1M	461	33123
voip	1000KB	513	>3.1M	710	47034

Table VII shows the great effect of fcq-FA on deflating the DFA of Snort rules. Especially, to build a matching system of 733 Snort web rules, 9MB static memory and 2KB dynamic memory is required here, which the traditional DFA implementation may cost larger than 10 GB.

V. Conclusion and Future Work

In this paper we find the essence of regex DFA explosion is to remember the matching history. Based on this observation, we propose a kind of extended DFA with several bit-queues (fcq-FA) to help recording the matching history. It eliminates most closures and length restrictions in a regex, which are the major causes of the explosion. Compared with other algorithms using auxiliary memory, fcq-FA achieves a better space performance and less generating and updating time. The essence of fcq-FA is utilizing bit-queues to record matching history to achieve the deflation of DFA. While, XFA uses instructions to handle regexes, which is a general CPU approach and requires lots of manual work to achieve optimization and is error-prone.

Acknowledgment

This work is supported by NSFC (60625201, 60873250, 60903182), 973 project (2007CB310701), 863 high-tech project (2007AA01Z216) and Tsinghua University Initiative Scientific Research Program

VI. References

[1] F. Yu et alt., "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection", in ANCS 2006

[2] R. Smith, C. Estan, Somesh Jha, "Deflating the Big Bang: fast and scalable deep packet inspection with variable-extended automata" SIGCOMM, August 2000

[3] S. Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, George Varghese, "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia" , in ANCS 2007

[4] R. Smith, C. Estan, Somesh Jha "XFAs: Faster signature matching with extended automata", IEEE Symposium on Security and Privacy (Oakland), May 2008

[5] B.C. Brodie, R.K. Cytron, and D.E. Taylor. "A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching." In ISCA, Boston, MA, June 2006.

[6] M. Becchi and S. Cadambi. "Memory-efficient regular expression search using state merging". INFOCOM, May 2007.

[7] Snort. http://www.snort.org/, 2008.

[8] S. Kumar, J. Turner and J. Williams, "Advanced Algorithms for Fast and Scalable Deep Packet Inspection", in ANCS 2006

[9] S. Kumar et alt., "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in ACM SIGCOMM, Sept 2006.

[10] Bro Intrusion Detection System. http://bro-ids.org/

[11] Perl Compatible Regular Expressions: http://www.pcre.org/

[12] "Application Layer Packet Classifier for Linux." http://l7-filter.sourceforge.net/.

[13] M. Becchi and P. Crowley, "An Improved Algorithm to Accelerate Regular Expression Evaluation" , ANCS 2007.

[14] R. Sommer and V. Paxson, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context," Proc. ACM CCS 2003