Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 56 (2015) 292 – 299

Procedia Computer Science

The 10th International Conference on Future Networks and Communications (FNC 2015)

A Memory Efficient DFA using Compression and Pattern Segmentation

Yeim-Kuan Chang, Yuen-Shuo Li, and Yu-To Chen

Department of Computer Science and Information Engineering
National Cheng Kung University, Taiwan

Abstract

The role of network security systems such as network intrusion detection system (NIDS) has become more important. The regular expression (a.k.a. regex) matching algorithm used for inspecting the payloads of packets is one of the most intensive tasks in NIDS. When multiple regular expressions are processed together, the corresponding Deterministic Finite Automata (DFA) becomes so complicated and needs a large amount of memory. In this paper, we propose a memory efficient parallel compatible DFA algorithm that uses the techniques of compression and pattern segmentation to reduce the memory usage. Extended from PFAC (Parallel Failureless-AC) algorithm4, the proposed compressed and segmented DFA (CSDFA) needs less numbers of states and transitions than δFA. Without considering the leading symbols ".*" in the regular expressions, the transition table can be compressed very efficiently by the run-length encoding. The number of transitions in CSDFA is about a half of the transitions needed in δFA, and uses only 74% of the memory consumed by δFA. Based on our experiments, the throughput of the proposed CSDFA is also much better than δFA.

Keywords: Network intrusion detection; pattern matching; regular expression;

1. Introduction

As the traffic of Internet is growing quickly, more malicious attacks and viruses spread over the Internet. The role of network security systems such as network intrusion detection system (NIDS), firewalls, and antivirus software has become more important. Snort and Bro are two NIDS examples that have been widely used to safeguard the security of network activities. NIDS inspects the payload of each packet with predefined rules or patterns. If the packet matches the rules, NIDS will perform the actions of dropping or blocking the packet predefined in the rules.

Pattern match algorithms can be divided into single-pattern and multi-pattern based schemes. The most famous

* Corresponding author. Tel.:+886-6-2757575-62539; fax:886-6-2747076.
E-mail address: ykchang@mail.csie.ncku.edu.tw

doi:10.1016/j.procs.2015.07.211

single-pattern matching algorithms are Boyer-Moore[8] and Knuth-Morris-Pratt (KMP)[3]. The most famous multi-pattern matching algorithms are Wu-manber[20] and Aho-Corasick (AC)[7] algorithms. Also, according to different forms of rules, patterns can be in the form of simple pattern or regular expression (regex). Processing regex is more complicated than processing simple patterns. In this paper, we focus on regex. A regex is a concise format to specify a set of simple patterns, instead of specifying them individually. For example, the pattern set containing three patterns, "apple", "angle", and "ankle" can be specified by the regular expression of "a(pp|ng|nk)le". The specification of regular expression is not unique. If at least one regex matches a particular pattern set then an infinite number of other regexes exist and also match this pattern set.

We can use regex engine to match the regular expression for the given string. Different regex engines may not be fully compatible with each other. But most regex engines will provide the same operations to handle regular expressions. Basically, a regular expression is composed of a sequence of atoms. An atom is what matches at a point in the target string. The simplest atoms are literals such as A, B, or C. The grouping parts surrounded by bracket of the pattern can also be seen as an atom too. There are some quantifiers telling how many atoms are allowed to occur. The common quantifiers are the question mark "?", the plus sign "+", and the asterisk "*". The question mark indicates there is zero or one of the preceding token, the asterisk "*" indicates there is zero or more of the preceding token, and the plus sign "+" indicate there is one or more of the preceding token.

Regex matching is one of the most intensive tasks in the NIDS. Because these systems must be performed online in line speed, the effective use of the pattern matching algorithm is very important. Generally, these algorithms have almost the same process to handle the regular expression. The first data structure we need is the parse tree[17] of the pattern. There are various rules like vertical bar "|", asterisk "*", question mark "?" or plus sign "+" in regular expressions and some of rules have higher priorities than the other. The parse tree helps us to know which rule will be handled first, and which one will be handled later. Some algorithms may transform the patterns into the post-order form with the same reason. The first symbol in post-order form will be handled first, and second symbol will be handled second. The parse tree is just another representation of the patterns. It has equal ability to handle the patterns. The left leaf node will be handled first, and the parent node only affects its child nodes.

Nondeterministic finite automaton (NFA) avoids state blowup problem. Unfortunately, the drawback of NFA is that a lot of active states must be maintained at each cycle. When an input symbol is read, NFA has to update all the active states to their next states. If the number of active states becomes large, the updating cost will also increase. In recent years, NFA approaches has been attracting much attention, because the speed of these approaches is much faster if GPU (Graphic Processing Unit), FPGA (Field-programmable gate array), or some special hardware are used.

The second algorithm builds a deterministic finite automata (DFA). DFA is extremely fast, but it may need a large amount of memory to store the DFA transition table because the given regular expressions cause the exponential state explosion. For this reason, we usually do some optimization for DFA to reduce the memory consumption. The difference between an NFA and a DFA is that a DFA requires only one state traversal per character, but it usually needs much more memory than an NFA. Therefore, many works have been proposed recently with the goal of memory reduction for DFAs. In this paper, we will propose two techniques to improve memory utilization of DFA, the run-length encoding scheme to compress the transition table and the pattern segmentation to solve the state explosion problem of DFA.

The rest of this paper is organized as follows: Section 2 reviews background and related work. In section 3, we introduce our proposed scheme in details and some instances to show the advantage of our ideas. We will show the search process of our proposed scheme. The analysis of Snort and Bro pattern set's experiment results are discussed in section 4. Finally, we conclude the paper in section 5.

2. Related work

Thompson NFA[10] is a very popular method to build NFAs. Its construction is linear in the numbers of states and transitions. However, this automaton has ε-transitions which can be passed through without reading a character. Another popular NFA constructing method is Glushkov's algorithm[9]. We call the automaton constructed by this method Glushkov NFA. An NFA does not require input symbols for state transitions which we usually use epsilon symbol (ε) represents this situation, and is capable of transitioning more than one next state for an input symbol, but Glushkov NFAs don't have ε-transitions which means it require input symbols for each state transitions, so we usually call it ε-free NFAs. It has only one initial state and may be one or more final states. And the other interesting property is that for any state, the state's incoming transitions are labeled by the same character. This NFA has only m + 1 states if we don't do any optimization, where m is the number of non-special characters in a pattern. But the number of transitions is m2 in the worst case which is larger than Thompson NFA's. NFA by Sidhu and Prasanna[15] is another way to construct an NFA. The proposed algorithm converts an arbitrary regex into an NFA capable of processing 2k symbols per cycle. The procedure of this algorithm is to convert the regular expression into a 1-character NFA, and then modify it to support multiple symbols.

The simplest way to construct a DFA is to build an NFA first and convert it into a DFA by subset construction[11]. The main idea of subset construction is as follows: when we traverse the text using a nondeterministic automaton, a number of transitions can be followed and a set of states become active. However, a DFA has exactly one active state at a time. So the corresponding deterministic automaton is defined over the set of states of the nondeterministic automaton. The key idea is that the unique current state of the DFA is the set of current states of the NFA.

Delta Finite Automata (δFA)[5] is a representation of DFA that considerably reduces the numbers of transitions. δFA examines one state per character only and thus reduces the number of memory accesses and speeds up the overall search process. δFA is orthogonal to the other algorithms such as XFAs[12] and H-cFA[13]. The main idea of δFA comes from the observations showing the most default transitions are directed to states closer to the initial states and the most transitions for a given input character are directed to the same state. By elaborating on this observation, it becomes evident that most adjacent states share a large portion of the same transitions. If we can "remember" the entire transitions of parent state in local memory, we don't need to store the entire transitions in the child state.

3. Proposed Scheme

Before introducing the proposed scheme, we will first describe the classical procedure of building DFAs. The first step of the DFA building procedure builds a parse tree for the given pattern. Various features that can represent the variety of the regular expressions include vertical bar "|", the asterisk "*", question mark "?" and the plus sign "+". Also, some of rules have higher priority than the others. The parse tree helps us to know which rule will be handled first, and which one will be handled later. The second step builds an NFA from the parse tree. Thompson NFA and Glushkov NFA algorithms are two typical methods to construct NFAs. The third step converts the NFA into a DFA by using subset construction algorithm. We can use the DFA to perform the regex matching. The advantage is that searching in DFAs is very fast, because it has only one active state in each cycle. However, DFAs have its own problem called exponential state blowup (or state explosion) where there are too many states and transitions in a DFA. State blowup problem leads to excessive memory requirement to store a DFA. Therefore, many works have been recently presented with the goal of memory reduction for DFAs like D^2FA[14] or δFA[5].

The main purpose of our proposed scheme is to reduce memory consumption by using two techniques, compression and pattern segmentation. We will describe the details of this scheme in the following sections starting from the observations on removing the symbols of ".*" from regular expressions in the DFAs, to the operations of

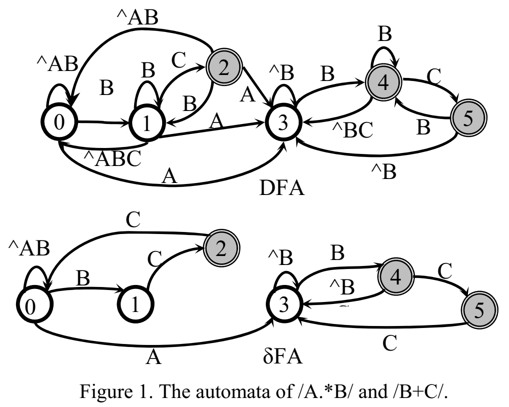

Figure 1. The automata of /A.*B/ and /B+C/.

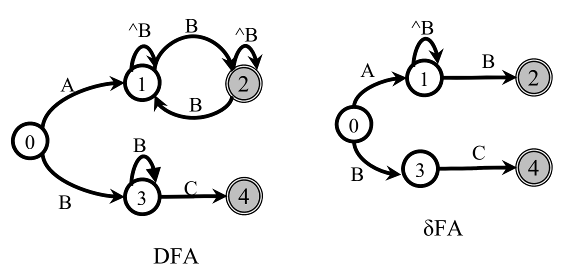

Figure 2. DFA and δFA of /A.*B/ and /B+C/ without leading ".*".

segmenting the patterns with "Counting Constraints" and "Kleene Star" which usually cause state blowup.

3.1 CSDFA

Compared to DFA, δFA[5] has remarkably less number of transitions. The main idea of δFA is from an observation that the most transitions for a given input symbol are directed to the same state. Let *P* be the parent state of a child state *C*. If we can "remember" all the transitions of parent state *P* in local memory, we have no reason to store all the transitions in child state *C* when *C*'s transition table is the same as *P*. δFA employs a search table that records 256 next states information on the fly. We only needs to store in *C* the differences of the transitions between states *P* and *C*. Figure 1 shows a DFA consisting of regexes /A.*B/ and /B+C/ and the corresponding δFA. We usually omit the ".*" from the beginning of patterns when we discuss the regular expressions. In this example, the pattern /A.*B/ and /B+C/ actually are /.*A.*B/ and /.*B+C/. Patterns aren't limited to be matched at the start of the input string. If we remove the ".*" from the beginning of the patterns, the DFA and δFA shown in Figure 2 have less states and transitions and so the memory requirement is lower.

A scheme called Parallel Failureless-AC (PFAC)[4] is a multi-threading pattern match algorithm designed for Graphic Processing Unit (GPU). PFAC assigns a thread to each symbol of input string to start its matching process. Every thread executes PFAC algorithm to find a matched pattern beginning at its starting position. If a thread cannot find a match, it terminates immediately. We can use the same technique to search with this δFA for reducing memory consumption and parallelization. The problem of this scheme is that it may have more than one active states in search time, so we have to maintain a search table recording entire transition information for each active state in the same time. These search table would consume lots of memory, so we need to solve this problem which make the search process quite slow.

In the DFA of a pattern set without leading '.*', there are more identical transitions in one state. Most input symbols will forward to the same next state in transition table. δFA, this scheme stores the transition table of a state by only recording the distinct part from its parent. In the proposed CSDFA, we removes all the transitions having the same next state as its former by using run length encoding. For example, there is a state which will transfer to the same next state with input symbol 'A' and 'B', we can remove the symbol 'B' in the transition table. Like δFA, CSDFA has a 256-bit bitmap to record whether the next state is changed. If the next state with an input symbol is different from its former transitions, the corresponding bit in bitmap table will be set to be 1. Figure 3 shows the data structure of transitions in a state. Symbol 'A' and 'B' share the same next state, so we remove the row with symbol 'B' of transition table, and set corresponding position of bitmap bm_x to 1 and 0, respectively. Symbol 'D' and 'E' have no forward link and so they are not recorded in the original transition table. We use another bitmap bm_y to record that Symbol 'D' and 'E' have no forward link and so we set the positions 68 and 69 to be 0 in this example.

Unlike δFA, CSDFA isn't required to maintain a search table to complete the search operations. CSDFA searches bitmap bm_y first. If the bit corresponding to the search symbol is 0, it means there is no transition with this symbol and the search process terminates immediately as in PFAC. However, if it is 1, we then access bitmap bm_x to compute the number of set bits at and before the bit position corresponding to the search symbol. Then we know the index of the compressed transition table from which we can obtain the next state.

3.2 CSDFA with pattern segmentation

Given an NFA with N states, the corresponding DFA may consist of 2^N states in the worst case theoretically. Although this upper bound isn't reached in most cases, the number of states in some conditions may be close to the

Symbol	Next state
65 (A)	1
66 (B)	1
67 (C)	3
70 (F)	2
71 (G)	2

(A) Original transition table.

Symbol	Next state
65 (A)	1
67 (C)	3
70 (F)	2

	65	66	67	68	69	70	71
bm_x	1	0	1	0	0	1	0
bm_y	1	1	1	0	0	1	1

(B) Transition table in CSDFA.

Figure 3. The transition table and bitmap table of a state.

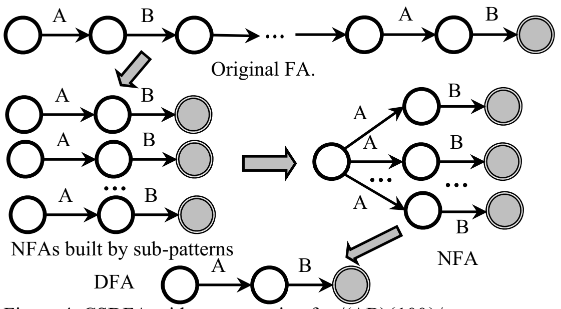

Figure 4. CSDFA with segmentation for /(AB){100}/.

worst case. We will focus on two conditions occurring frequently in pattern sets, namely "counting constraint" and "Kleene star (.*)".

A counting constraint condition is a repetition, that may occur in regular expressions such as /.{n, m}/ and /.{n}/. It can lead to state blowup during construction of FAs, even when the corresponding regular expression is compiled separately. It wastes memory to create states for the same atoms or sub patterns. So we segment the patterns into several pieces and convert each piece to an NFA separately. Then we concatenate all these NFAs by merging their initial states. Finally, this merged NFA is converted into DFA by the subset construction algorithm.

Figure 4 shows the building process for pattern /(AB){100}/. The purpose of this scheme is to merge similar parts of FAs for reducing the states and transitions. When we convert the NFA into a DFA, we also merge the states and transitions of the NFA branches corresponding the same sub-patterns. The benefits of this scheme doesn't require the pattern must be sequential or in a single pattern.

The Kleene star condition means zero or more times of repetitions for some atoms. It may lead to state blowup when converting NFA to DFA. Because in subset construction, every subset of NFA states containing self-loop states caused by Kleene star will affect all the following subsets, a lot of new DFA states will be created. It often happens in building for multiple patterns. Pattern segmentation may also lead the same problem in some rare cases when building DFA. Take an example of pattern /ABC.*ABDE/ shown in Figure 5. The final DFA in this example has more states and transitions than the original one. We can handle this problem by adjusting the cutting position of pattern segmentation to make all the self-loop states have less opportunity for being merged by other states.

Before we start to describe the details of our pattern segmentation, we define the term "depth" for NFA to be the maximum number of next states from initial state to the end states without any revisited state. The main purpose of our pattern segmentation is to reduce memory consumption. So we want to increase the opportunity of merging the states of NFA built from the sub-patterns segmented from original patterns. For this reason, the depths of these NFAs cannot be too different. It is inefficient to merge states if the some branches of NFA is much longer than the others. We use the maximum depth limit in our segmentation process as follows. We compute the depth of NFA for each possible sub-pattern. For above example, we will test "A" as the sub-pattern, compute the depth of the corresponding NFA, and check whether the depth reaches the maximum depth limit we set. If so, cut off the sub-pattern we have passed. If not, test "AB", "ABC", "ABC.*" and so on until the depth of corresponding NFA reaches

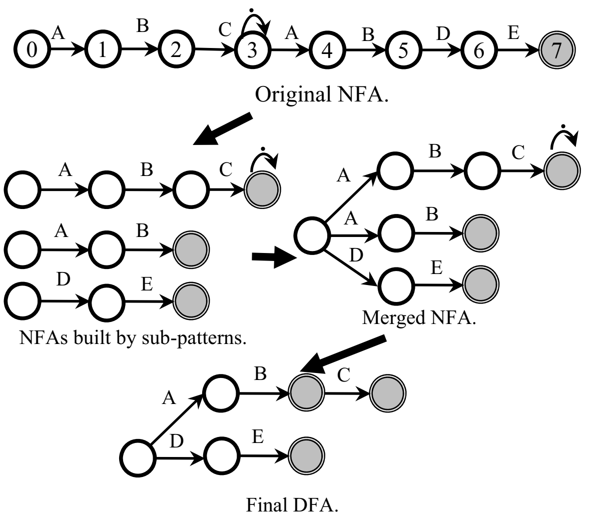

Figure 5. The automata for pattern /ABC.*ABDE/ in CSDFA.

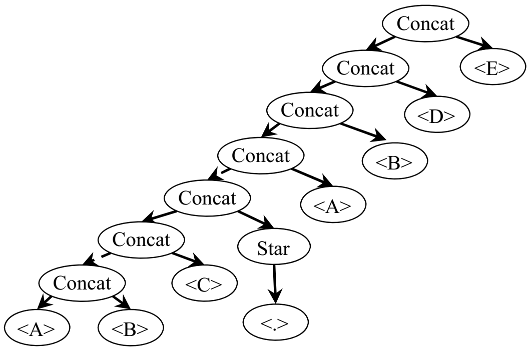

Figure 6. A parse tree for pattern /ABC.*ABDE/.

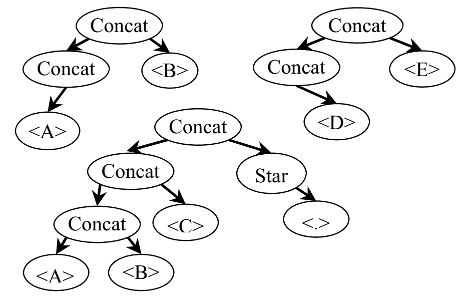

Figure 7. Sub-parse trees for pattern /ABC.*ABDE/.

Table 1. Computing the depth of NFA from the parse tree.

Node type	Depth
Symbol	1
Concat	Sum of all the children's depths
Alter	Return the largest depth of all children's depths.
Star	0

Table 2. Detailed characteristics of pattern sets.

	Snort25	Snort276	Bro217
# of patterns	25	276	217
ASCII length range	18-202	18-202	5-211
% of wildcards	60%	33%	1.38%
% of repetitions	28%	26%	0.46%

the maximum depth limit.

In our scheme, we will also cut off the pattern right after the Kleene star. In this case, the sub-pattern which has Kleene star will be appended to the prior sub-pattern to a longer one by ignoring the constraint of maximum depth limit. Figure 5 shows the NFAs built from segmented sub-patterns. In this case, we set the maximum depth to two, so the depth of each NFA doesn't exceed two except the NFA with self-loop states. As we can see, the segmented NFA having self-loop state is longer than other segmented NFAs and so it has less opportunity to make state explosion happen. Figure 5 also shows the final DFA for pattern /ABC.*ABDE/. Because the self-loop state is at the end of the branch of the NFA and this branch is longer than other branches, the state explosion doesn't happen in final DFA.

In practical implementation, we will convert the pattern to a parse tree which is the intermediate product of building NFA. The parse tree representation of a pattern is described subsequently. The parse tree is used to help us to know the relationship of atoms in a pattern. For example, we know that the repetition in pattern /(AB){100}/ is for "AB", not for "B". The parse tree built for pattern /ABC.*ABDE/ is shown in Figure 6.

We compute the depth of each node based on the node type defined in Table 1, and cut off the parse tree to be a new sub parse tree when the corresponding NFA's depth reaches the maximum limit or meet a Star Node. With the same idea, the sub-parse tree with Star node will be concatenated to their prior sub parse tree into a bigger one. The result of sub parse trees which we set the maximum depth to be two for above example is shown in Figure 7. Each of the sub parse trees will be converted to an NFA separately by using Glushkov's construction. All NFAs are concatenated into a single NFA by merging the initial states of these NFAs. Then the merged NFA is converted into DFA which is the same as the one shown in Figure 5.

Every pattern has different properties. Therefore, it is not easy to determine the best max depth. Some patterns with max depth of two are better, but some are not. For example, the best max depth is three for pattern /ABCABCABCABC/. We propose to test all the possible max depths, and find the best one that leads to the least number of the DFA's states. Finally, all the DFAs built from each pattern will be merged into a final DFA for search.

Because the pattern have been split into several segments, we need a mark called Sub Pattern ID (SPID) to identify its order in the original pattern. We also need to record the number of sub-patterns for each pattern. When we use DFA, we propose to search the patterns and find if the current state is accepted. We check not only the Pattern ID (PID) but also the SPID. We use the data structure called "Search State" to record current DFA state ID, SPID, and PID. In the beginning of search process, the 3-tuple of current state, SPID and PID is set to (0, Empty, Empty), where DFA's initial state ID to be zero. The next state is decided by the current state of the search state and input symbol. When the current DFA state is accepted and its corresponding SPID is zero, the current state of search state is back to the DFA's initial state again, SPID is incremented by one, and PID is set to be the pattern's ID we just met. From now on, this search state only accepts the state which matches the same SPID and PID. Repeat this process until the SPID of search state becomes the largest sub-pattern ID minus one of this pattern. There may have more than one searchstates when we reaches the accepted state in DFA, so we store these search states by a list called "slist".

4. Experiment results

In this section, we will show our simulation environment and the performance evaluation. We set up several experiments with different rule sets from Snort and Bro. In addition, we also compare the memory overhead and speed performance. All the experiments are executed in a single machine. The machine and environment specifications are Intel(R) Core(TM) i7-4770K CPU of 3.50GHz with 32GB DDR3, Ubuntu 14.04 LTS 64bit, and GCC 4.6.3 Compiler.

Three pattern sets are employed in our experiments. The first set called Snort25 contains 25 distinct regular expressions randomly selected from Snort version 2.9.5.6 virus database. Although Snort25 is small, it allows us to build its DFA and δFA for comparison purpose with various schemes. The second set called Bro217[19] consists of 217 patterns from Bro[2]. The third pattern set called Snort276 consists of 276 regular expressions also selected from Snort version 2.9.5.6 virus database by removing many redundant regular expressions. The information of the pattern sets are shown in Table 2, where we list the number of patterns, the ASCII code length range, the percentage of patterns including "wildcard symbols" (i.e. *, +, ?), and the percentage of pattern including repetitions with {n} or {n, m} format.

We compare the number of states, number of transitions, and average number of transitions for each state, and memory consumption in Table 3. The first three rows show the results for the patterns with ".*" prepended in the beginning, and the last four rows are the results for the patterns without ".*" in the beginning. As we can see, if the patterns start with ".*", they have larger numbers of states and transitions than that with no ".*" in the beginning. The memory required for the DFA of patterns starting with ".*" is about 2,485 times more than that without leading ".*" for pattern set snort25 in Table 3(a). We can see that the numbers of states and transitions in the last four rows are significantly smaller than that of the first three rows. The memory consumption in δFA is much smaller than the original DFA. δFA doesn't decrease the number of states, but reduces 98% of transitions needed in DFA. The number of transitions in CSDFA is significantly less than that needed in δFA. In fact, CSDFA needs only 73% memory consumed by δFA if we remove the ".*" from the beginning of patterns. The last row shows the results for the proposed CSDFA enhanced with technique of pattern segmentation. The performance of memory consumption in CSDFA with segmentation is better than the CSDFA without segmentation. CSDFA with segmentation improves the memory efficiency by reducing the numbers of states and transitions. Table 3(b) shows the memory consumption in Bro217 pattern set. This pattern set is simpler than Snort's, because its numbers of states and transitions are smaller. We still can see the performance in CSDFA is much better than δFA, since CSDFA needs only 16% of the memory consumed by δFA. CSDFA with segmentation is even better than CSDFA. In Figure 8, we also show the ratio of the number of transitions of δFA and the proposed CSDFA scheme over DFA, using Snort27 and Bro217 pattern sets. As we see, the number of states in δFA may have poor performance if we remove the ".*"

Table 3. Memory consumption.

	# of states	# of trans	Avg # of trans	Mem (KB)
Glushkov NFA	923	8,619	9.3	57
DFA	1,246,200	319,027,200	256.0	1,577,409
δFA	1,246,200	67,093,374	53.8	386,207
DFA, w/o .*	1,264	125,931	99.6	635
δFA, w/o .*	1,264	14,759	11.7	132
CS-DFA	1,264	7,658	6.1	97
CS-DFA with segmentation	1,108	5,277	4.76	78

(a) Snort25

	# of states	# of trans	Avg # of trans	Mem (KB)
Glushkov NFA	2,710	9,915	3.7	91
DFA	6,669	1,707,264	256	8,441
δFA	6,669	118,347	17.8	891
DFA, w/o .*	2,229	189,702	85.1	961
δFA, w/o .*	2,229	177,808	79.8	973
CS-DFA	2,229	9,658	4.3	152
CS-DFA with segmentation	1,874	4,437	2.4	114

(b) Bro217

	# of states	# of trans	Avg # of trans	Mem (KB)
DFA, w/o .*	16,464	429,254	26.1	2,355
δFA, w/o .*	16,464	154,765	9.4	1,530
CS-DFA	16,464	46,032	2.8	999
CS-DFA With segmentation	5,684	21,114	0.26	452

(c) Snort276

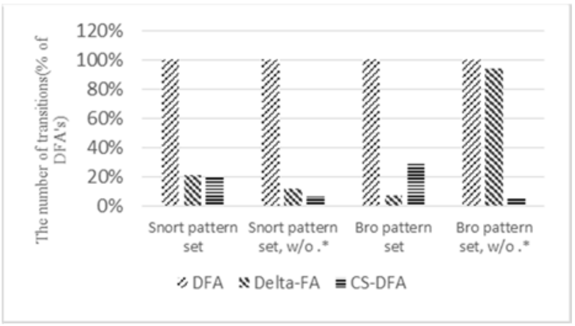

Figure 8. The number of transitions comparing DFA's in Snort's and Bro's pattern set.

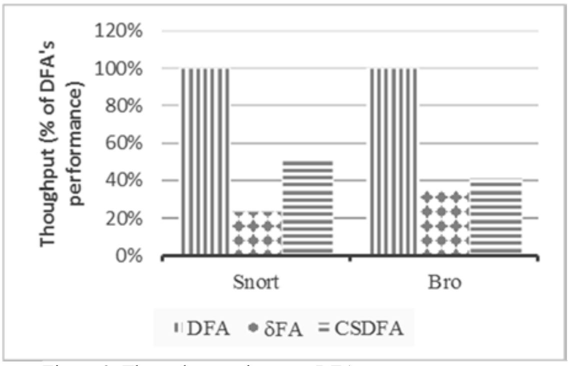

Figure 9. Throughput ratios over DFA.

from the beginning of patterns. For example, in Figure 8, the number of transitions in δFA has only 6% reduction compared to DFA in Bro217. The similar results for pattern set snort276 are shown in Table 3(c) where the results for patterns with '.*' prepended in the beginning cannot be computed due to state blowout problem. Figure 9 compare the throughputs of the proposed CSDFA and δFA in terms of throughput ratio over original DFA. The proposed CSDFA is much faster than δFA although both CSDFA and δFA are slower than original DFA.

We have compared the memory consumption of DFA, δFA, and CSDFA for the patterns with and without ".*" prepended. We show that these schemes for the patterns without ".*"actually requires smaller memory consumption. Without ".*", we can even use a larger pattern set to do the experiment. Table 3 shows the memory consumption in a larger Snort pattern set consisting of 276 patterns based on Snort. The technique of pattern segmentation perform very well for this larger patterns. As we can see, the CSDFA with segmentation has a better performance, since it needs only a half of the memory consumed by CSDFA without segmentation.

5. Conclusion

In this paper, we proposed a memory efficient parallel algorithm called CSDFA for regular expressions that uses compression and pattern segmentation. We focus on two conditions "Counting Constraints" and "Kleene Star" which cause state blowup frequently in pattern sets. From the idea of PFAC algorithm, CSDFA has less states and transitions than original δFA. Based on the observation that a DFA state transitions for most symbols go to the same next state. As a result, the transition table can be compressed by run-length encoding. The number of transitions in the proposed CSDFA is half of the transitions needed in δFA, and needs 74% of the memory consumed by δFA

References

1. Snort. [Online]: http://www.snort.org
2. Bro. [Online]: http://www.bro.org/
3. E. Knuth, J. H. M. Jr., V. R. Pratt, "Fast pattern matching in strings," SIAM Journal on Computing, vol. 6, no. 2, pp.323–350, 1977.
4. C.-H.Lin, C.-H. Liu, L.-S. Chien, S.C. Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs," IEEE Transactions on Computers, vol. PP, p.1, 2012.
5. D Ficara, S Giordano, G Procissi, F Vitucci, G Antichi, A Di Pietro, "An improved DFA for fast regular expression matching, "ACM SIGCOMM Computer Communication Review 38 (5), 29-40, 2008.
6. OpenMP Application Program Interface Version 3.1 May 2008, openmp.org.
7. V. Aho, M. J. Corasick, "Efficient string matching: An aid to bibliography search" Communications of the ACM, vol. 18, no.6, pp.333-340, 1975.
8. R. S. Boyer, J. S. Moore, "A fast string searching algorithm," Communications of the ACM, vol. 20, no 10, pp.762-772, Oct. 1977.
9. V.M. Glushkov, "The abstract theory of automata," Russian Math. Surveys, 16 (1961), pp. 1–53.
10. Ken Thompson, "Programming Techniques: Regular expression search algorithm,“ Communications of the ACM, v.11 n.6, p.419-422, June 1968.
11. M. O. Rabin, D. Scott. "Finite automata and their decision problems," IBM Journal of Research and Development, 3(2):114–125, 1959.
12. R. Smith , C. Estan , S. Jha , S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata,“ SIGCOMM Comput. Commu. Rev., vol. 38, no. 4, pp. 207-218, 2008.
13. S. Kumar , B. Chandrasekaran , J. Turner , G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in Proc. ACM ANCS, 2007, pp. 155-164.
14. S. Kumar , S. Dharmapurikar , F. Yu , P. Crowley , J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," Proc. ACM SIGCOMM, 2006, pp. 339-350.
15. N. Yamagaki, R. Sidhu, S. Kamiya, "High-Speed Regular Expression Matching Engine Using Multi-Character NFA," in Proc. Intl. Conf. FPL, Aug. 2008, pp. 697–701.
16. J. E. Hopcroft, R. Motwani, J. D. Ullman, "Introduction to Automata Theory, Languages and Computability, 2nd Edition," Addison-Wesley, Nov. 2000.
17. Alfred V. Aho, Ravi Sethi, Jeffrey Ullman, "Compilers: Principles, Techniques and Tools", Addison Wesley, 1986
18. R. McNaughton, H. Yamada, "Regular Expressions and State Graphs for Automata". IEEE Trans. on Electronic Computers 9 (1): 39–47, Mar 1960.
19. Regular Expression Processor. [Online]. Available: http://regex.wustl.edu/index.php/Main_Page.
20. Sun Wu, Udi Manber, "A fast algorithm For Multi-Pattern Searching," Technical Report TR 94-17, University of Arizona at Tuscon, 1994.