SDFA: Series DFA for Memory-Efficient Regular Expression Matching*

Tingwen Liu[1,2], Yong Sun[1,3], Li Guo[1,3], and Binxing Fang[3]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] Graduate University of Chinese Academy of Sciences, Beijing, China
[3] National Engineering Laboratory for Information Security Technologies, Beijing
{liutingwen,suny}@software.ict.ac.cn

Abstract. Regular expression (RegEx) matching plays an important role in various network, security and database applications. Deterministic finite automata (DFA) is the preferred representation to achieve online RegEx matching in backbone networks, because of its one single pass over inputs for multiple RegExes and guaranteed performance of $O(1)$ memory bandwidth per symbol. However, DFA may occupy prohibitive amounts of memory due to the explosive growth in its state size. In this work, we propose Series DFA (SDFA) to address the problem. The main idea is to cut a complex RegEx into several ordered and small RegExes carefully, and then concatenate their compact DFAs in series to match. Experimental results show that SDFA can achieve significant reduction in memory size at the cost of limited number of memory bandwidth.

1 Introduction

Deep Packet Inspection (DPI), which searches for predefined signatures over the content of packet payloads, is considered as a powerful and important method in network and security applications. Recently regular expressions (RegExes) are replacing exact strings as the de facto standard to specify signatures in most open-source tools [9, 6] and commercial devices. The primary reason is the expressive power, simplicity and flexibility of RegExes. Deterministic Finite Automata (DFA) is an ideal representation for high-speed RegEx matching, because multiple RegExes can be compiled into a composite DFA that performs matching over inputs in a single pass with a guaranteed robust performance of $O(1)$ memory bandwidth per byte. However, the composite DFA constructed for real-world RegEx sets may experience state explosion, as a result it usually consumes prohibitive amounts of memory.

In this paper, we focus on state reduction by cutting complex RegExes into well-designed and ordered RegEx fragments that can be compiled into compact DFAs. To match equivalently as uncutted RegExes, we propose Series DFA

* Supported by the National High-Tech Research and Development Plan of China under Grant No. 2011AA010703; the National Natural Science Foundation of China under Grant No. 61070026 and No. 61003295.

N. Moreira and R. Reis (Eds.): CIAA 2012, LNCS 7381, pp. 337–344, 2012.

(SDFA) that concatenates the compact DFAs with epsilon transitions in the order of their appearance. We further introduce some optimizations to improve the memory consumption and memory bandwidth of SDFA. Different from prior work [1], SDFA works over RegExes directly to achieve the reduction of states, which makes it being constructed easily and quickly even for large-scale RegEx sets. We perform a systematic experimental study on real RegEx sets and our synthetic RegEx set. The results show that SDFA achieves significant memory reduction, and shows almost the same matching speed comparing with the composite DFA.

2 Related Work

With the widespread use of RegExes in various applications, research interests focus on designing data structures, algorithms and architectures to support fast and memory-efficient RegEx matching. In this context, how to reduce the huge memory consumption is the hotspot of related researches for those matching solutions based on DFAs. In general, prior work can be classified into three categories: *DFA compression*, *partial determinization* and *history auxiliary*.

DFA compression solutions try to achieve memory reduction by compressing the transition table for a given DFA [5, 3, 8, 7]. They are based on the observation of many common values in the table. However, the memory usage, which have been reduced by 95% after compressing, are still very huge as the composite DFA for real RegEx sets usually costs multiple terabytes. These solutions are orthogonal to our work and can be used to compress the compact DFAs in SDFA.

Partial determinization solutions address the problem by constructing hybrid automata [1] or multiple parallel DFAs [13, 10] at the cost of determinacy by allowing multiple states active during the matching process. Our work improves upon these solutions because our DFAs constructed for the cutting RegEx fragments are compact enough, and are activated when necessary.

History auxiliary solutions introduce counters, queues and other data structures as auxiliary memory to avoid duplication of states by recording matching history [4, 11]. However, the benefit of state reduction does not come for free. They either experience an exponential growth in the size of auxiliary memory, or require much time to update auxiliary memory after processing each symbol.

3 Technical Overview of Series DFA

3.1 State Complexity for RegExes

An analysis of state complexity for DFA of individual RegEx that does not have OR relationship (`|`) is represented in [13]. Here we consider RegExes in the combination of `^`, one unconstrained repetition `*` and one constrained repetition (three types: fixed repetition `{j}`, range repetition `{j,i}` and at-least repetition `{j,}`) of wildcards, and the detail is shown in Table 1.

Table 1. State complexity for individual RegEx with k characters

RegEx Feature	Example	State Complexity
without constrained repetitions of wildcards	`^abcd`, `abcd` `^ab.*cd`, `ab.*cd`	$O(k)$
with `^`, one fixed or one at-least repetition	`^ab.{j}cd` `^ab.{j,}cd`	$O(k+j)$
with `^` and one range repetition	`^ab.{j,i}cd`	$O(k(i-j))$
with only one fixed or one at-least repetition	`ab.{j}cd` `ab.{j,}cd`	$O(k+2^j)$
with only one range repetition	`ab.{j,i}cd`	$O(k(i-j)+2^i)$

From the table, we can find that unconstrained repetitions do not cause state explosion when individual RegEx is compiled into a DFA in isolation (case 1). Constrained repetitions of wildcards lead to exponential growth in DFA state size for individual RegEx not starting with `^` (case 4 and case 5). Because the DFA needs to record the prefix part within each wildcard. The situation becomes even worse when multiple RegExes with constrained repetitions are compiled together into a composite DFA. Because there are more combinations of prefixes and more wildcards in these RegExes.

By comparison, RegExes of the former three cases do not result in a large DFA. Therefore, if we cut RegExes of the latter two cases into multiple RegEx fragments of the former three cases, we can construct a compact DFA for each fragment. In this paper, we investigate its feasibility to reduce DFA state size.

3.2 Main Idea of SDFA

In order to facilitate description, we call a RegEx as its fragments' *father*, each fragment as its *son*. For a given RegEx, the first (last) fragment is called its *eldestson* (*youngestson*), correspondingly other fragments are *non-eldestsons* (*non-youngestsons*). To match multiple RegExes together in a single pass, all the eldestsons are compiled into a composite DFA, and each non-eldestson is compiled into an individual DFA. SDFA organizes all the DFAs in series and perform matching in the follow way: at the beginning only the initial state of the composite DFA is active, all the individual DFAs are sleep; SDFA will add a new instance of the initial state of one individual DFA when its preceding DFA matches successfully, and delete an instance when it moves to the dead-state.

We use an example of two RegExes `ba[^a]*bad.{2}cd` and `de[^e]{3}` to show how SDFA works in detail. It first locates all unconstrained and constrained repetitions in the two RegExes, and then cut them into five fragments: `ba`, `^[^a]*bad`, `^.{2}cd`, `de`, `^[^e]{3}` at these positions. Note that all the non-eldestsons begin with ^, because a fragment begins to match from position $j+1$ of input string only when its preceding fragment matches successfully at position j. Fragments `ba` and `de`, which are the eldestsons of the two RegExes, are compiled into a composite DFA. Now we describe how to construct a SDFA with the

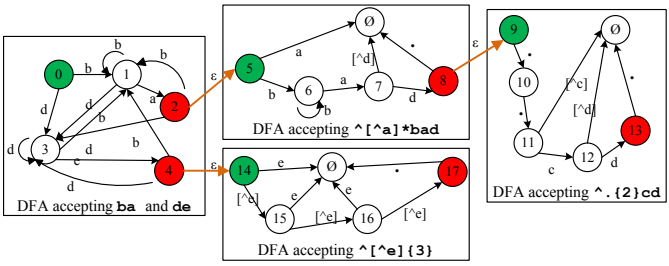

Fig. 1. SDFA accepting `ba[^a]*bad.{2}cd` and `de[^e]{3}`. For each DFA, the state in green (red) is its initial (accepting) state. Transitions to the initial states are omitted.

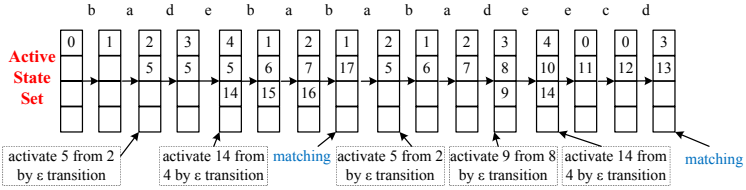

Fig. 2. SDFA traversal with input `badebababadeecd`

four DFAs, as shown in Fig. 1. The initial state of the composite DFA (state 0) is the initial state of the SDFA, and the accepting states of DFAs constructed for the youngestsons (state 13 and 17) are the accepting state of the SDFA. For the accepting states of the other DFAs, adding an epsilon (ϵ) transition that does not consume any symbol to the initial state of the DFA constructed for its following brother. As shown in Fig. 1, the SDFA accepting `ba[^a]*bad.{2}cd` and `de[^e]{3}` has 21 states, while the state-minimized DFA has 58 states (omitted here for readability).

In Fig. 2, we show the matching process of the SDFA in Fig. 1 over input string `badebababadeecd`. For example, fragment `de` is matched two times at the fourth and the twelfth symbol, and then SDFA activates state 14 along an epsilon transition. The first activation reports a successful match of `ba[^a]*bad.{2}cd` after processing the seventh symbol, while the second de-activates immediately because state 14 moves to the dead-state along the next symbol `e`.

4 Optimization for Series DFA

Essentially SDFA trades memory size (size of states) with memory bandwidth (size of *active state set*). In this section, we propose some techniques to optimize the two metrics by improving the cutting process and matching process of SDFA.

4.1 Optimization in Cutting Process

Determining the cutting positions is the main challenge for the construction of good SDFA. Cutting at the repetitions of any character range will have low

memory size but high memory bandwidth as each fragment is too short. In contrast, cutting only at the repetitions of wildcards will have low memory bandwidth but high memory bandwidth. Here we give a simple but striking way to finish the determination quantitatively. We define the number of characters allowed in a character range as its size. Then we introduce a threshold μ: if the size of a character range is more than μ, we think the range is large enough to be cut at the positions of its repetitions. When μ is set to 256, the SDFA is essentially a composite DFA for the complete RegEx set because no RegEx is cut.

Furthermore, to obtain good SDFA, the cutting process should comply with the following three rules. In fact, we can also consider these rules trying to combine several adjacent fragments into one.

Rule 1: No constrained repetitions or unconstrained repetitions in any eldestson. Because repetitions of large character ranges need to duplicate states to record all possible prefixes when multiple RegExes are compiled together as mentioned before. Therefore the composite DFA constructed for eldestsons that violate this rule will experience state explosion.

Rule 2: No constrained repetitions after unconstrained repetitions in each fragment. Obviously eldestsons that satisfy rule 1 also follow this rule. For each non-eldestson, if it violates this rule, it may belong to case 4 or case 5 in Table 1, and cause exponential growth of state size in the worst case.

Rule 3: No constrained repetitions or unconstrained repetitions after range repetitions in each fragment. All the eldestsons also follow this rule just as described above. Any non-eldestson failing to comply with this rule falls into case 3 in Table 1, whose complexity is product.

These rules allow non-eldestsons to have more than one unconstrained repetitions. One vivid example is RegEx `Cookie\s+Monster\s+server\s+engine` in Snort system. It can be cut into fragments `Cookie` and `^s+\s+Monster\s+server\s+engine` if set μ no less than the size of `\s`.

The point that need to be made is that these rules are sufficient conditions but not necessary conditions to combine adjacent fragments. An example is RegEx `ba[^a]*bad.{2}cd` in Fig. 1. The fragment `^[^a]*bad.{2}cd` obviously violate Rule 2, however its DFA does not experience exponential growth in state size. Because the occurrence of `a` makes `^[^a]*` fails to consume `bad`, as a result the DFA needn't to take into consideration that `bad` may appear in the constrained repetition `.{2}`. Snort and other intrusion detection systems have many RegExes of this type, for example `\/OvCgi\/[^\.]*\.exe[^\x20]{2000,}`.

4.2 Optimization in Matching Process

Most DPI applications such as Snort and L7-Filter are only interested in knowing the set of patterns to be fired by a packet. We call this type of matching as left-most matching, which is formally defined as below.

Left-Most Matching: Consider the matching process M as a function from a pattern P and a string S to a power set of S, such that, $M(P, S) = \{$substring S' of $S|S'$ is the left-most substring which is accepted by the DFA of $P\}$.

Table 2. Primary information of experimental RegEx sets ($\mu = 1$)

RegEx set	# of RegExes	% of * repetitions	% of {} repetitions	min-length range	# of NFA states	# of 7-DFA states
l7filter	107	46.7	21.5	1–76	3325	29047
backdoor	158	36.1	1.3	2–77	3580	6164
synset	300	59	18.7	11-225	19751	$> 10^6$

This specialty can be exploited to decrease memory bandwidth. As left-most matching is enough to know the fired RegExes, once a RegEx is reported it is safe to set its all non-eldestson DFAs inactive forever. To our knowledge, SDFA is the first automata that uses left-most matching to improve matching process. Because all kinds of previous methods must go through the step of constructing a sort of composite finite automata for the complete RegEx set. When a RegEx is matched, they cannot guarantee that the states that have been traversed by the RegEx will not be accessed by other RegExes. On the contrary, SDFA is able to ensure that the fragment DFAs of one RegEx will never be accessed by other RegExes. For the same reason, the composite DFA in SDFA needs to have an always active instance.

5 Experimental Results

We design three representative RegEx sets, as shown in Table 2. Column 3 (4) is the percent of RegExes containing constrained (unconstrained) repetitions of character ranges in each set. The first RegEx set is extracted from L7-Filter [6] system, and the second set is from *backdoor* rule file in Snort [9] system. The third RegEx set is generated by open-source RegEx generator [2]. As shown in column 7, the three RegEx sets can be compiled into 7 DFAs of 29047, 6164 and more than 10^6 states respectively with multiple parallel DFAs [13].

We make experiments using two real traffic traces from different links: one trace named *download* is downloaded from [12], the size is 254 MB; the other trace named *capture* is captured in the interface of a backbone network, the size is 1,538 MB. We also generate some synthetic traces of 50 MB with open-source trace generator [2] under $p_m = \{0, 0.15, 0.3, 0.45, 0.6, 0.75, 0.9\}$. Value p_m is used to model the likelihood of experiencing malicious traffic.

5.1 Evaluation of Memory Consumption

In this section, we use the size of DFA states to evaluate memory consumption of SDFA for the three RegEx sets. Table 3 shows the summary of state size for different values of μ. We can draw the following conclusions from Table 3.

First, DFA-based solutions are infeasible to perform matching for large RegEx sets containing constrained repetitions and unconstrained repetitions. As mentioned before, SDFA is in fact a composite DFA when μ is 256. However, the state size is more than 10^7 (inf) in this case for all experimental RegEx sets.

Table 3. State size of SDFA on varying μ

value of μ	state size of composite DFA / state sums of individual DFAs / # of DFAs		
	l7filter	backdoor	synset
1	5689 / 3293 / 103	2034 / 3009/144	9507 / 20322 / 321
64	6438 / 3246 / 93	45072 / 1451/58	*inf* / *inf* / 173
128	*inf* / 2618 / 56	45072 / 1451/58	*inf* / *inf* / 173
256	*inf* / 0 / 1	*inf* / 0 / 1	*inf* / 0 / 1

Second, the number of DFAs decreases as the increase of μ while the state size of the composite DFA grows with μ. The primary reason is that high μ makes some character ranges become small, and SDFA does not cut RegExes at the occurrence of unconstrained repetitions and constrained repetitions of small character ranges. As a result, the eldestsons have more symbols especially more repetitions, which lead to the rapid increase in the state size of the composite DFA. However, the sum of states in individual DFAs appears complexly. The primary reason is that constrained repetitions may appear in the middle of non-eldestsons for large μ, which results in exponential growth in state size even for an individual DFA.

Third, SDFA can greatly reduce memory consumption. When μ is 1, the three SDFAs have 8982, 5043 and 29379 states respectively in all, which can be encoded in on-chip memory directly even without compression. The result is closed to that of NFA, and better than that of multiple parallel DFAs (7-DFA).

5.2 Evaluation of Matching Performance

In this section, we evaluate matching performance of SDFA, which is measured by the size of active state sets. We construct a SDFA for each given RegEx set with $\mu = 1$, and observe its active set size on real traces and synthetic traces in average case and maximum case. In fact, both the average size and the maximum size of active sets increase with μ. When $\mu = 1$, SDFA has the worst average size and maximum size, because it cuts RegExes at the occurrence of repetitions of any character range, as a result fragments are matched frequently.

The result of backdoor set on its synthetic traces is shown in Fig. 3. As l7filter and synset have the similar behavior, we omit them here due to page limitation. We can find that: First, left-most matching can really improve the matching performance of SDFA, especially in average size. Second, the average size grows slowly with the increase of p_m, while the change of maximum size is uncertain.

Fig. 4 shows the results of active set size on real traffic traces for each RegEx set. Each connection carries an application protocol, so almost every packets can be matched by RegExes in l7filter set. As a result, we can regard the behavior of l7filter set as the performance of SDFA under an attack. From Fig. 4 we can find that SDFA works well under attacks, although the maximum size is a little big. The average active set size of SDFA is very close to that of a composite DFA. As each RegEx set can be constructed into 7 DFAs, its average size and

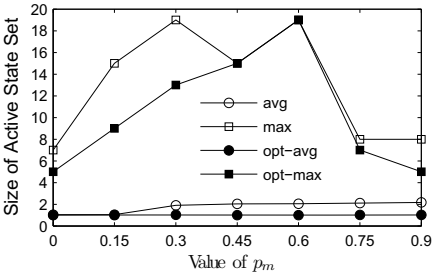

Fig. 3. Size of active state sets for backdoor set on its synthetic traces

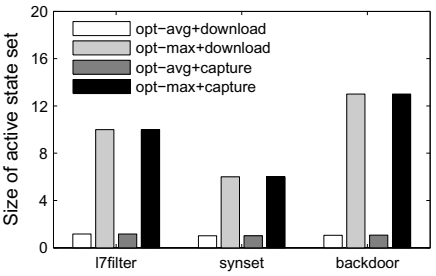

Fig. 4. Size of active state sets for three experimental RegEx sets on real traces

maximum size are both 7. Obviously SDFA is suitable to perform large-scale RegEx matching in different high-speed network environments.

References

1. Becchi, M., Crowley, P.: A Hybrid Finite Automaton for Practical Deep Packet Inspection. In: Proc. ACM CoNEXT Conference, pp. 1–12 (2007)
2. Becchi, M.: Regular Expression Processor, `http://regex.wustl.edu/`
3. Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G., Di Pietro, A.: An Improved DFA for Fast Regular Expression Matching. ACM SIGCOMM Computer Communication Review 38(5), 29–40 (2008)
4. Kumar, S., Chandrasekaran, B., Turner, J., Varghese, G.: Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia. In: Proc. ACM/IEEE ANCS, pp. 155–164 (2007)
5. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In: Proc. ACM SIGCOMM, pp. 339–350 (2006)
6. Levandoski, J., Sommer, E., Strait, M.: Application Layer Packet Classifier for Linux, `http://l7-filter.sourceforge.net/`
7. Liu, T., Yang, Y., Liu, Y., Sun, Y., Guo, L.: An Efficient Regular Expressions Compression Algorithm From A New Perspective. In: Proc. IEEE INFOCOM, pp. 2129–2137 (2011)
8. Liu, Y., Guo, L., Liu, P., Tan, J.: Compressing Regular Expressions' DFA Table by Matrix Decomposition. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 282–289. Springer, Heidelberg (2011)
9. Roesch, M.: Snort - Lightweight Intrusion Detection for Networks. In: Proc. USENIX LISA, pp. 229–238 (1999)
10. Rohrer, J., Atasu, K., van Lunteren, J., Hagleitner, C.: Memory-Efficient Distribution of Regular Expressions for Fast Deep Packet Inspection. In: Proc. IEEE/ACM CODES+ISSS, pp. 147–154 (2009)
11. Smith, R., Estan, C., Jha, S.: XFA: Faster Signature Matching with Extended Automata. In: Proc. IEEE S&P, pp. 187–201 (2008)
12. The Shmoo Group: Internet Traffic Traces, `http://cctf.shmoo.com/`
13. Yu, F., Chen, Z., Diao, Y., Lakshman, T.V., Katz, R.H.: Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In: Proc. ACM/IEEE ANCS, pp. 93–102 (2006)