An Efficient Pre-filter to Accelerate Regular Expression Matching

Chengcheng Xu[1(✉)], Shuhui Chen[1], Xiaofeng Wang[1], and Jinshu Su[1,2]

[1] College of Computer, National University of Defense Technology, 410073 Changsha, China
{xuchengcheng,shchen,xf_wang,sjs}@nudt.edu.cn
[2] National Key Laboratory for Parallel and Distributed Processing, National University of Defense Technology, 410073 Changsha, China

Abstract. Regular expression matching is widely used in content-aware applications, such as NIDS and protocol identification. However, wire-speed processing for large scale patterns still remains a great challenge in practice. Considering low hit rates in NIDS, a compact and efficient pre-filter is firstly proposed to filter most normal traffics and leave few suspicious traffics for further pattern matching. Experiment results show that, the pre-filter achieves a big improvement in both space and time consumption with its compact and efficient structure.

Keywords: Regular expression matching · Pattern matching · Deep packet inspection · DPI · Pre-filter

1 Introduction

With the rapid growth in big data, massive network data needs to be captured and analyzed in real-time, which poses great challenges to traditional network security field, especially for Deep Packet Inspection (DPI) [7]. DPI requires to inspect the packet payload for further processing, which also needs real-time acquisition and analysis for massive network data. Currently, DPI is widely used in load balancing, traffic billing, Network Instruction Detection (NIDS) and protocol identification. The inspection process is to match the stream or packet payload with a set of pre-defined patterns, and the result indicates whether the stream satisfies some special features, such as a virus or an application-level protocol.

Regular expression is widely used in pattern matching scenarios for its powerful and flexible expression ability, for instance, the open source NIDS of Snort [3] and the Linux application protocol classifier [1] (L7-filter). Patterns are compiled to finite state machine (FSM) for automatic processing, and the matching process is represented with FSM state traversal which is driven by stream payload. Deterministic finite automata (DFA) and nondeterministic finite automata (NFA) are traditional FSMs which have opposite performance in memory occupancy and time consumption. NFA state number is linear with pattern length,

© Springer International Publishing Switzerland 2015
G. Wang et al. (Eds.): ICA3PP 2015 Workshops, LNCS 9532, pp. 111–119, 2015.
DOI: 10.1007/978-3-319-27161-3_10

while multiple potential states need to be traversed for an input symbol. On the contrary, only one DFA state needs to be accessed for each character, however, DFA may bring state explosion, which even makes it infeasible to construct an integrated DFA in many cases.

Current researches mainly exploit alternative FSMs for trade-off between FSM size and memory access requirements of per-character processing [5,6,10–12,14,15,17]. Despite of massive proposals, none of them has solved the problem satisfactorily, especially for large scale (namely hundreds to thousands) complex patterns. Yu [16] implemented popular solutions on GPU for large scale patterns, results showed that the highest performance is about 0.2 Gbps which is orders of magnitude lower than needed. In this work, we firstly propose a filtering mechanism to solve the contradiction between memory requirement and matching performance thoroughly.

2 Motivations

In matching process, for each input symbol, all current active states should be traversed to get the next active state set. As FSM is kept as state transition table (STT) in memories, time is mainly consumed for memory access. For large scale patterns, STT can only be deployed on high-latency global memories such as DDR SDRAM, resulting tens to hundreds cycles for per-character processing [8]. However, most streams cannot hit any of these patterns in applications such as NIDS and protocol identification. Thus, it's a huge waste to match all streams with whole STT in global memories.

In practice, most streams cannot match any pattern, and they are even not similar with these patterns especially for NIDS. Suppose a filtration process is carried out to trim the normal flows, then only the left small fraction of suspicious flows need to be matched with the whole STT. Based on this, if the pre-filter is compact enough to be deployed on fast on-chip memories, the performance of filtration process can be orders higher than that of whole pattern matching in global memories. Furthermore, if the pre-filter is accurate enough to trim an overwhelming majority of normal streams, the overall performance can be greatly improved. After filtration, the left suspicious streams should be matched with the whole pattern set deployed in high-latency global memories for further inspection and confirmation. In fact, each stream only needs to be matched with one or several patterns as the filtration process has indicated which rules the stream may belong to. Thus, there is no need to construct an integrated FSM for the whole pattern set. One FSM each pattern strategy is adopted to avoid state inflation brought by pattern interaction, and the strategy is definitely a practical method to solve state explosion. Figure 1 illustrates the matching process with Pre-filter. The patterns are compiled to an integrated FSM for filtration in front-end and separate FSMs for whole matching in back-end. Each stream should be matched with the Pre-filter FSM firstly, and only matched streams in front-end Pre-filter need to be sent to the corresponding back-end FSMs for further whole matching. As the back-end FSMs are deployed in large capacity memories where storage is not a problem, DFAs are employed for fast back-end matching.

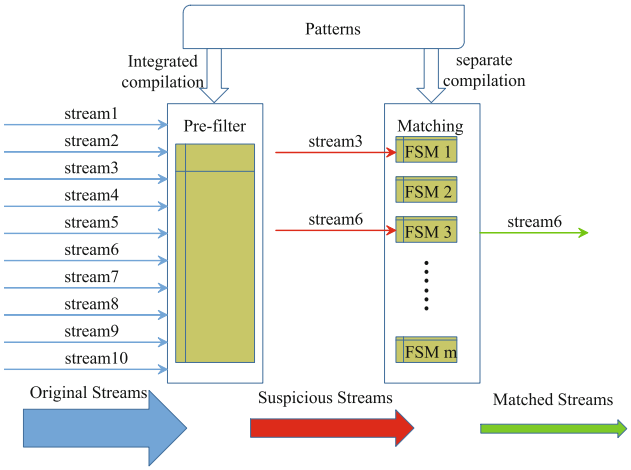

Fig. 1. Pattern matching process with Pre-filter

3 Pre-filter Design

To achieve the expected high throughput, the Pre-filter must be compact and accurate enough to filter most normal streams. Furthermore, to keep matching correctness, false positive can be eliminated by back-end traditional FSMs. While for false negative, the languages identified by the Pre-filter must be a superset of languages identified by the original pattern sets. For above purposes, we propose to construct a compact Pre-filter by extracting string segments from original regular patterns. The idea is motivated by observations that most streams are benign, and they are not even similar with malicious patterns, which means that they even don't contain the sub-strings in these patterns. By extracting string segments, a compact and efficient DFA-like automata can be constructed to filter most benign streams. The Pre-filter is composed of an improved AC [4] automaton and a state dependency table which records the order and dependencies among these extracted string segments. Traditional AC algorithm can only handle exact strings, with assistance of the state dependency table, the enhanced AC can deal with languages described by a fixed sequence of exact string segments.

For example, the integrated minimum DFA for pattern set of $regular[a-z]\{10\}pattern.*set$ and $ab\backslash d+cd$ has 198 states,while the corresponding NFA has only 36 states. State explosion comes from dot-star and character class with length restriction. However, most normal streams even do not contain the ordered string segments in these patterns. A Pre-filter as shown in Fig. 2 where some transitions have been omitted for clarity, associated with the state dependency Table 1 can trim such normal streams. The dependency table is similar with tables in [9,13], while their target are solving all kinds of regular expressions which will have lots of limitations especially for overlaps. Our method has no

such limitations, because our goal is filtration not matching. In Fig. 2, the original pattern $regular[a-z]\{10\}pattern.*set$ is split into segments of $regular$, $pattern$, set which are matched simultaneously. A stream can pass through the Pre-filter for fully matching only when all these segments are matched in order, which is guaranteed by inquiring and updating the dependency status in Table 1. The match of any segment may trigger two actions, test and set operations. If the status label of the previous segment denoted by dependency state region is 0, which means the prior segment has not appeared, then no more operations will be taken. Else, the status label of the current segment will be set to 1. Further, if this segment is the last segment, a pre-filtration matching is hit and the stream will be sent to the corresponding back-end FSM for exact matching, as the stream 3 and stream 6 in Fig. 1. Finally, only stream 6 matches the original regular pattern 3. If a stream cannot hit any of these extracted patterns, it will never match the original patterns, as the other streams in Fig. 1, obviously there is no need to send these streams to back-end matching.

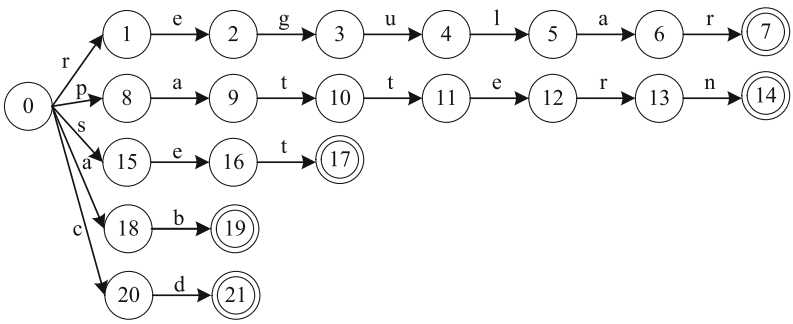

Fig. 2. FSM part of Pre-filter for patters $regular[a-z]\{10\}pattern.*set$ and $ab\backslash d+cd$

Table 1. The state dependency table for Pre-filter in Fig. 2

State id	Status label	Dependency state	Matched rule
7	0/1	-	-
14	0/1	7	-
17	0/1	14	1
19	0/1	-	-
21	0/1	19	2

Next, we will give some examples covering all possible situations to illustrate the matching process. For input sequence $regularabset$, state 7 is activated after processing $regular$, as state 7 is the last state of segment $regular$ and there is no dependency state for it, the status label of state 7 is set to 1. Then for input

ab, state 19 becomes the current active state, and just like state 7, there is no dependency state for state 19, thus its label is set to 1. Next for input set, state 17 will be activated, as it is the last state of segment set, its dependency state (state 14) should be checked for further operation. As the label of state 14 is 0, thus the label of state 17 cannot be set to 1 and no matching occurs for Pre-filtration even it has matched the last segment of the first pattern. For another input sequence $abefcd$, the input ab will activate state 19 and set the corresponding label to 1. Then the following e will make a transition to state 0, and f will stay in state 0. Next, the input cd will activate state 21, as the label of corresponding state 19 is set to 1, the label of state 21 will be set to 1 and a pre-filtration matching occurs. Then the whole sequence $abefcd$ will be sent to back-end FSM corresponding to the original pattern $ab\backslash d + cd$ for confirmation, obviously no matching occurs in the back-end FSM. While for another input sequence $ab123cd$, it can also pass through the pre-filtration as $abefcd$, and further it can make a full matching in the back-end FSM. In practice, the probabilities for above three situations are in a descending order, and most streams belong to the first situation. This distribution is very significant as we can trim most streams with a very compact and efficient FSM deployed in fast memories. In other words, a great improvement can be achieved with the Pre-filter mechanism.

4 Experiments

The experiments were conducted on an Intel Core i7 3770 platform (CPU: 3.40 GHz, L1 Cache: 32 KB, L2 Cache: 256 KB, L3 Cache: 8 MB) with 8 GB RAM and Linux system. We chose pcre-type rules from Snort pattern file backdoor.rules and traces from DARPA [2] intrusion detection data sets. We compared the Pre-filter scheme with traditional NFA, DFA and state of the art Hybrid-FA [5] from FSM construction time, memory footprint and matching speed. Table 2 shows part of the state number statistics for different FSMs with increasing pattern scales. As memory footprint is nearly linear with state number, results in Table 2 can be regarded as memory consumption comparisons among these automatons. Despite of the back-end DFAs in Pre-filter, its size is orders lower than Hybrid-FA and DFA, and even comparable with traditional NFA. The main reasons are that separate compilation hinders the interactions among different rules and little explosion occurs in a single rule. Comparison between column 3 and column 7 also give a visual representation of how the pattern interaction can contribute to state explosion. Furthermore, compressing algorithms such as D^2FA [11] can be employed in back-end standard DFAs to achieve more than 90 % space reduction.

The construction time of Pre-filter scheme consists of two parts: front-end filter compilation and back-end DFAs compilation. As the back-end DFAs can be compiled separately on parallel platforms, we only focuse on the filter compilation time. Figure 3 shows construction time comparisons among these automatons, and DFA is omitted for clarity as it is orders higher than the others. Results show that construction time of both NFA and Pre-filter are linear with increasing

Table 2. State number statistics for different FSMs with increasing pattern scale

No of rules	NFA size	DFA size	HFA		Pre-filter	
			head size	tail size	front size	back size
10	386	299	299	0	348	313
30	1151	13943	9101	41	952	1063
50	1811	120675	21492	109	1553	3863
100	3069	>2M	69218	371	2841	13180

pattern number. While, the construction time for Pre-filter is a little more than NFA as it needs additional time for AC determination and building state dependency table, but it is far more less than that of Hybrid-FA and DFA because no state explosion occurs in Pre-filter.

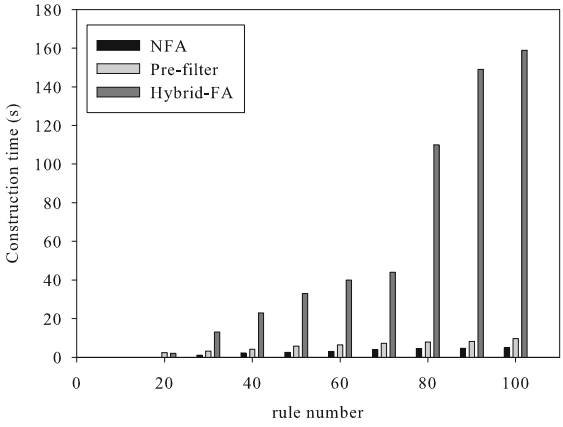

Fig. 3. Construction time comparison among NFA, Hybrid-FA and Pre-filter

The performance of Pre-filter highly depends on the filtration rate, if most streams need to be further processed in back-end DFAs, the overall throughput will drop rapidly. We employed four different DARPA files with average size of more than 300 MB to test the filtration effect, and results are presented in Fig. 4. All the traces achieve similar filtration ratios, and the filtration ratio declines with the increasing pattern set as streams filtered out by smaller filter may match with filter for larger rule set. Even with the declining trend, the filtration rate can still reach more than 95 % with 100 patterns, which contributes a lot to the overall high speed. Performance estimation is shown in Fig. 5, only part of DFA result is displayed as no DFA has been constructed for more than 70 rules in our platform. DFA and Hybrid-FA perform better when the pattern number is less than 20, it is because most transitions are accessed in the high-speed caches.

With the increase of pattern set, state explosion results in rapid speed decline for both DFA and Hybrid-FA. While for Pre-filter, the matching speed is insensitive to the pattern scale as the memory footprint for Pre-filter is linear with pattern size. As most streams have been filtered as shown in Fig. 4, matching speed for Pre-filter declines very slowly with the increasing patterns.

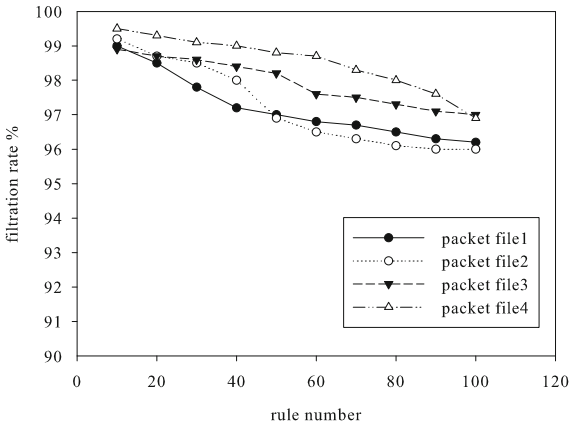

Fig. 4. Filtration rate statistics with different trace files and increasing pattern scale

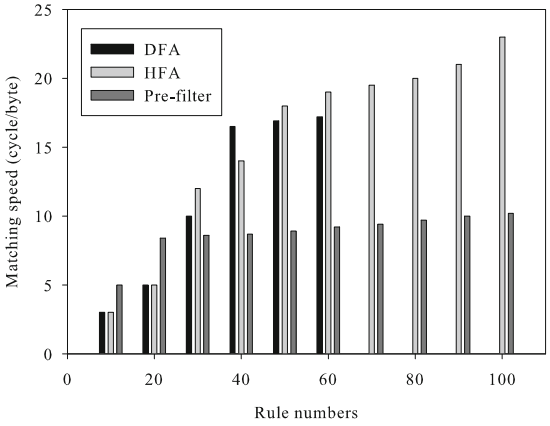

Fig. 5. Average cycles to process one input symbol

5 Conclusion

Considering that few streams can match the regular expression patterns in NIDS applications, we firstly proposed a Pre-filter matching mechanism which employs

a compact and efficient filter to eliminate most benign streams and leaves the left suspicious streams for whole pattern matching. As the filter is compact enough to deployed in small fast memories, most streams can be processed efficiently. Further, benefiting from the high filtration rates in front-end, this method achieves high overall throughput as few streams need further whole matching. Compared with the state of art Hybrid-FA, our method performs better both in memory consumption and time complexity.

Future work will improve from the following aspects to achieve lower memory consumption and higher matching speed: (1) extending the front-end filter with multi-stride, (2) implementing the matching prototype with FPGA or ASIC, (3) employing compressing algorithms in back-end DFAs.

Acknowledgements. This work is sponsored by National Natural Science Foundation of China under Grant No. 61379148.

References

1. Application layer packet classifier for linux (2009). http://l7-filter.sourceforge.net/
2. Darpa intrusion detection data sets (1999). http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html/
3. Snort v2.9 (2014). http://www.snort.org/
4. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
5. Becchi, M., Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In: Proceedings of the 2007 ACM CoNEXT conference, p. 1. ACM (2007)
6. Becchi, M., Crowley, P.: A-dfa: a time-and space-efficient dfa compression algorithm for fast regular expression evaluation. ACM Trans. Archit. Code Optim. (TACO) **10**(1), 4 (2013)
7. Chen, C.P., Zhang, C.Y.: Data-intensive applications, challenges, techniques and technologies: a survey on big data. Inf. Sci. **275**, 314–347 (2014)
8. Chen, S., Lu, R.: A regular expression matching engine with hybrid memories. Comput. Stan. Interfaces **36**(5), 880–888 (2014)
9. Khalid, A., Sen, R., Chattopadhyay, A.: Si-dfa: Sub-expression integrated deterministic finite automata for deep packet inspection. In: 2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR), pp. 164–170. IEEE (2013)
10. Kumar, S., Chandrasekaran, B., Turner, J., Varghese, G.: Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications systems, pp. 155–164. ACM (2007)
11. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. ACM SIGCOMM Comput. Commun. Rev. **36**(4), 339–350 (2006)
12. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. ACM SIGCOMM Comput. Commun. Rev. **38**(4), 207–218 (2008)

13. Wang, K., Li, J.: Towards fast regular expression matching in practice. In: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, pp. 531–532. ACM (2013)
14. Xu, Y., Jiang, J., Wei, R., Song, Y., Chao, H.J.: TFA: A tunable finite automaton for pattern matching in network intrusion detection systems (2014)
15. Yang, Y., Prasanna, V.K.: Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In: 2011 IEEE Proceedings of INFOCOM, pp. 1853–1861. IEEE (2011)
16. Yu, X., Becchi, M.: Gpu acceleration of regular expression matching for large datasets: exploring the implementation space. In: Proceedings of the ACM International Conference on Computing Frontiers, p. 18. ACM (2013)
17. Zheng, K., Cai, Z., Zhang, X., Wang, Z., Yang, B.: Algorithms to speedup pattern matching for network intrusion detection systems. Comput. Commun. **62**, 47–58 (2015)