LETTER

Regular Expression Filtering on Multiple *q*-Grams

Seon-Ho SHIN[†], **HyunBong KIM**[†], ***Nonmembers*, *and*** **MyungKeun YOON**[†a)], ***Member***

SUMMARY Regular expression matching is essential in network and big-data applications; however, it still has a serious performance bottleneck. The state-of-the-art schemes use a multi-pattern exact string-matching algorithm as a filtering module placed before a heavy regular expression engine. We design a new approximate string-matching filter using multiple *q*-grams; this filter not only achieves better space compactness, but it also has higher throughput than the existing filters.

key words: *regular expression, string matching, q-gram, intrusion detection, deep packet inspection*

1. Introduction

Regular expression is widely used to analyze big data, network traffic, web contents, etc. Many network and security applications provide the functionality where users can define their own regular expression rules. In this letter, we focus on how to accelerate regular expression matching in network intrusion detection, but our proposed scheme can work for any computing application.

Although regular expression matching is widely used, it still has a serious performance bottleneck. Numerous studies have proposed new designs and implemented new regular expression engines [1]–[3]. Efficient data structures, such as the bitmap and the hash table, help increase the throughput; however, they rely on additional ternary content-addressable memory (TCAM) or field-programmable gate array (FPGA) hardware modules.

To improve the processing speed by software, most regular expression applications adopt a filtering module as illustrated in Fig. 1, which is the basic architecture assumed in this paper. The filtering module is generally implemented as a multi-pattern exact string-matching algorithm, such as Aho-Corasick [4]. A representative simple string is defined for each regular expression rule, and the set of representative strings are fed to the filtering module in the preparation phase. When a packet arrives, the filtering module first checks if the packet contains any simple string of the set. If a match is found, the packet is sent to the regular expression engine for further evaluation. The filter also notices the engine of the specific rule numbers as index so that the engine only checks these indexed rules [5], [6].

Manuscript received August 16, 2017.
Manuscript revised September 27, 2017.
Manuscript publicized October 11, 2017.

[†]The authors are with Kookmin University, Seoul, South Korea.
a) E-mail: mkyoon@kookmin.ac.kr

DOI: 10.1587/transinf.2017EDL8180

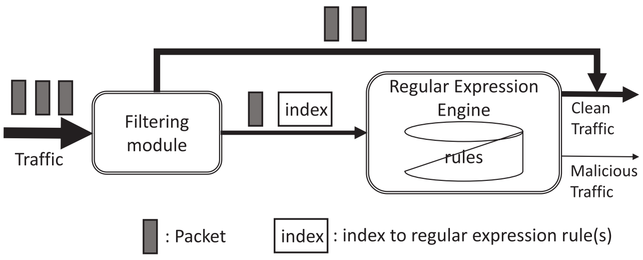

Fig. 1 Architecture of regular expression matching: a filtering module is placed before the regular expression engine to filter out benign packets.

Our simple experiments reveal that the throughput becomes as low as 2.3Mbps on a modern general-purpose computer with two CPUs of six cores when no filtering module is used. We directly applied a PCRE2 [7] or an RE2 [8] engine to the Snort regular expression rules with real Internet packets [9]. Adding an Aho-Corasick filter raised the matching throughput from 2.3Mbps to 2.7Gbps. This significant improvement is obtained from the selective invocation of regular expression rules indexed by the filter rather than the whole set of rules.

Although the filter improved the throughput, multi-pattern exact string-matching algorithms, such as Aho-Corasick, Commentz-Walter, and Wu-Manber, were not originally designed for the purpose of fast regular expression filtering. They are designed to find exact strings comprising some alphabets in a specific order. Therefore, these algorithms are quite expensive as a regular expression filter because they need to keep track of states in automata and multiple pointer operations are required. This motivated us to design a new filtering algorithm for regular expression matching that is faster and simpler than the existing algorithms. The contributions of this letter are as follows:

- A family of new filtering algorithms is designed for regular expression matching, called regular expression filter (REF), θ-REF, and dynamic-REF. These algorithms use a stateless data structure on *q*-grams.
- Through experiments with Internet traffic, we prove the enhanced throughput and space compactness of REF. The throughput is improved by more than three times, and the memory space is reduced by a factor of four.

2. Design of Regular Expression Filter

We present REF, a new filtering algorithm for multi-pattern approximate string-matching, which consists of two phases: rule graph generation and regular expression filtering. We first explain these phases in detail and propose two advanced schemes: θ-REF and dynamic-REF.

2.1 Rule Graph Generation

The rule graph is generated by following a specific process. For a set of regular expression rules $R = \{r_1, r_2, \ldots, r_n\}$, we first generate a set of q-grams for each rule, which is denoted by $x_q(r_i)$ for r_i. Some regular expression applications require their users to provide a representative simple string for any user-defined regular expression rule. For example, Snort requires its users to provide a *content* field for a regular expression rule that is a just simple string. We can also extract extra q-grams by removing the special characters from the regular expression rules. For example, we can extract 4-gram of "$SSH-$" from a regular expression rule of "$\hat{}SSH-[12]\backslash.\backslash d+$" by eliminating special characters of '^', "[12]", and "\ d+". We define a set of all q-grams as $S = \bigcup_{i=1}^{n} x_q(r_i) = \{s_1, s_2, \ldots, s_m\}$.

Special care should be taken when q-grams are extracted from a regular expression rule containing such characters as '!' and '|', which respectively refers to the NOT and OR operations; otherwise, false-negative matching may happen. For example, suppose that a regular expression rule "(POST|HEAD)" is given. If we remove '|', this rule will contribute two 4-grams of "POST" and "HEAD". If a packet contains only "POST", the rule will never be matched. To solve this problem, we removed not only special characters but also their associated simple strings. For example, "(POST|HEAD)HTTP" contributes only one 4-gram of "HTTP". How the q-grams can be optimally extracted remains as an open problem.

In general, enough q-grams are extracted from a regular expression rule and its representative simple string. Our experiments with Snort [9] showed that we could extract at least one q-gram from all the regular expression rules when $q \leq 4$. Most rules generated more than two q-grams. In case no q-grams can be extracted from a regular expression rule, we may have a separate process to evaluate packets with those rules that do not have q-grams. Experiments show that a q value of less than 5 optimizes the performance.

After S is obtained from R, a bipartite graph of $G = (V, E)$ is defined as follows [10]: $V = S \cup R$ with r_i located on the left side of G and s_j on the right side, $1 \leq i \leq n$ and $1 \leq j \leq m$. An edge is drawn between r_i and s_j and is denoted as $e_{ij} \in E$, if s_j is derived from r_i, i.e., $s_j \in x_q(r_i)$. Figure 2 shows the bipartite graph generation for R=$\{r_1, r_2, r_3\}$. Representative simple strings are not included in this example for simplicity. Note that $x_q(r_1)$=$\{SSH-\}$, $x_q(r_2)$=$\{/ff., ff.p, f.ph, .php\}$, and $x_q(r_3)$=$\{/q.p, q.ph, .php\}$.

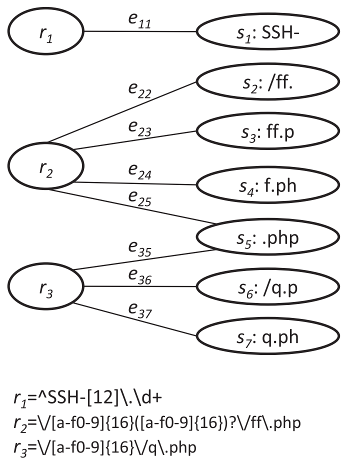

Fig. 2 A basic rule graph derived from $R = \{r_1, r_2, r_3\}$. $x_q(r_1)$=$\{SSH-\}$, $x_q(r_2)$=$\{/ff., ff.p, f.ph, .php\}$, and $x_q(r_3)$=$\{/q.p, q.ph, .php\}$, when q=4.

2.2 Regular Expression Filtering

The basic bipartite rule graph can be used for filtering out benign packets. A regular expression rule is only activated when all its q-grams are found in a packet. For every packet, we first generate all possible q-grams from it; if the packet is of size α bytes, $(\alpha - q + 1)$ grams are generated. For each q-gram in the packet, we look up the nodes of S that are implemented as a hash table of q-grams. If a q-gram matches any s_j, we remove all the edges connected with s_j. If any r_i node is disconnected from the graph, the regular expression engine is activated to evaluate the packet with r_i.

Once a packet arrives, all possible q-grams are extracted, and any s_j node in the rule graph matching q-grams is searched for. When s_j matches a q-gram from the packet, the edges connected with s_j are removed. Finally, any r_i that is disconnected from the graph activates the regular expression engine. For example, suppose that a packet of "SSH-/ff.php" arrives. Its 4-grams are "SSH-", "SH-/", …, and ".php". Five s_i's are matched from s_1 to s_5, and two rules of r_1 and r_2 are disconnected.

REF approximately identifies the benign packets; it does not belong to a family of exact multi-pattern string-matching algorithms such as Aho-Corasick. Some packets are unnecessarily sent to the regular expression engine by REF, which can be called false positives. For example, if a packet includes "q.php", Aho-Corasick does not activate the regular expression engine, whereas REF does so (see Fig. 2). However, REF runs much faster than the traditional multi-pattern string-matching algorithms, and it saves more memory space, as experimentally verified.

In this letter, we implement a hash table of q-grams derived from S; therefore, all q-grams are inserted in the table. A hash table can be looked up by $O(1)$; q-grams from a packet are searched for quickly. For a packet of size α bytes, the hash table should be looked up $(\alpha - q + 1)$ times.

A sophisticated attack can fabricate packets to include as many triggering strings as possible so that the regular expression engine is frequently invoked. This attack can compromise not only our proposed scheme but also any filtering-based schemes. One simple countermeasure is to limit the maximum number of engine invocations per packet.

2.3 θ-REF

In θ-bounded REF, or θ-REF, the maximum number of edges from r_i is not larger than threshold θ. The intuition behind θ-REF is that only a few q-grams from a regular expression rule can make a unique identifier for that rule if selected appropriately. Two new properties are defined for each node in the rule graph for θ-REF, *connected* and *degree*, denoted as c and d, respectively. The degree property specifies the number of physical edges around the node. The connected property specifies the number of edges around the node that have been selected until now.

We propose a heuristic edge-selection algorithm. The idea is that we can identify a rule or at least a small set of candidate rules quite accurately with small θ, if each rule is connected with as many different q-grams as possible. Note that the REF can activate any rule after the rules are disconnected from the rule graph. If two rules have the same q-grams, they would be disconnected from the graph simultaneously when these q-grams are present in a packet. This is not desirable because multiple regular expression rules should not be evaluated by the engine. Another design principle is to give high priority to a small-degree node because large-degree nodes generally have more freedom in selecting edges that are not shared by others.

We explain the selection algorithm. The rule node with the smallest c is selected first. If there are multiple rule nodes with the same smallest c, we select the one with the smallest d value. If there are multiple rule nodes with the same smallest c and d, we select one of them randomly. Then, we need to choose one edge connected with the selected rule node. Again, we select the q-gram node with the smallest c. For tie breaking, we compare the d values, and finally select the most appropriate q-gram, which determines an edge. After the edge is selected, we increase the c values of the end nodes by one. If c equals d or θ in a rule node, we remove that rule node from the graph. This is because the rule node has been selected θ times, or no more available edges are left at the node. We repeat this edge selection process until every rule node is removed from the graph.

When a packet arrives, all its possible q-grams are extracted, and the hash table of the set S is looked up for those s_j's that match any q-grams. If a matching node is found, all the edges connected with that are selected and removed, and the c value of the connected nodes are decreased by one. If any r_i is disconnected from the graph, that is, its c value becomes zero, the regular expression engine is activated to evaluate the whole packet content with the rule r_i.

2.4 Dynamic-REF

Some runtime environments may have static data and traffic patterns. For example, if an intrusion detection system runs for years in front of a web server with static contents, the system would see repeated traffic patterns periodically or steadily. We propose how to use this property to generate a more optimal rule graph and minimize the number of regular expression engine activations. This version of REF is called *dynamic*-REF because it enhances performance by using the statistical information about the previous traffic.

Let us suppose that we know the occurrence frequency of each q-gram in S. When selecting an s_j during the rule graph generation, we should first consider those q-grams that have rarely occurred. This is because we can expect that the edges connected with these s_j's would be hardly removed; therefore, the corresponding rules are less frequently evaluated with the regular expression engine. We add a new property to s_j, f for frequency, which indicates how many times s_j has occurred. This information can be collected during a certain period of the learning phase. The rule graph construction is the same as θ-REF; the only difference is that we consider the c, f, and d values for each node in dynamic-REF. After the c values are compared, the node of small f is given high priority.

3. Experiments

We evaluate REF through experiments using real Internet traffic traces. The experiments show that REF outperforms Aho-Corasick in terms of both throughput and memory use simultaneously. We chose Aho-Corasick because it is widely used, including Snort [9], and it performs well for both average and worst cases.

The experimental results are compared in terms of the throughput and memory space. The throughput is defined as the total bytes of the traffic trace divided by the processing time. For dynamic-REF, we use the first half of the traffic trace to count the occurrence of each q-gram. Then, the rule graph is generated, and dynamic-REF runs against the remaining traffic trace.

We use 1,200 distinct regular expression rules from Snort version 2.9 after removing the redundant rules. The *pcre* and *content* option fields are examined to generate q-grams. For example, the q-gram set of $\{13D1, 3D12, 1\%3D\}$ is obtained from the snort rule of [alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (content:"13D12"; pcre:"/or\++1%3D/";)].

A Linux server was used for the experiments with two Intel Xeon CPUs (2.6 GHz six-core processors) and 16GB DDR3 memory (1,600 MHz). The 64-bit CentOS version 6.6 is installed, and 12 full threads run for all the experiments.

The traffic trace is collected from a campus network. The same traffic was used for both REF and Aho-Corasick in the experiments for fair comparison. We briefly explain the traffic distribution: IP packets account for 88.3%, and TCP/UDP packets for 83.3%. Most frequently-observed ports are secure socket layer (SSL), web, domain name systems (DNS), as usual with the Internet access points in campus networks. Most threat alarms were related with web, DNS, and distributed denial of service (DDoS) attacks.

The first set of experiments compares θ-REF, dynamic-

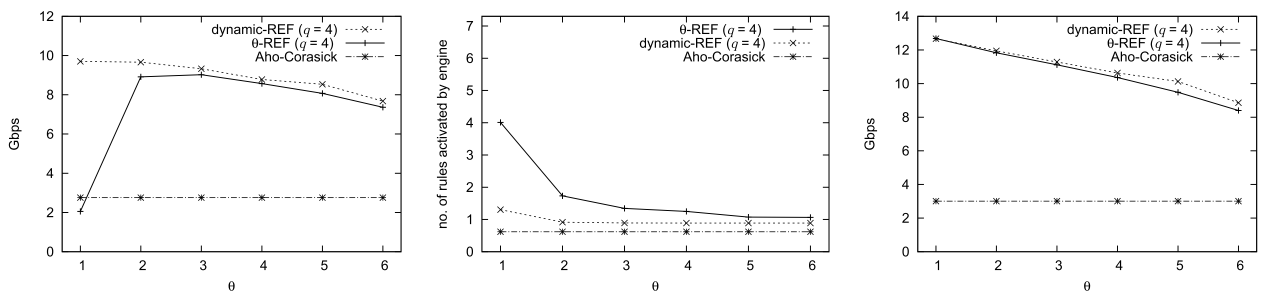

Fig. 3 Comparison of θ-REF, dynamic-REF, and Aho-Corasick in terms of total throughput, average number of rules activated per packet, and filtering throughput, from left to right.

REF, and Aho-Corasick in terms of their throughputs. Figure 3 presents the experimental results; the left plot shows the total throughput that measures both the filtering time and the regular expression engine time. The plot shows that Aho-Corasick processes packets at 2.7 Gbps, whereas θ-REF and dynamic-REF have their maximum throughputs higher than 9 Gbps. The middle plot compares the average number of regular expression rules that are finally activated by the rule engine per packet. Aho-Corasick causes the smallest number of rules to be activated. In REF, this number decreases sharply with θ. The left and middle plots of Fig. 3 imply that the filtering module of REF runs much faster than the Aho-Corasick filter despite activation of more rules. This is shown in the right plot of Fig. 3; the filtering throughput of REF is significantly higher than its total throughput. Therefore, the total throughput of REF would significantly improve with a faster regular expression engine.

Dynamic-REF already knows which q-grams would be seen more frequently; therefore, this algorithm performs best when $\theta = 1$. This is because a small θ implies a faster lookup in the q-gram hash table. Dynamic-REF would result in a poor throughput if the traffic patterns changed abruptly. We believe that this does not devalue REF because θ-REF shows competitive throughput without any learning phase of previous traffic patterns.

Finally, we compare the runtime memory sizes of dynamic-REF and Aho-Corasick. Dynamic-REF requires the memory space in proportion to the number of hash table slots while Aho-Corasick requires memory space in proportion to the number of states. From the experiments described above, we confirmed that Dynamic-REF consumed only a quarter of the memory used by Aho-Corasick while the throughput was improved by more than three times.

4. Discussion

When REF or Aho-Corasick finished the filtering process with a packet, a few rule index, usually one, were selected. Then, the regular expression engine finally checked the whole contents of the packet only with these rule index. Therefore, false positive or false negative errors are totally dependent on the regular expression engine irrespective of the filtering module. In the experiments, we confirmed that using REF or Aho-Corasick did not affect the errors of intrusion detection.

5. Conclusions

This paper proposed a new multi-pattern approximate string-matching algorithm for accelerating regular expression matching. The new algorithm enhances throughput by more than three times, and it uses only a quarter of the memory, compared with previous algorithms.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. 2017-0-00513-001, Developing threat analysis and response technology based on Security Analytics for Heterogeneous security solution)

References

[1] J. Patel, A.X. Liu, and E. Torng, "Bypassing space explosion in high-speed regular expression matching," IEEE/ACM Transactions on Networking, vol.22, no.6, pp.1701–1714, 2014.

[2] A.X. Liu and E. Torng, "An overlay automata approach to regular expression matching," Proceedings of IEEE INFOCOM'14, pp.952–960, 2014.

[3] T. Liu, A.X. Liu, J. Shi, Y. Sun, and L. Guo, "Towards fast and optimal grouping of regular expressions via dfa size estimation," IEEE Journal on Selected Areas in Communications, vol.32, no.10, pp.1797–1809, 2014.

[4] A.V. Aho and M.J. Corasick, "Efficient string matching: an aid to bibliographic search," Commun. ACM, vol.18, no.6, pp.333–340, 1975.

[5] G. Vasiliadis, M. Polychronakis, S. Antonatos, E.P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," Proceedings of RAID'09, vol.5758, pp.265–283, 2009.

[6] M.A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a highly-scalable software-based intrusion detection system," Proceedings of the 2012 ACM CCS'12, pp.317–328, 2012.

[7] http://www.pcre.org/, 2017.

[8] https://github.com/google/re2/, 2017.

[9] https://www.snort.org/, 2017.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms, Third Edition, The MIT Press, 2009.