A Heuristic for Constructing Smaller Automata Based on Suffix Sorting and Its Application in Network Security*

Inbok LEE†††, Member, Victor C. VALGENTI††, Min S. KIM††, and Sung-il OH††††, Nonmembers

SUMMARY In this paper we show a simple heuristic for constructing smaller automata for a set of regular expressions, based on suffix sorting: finding common prefixes and suffixes in regular expressions and merging them. It is an important problem in network security. We applied our approach to random and real-world regular expressions. Experimental results showed that our approach yields up to 12 times enhancement in throughput.

key words: string algorithm, regular expression, automata, network security

1. Introduction

We are interested in the following problem.

Problem 1: Given a language $L$, compute the minimal automata which accepts $L$.

By “minimal” we mean the minimum number of states in the automata. If $L$ is regular, then we may build non-deterministic finite state automata (NFA) first, convert it into a deterministic finite state automata (DFA), and minimize it using the classical technique in [2]. However, as the number of states in DFA may be exponential, we cannot do it in real world and we do not know a better method.

The main motivation for the problem lies in network security. An intrusion detection system (IDS) reads incoming and outgoing packets, and issues warnings if they contain a string in the language of predefined regular expressions. We need one or more regular expressions for describing a network attack and now we have thousands of such regular expressions. From now on we consider $L$ is regular.

In practice, Snort [7], de facto standard of IDS, builds a DFA for each regular expression. For each regular expression, it first create “candidates”, short string contained in every word in its language. When it reads packets, it finds candidates by Aho-Corasick automata [1], then upload the corresponding DFA one by one.

However, uploading automata one by one is not very efficient: each time the automata should be moved to the cache and after a while, it should be brought back to the RAM again. If we build a smaller automata for the whole language that fits into the cache, then we will obtain better throughputs.

Worker on intrusion detection systems focus on obtaining better throughputs by hardware-based approaches (for example, using graphical processors as in [9]) or by software-based ones (for example, Hybrid Finite Automata [3]).

So far as we know, our previous work [11] is the only one which tried to reduce the size of the automata. We gave two heuristics based on finding common prefixes and/or suffixes among regular expressions, but it was rather ad-hoc and its steps and running were not analysed theoretically.

Here we show a heuristic for the following problem which runs in time proportional to the sum of the lengths of regular expressions, based on suffix sorting.

Problem 2: Given a set of regular expressions and two integers $\theta_1$ and $\theta_2$, divide them into disjoint sets such that every pair of regular expressions in the same set share a common prefix of length $\theta_1$ and/or a common suffix of length $\theta_2$.

2. Algorithms

Here we will assume that the language $L$ consists of a set of regular expressions. Note that a simple string itself is a regular expression.

Our approach is based on the simple observation: regular expressions in real world frequently share a common prefix and/or a common suffix. For example, in Fig. 1, two regular expressions from the network intrusion detection rules differ by one letter.

That is, two regular expressions $r_1 = wuy$ and $r_2 = wyy$ can be merged into one $r' = w(u | v)y$. Once we divide regular expressions into disjoint sets sharing a common prefix and/or a common suffix, we can merge all regular expressions in the set into a single concise regular expression. By merging regular expressions with common prefixes, we can reduce the number of regular expressions and the number of cache misses as they will be loaded into the cache together. By merging regular expressions with common suffixes, we can find the additional redundancy in them. For example,
the Aho-Corasick algorithm consider strings with the same prefixes and the automata reflects this fact. However, it does not consider strings with the same suffixes.

There is one drawback with our approach: each regular expression may be associated with a distinct meta information (for example, the name of network attack associated with the regular expression) and merging them will result in a loss of information without further examination. However, in practice it is not much of an issue as similar regular expressions have almost always the same meta information.

Here we will show our heuristic for Problem 2 based on suffix sorting. We will explain the procedure for finding both a common prefix and a common suffix.

**Step 1:** We build a generalised suffix array for the rule set (with LCP values). For example, if we have three regular expressions, \(a[cd]e, ab*e, \) and \(be, \) then we build the suffix array of \(a[cd]e\#ab*e\#be\# \) as in Fig. 2 where \# denotes the end of a regular expression and \(\# \) does not match with itself. Note that we will consider regular expressions as simple strings: the meanings of special characters \([, ]\), and * are ignored.

**Step 2:** We group regular expressions by the common prefixes of length \(\theta_1\) using a simple line sweeping on the suffix array. We first find the first regular expression in the suffix array, not one of proper suffixes. In our previous example, the first regular expression in the suffix array is \(a[cd]e\) followed by \(ab*e\). The LCP value is 1, which tells they share a common prefix \(a\). Note that we ignored \# and following characters in suffixes. We keep on the line sweeping until the LCP value goes below \(\theta_1\). From the next suffix we continue the line sweeping.

We do the same until we hit the last row of the suffix array. The results with our example are \(\{a[cd]e, ab*e\} \) and \(\{be\} \). We restricted ourselves to prefixes and suffixes as that way we can guarantee no change to the semantics of the underlying regular expressions.

**Step 3:** For each group, again we group regular expressions by the common suffixes of length \(\theta_2\). The idea is almost the same with the one in Step 2. For each regular expression \(r\) in the group, we create \(r^\delta\), its reverse, and concatenate them into one string. In our previous example, from the set \(\{a[cd]e, ab*e\}\), we create \(e[dc]a#e*ba#\) and do the same line sweeping again with \(\theta_2\).

**Step 4:** For each group, we create one regular expression as prefix plus alternates plus suffix. It is added to the final set, representing all the regular expressions in the group. The final set with our example is \(\{a([cd])b*e, be\}\).

**Theorem 1:** The above procedure runs in \(O(n)\) time, where \(n\) is the sum of lengths of regular expressions.

**Proof.** Step 1 runs in \(O(n)\) time using the techniques in [5], [6]. Step 2 runs in \(O(n)\) time as for each row of the suffix array, we just check whether the LCP value is equal to or greater than \(\theta_1\). Step 3 also runs in \(O(n)\) time, as one regular expression can be contained by exactly one group. It is easy to show that Step 4 also runs in \(O(n)\) time. \(\Box\)

### 3. Experimental Results

For evaluation, we first built the NFA after modifying the regular expressions in the set according to our heuristics. Then we look at the improvement from using the new NFA over a normal NFA, or a Hybrid Finite Automata (HFA) [3]. In particular we look at automata size and matching efficiency as a measure of throughput. After some experiments, instead of fixing \(\theta_1\) and \(\theta_2\), we set \(\theta_1\) as 10% of the length of the shortest string in a group and \(\theta_2\) as 5%.

We used five sets of regular expressions. Sets called as ClamAV [4], SpamAssassin [8], and Snort [7] are regular expressions used in detecting spams, viruses, and network intrusions. The set Proprietary is regular expressions used in a telecommunication company to detect network intrusions. The set Random is a completely random regular expression generated by [10]. We employed a tool [10] to generate synthetic packet captures for evaluation. First, we created a packet capture of one million IP packets, each 1514 bytes long, containing uniformly random data (termed Rand). Second, we generated synthetic packets so that each matches at least one regular expression in the set. We termed this traffic 'Matching' and created one million 1514 byte IP packets for each set.

We first consider the size reduction obtained by merging regular expressions. Size Reduction is defined as the size of the new NFA divided by that of the original NFA for a given regular expression set. As we just changed the set of regular expressions, we can build HFA with the new set too and we also calculated the size reduction. Figure 3 shows...
the size reduction obtained by our approach. ClamAV and Random sets showed little size reduction as they were made up with almost no redundancy. With the remaining sets we were able to obtain the size reduction between 5% and 13%.

Then we considered the throughput enhancement over two traffic sets we previously mentioned as Rand and Match. In Fig. 4, we showed the throughput of the new NFA (or, the new HFA) divided by a normal NFA. Though it is difficult to see in Fig. 4, all sets saw a speed-up, even if small. With the ClamAV set we saw the least enhancement (about 1%). With the Proprietary set we were able to obtain about 1200% throughput enhancement.

4. Conclusion

We showed that the classical idea of suffix sorting can be applied to building a smaller automata for a set of regular expressions. Also, experimental results showed that our approach can be used in the real world problem of network intrusion detection. We were able to obtain up to 12 times enhancement in throughput.

The main merit of our approach is that it can be used with any previous approach for faster intrusion detection, be it based on the hardware [9] or the software [3]. Also it is simple and easy to implement.

We believe that basic string algorithms and indexing techniques can be used in the network security as traffics and intrusion detection rules are regular expressions.

References

[1] A.V. Aho and M.J. Corasick, “Efficient string matching: an aid to bibliographic search,” CACM, vol.18, no.6, pp.333–340, 1975.
[2] J.E. Hopcroft, R. Motwani, and J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, 2nd ed., Addison-Wesley, 2001.
[3] M. Becchi and P. Crowley, “A hybrid finite automaton for practical deep packet inspection,” CoNext, 2007.
[4] ClamAV: Open Source Antivirus Engine 0.98.6, Sourcefire, 2015, http://www.clamav.net.
[5] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, “Linear-time LCP computation in suffix arrays and its applications,” CPM 2001, pp.181–192, 2001.
[6] J. Kärkkäinen, P. Sanders, and S. Burkhardt, “Linear work suffix array construction,” J. ACM, vol.53, no.6, pp.918–936, 2006.
[7] Snort: Network intrusion detection and prevention system, http://www.snort.org
[8] SpamAssassin: the open-source spam filter 3.3.2, Apache, 2015, http://spamassassin.apache.org.
[9] G. Vasilaidis, S. Antonatos, M. Polychronakis, E.P. Markatos, and S. Ioannidis, Gnort: High Performance Network Intrusion Detection Using Graphics Processors, RAID 2008: 116–134.
[10] V.C. Valgenti and M.S. Kim, “Increasing diversity in network intrusion detection system evaluation,” GLOBECOM 2015, pp.1–7, 2015.
[11] V.C. Valgenti, M.S. Kim, S.-I. Oh, and I. Lee, “REduce: Removing redundancy from regular expression matching in network security,” ICCCN 2015, pp.1–10, 2015.