BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2018
http://excel.fit.vutbr.cz

Reduction of Automata used in Network Traffic Filtering

Jakub Semrič*

Abstract
The aim of this work is to propose flexible methods for reducing non-deterministic finite automata used in network traffic filtering. We introduce several approaches, which are then combined into a single algorithm with parameters. To achieve a substantial reduction of automata, we use language non-preserving techniques with a primary focus on language over-approximation, since language preserving methods may not be sufficient. We implemented the methods and computed the error caused by the reduced automata on real traffic. Despite the fact that our approach does not provide any formal guarantee wrt unseen input traffic, it can be smoothly used for non-deterministic automata of any size, which is a significant problem for existing methods with very high time complexity.

Keywords: Finite automata — Automata reduction — Deep packet inspection — Network intrusion detection system

Supplementary Material: Downloadable Code, https://github.com/jsemri/ahofa

*xsemri00@fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

In the past years, there has been a considerable increase in cybercrime, including intrusion into Internet networks. This fact has aroused a need for using network intrusion detection systems, which try to detect and prevent such malicious activities. This detection is carried out by deep packet inspection, which searches for a particular pattern in the packet payload. The patterns are usually described by regular expressions (RE). Due to increasing speed of networks and the need of real-time packet inspection, it is necessary to implement these systems in hardware. In other words, detection systems are required to analyze a packet and invoke a corresponding reaction immediately after it has been received. In general, REs are represented in hardware as finite automata. There exist various architectures for pattern matching primarily based on FPGA technology. However, huge automata take a lot of space on a chip. Hence, their hardware realization would be very expensive or even not possible. Moreover, the hardware implementation of the automaton has to be copied several times if we want to achieve a higher speed of packet processing (for 400 GiB per sec. it is approximately 63 times [1]). To tackle this problem we propose reductions which try adjust the size of automata with a reasonable trade-off between the error and the number of states of the automaton.

If we limit ourselves to deterministic finite automata (DFA), one can use Hopcroft's algorithm [2], to obtain minimal DFA. Nevertheless, a problem appears when one wants to use this technique on immense non-deterministic automata (NFA), where state explosion caused by converting NFA to DFA may occur. On the other hand, there also exist algorithms of diminishing size of non-deterministic automaton directly, without need of determinization. These approaches are based on various simulation techniques discussed, e.g., in [3, 4]. Although minimized automaton using the mentioned techniques can be several times smaller than original one, the reduction may still not be as sufficient as it is desired (e.g. we want smaller NFA that a minimal NFA).

There are not many language non-preserving reduction methods, but we can find one described in [5].

This approach, driven by the probabilistic distance, has quite favorable results. In addition, it provides a formal guarantee wrt unseen data. However, the most significant drawback is that it relies on general network traffic model represented by a probabilistic automaton. Acquiring of exact model is extremely difficult because of the diversity of traffic containing myriads of protocols. The algorithms for learning probabilistic automata are not suited for such samples including a lot of noise represented by binary and tunneled data.

To significantly reduce the size of automata in a fast and flexible way, we propose a different and rather experimental approach based on language over-approximation. Although the language of reduced automaton is not the same as the language of the original one, we can considerably decrease the number of states and transitions. Note that there undoubtedly will be some false positives. However, the language over-approximation assures that filtering based on the reduced automaton omits no malicious packets which should be classified by the original automaton. In practice, the hardware devices serve as a traffic prefilter, which sends suspicious packets for further inspection to software. In other words, once a packet has been classified, which might have been classified either correctly or incorrectly, it is subsequently validated in software to achieve that all packets are handled faultlessly.

Our proposed methods modify the structure of an input automaton in order to find sequences of states, which principally contributes to the classification process. These states are retained, and the rest is modified based on algorithm's parameters, including the reduction rate. To decide which states are more important, packet frequency is used, which is basically computed on some training samples.

The proposed methods were carried out and tested on various automata. The vast majority of datasets we used was supplied by ANT@FIT research group. The rest of samples were acquired by Darpa traffic dumps [6]. We achieved quite encouraging results, which have shown a great potential of our approach. Moreover, we also managed to reduce huge automata in quite reasonable time.

2. Preliminaries

In this section, we provide a few definitions, which are essential to comprehend concepts in the following sections.

Definition 1. *Formally, a finite automaton is defined as a structure* $M = (Q, \Sigma, \delta, s, F)$ *where*

- Q *is a finite set of states,*
- Σ *is an input alphabet,*
- $\delta : Q \times \Sigma \to 2^Q$ *is a transition function,*
- $s \in Q$ *is an initial (start) state,*
- $F \subseteq Q$ *is a set of final (accepting) states.*

We say that the automaton $M = (Q, \Sigma, \delta, s, F)$ accepts or recognizes a language L, we write $L(M)$, when $\forall w \in L$ holds $\widehat{\delta}(s, w) \subseteq F$, where a function $\widehat{\delta}$ maps a state $q \in Q$ and a string w to a new set of states [7].

We also recognize two types of a finite automaton, a deterministic (DFA) and nondeterministic (NFA) one. If a finite automaton is DFA, it holds that $|\delta(q, a)| \leq 1$, for $\forall q \in Q$ and $a \in \Sigma$ (note that $|S|$ denotes a cardinality of the set S). In general, instead of having one initial state a NFA can have a set of initial state, but this fact does not change anything on automaton semantic. However, a DFA can have only a single initial state.

For every regular languages there exists a minimal deterministic automaton that accepts it. There are minimal NFA whose size can be even exponentially smaller than equivalent DFA accepting the same language.

Definition 2. *We call* $M' = (Q', \Sigma, \delta', s', F')$ *a subautomaton of* $M = (Q, \Sigma, \delta, s, F)$ *where*

- $Q' \subseteq Q$,
- $\delta' = \delta$ *restricted to* Q',
- $s' \in Q'$,
- $F' \subseteq F \cap Q'$.

Generally speaking, a subautomaton is some interconnected subset of states of an automaton. Naturally, a finite automaton can have several subautomata.

3. Automata Reduction

In this section, we describe our proposed methods for the reduction of non-deterministic automata. Initially, the basic concept of automata used in traffic filtering will be discussed. After this, we propose a refinement of this method, give some comparisons, and discuss advantages and disadvantages. At the end of the section, we will describe our final approach, which combines both previous methods by using particular parameters.

Automata used in Network Traffic Filtering

In packet pattern matching, each automaton contains several smaller independent subautomata. The final states of each subautomaton identify particular rules, which are used for classification, e.g., recognizing a type of attack or protocol. Because we search for patterns in traffic, not a full match we only need to know which finals states were reached by a packet during its processing. Basically, after a packet has been

processed by some automaton, we obtain a bit vector representing which subautomata or rule matched the packet.

If we compare regular finite automaton and automaton for traffic filtering, the main difference is that once a packet has reached the final state of particular subautomaton, it is matched, and we turn the corresponding bit on (from 0 to 1). However, concerning a finite automaton, the packet is accepted until it is read completely and we end in at least one final state. According to this features of our automata, we can spare some useless transitions leading from final states.

State Pruning

The first method identifies less important states of an automaton and removes them appropriately. The removing of the states is quite straightforward. Once the states have been marked as not important, we simply sever them from automaton, including transitions which lead into them. Because we want to achieve over-approximation of the input language, we mark predecessors of removed states as final states. We may also add a self-loop over the alphabet[1] to them, however, as we are concerned only about classification we may neglect this step. In other words, we focus only on some parts of the target language, more precisely on a prefixes of the strings in original languages. Figure 1 illustrates this approach, where we can see the automaton before and after pruning.

In order to decide which state to prune we remove states with the lowest *packet frequency* (or *state frequency*). Its a non-negative number associated with each state of NFA. This number denotes how many packets from input traffic sample went through a particular state during the packets processing. One may say that it is similar to symbol frequency mentioned in [8] when building prefix tree acceptor or frequency automaton. However, the main difference is that one specific packet can be counted at most once. Figure 2 illustrates a packet (state) frequency heat map of some automaton. The states in the close proximity of the initial state have higher frequent (red color), while the others are less visited (green and blue color).

If we remove a state by pruning, we can compute the upper bound of an error (related to an input traffic sample), which is equal to the sum of the frequencies of the pruned states. It means that we will in the worst case make a mistake on this number of packets wrt input traffic sample. In addition, due to nondeterminism, we have to mark all paths in the NFA that can be touched by a packet. Algorithm 1 shows how packet

[1] A transition from a state over the alphabet to the state itself.

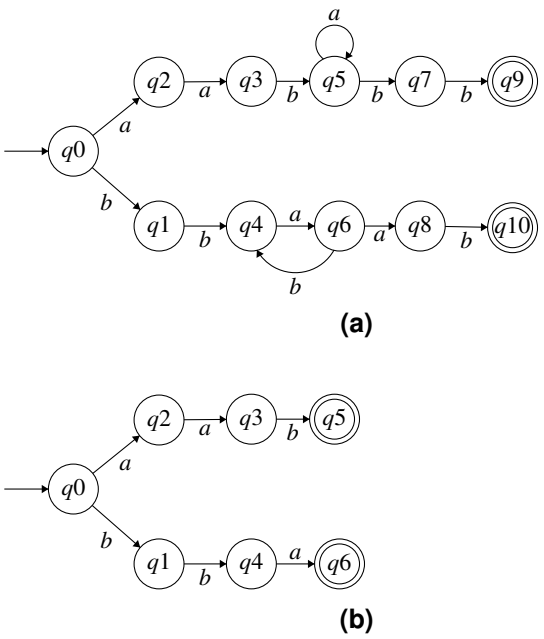


Figure 1. Pruning of the automaton (a) to (b). The states $q7, q8, q9$ and $q10$ were severed and their predecessors, states $q5$ and $q6$ became final states. Note that previous transitions from $q5$ and $q6$ were removed too. Since we care only about classification, not string acceptance, the transitions are redundant.

frequency is used for deciding which states to remove. At first, we sort the frequencies in ascending order. In the next step we, mark states we want to prune until we reach the desired number of states of output automaton. Finally, we sever marked states and propagate final states to their predecessor states.

Algorithm 1: State Pruning Reduction

Input: automaton $M = (Q, \Sigma, \delta, s, F)$, state frequencies mapping $freq : Q \to \mathbb{N}$, reduction ratio r
Output: reduced automaton M_r

1: $s := sort(M, freq)$ `/* sort according to frequency */`
2: $cnt := 0$
3: $marked := \emptyset$
4: **while** $r > cnt/|Q|$ **do**
5: $\quad marked := marked \cup s[cnt]$
6: $\quad cnt := cnt + 1$
7: **end**
8: $M_r := RemoveStates(M, marked)$ `/* remove marked states */`
9: **return** M_r

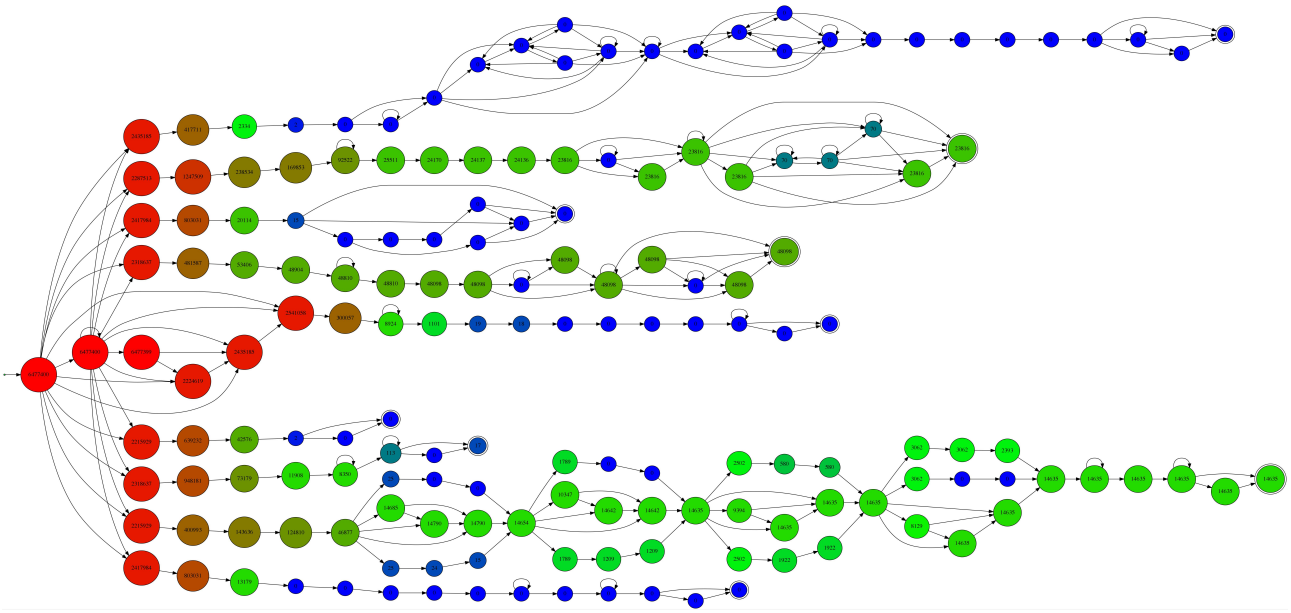


Figure 2. The packet frequency heat map of the automaton (apparently without transition labels). The red states are the most frequently visited, the green are medium frequent and the blue one are almost not visited at all.

State Merging Refinement

The state pruning approach is quite efficient when packets visit only a small part of the automaton. In such cases, we can cut off many states of the automaton, while we obtain a relatively small error. The state pruning is, however, not suitable for all regular expression, for instance, `GET HTTP 1.1 \x0d\x0a\x0d`. In this expression the prefix `GET HTTP 1.1` is very common in traffic, but the sequence of bytes `\x0d\x0a\x0d` is not. Therefore, the reduction would not do much, because many states would have high packet frequency.

To achieve a more significant reduction, we use *state merging*. The fundamental idea is to maintain only parts of automata which are very specific and therefore would cause small error. It involves merging of states which are usually visited by same set of packets. If the frequency of state is similar (with respect to some threshold) to its predecessing state, the states are then merged. This method extends the language in a different way compared to pruning. It merely does not only cut some prefixes but adds iterations of some symbols to the language.

Note that after this method has been applied, the state pruning is used. In the most cases, this refinement led to better results. An example of state merging (without pruning) is depicted in Figure 3, where we reduce the automaton (a) in Figure 1.

Iterative Merging

We can extend state merging by repeating this procedure several times on different samples, which involves in general, different state frequencies. We call this approach *iterative merging*. The point is to anticipate how language was modified after merging reduction (without pruning) and try to join more states together. In other words, we want to merge more states since pruning, which is applied after merging, can yield higher errors.

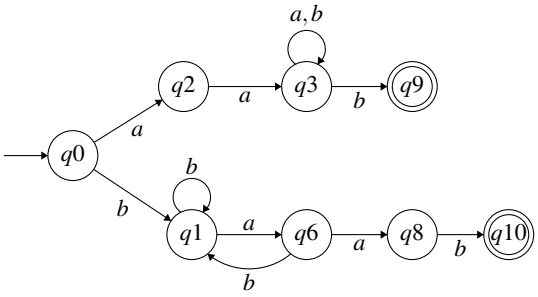


Figure 3. The merging of automaton **(a)** in Figure 1. Notice that states $q5$ and $q7$ were merged to state $q3$, whilst $q4$ was merged to $q1$.

We generalize all methods into one algorithm. The algorithm has two parameters. The reduction ratio r and the number of iterations i. The reduction ratio specifies a proportional number of states of input automaton, which our reduced automaton will have. The $i = 0$ corresponds to pruning, while $i = 1$ is one step merging. For the $i > 1$, we use several merging steps for reduction.

4. Experiments

The proposed reduction methods were implemented and tested on real traffic. In this section, we provide a brief report of obtained results.

Concerning the obtained reduced automata, we

were primarily interested in the two following statistics:

- *accuracy*, which stands for the number of correct classifications divided by the total number of packets with payload. The correct classification means that we visited the same set of final states for the reduced NFA and the original one. Since we are over-approximating the target language, we cannot activate less states than the original NFA.
- *precision*, is the ratio of correctly predicted positive observations (same final states as the original NFA visited) to the total predicted positive observations (at least one final state visited)

The accuracy is the most interesting variable, provided that there is subsequent processing in software after a packet has been classified. However, if we do not have this further processing, we might be then also interested in precision. This tells us how many false alarms we will encounter.

Here we present the results of our methods used on two automata. The first one is called **sprobe** (approximately 160 states). Figure 4 shows results of the state pruning reduction and relevant variables accuracy and precision. Naturally, with increasing reduction ratio accuracy and precision are also increasing. As regard to merging (not displayed in the plot), it yielded only negligibly better results.

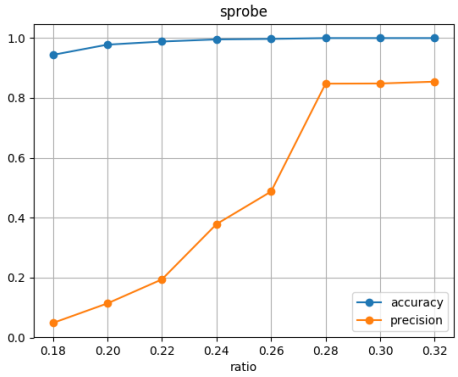
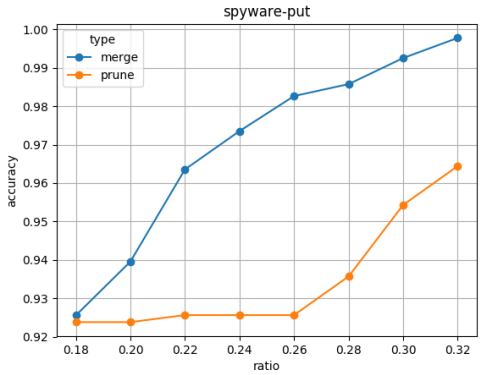
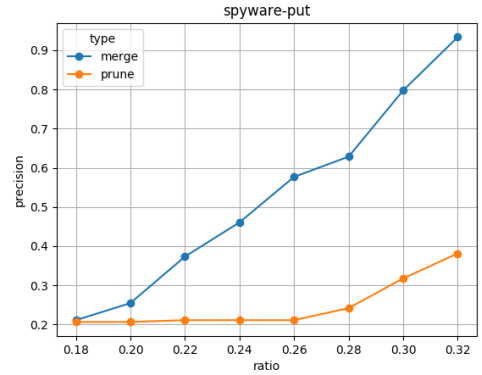


Figure 4. The results of pruning reduction of the automaton **sprobe** with different reduction ratios on the x-axis.

The second automaton, which was acquired from Snort [9], is named **spyware-put.rules** (around 12 500 states). Plot (a) in the Figure 5 highlights results of accuracy for pruning and merging. We can see that in the most cases merging considerably improved accuracy. On the other hand, the second plot (b) shows information about the same automata and reduction ratio, but this time for precision. Again, we can observe that merging has better results.



(a)



(b)

Figure 5. The iterative merging reduction of the automaton **spyware-put.rules**. The graph (a) shows accuracy, while the (b) highlights data about precision with reduction ratio on x-axis.

As regard iterative merging, it may yield slightly higher accuracy and precision. Nevertheless, the difference is not so significant as comparing merging and pruning.

5. Conclusions

By way of conclusion, we proposed several methods for automata reduction based on packet frequency. We saw that even the simple pruning method could produce impressive results. Moreover, we refined pruning with merging, which in some cases improved accuracy.

Although these methods do not provide any formal guarantee wrt input traffic, the results are quite encouraging. Furthermore, we managed to reduce several automata to around 20% of original size, with quite reasonable accuracy. Due to the flexibility of our approach, it is also possible to reduce huge automata with thousands of states, which could not be achieved using previous methods.

The next step of this work will use reduced automata in real-time network traffic. A quite cheap

solution could be computing accuracy just in software on sampled traffic. This approach will show us whether packet frequency, on which basis reductions were made, calculated on our datasets was sufficient. Provided that it will be successful, automata can be synthesized to FPGA and tested more rigorously.

Acknowledgements

I would like to thank profusely my supervisor Tomáš Vojnar, and my advisers Vojtěch Havlena, Ondra Lengál for their guidance and assistance.

References

[1] Denis Matoušek, Jan Kořenek, and Viktor Puš. High-speed regular expression matching with pipelined automata. *2016 International Conference on Field-Programmable Technology (FPT)*, pages 93–100, 2016.

[2] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.

[3] Lorenzo Clemente and Richard Mayr. Advanced automata minimization. *CoRR*, abs/1210.6624, 2012.

[4] Lucian Ilie, Gonzalo Navarro, and Sheng Yu. *On NFA Reductions*, pages 112–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[5] Milan Ceska, Vojtech Havlena, Lukás Holík, Ondrej Lengál, and Tomás Vojnar. Approximate reduction of finite automata for high-speed network intrusion detection. *CoRR*, abs/1710.08647, 2017.

[6] Darpa intrusion detection evaluation.

[7] Dexter C. Kozen. *Automata and Computability*. Springer Verlag, New York, Inc, Secaucus, NJ, USA, 1997.

[8] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

[9] Jeffrey Carr. Snort: Open source network intrusion prevention, 2007. [ONLINE 21.03.2018].