BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

REDUCTIONS OF AUTOMATA USED IN NETWORK TRAFFIC FILTERING

REDUKCE AUTOMATŮ POUŽÍVANÝCH VE FILTRACI SÍŤOVÉHO PROVOZU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JAKUB SEMRIČ**
AUTOR PRÁCE

SUPERVISOR **Prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2018

Abstract

The aim of this work is to propose scalable methods for reducing non-deterministic finite automata used in network traffic filtering. We introduce two approaches of NFAs reduction based on states elimination. 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 provide sufficient reduction. We implemented the methods and evaluated the accuracy of the reduced automata on real traffic. Our approach does not provide any formal guarantee wrt unseen input traffic, but on the other hand, it can be smoothly used on automata of any size, which is a significant problem for existing methods that have very high time complexity and cannot be applied on really large automata.

Abstrakt

Cieľom tejto práce je navrhnúť škálovateľné metódy pre redukciu nedeterministických konečných automatov používaných vo filtrácii paketov. Uvádzame dva prísty redukcie automatov založené na elminácii stavov. Aby sme dosiahli významnú redukciu automatu, používame techniky nezachovávajúce jazyk so zameraním na nad-aproximáciu, keďže redukcie so zachovaním pôvodného jazyka nemusia byť dostatočne účinné. Implementovali sme dané metódy a vyhodnotili presnosť redukovaných automatov na reálnych vzorkoch. Náš prístup neposkytuje žiadne formále záruky vzhľadom na nepoužité dáta, ale može byť hladko použitý na automaty akejkoľvek veľkosti, čo je hlavný problém existujúcich metód, ktoré majú vysokou časovou zložitosťou a nemôžu byť aplikované na veľké automaty.

Keywords

finite automata, automata reduction, deep packet inspection, network intrusion detection system

Klíčová slova

konečné automaty, redukcia automatov, hlboká analýza paketov, sieťový detekčný systém narušenia

Reference

SEMRIČ, Jakub. *Reductions of Automata Used in Network Traffic Filtering.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Prof. Ing. Tomáš Vojnar, Ph.D.

Reductions of Automata Used in Network Traffic Filtering

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of prog. Ing. Tomáš Vojnar Ph.D. The supplementary information was provided by Ing. Vojtěch Havelna and Ing. Ondrej Lengál Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Jakub Semrič
May 16, 2018

Acknowledgements

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

Contents

Chapter 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. Often, 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 or used hardware packet preprocessing. Detection systems are required to analyze a packet and invoke a corresponding reaction immediately after it has been received with minimal latency.

In general, REs are represented in hardware as finite automata. 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 (e.g., for filtering the network traffic at 400 GiB per sec. approximately 63 copies of the base automaton were used in [19]). To tackle this problem, we propose reductions which attempt adjust the size of automata with a reasonable trade-off between the classification error and the number of states of the automaton.

If we limit ourselves to deterministic automata, one can use Hopcroft's algorithm [14], to obtain minimal deterministic automata. Nevertheless, a problem appears when one wants to deal with immense non-deterministic automata (NFA), where state explosion during determinization may occur. On the other hand, there also exist algorithms of reducing the size of non-deterministic automata directly, without need of determinization. These approaches are based on various simulation techniques discussed, e.g., in [9, 15]. Although a reduced automaton using the mentioned techniques can be several times smaller than the original ones, the reduction may still not be as sufficient as it is desired, which is illustrated in our experiments too.

There are not many language non-preserving reduction methods. Out of them, the closest to this work is described in [8]. This approach is driven by the probabilistic distance between regular languages and we will shortly discuss it in a later chapter. The method provides a formal guarantee wrt unseen data. However, there are two major drawbacks. Firstly, the method has high time complexity which makes it hard to apply on large automata. Secondly, it relies on general network traffic model represented by a probabilistic automaton. Acquiring of an exact model is extremely difficult because of the diversity of traffic containing various content. The algorithms for learning probabilistic automata are

not suited for such samples including a lot of noise represented by binary and tunneled data as we also discuss in the work.

To significantly reduce the size of automata in a fast and flexible way, we propose a different approach based on language over-approximation. Although the language of reduced the 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, 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, the 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 the ANT@FIT research group. The rest of the samples were acquired from the DARPA traffic dumps [2]. 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.

This work is organized as follows. First and foremost, we give some preliminaries in Chapter 2, which introduces the reader to a basic background of the theory of automata and regular languages. Further, Chapter 3 briefly discusses the network traffic filtering and involvement of finite automata in this process. Following this, in Chapter 4, we give a detailed account to proposed reduction methods including their algorithmic description. In Chapter 5, besides describing the implementation of the proposed reductions, we also talk about error evaluation used after reduction. Furthermore, Chapter 6 provides an experimental evaluation of the proposed reduction techniques, primarily their error and the reduction rate. Finally, a summary of acquired results and future ideas of our approach can be found in Chapter 7.

Chapter 2

Preliminaries

This chapter covers fundamental definitions, which are used throughout the thesis. If all these concepts are familiar to the reader, the chapter can be freely skipped. Definitions in this section are mostly inspired by books and online references, which can be found in [16, 23, 10].

Just before we jump into the main concepts, we describe the most fundamental basics of the considered area, which are strings and languages. Then, we introduce finite automata which are one of the possible representation of regular languages. We also discuss regular expressions, including a formal definition, and give some examples. Finally, we describe automata reductions and other types of automata such as probabilistic ones.

2.1 Strings and Languages

Definition 1. *An* alphabet *is any finite set, which is denoted as* Σ*. We call elements of an alphabet* symbols *or* letters*. A* string *or* word *over* Σ *is any finite-length sequence of symbols of* Σ*.*

An example of a string could be $x = abba$ over the alphabet $\Sigma = \{a, b\}$. Next, the *length* of a string w is denoted as $|w|$ and represents the number of symbols in the string. For example $|abba| = 4$. There exist a unique *empty string* over Σ. It is denoted by the Greek letter ε and has the length equal to zero ($|\varepsilon| = 0$).

Further, we denote the set of all strings over alphabet Σ as Σ^*. A formal language L over the alphabet Σ is then any subset of Σ^*. There also exist a special kind of language, called the *empty language*, which contains no strings at all, and we write $L = \emptyset$.

Besides basic set operations (intersection, union, symmetric difference ...) , which can be applied on languages, there are other common operations:

- Concatenation L_1L_2: the set of strings that can be obtained by concatenating a string in L_1 and a string in L_2. For instance, $\{xy, w\}\{a, bc\} = \{xya, xybc, wa, wbc\}$.
- Kleene star L^*: this is the set containing all strings that can be acquired by concatenating any finite number (including zero) of strings from L. For example, $\{x, y\}^* = \{\varepsilon, x, y, xy, yx, xx, yy, \dots\}$.

2.2 Automata

Next, we give a definition about one of possible representation of regular languages, which is a finite automaton.

Definition 2. *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 are often interested in strings and their relations with finite automata (FA). For that we give a definition of the function $\widehat{\delta}$.

Definition 3. *We define a function* $\widehat{\delta} : Q \times \Sigma^* \to 2^Q$ *from* δ *by induction on the length* x*:*

- $\widehat{\delta}(q, \varepsilon) \stackrel{def}{=} q$,
- $\widehat{\delta}(q, xa) \stackrel{def}{=} \delta(\widehat{\delta}(q, x), a)$.

In other words, the function maps a state $q \in Q$ and a string w to a new set of states $\widehat{\delta}(q, w)$. Furthermore, we say that the automaton $M(Q, \Sigma, \delta, s, F)$ accepts or recognizes a language L, we write $L(M)$, when for all $w \in L$ $\widehat{\delta}(s, w) \subseteq F$.

We also recognize two types of a finite automaton: deterministic (DFA) and nondeterministic (NFA). If a finite automaton is a DFA, it holds that $|\delta(q, a)| \leq 1$ for $q \in Q$ and $a \in \Sigma$ (note that $|S|$ denotes a cardinality of the set S). In general, instead of having one initial state an NFA can have a set of initial states. However, a DFA can only have a single initial state.

For every regular language, there exists a minimal deterministic automaton that accepts it. However, for some regular languages, there are NFAs whose size can be even exponentially smaller than that of the minimal DFA recognizing the same language. The transformation of an NFA to a DFA is called determinization, which is done by subset construction. The application of the subset construction method on an NFA creates states, which are subsets of Q. Thus, in the worst case, the number of states is equal to $2^{|Q|}$.

Figure 2.1 shows the difference between DFAs and NFAs. Concerning the NFA (the automaton on the right), we see that there are more transitions under one symbol leading from one state (in this case, symbol a and state $q0$).

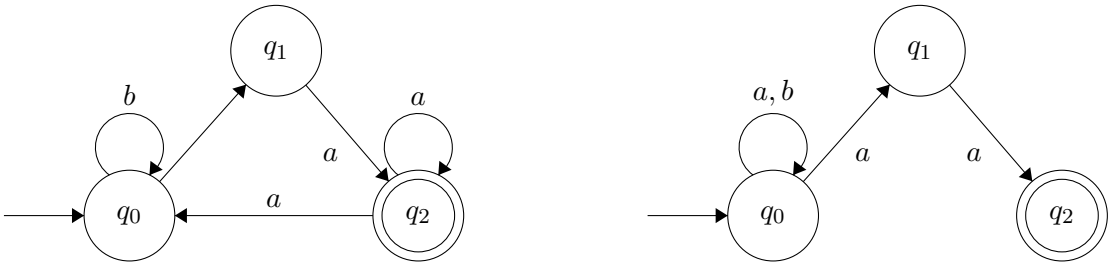

Figure 2.1: DFA (on the left) and NFA (on the right) accepting the same language. Note that q_0 in NFA can reach two states after reading symbol a (that is why it is called non-deterministic).

Definition 4. *We call $M' = (Q', \Sigma, \delta', s', F')$ a subautomaton of $M = (Q, \Sigma, \delta, s, F)$ iff*

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

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

2.3 Regular Expressions

Formally, regular expressions (RE) are defined as follows (taken from [1]).

Definition 5. *Let Σ be an alphabet. The regular expressions over Σ and languages they denote are defined as follows:*

- $\emptyset$ *is an RE denoting the empty set,*
- ε *is an RE denoting* $\{\varepsilon\}$,
- a, *where* $a \in \Sigma$, *is an RE denoting* $\{a\}$.

Let r and s be regular expressions denoting languages L_r and L_s, in turn; then

- $(r \cdot s)$ *is an RE denoting* $L = L_r L_s$,
- $(r + s)$ *is an RE denoting* $L = L_r \cup L_s$,
- (r^*) *is an RE denoting* $L = L_r^*$

Use of REs is very popular in computer science and engineering because even with a small string expression, we can describe a quite complex language. Moreover, REs can be handled in an effective way, typically by translation to automata. For instance, example of RE describing the same language as the automaton in Figure 2.1 can be written as `(a|b)*aa`. Besides standard RE symbols mentioned in the definition above, there have been added several special symbols and flags for simpler language description. These symbols are for instance `^` (matches start of the string), `[]` (set of characters), etc.

2.4 Automata Reductions

Concerning DFA reductions, we can use algorithms based on the Myhill-Nerode relations. These methods distinguish states of the DFA into equivalence classes. However, the size of a minimized DFA can be exponentially larger than the size of an NFA accepting the same language. Nevertheless, there also exists approaches for reducing an NFA directly. One of those approaches is based on merging states according to the simulation equivalence relation. The definition of the simulation equivalence is given below [13].

Definition 6. *A (forward) simulation on an NFA* $A = (Q, \Sigma, \delta, s, F)$ *is a binary relation* $R \subseteq Q \times Q$ *such that, for any state* $p, q \in Q$ *and* $a \in \Sigma$, $(p, q) \in R$ *holds, iff*

- $p \in F \implies q \in F$, *and*
- *for every* $p' \in \delta(p, a)$ *there exists* $q' \in \delta(q, a)$ *such that* $(p', q') \in R$

For each NFA there also exist a unique largest simulation, called *simulation preorder*. The *simulation equivalence* for the simulation preorder $\preceq$ is then given as $\preceq \cap \preceq^{-1}$. These methods reduce the NFA just by state merging, but they can be combined, e.g., with removing of transitions as used in [9].

2.5 Probabilistic and Frequency Automata

In the following chapters, we will often use a term *frequency*. Generally, we consider frequency as the number of times an event occurs, typically over some period of time. For example, it could be the number of sunny days in the previous week. In terms of regular languages, we can also have a frequency finite automata (FFA), which is defined as follows.

Definition 7. *A frequency finite automaton is a tuple* $A = (Q, \Sigma, \delta, \delta_{fr}, I_{fr}, F_{fr})$ *where*

- Q, δ, Σ *are the same as in the definition of a finite automaton,*
- $I_{fr} : Q \to \mathbb{N}$ *are initial-state frequencies,*
- $F_{fr} : Q \to \mathbb{N}$ *are final-state frequencies,*
- $\delta_{fr} : Q \times \Sigma \times Q \to \mathbb{N}$ *is the transition frequency function.*

For a deterministic frequency finite automaton (DFFA) should hold that for each state the sum of the entering frequencies is equal to the sum of the leaving frequencies. DFFAs are often used for learning some distributions over regular languages by some learning algorithm such as Alergia, MDI or DSAI [12]. The result is then a probabilistic automaton, which can be defined similarly as an FFA, where instead of frequencies we have probabilities (relative frequencies).

Chapter 3

Network Traffic Filtering

Here, we introduce the area of network traffic filtering, including its possible HW-acceleration, and a use of automata involved in this process. The most of this part is inspired by [18, 21, 22, 20]. First and foremost, we remind basic concepts related to network architecture and packet classification. Following this, deep packet inspection and intrusion detection systems are briefly discussed. Finally, we describe the purpose of automata in pattern matching and its hardware realization.

3.1 Network Architecture

In general, the Internet consists of thousands of smaller networks. These networks are composed of nodes (computers, routers, etc.) connected together. The communication on the Internet then proceeds by sending data (packets) between nodes using various protocols in several network layers.

Today, we can distinguish network layers in the five-level model (sometimes called TCP/IP model). This model was developed primarily empirically as people gained experience with the actual problems of working with computer network connections and with the solutions to those problems. Firstly, the lowest layer is called the physical layer. It consists of hardware devices and some medium (cables, radio waves) which connects them. Ethernet, FDDI, and Token Ring are the most common technologies used in this layer. Secondly, link layer connects nodes within local area network (LAN) and provides routing based on MAC addresses. Thirdly, we have the Internet layer, which extends LAN to a wider network so that computers can communicate remotely outside their local network. Nowadays IPv4 protocol still dominates the Internet but is being slowly replaced with its newer version IPv6. Next, the transport layer assures that data is delivered either with some guarantee (TCP) or with none (UDP). Finally, we have the application layer, where we can find myriads of various application protocols, including the most popular HTTP.

According to the Cisco model, the application layer further consists of three levels, including presentation, session, and application layer which are denoted as L5, L6, and L7 respectively.

If we look into a packet structure, we can recognize these layers by protocols. For instance, an HTTP packet may consist of an HTTP, TCP, IP, and finally, Ethernet header.

3.2 Classification

One of the primary tasks of network devices (routers, switches) is packet classification. Besides packet routing based on IP address, classification also serves for traffic filtering in firewalls. Filtering splits the traffic into several categories, which are then handled by a specific function which determines what happens next with the packet.

The basic classification looks into headers on the IP and transport layer and searches for specific fields. Following this, with respect to the set of filtering rules, the next action takes place. For instance, the packet is thrown away, redirected, analyzed, processed, etc. However, there are also classifications which do not parse the packet header, but its content (packet payload). This kind of classification is called deep packet inspection (DIP) and is described in the following section.

3.3 Deep Packet Inspection

In general, DPI is an advanced approach, which examines the contents of packets passing through a particular node and makes real-time decisions based on the set of rules assigned by an Internet service provider (ISP) or network manager, depending on what a packet contains.

Besides extensive use in Intrusion Detection Systems (IDS), which are described in the next section, it can also be used in network management to streamline the flow of network traffic. For example, a message tagged as high priority can be routed to its destination ahead of less important or low-priority messages or packets involved in casual Internet browsing. DPI can also be used for throttled data transfer to prevent peer-to-peer abuse, therefore, improving network performance.

According to [20], DPI has at least three significant limitations. First , while protecting against some existing vulnerabilities it can create new ones. For instance, while effective against buffer overflow attacks, certain types of malicious software, and (D)DoS[1] attacks, DPI can also be exploited to facilitate attacks in those same categories.

Second, deep packet inspection contributes to the complexity and cumbersome nature of existing firewalls and other security-related software. Deep packet inspection systems require periodic revisions and updates to remain optimally effective again newly developed attacks.

Third, DPI can reduce network speed because it increases the burden on firewall processors. This is also a reason why there is a need to implement DPI systems in hardware (often based on the FPGA architecture[2]). Hardware processing is inevitable in routers in high-speed networks.

3.4 Intrusion Detection System

An intrusion detection system is a system for monitoring suspicious activity in network traffic and providing alerts once such behavior has been discovered. Besides detection and reporting of malicious activity or anomalous traffic, some IDSs are capable of taking counteractions when an intrusion is detected, including blocking traffic sent from suspicious IP addresses.

[1] (Distributed) Denial-of-Service
[2] Field Programmable Grid Array

Intrusion detection systems monitor network traffic in order to detect when an intrusion is being carried out by unauthorized entities. IDSes do this by providing some or all of the following functions to security professionals:

- reacting to violators by blocking them,
- monitoring the operation of routers, firewalls, key management servers, and files that are needed by other security controls aimed at detecting, preventing, or recovering from cyberattacks,
- providing a user-friendly interface so non-expert staff members can assist with managing system security,
- providing administrators a way to tune, organize and understand relevant operating system audit trails and other logs that are often otherwise difficult to track or parse,
- recognizing and reporting when the IDS detects that data files have been altered,
- including an extensive attack signature database against which information from the system can be matched,
- generating an alarm and notifying that security has been breached.

3.5 Finite Automata Used in Filtering

DPI filtering rules for pattern matching are represented by regular expressions and are implemented as an FA. Due to increasing the throughput of network devices the solutions based solely on software processing are not sufficient. To obtain speeds suitable for real-time pattern matching it necessary to implement automata in hardware. In hardware, an FA is a sequential circuit, which compared to a combinatorial circuit has its own memory.

There are two primary architectures based on DFAs or NFAs. Concerning DFAs, we can have only one active state at once, which allows us to store the transition function into RAM memory. This is also useful when we want to change the transition function because there is no need to make any changes in hardware. However, as we mentioned in the previous chapter, the size of a DFA can be exponentially larger than an NFA accepting the same language.

When using NFA-based approach, the automata are implemented in hardware including the transition function because we can have more active states. Although NFAs can be relatively small compared to DFAs the problem occurs when we want to update the transition function.

Another solution is to use *delayed input* DFAs (D2FA), which are DFAs with some non-deterministic features [17]. In D2FA, each state may have at most one unlabeled outgoing default transition. These automata have the same advantage as DFAs, so their transitions can be stored in RAM. Moreover, they consist of fewer transitions than DFAs, which is the main reason why they are used.

3.6 Filtering Rules

We obtained regular expressions from the the research group ANT@FIT in the PCRE[3] format. This format is widely used in many programming languages including Perl and PHP. In the following subsection, we describe systems and software products, which these REs come from.

Snort

Snort is an open source network-based intrusion detection system capable of real-time traffic analysis and packet logging on Internet networks [7]. The Snort main features are protocol analysis, content searching and matching. In particular, it is used to detect attacks and other malicious activities such as operating system fingerprinting attempts, semantic URL attacks, buffer overflows, (D)DoS, server message block probes, and stealth port scans.

An example of REs from Snort is given in Figure 3.1 below. After the REs were converted to an automaton, we obtained an NFA with around 1 300 states and 8 000 transitions.

```
/awstats.pl?[^\r\n]*configdir=\x7C/Ui
/awstats.pl?[^\r\n]*logfile=\x7C/Ui
/calendar(|[-_]admin)\.pl/Ui
/db4web_c(\.exe)?\/.*(\.\.[\#\/]|[a-z]\:)/smiU
/evtdump\x3f.*?\x2525[^\x20]*?\x20HTTP/i
/instancename=[^&\x3b\r\n]{513}/smi
/itemid=\d*[^\d\&\;\r\n]/i
/pwd=(\!|\%21)CRYPT(\!|\%21)[A-Z0-9]{512}/i
/ShellExample.cgi\?[^\n\r\&]*\x2a/Ui
/SoftCart.exe\?[^\s]{100}/Usmi
```

Figure 3.1: Regular expressions in the PCRE format of web-cgi.rules. These REs are searching for a suspicious content of a packet. For instance, the rule `ShellExample...` detects remote attackers attempting to list arbitrary directories via a URL with the desired path and a * (asterisk) character.

Besides Snort, there are plenty of other products providing similar capabilities. A system developed by Bro[4] is an example of such framework for network security monitoring.

L7-filter

L7-filter is a software product that provides a classifier for Linux's Netfilter subsystem which can categorize Internet Protocol packets based on their application layer data (thus the L7 layer) [5]. The major goal of this tool is to make possible the identification of peer-to-peer programs, which use unpredictable port numbers in TCP or UDP source and destination port fields. The program uses regular expressions for the network protocol identification. This technique, used in conjunction with Linux's QoS (Quality of Service) system, allows

[3] Perl Compatible Regular Expressions https://pcre.org/
[4] https://www.bro.org/

application-specific yet port-independent traffic shaping. An example of some rules are shown in Figure 3.2.

```
/<html.*><head>/
/User-Agent: DA [678]\.[0-9]/
/User-Agent: FreshDownload\/[456](\.[0-9][0-9]?)?/
/http\/(0\.9|1\.0|1\.1).*(user-agent: itunes)/
```

Figure 3.2: An example of filtering rules obtained from L7-filter. We can see that the first rule is searching for an html content, while the other rules are used for analyzing user-agent request headers. In general, user-agent headers contain a characteristic string that allows the network protocol peers to identify the application type, operating system, software vendor or software version of the requesting software user agent.

Chapter 4

Automata Reduction

In this chapter, we describe our solution of approximate automata reduction. Initially, the basic concept of approximate reduction will be discussed. Next, we explain some important difference between regular NFAs and NFAs used in network traffic filtering. Then, we will introduce our initial proposal of approximate reduction. After this, we describe a refinement of this reduction method and discuss its advantages and disadvantages.

4.1 Approximation

As regards NFA reduction methods which preserve language of the orignial automaton, they can yield excellent results. For instance, the number of states can be even more than twice lower than in the original automaton. However, one may want to shrink an automaton to a greater extent, but the reduction is limited by retaining the original language. For that reason, other methods which do not preserve the language are necessary. Although they would certainly yield some error, they allow us to achieve more significant reductions.

If we want our reduction to modify the original language, we have basically three options. First, we can under-approximate the language, e.g., by neglecting several rules (subautomata) in a given NFA. To decide which subatomata to remove, we can choose those which final states were the least visited. However, following this approach, we would lose the opportunity to handle many attacks. Therefore, our IDS would be vulnerable against particular attacks, which is too dangerous.

Secondly, we can change the language that it will neither be under-approximation nor over-approximation of the target language. Then the erroneously classified packets would lie in the symmetric difference of the reduced and the original language. Despite the fact that we could obtain a quite low error, there still would be unsolved issues in the system. These issues include false alarms and undetected attacks.

Finally, we can use over-approximation whose error would consist only of false positives. Generally, ignoring an attack is a greater risk than having false alarms. This approach ensures that no attacks are ignored, which is more desirable when protection is needed. But what to do with these false alarms? By making our system too permissive with a great rate of false alarms, we would block a large amount of casual network traffic. The solution is subsequent processing in the CPU of a router or any other device where these systems should be installed. Figure 4.1 illustrates the classification process and shows that after a packet has been classified as potentially dangerous by hardware, it is then checked in software for a false alarm.

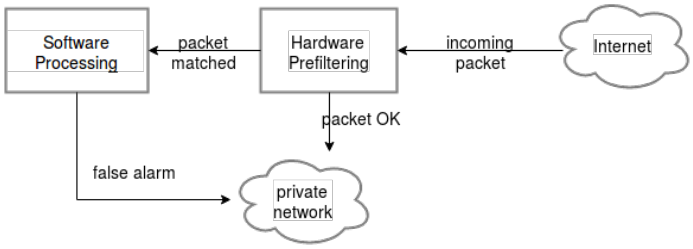

Figure 4.1: The diagram shows how the problem with false alarms is tackled. First, the packet is processed in hardware (reduced automata). Then, if the packet is matched, it is subsequently verified in the software running on the CPU of a network device.

The approach based on over-approximation can reduce the size of automata more significantly than language-preserving reduction. Since we also want to reduce the possible errors, we will focus on a language approximation illustrated in Figure 4.2. It shows how the original and reduced language should be related with network traffic. The language of reduced automata can extend a language vastly, but concerning the frequent traffic (red dashed circle), it should remain very close to the original language.

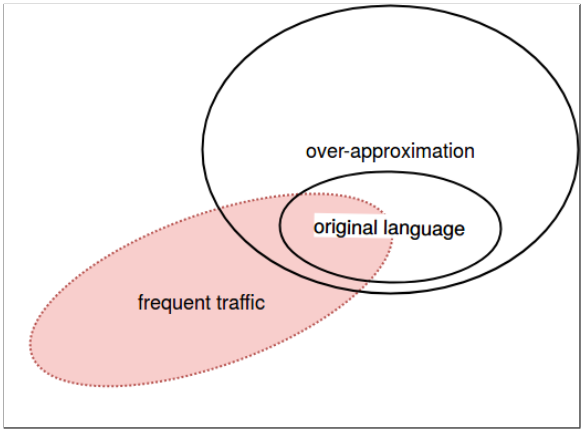

Figure 4.2: Diagrams of languages of the original automaton, the reduced automaton, and frequent traffic. The reduced automaton attempts to over-approximate the target with low false positive rate (intersection with red ellipse outside the original language).

4.2 NFA Differences

In packet pattern matching, we can have several filtering rules which are represented by different REs. In the automaton, the rules are represented by several smaller subautomata. The final states of each subautomaton identify particular rules, which are used for classification, e.g., recognizing a type of attack or protocol. So the reduction should not merge all final states, because we would lose the information about which filtering rule to apply.

In contrast to string acceptance in regular automata, the classification process is a bit different. Since we search for substrings in packets, not a full matches, we only need to know which finals states were reached by a packet during its processing. This allows us to drop self-loops over the final state.

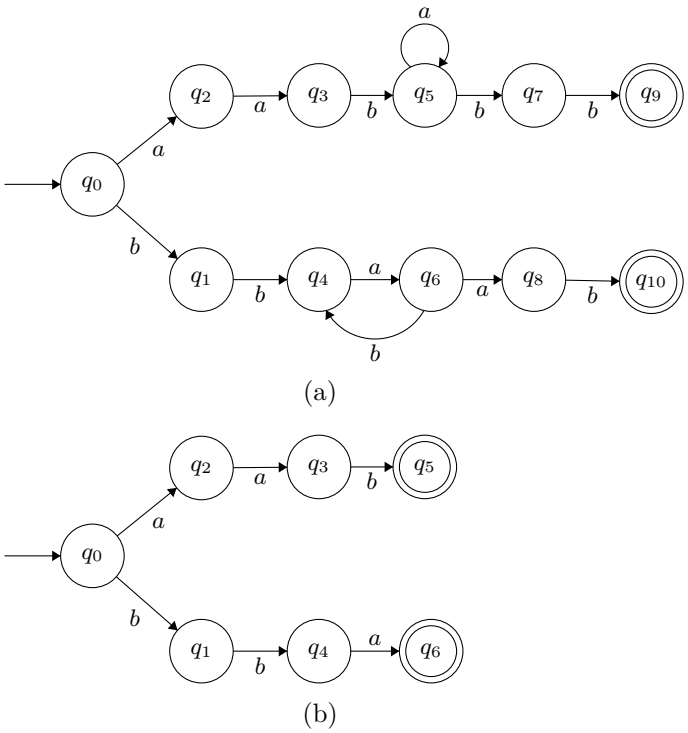

Figure 4.3: Pruning of the automaton (a) to (b). The states q_7, q_8, q_9 and q_{10} were severed and their predecessors, states q_5 and q_6 became final states. Note that the transitions from q_5 and q_6 were removed too, since we are concerned only about which final states are visited by a packet.

4.3 State Pruning

In this chapter, we will describe our first method used for NFA reduction, which we call *state pruning*. This approach follows simple ideas and is not difficult for both understanding and implementation. The following sections describe how our approach reduces an automaton and how it modifies its language after reduction.

4.3.1 Basic Idea

The state pruning reduction 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 to them. Because we want to achieve over-approximation of the input language, we mark immediate predecessors of the removed states as final states. We may also add a self-loop over the alphabet[1] to them, however, as we are only concerned about which final states were visited we may neglect this step. Figure 4.3 illustrates this approach, by showing the automaton before and after pruning.

To better understand this principle, imagine a simple regular expression r over the alphabet $\Sigma = \{a, b, c\}$ $r = abcabcccaa$, whose NFA representation we want to reduce. Once

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

it has been pruned, we may obtain something like $abca(a|b|c)^*$. We see that we neglected the last six characters and substituted them with a self-loop.

4.3.2 State Frequency

In order to decide which state to prune, we use *packet frequency* (or also *state frequency* denoting the same). Its a non-negative number associated with each state of an NFA. This number denotes how many packets from the input traffic sample went through a particular state during the packets processing. The states with the low packet frequency are then removed. The main difference between classic frequency and packet frequency is that we do not compute frequency for each character, but for each packet. Figure 4.4 illustrates a packet frequency heat map of some automaton. The states in the close proximity of the initial state have higher frequency (red color), while the others are less visited (green and blue color). This distribution of packet frequency is typical for the majority of NFAs obtained from Snort.

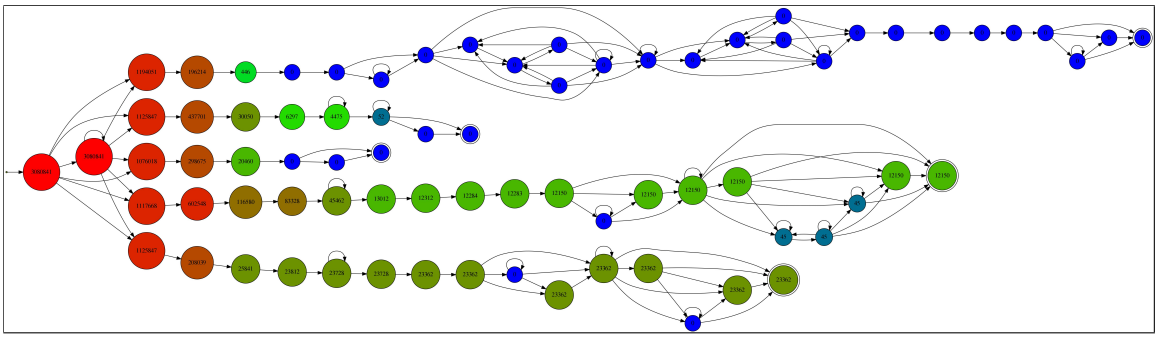

Figure 4.4: The packet frequency heat map of the automaton (apparently without transition labels). The red states are the most frequently visited, the green are medium frequent and the blue one are almost not visited at all.

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 the input traffic sample. In addition, due to nondeterminism, we have to mark all paths in the NFA that can be visited by a packet. Algorithm 1 shows how packet frequency is used for deciding which states to remove. Besides the automaton we want to reduce, it has two parameters a reduction ratio r and a frequency mapping $freq$. The reduction ratio r specifies the number of state of the reduced automaton proportional to the number of states of the input automaton (e.g., for $r = 0.2$ the number of states of the reduced NFA is 20 % of the original NFA). The parameter $freq$ maps each state to its packet frequency obtained from sample.

First, we sort the frequencies in the ascending order. In the next step we, mark states we want to prune until we reach the desired number of states of the output automaton. Finally, we sever marked states and propagate final states to their predecessor states.

Algorithm 1: State Pruning Reduction

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

Output: NFA M' obtained by reducing M, where $L(M) \subseteq L(M')$

```
1: s := sort(M, freq)  // sort according to frequency
2: cnt := 0
3: marked := ∅
4: while r > cnt/|Q| do
5:     marked := marked ∪ {s[cnt]}
6:     cnt := cnt + 1
7: end
8: M' := RemoveStates(M, marked)  // remove marked states
```

4.3.3 Discussion

The reduction based on removing states of the NFA with the lowest packet frequency is a simple and effective approach. The time complexity depends on computing the packet frequency and the pruning process. Due to non-determinism, computing of the packet frequency can be hard, because we have to expand all reachable states for each symbol read. This problem aggravates when an NFA contains a lot of self-loop transitions, mostly close to the initial state (so they are visited often). So these states are continuously being expanded, which slows down computing. This problem is addressed by joining these states together (more details in Chapter 5). The pruning reduction itself is not time consuming. We only sort an array of packet frequencies of states and remove selected states. Following this step, we then propagate the final states to predecessors of erased states.

After applying state pruning reduction to some automata, we obtained quite good results. More detailed description about experiments including reduced NFAs evaluation is provided in Chapter 6. In average, we could reduce NFAs to 20 % of their size with around 1 % error on traffic. The reduction was quite significant on automata obtained from Snort. The main reason why reduction was so successful lies in the two following factors:

The first one is that the degree centrality of the states is quite low. The degree centrality is a simple centrality measure in graph structures that count how many neighbors a node has. If a graph is directed (NFA), we can either choose whether this number represents the count of predecessor or successor nodes. Concerning NFAs, which can be considered as a directed graph, we use degree centrality as the number of successor states. The pie charts in Figure 4.5 shows the distribution of this feature on some automata obtained from Snort. We can see that the most common degree centrality or a number of successors is equal to one. This contributes to the fact, that if we remove a state it will not cause significant modification to a language. On the other hand, if states had very high degree centrality, the language would be changed more considerably after pruning, and thus the error would be more increased.

The second factor which considerably contributes to the effectiveness of the pruning reduction is that the most of the packets, which are accepted by automata for attack detection, are rare in casual network traffic. For instance, if we look at the regular expression in Figure 3.1, we can see the RE "`instancename=[^&\x3b\r\n]{513}`". This RE is both quite specific (`instancename` followed by 513 characters in a set `[^&\x3b\r\n]`) and also has

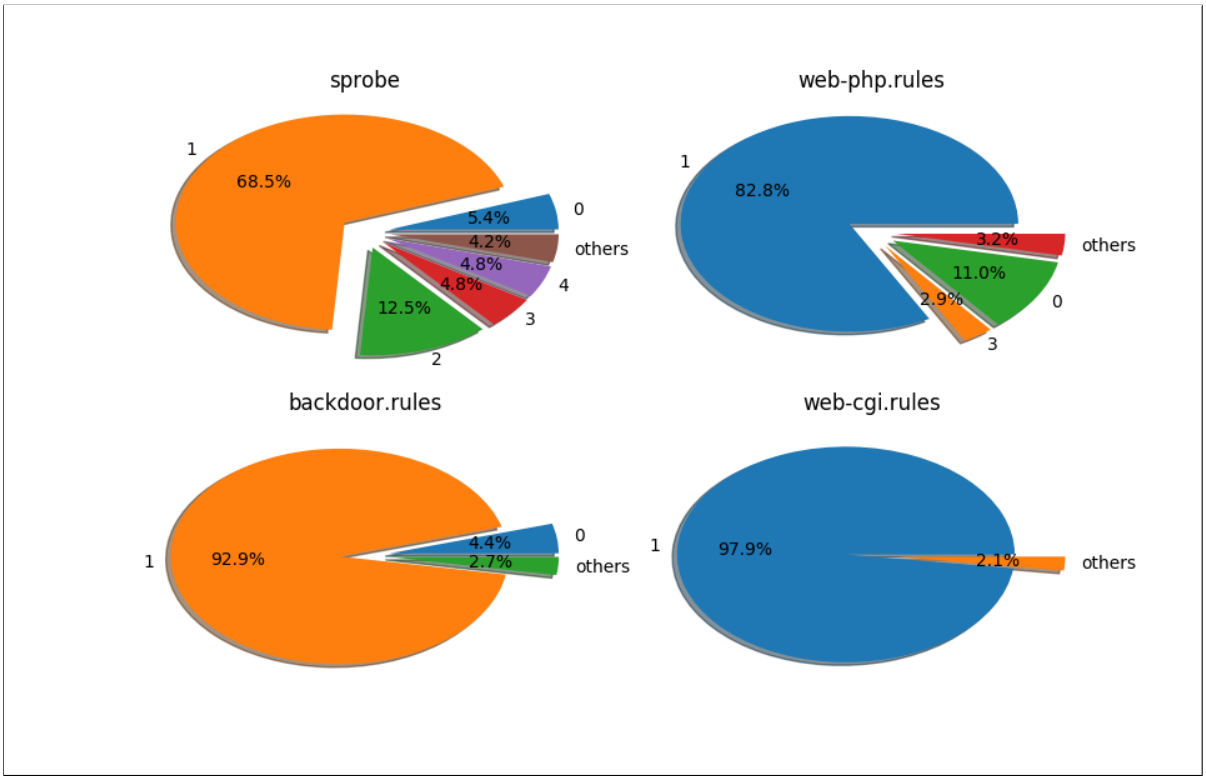

Figure 4.5: The pie charts illustrating the degree centrality or the number of successor states of several automata obtained from Snort. We can see that the majority of states has the degree centrality equal to one.

meager degree centrality (no iterations, so subautomaton would consist of a long sequence of states).

4.4 State Merging Refinement

Although the pruning reduction yields quite feasible results, it can be slightly improved using the approach that we propose below. This improvement consists in state merging, which is described in the following sections.

4.4.1 Basic Concept

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

To achieve a more significant reduction in cases similar to the above, we propose an approach based on *state merging*. The fundamental idea is to maintain only parts of automata which are very specific, regardless of where they appear in the NFA. The reduction

merges adjacent states if their packet frequency is similar wrt to some threshold. This method approximates the language in a different way compared to pruning. It does not just cut some parts of the NFA as pruning does, but adds iterations of some symbols to the language. However, this modification of the language can cause a much higher error. Therefore, the merging should be done carefully.

4.4.2 Merging Procedure

Firstly, we describe how we merge two states together into a single state while preserving over-approximating the original language. For example, when merging a state p with q in the automaton $M = (Q, \Sigma, \delta, s, F)$, we proceed as follows. Initially, we substitute p for q on the left side of each rule in δ where q appears. For instance, $\delta(q, a) \to q'$ will be changed to $\delta(p, a) \to q'$. Then we redirect all transitions to p, such that they lead to q, e.g., $\delta(q', a) \to q$ is changed to $\delta(q', a) \to p$. If q was a final or initial state, we make p final or initial as well. Finally, we remove q from the set of states Q and F.

Figure 4.6 illustrates how the merging of two states is done. Notice that state q_1 gained a self-loop because of the former transitions to the merged state.

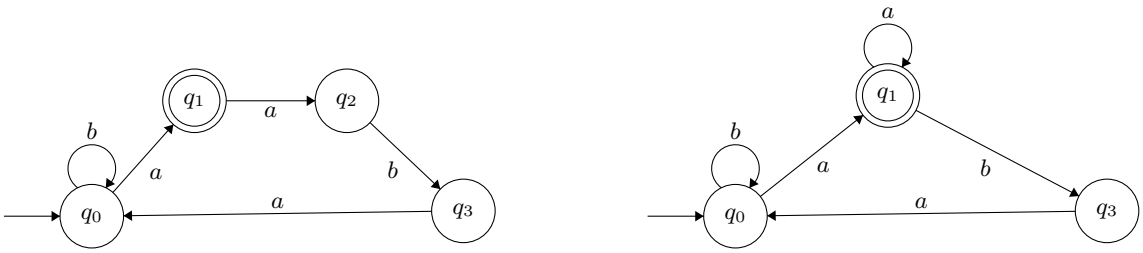

Figure 4.6: The merging of the state q_2 into q_1 of the automaton on the left.

Now let us have a look at merging reduction of the automaton as depicted in Algorithm 2. Besides the state frequency, we have two extra parameters a threshold th and a maximal frequency ratio $freq_{max}$. The threshold th says what the maximal difference between frequencies of two states that we allow to be merge is. For example, if we want to merge q into p, where $freq[q] = 1000$, and $freq[p] = 970$ the threshold must be less than $970/1000$. The second parameter, $freq_{max}$, is a ratio between the maximal packet frequency and the packet frequency of the state allowed to be merged. This parameter limits the merging of states with the high packet frequency, which may yield a high error after merging. For instance, setting this parameter to 0.1 means that only states which have less than 10 % of total packets used for computing frequency are allowed to be merged.

The algorithm iterates over all states using the breadth-first search method. It starts with the initial set of states *actual*, which contains the initial state of the NFA. We also have the set *visited* representing visited states. We loop over the states in *actual*. Then, if the frequency of a state is nonzero and not higher than the maximal frequency allowed for merging, we iterate over this state successors states. If the packet frequencies of the states are similar we mark them for merging. Next, we update sets *actual* and *visited*. Once the main while loop ends, we merge marked states together.

The reduction ratio parameter in merging is not included in the algorithm. This is because the merging reduces around 7 % of states on average in the automata we considered.

Algorithm 2: State Merging Reduction

Input: automaton $M = (Q, \Sigma, \delta, s, F)$, state frequencies mapping $freq : Q \to \mathbb{N}$, threshold $th \in (0, 1)$, maximal frequency ratio $freq_{max} \in (0, 1)$

Output: NFA M' obtained by reducing M, where $L(M) \subseteq L(M')$

```
1:  actual := {s}
2:  visited := {s}
3:  marked := ∅
4:  while actual ≠ ∅ do
5:      next := ∅
6:      foreach state p ∈ actual do
7:          t := freq[p]/max(freq)
            // get successor states of p
8:          suc := succesors(M, p)
9:          if t ≤ freq_max AND freq[p] > 0 then
10:             foreach state q ∈ suc do
                    // compute frequency ratio between p and q
11:                 freq_ratio := min(freq[q], freq[p])/max(freq[q], freq[p])
12:                 if freq_ratio > th then
13:                     marked := marked ∪ {(q, p)}
14:                 end
15:             end
16:         end
17:         next := next ∪ suc
18:     end
19:     actual := next \ visited
20:     visited := visited ∪ next
21: end
22: M' := MergeStates(M, marked)
```

After merging we use pruning, which is not constrained by any conditions (we can remove as many states as we want). In practice, we reduce to around 10 - 30 % of the original size of the NFA, so the reduction ratio parameter would be useless in merging.

We can also skip the merging states with final states. This is because, in the majority of the obtained NFAs, the final states do not have any successor states. Thus merging a state with its successor final state would mean pruning. Since we want to distinguish the impact of the merging from the pruning in our experiments, we do not merge final states.

4.4.3 Iterative Merging

We can extend state merging by repeating this procedure several times on the same automaton. We call this approach *iterative merging*. The point is to merge more states, since the pruning, which is applied after the merging, can yield higher errors. The merging can end after reaching several iterations, or when no states were merged in the last step.

Concerning computing of the packet frequency, it can be either done by dividing a dataset into equally large parts according to the number of merging steps, or by using the same dataset again.

After having performed some experiments with iterative merging, we found out that it indeed improved the accuracy of the reduced NFAs in some cases. However, the improvement was not so significant. For that reason, we decided to not focus on this approach.

4.5 Other Approaches

In this section, we describe several approaches, which are different from the state pruning and merging discussed above. First, we describe in a nutshell a recently proposed method for approximate reduction driven by a probabilistic distance. Subsequently, we discuss a solution of state merging inspired by abstract regular model checking.

4.5.1 Probability Driven Approach

In general, our proposed state pruning and merging reduction do not provide any formal guarantee wrt to traffic, so their capabilities, when used on a small dataset, are questionable. Nevertheless, there is a method which supplies this guarantee, recently proposed in [11]. The approach is based on having a probabilistic automaton (PA) described by the traffic and provides guarantee wrt this PA.

However, the problem is in obtaining a PA, which represents the traffic precisely. In theory, one can build such PA manually, semi-automatically as used in [8] (give an FA and learn the probabilities), or use a fully automatic approach using learning algorithms such as Alergia.

The majority of algorithms for learning a PA, firstly build a prefix tree acceptor, which is a frequency automaton with a structure as an N-ary tree without loops. Then the prefix tree acceptor is transformed to the probabilistic automaton usually by merging states defined by learning algorithms.

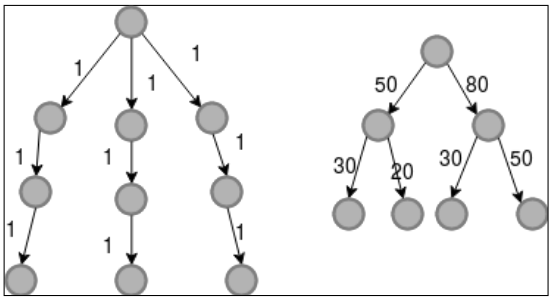

Figure 4.7: The oriented graphs represent two prefix tree acceptors (frequency automata). The circles are states and transitions are labeled with frequencies. The degraded tree is on the right, while the model on the right suits better for learning algorithms.

When learning such models on packets, we encounter one big problem right at the beginning. Just after a few thousands of packets, we are usually out of the computer memory. The prefix acceptor is being degraded during learning, because of a high diversity of network traffic. In other words, the frequency automaton has the majority of branches with the frequency equal to one. Besides, some packets are too long (around 1 500 bytes), which only aggravates the problem (more branches with low frequencies). Even if we use computers with better parameters, the problem still remains, the tree degradation is notable in a high degree. Learning algorithms are not suited for such samples. They expect samples which were generated from some distribution. The difference between a prefix tree acceptor

which works well with learning algorithms and the degraded prefix tree acceptor is depicted in Figure 4.7.

To sum up, unless some other learning approach is used or one invests into manual modeling of the traffic, which is itself very hard and challenging task, this reduction method may yield inaccurate predictions about errors derived from the trained probabilistic automaton.

4.5.2 Merging Based on the Set of Prefixes

We made several experiments of automata reduction inspired by abstract regular model checking described in the article [6]. The reduction proceeds as follows.

We take a sample S of packets, which do not belong to the language recognized by the input NFA we want to reduce. Let P be the set of prefixes of the strings in S. We mark each state state q by the maximal subset $P_q \subseteq P$ such that q can be reached by each packet $w \in S$. Then we merge the states with the same sets of prefixes, including empty sets. Since there may be only few states with equal sets of prefixes, we can also merge states which sets of prefixes are similar wrt some threshold. For instance, for the threshold equal to 0.9, we merge states whose sets of prefixes have 90 % elements the same. When comparing the two sets P_1 and P_2, we compute their intersection I. Then we calculate the similarity rate

$$s = \frac{|I|}{max(|P_1|, |P_2|)}$$

and compare it to the threshold. If the similarity rate s is higher or equal to the threshold, we can merge the states.

Chapter 5

Implementation

The design and implementation of the prototype tool based on the proposed methods are described in this chapter. The whole software tool is called `ahofa`, which stands for Approximate Handling Of Finite Automata. This tool consists of automata reduction, packet frequency computation, and the reduced NFA error evaluation.

The NFA reduction is written in Python, while the NFA error evaluation and packet frequency computation are written in C++. Both languages are object-oriented, which allowed us to use class encapsulation and other useful features.

5.1 The Design

In this section, we focus on the architecture of the reduction tool `ahofa`. The diagram in the Figure 5.1 shows the connection between several programs and data resources. Firstly, we convert REs stored in PCRE format to non-deterministic automata. Then we simplify the input automaton by removing states which burden further computation with the automaton. Following this, we apply reduction on the simplified automaton using some packet dataset for computing the packet frequency for each state. Once we have obtained the reduced automaton, we can validate it by computing the error on some datasets. In the end, we can use language-preserving reductions (e.g., `Reduce` [3]) to intensify the reduction of the original automata.

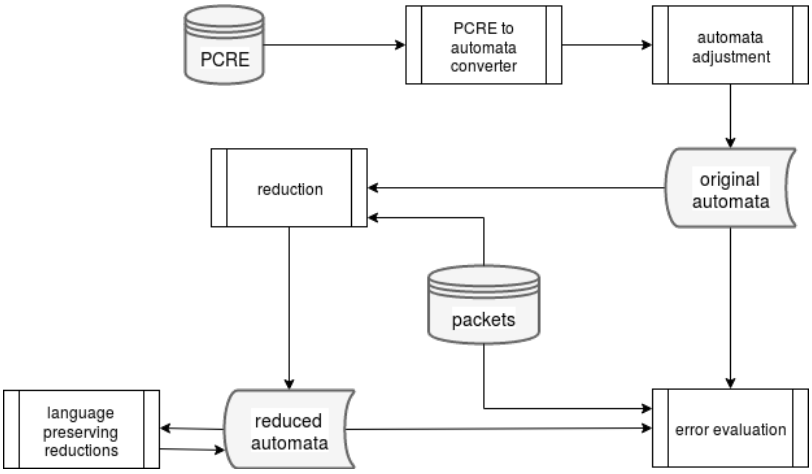

Figure 5.1: The architecture of the reduction tool `ahofa`.

5.2 Implementation Details

This section contains information about technical details of this work. The code related to automata reduction and error evaluation can be found in `ahofa`[1] repository. The structure of this section is following. First, we describe the way how NFAs and samples are obtained. Secondly, we explain what is done before the reduction, which we call automata simplification. Next, we focus on the automata reduction algorithms and error evaluation of reduced automata.

5.2.1 A PCRE Converter and PCAP Manipulation

The RE converter and tools for automata language-preserving reduction that we use are external programs, which we did not implement. The transformation of an RE to a finite automaton is provided by a NETBENCH tool[2] developed by the research group ANT@FIT. This tool is capable of converting the majority of REs to NFAs, except some complex REs. The output of the original tool is an NFA in the Timbuk format [4], but we reimplemented it to yield our own format.

This NFA format is line-based and has the extension `.fa`. We tried to make it as intuitive and simple as possible with the following syntax:

```
<state>  // initial state
<symbol> <state> <state>  // transitions
...
<state>  // final states
...
```

The symbols `<state>` can be any non-negative integral value, while `<symbol>` is a symbol of the byte alphabet (2^8 values) and ranges from `0x00` to `0xff` in hexadecimal. The prefix `0x` must be included to distinguish it from state label. We neglect epsilon transitions since

[1] https://github.com/jsemric/ahofa
[2] https://github.com/vhavlena/appreal

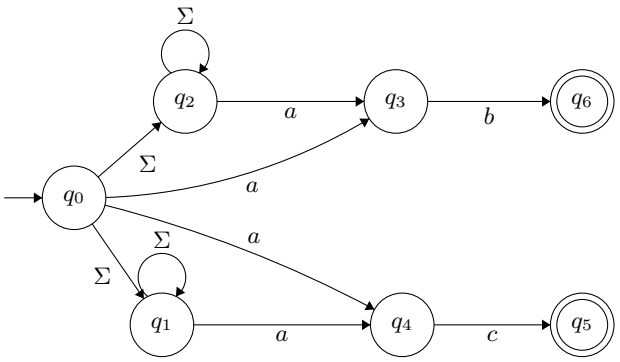

Figure 5.2: The stats q_1 and q_2 are redundant and can be merged together without changing the language defined by the NFA. The label Σ denotes the transition over the all symbols of the alphabet.

we do not work with them. A transition `<symbol> <state> <state>`, for instance, a line `0x32 0 1`, means that from the state `0` there is a transition to the state `1` under a byte of the hexadecimal value `0x34`.

Packet capture files (PCAP files) are binary files, which store information about some packets, including Ethernet headers, in the compressed format. For reading PCAP files we use libpcap[3] library. This library provides an API for manipulation with PCAP files in the C or C++ programming languages. Nevertheless, payload extraction is missing, so we had to implement it ourselves. Another solution could be the Python Scapy[4] module which can read the packet payload directly. However, since it is a way slower than libpcap, we decided not to employ it.

5.2.2 NFA Simplification

Just before we run our approximation reduction on the automata, we remove some states which slow down any computation with NFA. The PCRE converter yields an NFA that contains a lot of redundant states. Moreover, it adds a state with self-loop over the whole alphabet at the beginning of each subautomaton as illustrated in Figure 5.2. The problem is not so obvious, but if we have, for instance, 100 rules, we have 100 additional states with self-loop over the whole alphabet. When using NFA operations in software, we have to expand transitions of these states after reading a single character, which is a large burden when working with automata containing a lot of rules.

However, by merging these states into one, we can speed up NFA operations which work with the given NFA and a string, e.g, computing the packet frequency. We measured the time duration of computing packet frequency on 20 000 packets for a single automaton with this adjustment and without it. The automaton we used had 154 rules and around 4 000 states. The duration of computing the packet frequency of the simplified NFA was 3.9 seconds, whereas the evaluation of the second one lasted approximately 6 minutes and 53 seconds on our system. We can see a remarkable improvement (almost 106 times faster) in the performance in NFA operations with the described removal of redundant states at the beginning of each subautomaton.

[3] http://www.tcpdump.org/
[4] http://www.secdev.org/projects/scapy/

5.2.3 Approximate NFA Reduction

The Python script `app-reduction.py` serves for the NFA reduction. It is also possible to evaluate the error of the NFA immediately after the reduction by running the program with particular parameters.

Concerning reduction methods (pruning and merging), they are implemented as single functions, since the class-oriented implementation was not necessary. Both functions use the packet frequency of the input NFA obtained by calling C++ program `state_frequency`. Then they mark states for merging. The merging is done by calling method `merge_states` of the class `Nfa`, which is the class serving for manipulation with NFAs.

5.2.4 Error Evaluation

The validation of the reduced automata is done by computing their error wrt input traffic. We usually run this operation on a great number of samples to get more information about how the reduction changed the automaton. This evaluation is provided by the executable file `nfa_eval`. It was designed to speed up the process of parsing packets by the automaton and comparing packet classifications. It takes three or more positional arguments, in the following order: the original automaton, the reduced automaton, and at least one PCAP file. Besides, positional arguments there are some optional parameters such as output file name or the number of threads to run in parallel.

For dealing with packets, we use the implementation of an NFA which stores transitions in a single one dimensional array where each item is a set of states. The access to an item then consists of the label of states plus the symbol value. Despite the fact that this solution is very greedy on memory, it is much faster than using hash tables for manipulation with transitions.

Chapter 6

Experiments

In this chapter we present the experimental evaluation of the proposed reduction methods. Besides the reduction results, we also show outcomes of applying state-of-the-art language-preserving reduction techniques on reduced automata.

6.1 Reduction Results

In this section, we present the reduced NFAs statistics on real samples. The vast majority of packets were obtained by research group ANT@FIT, while the rest was acquired from DARPA training traffic dumps.

Concerning the obtained reduced automata, we were primarily interested in the following statistics:

- *Acceptance Error* (AE) stands for the number of incorrectly accepted packets by the reduced NFA which are not accepted by the original NFA over the the total number of packets.
- *Classification Error* (CE). Let $\Phi(M, w)$ be a subset of final states of the automaton M visited during processing the string w. Then the classification error is the number of packets w such that $\Phi(M, w) \neq \Phi(M', w)$ where M' is the reduction of the automaton M, over the total number of packets. Since the NFA M' is an over-approximation of M, and the number of final states is preserved after the reduction, for all packets w holds that $\Phi(M, w) \subseteq \Phi(M', w)$.
- *Acceptance Precision* (AP) is the ratio of correctly accepted packets to the all accepted packets by the reduced NFA.
- *Classification Precision* (CP) is the ratio of correct packet classifications to the total number of packets classified (at least one final state has been visited).

Further, it also holds that $AE \leq CE$, since all accepted packets are classified, but there are many ways how a packet can be classified (visiting more final states than in the original NFA). The CE 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 processing, then we might also be interested in CP. The variables AE and AP are used for showing the relation between accepting and classifying a packet.

6.1.1 Pruning Results

In this section, we present the results of the pruning reduction of the automaton `sprobe`. After PCRE parsing and subsequent NFA simplification, the NFA has 168 states and 5 108 transitions.

Table 6.1: Error and precision of the pruning reduction of `sprobe`.

ratio	AE	CE	AP	CP
0.14	0.277599	0.277609	0.000175	0.000140
0.16	0.170191	0.170197	0.000285	0.000251
0.18	0.084576	0.084580	0.000574	0.000532
0.20	0.015422	0.015424	0.003141	0.003010
0.22	0.005490	0.005492	0.008773	0.008446
0.24	0.002262	0.002264	0.021030	0.020356
0.26	0.000473	0.000475	0.093147	0.090241
0.28	0.000118	0.000119	0.292226	0.283616
0.30	0.000118	0.000119	0.292300	0.283688

Table 6.1 shows the statistics about the automaton reduced by the pruning reduction with different reduction ratios. The results were acquired on the dataset consisting of 23 millions of samples, while the dataset used for computing the packet frequency consisted of around 1 million of packets. If we look at details, we can see that there is only a small difference between AE and CE. However, the difference between AP and CP is slightly higher, but still not very high. Since the CE and AE are similar we will be interested only in CE and CP in the rest of this chapter.

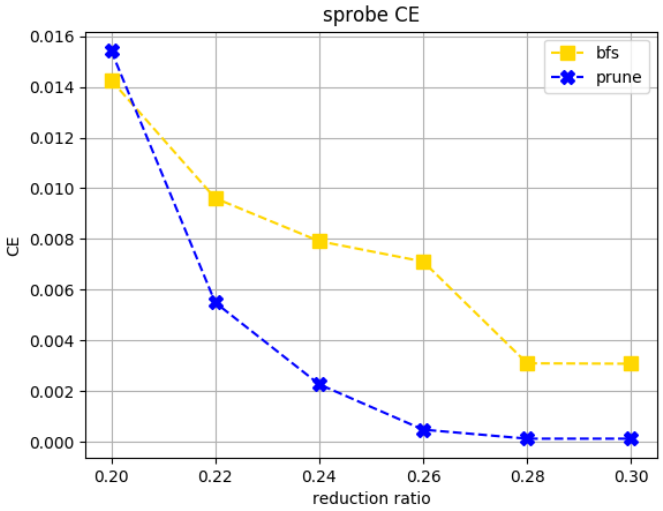

Figure 6.1: CE of pruning and BFS method of the NFA `sprobe`.

The pruning reduction with the reduction ratio 0.2 (20 % of the original NFA size in terms of the number of states) yields the reduced NFA, which has less than 2 % error (CE).

By using the `Reduce` tool for precise reduction, the resulting NFA has 128 states, where the calculated reduction ratio is approximately 0.76. Although the pruning reduction yields some inaccuracies, which is not a big problem when we subsequently use software processing, the reduction is almost three times more significant than the precise reduction provided by the `Reduce` tool. Moreover, we can apply the precise reduction after the pruning and achieve even greater reductions.

The figure 6.1 shows the CE of the NFA reduced by the pruning and the *BFS* reduction. BFS is a blind reduction technique, which uses breadth-first search for removing the states of the input NFA. In other words, it starts removing the states, which have the largest distance[1] from the initial state in a same way as the pruning. Although, the BFS reduction has lower CE with reduction ratio equal to 0.2, in all other cases is the pruning better (see Table A.2 for lower reduction ratios).

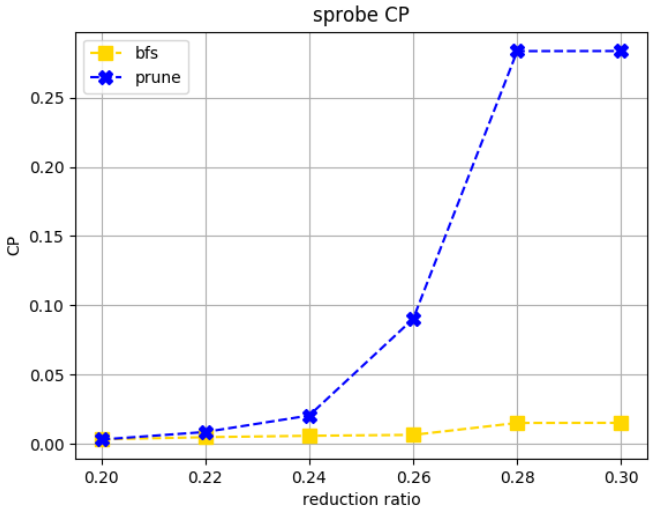

Figure 6.2: CP of pruning and BFS of the NFA `sprobe`.

Looking at the CP values of the pruning and the BFS method illustrated in Figure 6.2, we can see that the pruning has noticeable higher precision than BFS. With the increasing reduction ratio, the precision in BFS rises negligibly, while the precision pruning grows considerably.

Sometimes we want to know how the error of the reduced NFA is distributed. Figure 6.3 illustrates the dispersion of the CE. The small points represent CEs of the reduced NFA computed on particular PCAP files. With the decreasing reduction ratio on the y-axis, we can see that not only the average CE is increasing but also the dispersion of CE steadily grows.

6.1.2 Merging Results

In this section, we show the results of the merging reduction (see Table A.2 for exact numeric values). Here we also perform the reduction on the automaton `sprobe` and compare

[1] we consider the distance as the minimal number of symbols read to get from a particular state to a given state

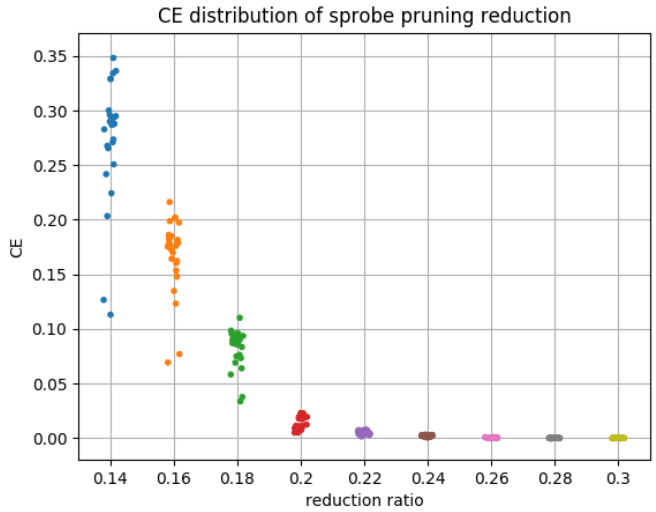

Figure 6.3: The dispersion of the CE of the reduction of the NFA `sprobe`.

obtained results with the pruning reduction. The samples used for the packet frequency and the error evaluation are the same as in the previous section.

Figure 6.4 shows the difference between pruning and merging in CE and CP. The merging outperforms the pruning by having two times smaller CE on average. Concerning CP, the merging has also better results, e.g., having CP at around 0.27 at the reduction ratio 0.26, while the pruning has CP approximately 0.09, which is three times less.

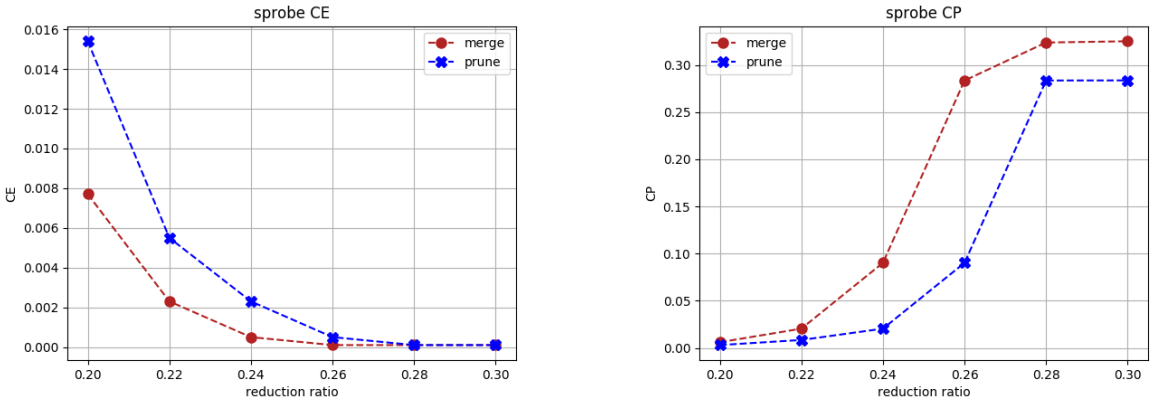

Figure 6.4: CE and CP of the merging and pruning methods used on the NFA `sprobe`.

For the mentioned results, we used reduction parameters the threshold and the maximal frequency equal to 0.995 and 0.1 in turn. However, we also made some experiments with different merging parameters. For that the automaton `backdoor.rules` was used, which has around 4 000 states. Figure 6.5 illustrates the values of the CP obtained by reducing the NFA with the combination of the merging parameters. In the figure, the threshold parameter is on the x-axis, while the y-axis consists of values of the maximal frequency parameter. The reduction ratio 0.18 was the same for all reductions.

We can see that changing these parameters indeed have some impact on CE. Moreover, the threshold seems to have a greater influence than the maximal frequency parameter, since the most of the values at the particular rows are the same. The yellow area shows where the error was the highest. Primarily the combination of the high maximal frequency parameter and the low threshold increases the error.

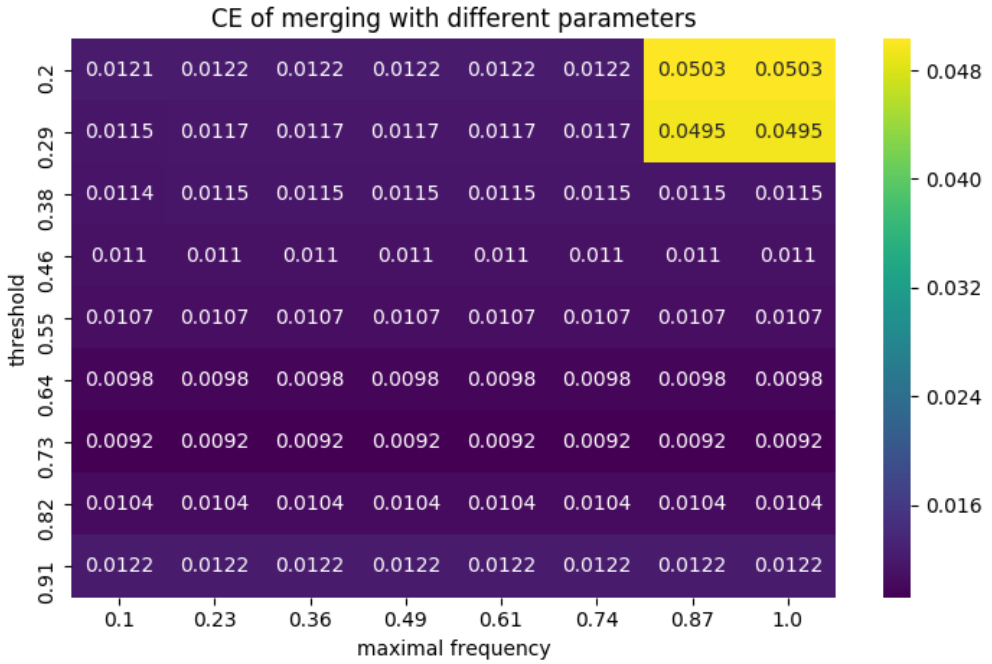

Figure 6.5: CE of merging with different the threshold and the maximal frequency parameters.

6.1.3 Merging Based on Similar Sets of Prefixes

The next experiments involve evaluation of the reduction method inspired by abstract regular model checking. For computing the packet frequency and sets of prefixes we used a dataset of 10 000 packets, while for the error evaluation we used 40 000 packets. The automaton `sprobe` was used as the target of the reduction.

In Table 6.3 we can find the results of several methods. The reduction that merges states, which have a similar set of prefixes including empty sets is denoted as *armc*. However, this method is not scalable (we cannot set the reduction ratio) and yields very high error (0.75). For that reason, we changed this method by merging only states with non-empty sets. The other states are then removed by the pruning reduction. This reduction is denoted as *armc+prune* and is also scalable with the reduction ratio parameter. By using a different threshold parameter we obtained lower CE compared to the pruning with the same reduction ratio. However, after we used the precise NFA reduction (`Reduce`), we acquired larger NFAs than in the pruning reduction (column `Reduce` states). That was presumably caused by the merging of the equivalent states wrt the language defined by the target NFA. Then we used the pruning reduction with the 0.22 ratio and after precise

reduction, we obtained both the smaller error and the smaller NFA than *armc+prune* method.

Table 6.2: Comparison between pruning and merging based on set of prefixes.

method	ratio	threshold	CE	states	`Reduce` states
armc	-	0.7	0.7421	30	29
armc + prune	0.2	0.7	0.0088	34	32
armc + prune	0.2	0.5	0.0088	34	32
armc + prune	0.2	0.1	0.0113	34	29
prune	0.2	-	0.0149	34	26
prune	0.22	-	0.0071	37	29

One major drawback of the *armc* method is that it works with sets of prefixes, which is computationally harder than calculating packet frequencies. For huge NFAs and large datasets, there may also be a problem with not enough memory, because we have to remember each prefix label. Although the method did not bring better results than pruning for the examined NFAs, it may be worthwhile to do some further investigation and evaluation on more NFAs.

6.1.4 Large Automata Reductions

We picked four large automata (three from Snort, one from L7-filter) on which we performed reduction and error evaluation. Table 6.3 shows their size consisting of the number of states and the number of transitions. The sizes were computed after all NFAs have been simplified (merging of some redundant states). We reduced the automata with both merging and pruning reductions. Concerning the merging reduction, the threshold and the maximal frequency parameters were set to 0.995 and 0.1 respectively. For computing packet frequency we used one PCAP file consisting of around 1 million packets. The testing sample on which the error was evaluated consisted from around 23 million packets with a payload.

Table 6.3: The size of NFAs.

NFA name	**states**	**transitions**
backdoor.rules.fa	3,898	100,024
imap.rules.fa	5,637	1,348,955
spyware-put.rules.fa	12,809	279,275
l7-all.fa	7,280	2,647,220

The first NFA, whose results we present, is called `backdoor.rules`. Besides pruning and merging reductions, we also added the results of the BFS method. The CE and CP for various reduction ratios are illustrated in Figure 6.6. If we look at CE, we can see that merging caused a smaller error than other two methods for reduction ratio ranging from 0.14 to 0.22. But from 0.22 the pruning reduction has a slightly better results than merging. The CE of BFS is lower than pruning for the ratios from 0.2 down to 0.16. However, at 0.14 the error rockets above 0.5 (see Appendix A for the precise CE values). Another interesting point is the CP in merging, where it grows slowly and then plummets from approximately 0.23 to almost 0.99.

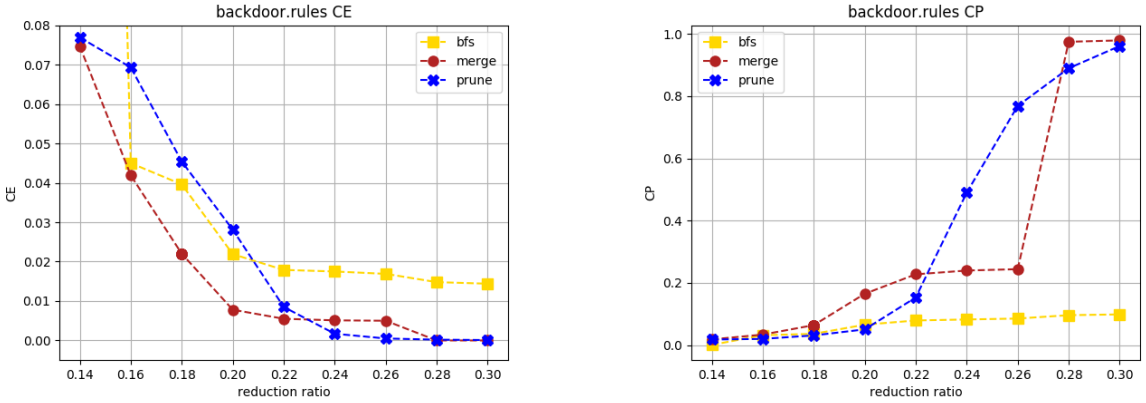

Figure 6.6: CE and CP comparison between merging, pruning, and the BFS method of the NFA `backdoor.rules`. For the ratio 0.14 the exact value of CE of the BFS reduction can be found in Table A.3

Let us move to larger NFAs. Figure 6.7 shows CE of two automata, `imap.rules` and `spyware-put.rules`. The reduction of `imap.rules` is quite impressive. We managed to reduce to 2 % of the original size and yield less than 5 % CE using the merging reduction. The merging reduction is also better than pruning for `spyware-put.rules`, where, e.g., for the reduction ratio 0.24 the CE is more than five times smaller. The numeric results of CP for the both NFAs can be found in Table A.1 and Table A.4 in Appendix A.

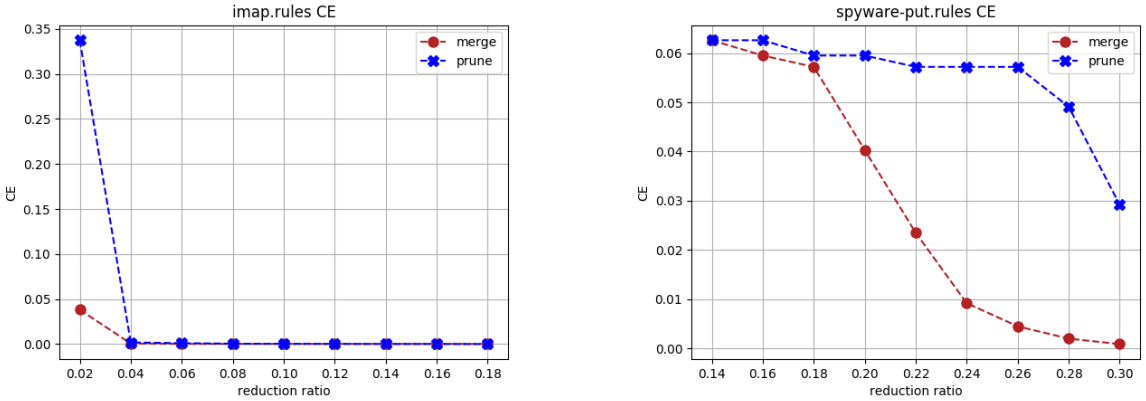

Figure 6.7: CE of the reduction of NFAs `imap.rules` (ont the left) and `spyware-put.rules` (on the right).

The automaton `l7-all.fa` is the NFA containing all L7 filtering rules. The reduction results are illustrated in Figure 6.8. In the previous experiments, the merging reduction had better results on average than pruning. However, in this case, the merging is not appropriate. As we can see from the graph, the merging causes some big error around 0.28, which does not considerably improve with increasing the reduction ratio. Although the pruning has worse results when reducing under 35 %, the error constantly declines, reaching around 0.03 at the reduction above 45 %. Also, it is noticeable that the reduction of this NFA is less significant than the reduction of the previous ones, where the trade-off between the number of states and CE is better.

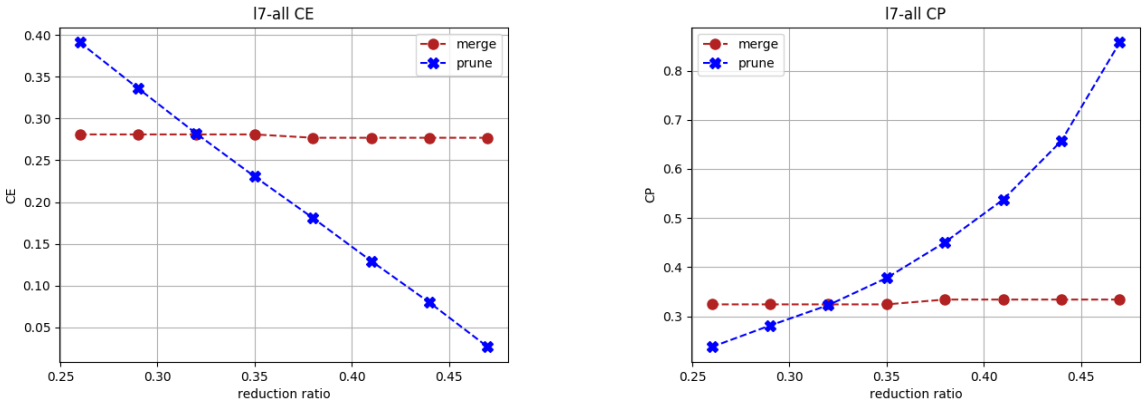

Figure 6.8: Comparison of CE and CP statistics between the pruning and merging the automaton `l7-all`. The numeric values can be found in Table A.5.

6.2 Precise Reductions

In this section, we describe experiments with language-preserving reductions, namely DFA minimization and the NFA simulation reduction. Since the output of the PCRE parser is an NFA with many redundant states, it is possible to diminish the size of the automata considerably and still preserve the language. In the following experiments, we first reduced the NFA named `backdoor.rules` with approximate reductions (pruning) and then applied the precise reductions. The reason why we did not reduce the original NFA directly is that we could not manage to perform such reduction on the most of NFAs, since we were limited to our system resources.

For a DFA determinization and minimization we used the `Symboliclib`[2] library, whereas for the NFA reduction we used the `Reduce` tool. The chart in Figure 6.9 compares the data about the number of states of the two mentioned precise reductions applied on the NFAs reduced by the pruning with different reduction ratios.

Looking at the details, we can see that for the ratios 0.1 and 0.12, DFA determinization & minimization caused a small growth in the state count. However, for higher reduction ratios the number of states rapidly rose. Concerning the NFA reduction, the resulting NFAs were on average twice time smaller than the NFAs obtained directly after approximate reduction.

We can also measure the number of transitions in all three cases and compare their count. Table 6.4 shows the data, where the number of transitions in the DFAs were exponentially higher than in the NFAs. For instance, for the reduction ratio 0.18 the NFA precise reduction yielded automata only with around 3 500 transitions, while the automata obtained after DFA minimization had more than one million transitions.

[2] https://github.com/Miskaaa/symboliclib

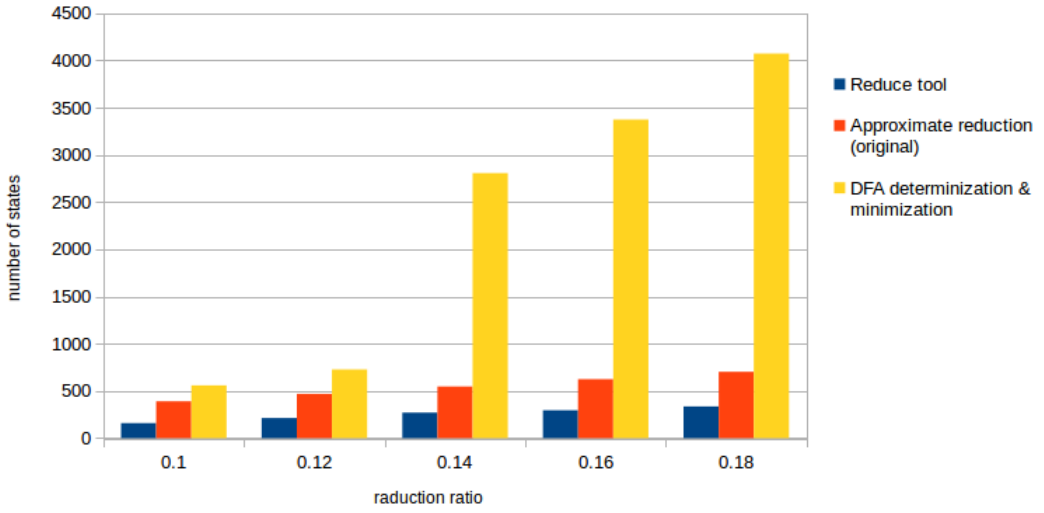

Figure 6.9: An impact on the count of states of NFA `backdoor.rules`, after applying the language-preserving reductions on the reduced automata by the pruning.

ratio	NFA simulation	reduced original	DFA minimization
0.1	1,364	2,333	147,841
0.12	1,471	2,486	197,259
0.14	2,336	5,661	805,684
0.16	2,644	10,379	992,647
0.18	3,482	15,096	1,226,540

Table 6.4: The number of transitions of `backdoor.rules` for approximate reduction and its DFA and NFA precise reductions.

To sum up, the applying of the `Reduce` tool after the approximate reduction gains on reduction strength. Concerning DFAs, their size grows significantly and may not be manageable even after transforming into delayed input DFAs.

Chapter 7

Conclusion

In the work, we proposed several methods for approximate automata reduction based on packet frequency. Namely, we proposed the pruning method and its refinement with state merging. We saw that even the simple pruning method can produce impressive results and can be improved by merging, which in many cases decreased error.

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 4 % of original size, with the error below 1 %. Due to the low time complexity of our approach, it is also possible to reduce huge automata with thousands of states, which could not be achieved using previous methods. Moreover, the methods do not lack intelligibility. In other words, we know precisely what they do, on what basis they remove or merge states, and when they may fail. This comprehensibility is essential for people with non-technical background involved in projects such as managers and stakeholders who have the main say where money is invested.

The next step of this work will be to use reduced automata in real-time network traffic. A quite cheap solution could be computing the error just in software on sampled traffic. This approach will show us whether the packet frequency, on whose basis the reductions were made, calculated on our datasets, was sufficient. Provided that it will be successful, the automata can be synthesized to FPGA and tested more rigorously.

Bibliography

[1] Course Formal Languages and Compilers. [online].
Retrieved from:
https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj03-en.pdf

[2] DARPA Intrusion Detection Evaluation.
Retrieved from: https://www.ll.mit.edu/ideval/data/1999data.html

[3] Rabit and Reduce tool. [online].
Retrieved from: http://www.languageinclusion.org/doku.php?id=tools

[4] Timbuk for Reachability Analysis and Tree Automata Calculations. [online].
Retrieved from: http://people.irisa.fr/Thomas.Genet/timbuk/

[5] Application Layer Packet Classifier for Linux. 2009. [online].
Retrieved from: http://l7-filter.sourceforge.net/

[6] Bouajjani, A.; Habermehl, P.; Rogalewicz, A.; et al.: Abstract Regular Tree Model Checking. *Electronic Notes in Theoretical Computer Science*. vol. 149, no. 1. 2006: pp. 37 – 48. ISSN 1571-0661. doi:https://doi.org/10.1016/j.entcs.2005.11.015. proceedings of the 7th International Workshop on Verification of Infinite-State Systems (INFINITY 2005).
Retrieved from:
http://www.sciencedirect.com/science/article/pii/S1571066106000521

[7] Carr, J.: Snort: Open Source Network Intrusion Prevention. 2007. [online].
Retrieved from: https://www.esecurityplanet.com/network-security/Snort-Open-Source-Network-Intrusion-Prevention-3681296.htm

[8] Ceska, M.; Havlena, V.; Holík, L.; et al.: Approximate Reduction of Finite Automata for High-Speed Network Intrusion Detection. *CoRR*. vol. abs/1710.08647. 2017. 1710.08647.
Retrieved from: http://arxiv.org/abs/1710.08647

[9] Clemente, L.; Mayr, R.: Advanced Automata Minimization. *CoRR*. vol. abs/1210.6624. 2012. 1210.6624.
Retrieved from: http://arxiv.org/abs/1210.6624

[10] Goyvaerts, J.: Regular-Expressions.info. [online].
Retrieved from: https://www.regular-expressions.info/

[11] Havlena, V.: *Comparing Languages and Reducing Automata Used in Network Traffic Filtering*. Master's Thesis. Brno University of Technology Faculty of Information Technology. 2017.

[12] de la Higuera, C.: *Grammatical Inference: Learning Automata and Grammars*. New York, NY, USA: Cambridge University Press. 2010. ISBN 0521763169, 9780521763165.

[13] Holík, L.: Simulations and Antichains for Efficient Handling of Finite Automata. *CoRR*. vol. abs/1706.03208. 2017. 1706.03208.
Retrieved from: http://arxiv.org/abs/1706.03208

[14] Hopcroft, J. E.; Ullman, J. D.: *Introduction To Automata Theory, Languages, And Computation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.. first edition. 1990. ISBN 020102988X.

[15] Ilie, L.; Navarro, G.; Yu, S.: *On NFA Reductions*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2004. ISBN 978-3-540-27812-2. pp. 112–124.
doi:10.1007/978-3-540-27812-2_11.
Retrieved from: https://doi.org/10.1007/978-3-540-27812-2_11

[16] Kozen, D. C.: *Automata and Computability*. Secaucus, NJ, USA: Springer Verlag, New York, Inc. 1997. ISBN 0-387-94907-0.

[17] Kumar, S.; Dharmapurikar, S.; Yu, F.; et al.: Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. *SIGCOMM Comput. Commun. Rev.*. vol. 36, no. 4. August 2006: pp. 339–350. ISSN 0146-4833.
doi:10.1145/1151659.1159952.
Retrieved from: http://doi.acm.org/10.1145/1151659.1159952

[18] Matoušek, P.: *Síťové služby a jejich architektura*. Publishing house of Brno University of Technology VUTIUM. 2014. ISBN 978-80-214-3766-1. 396 pp.
Retrieved from:
http://www.fit.vutbr.cz/research/view_pub.php.cs.iso-8859-2?id=10567

[19] Matoušek, D.; Kořenek, J.; Puš, V.: High-speed regular expression matching with pipelined automata. *2016 International Conference on Field-Programmable Technology (FPT)*. 2016: pp. 93–100.

[20] Rouse, M.: deep packet inspection (DPI). [online].
Retrieved from: http://searchnetworking.techtarget.com/definition/deep-packet-inspection-DPI

[21] Rouse, M.: intrusion detection system (IDS). [online].
Retrieved from: http://searchsecurity.techtarget.com/definition/intrusion-detection-system

[22] Wyllys, R.; Dotty, P.: The 5-Layer Model (the TCP Model). 2000. [online].
Retrieved from: https://www.ischool.utexas.edu/~l38613dw/readings/NotesOnInterconnection.html

[23] Češka; Vojnar: Course Theoretical Computer Science. [online].
Retrieved from: http://www.fit.vutbr.cz/study/courses/TIN/public/Prednasky/tin-pr01-rj1.pdf

Appendix A

Tables with Results

Table A.1: imap.rules

	CE		CP	
method ratio	merge	prune	merge	prune
0.02	0.0382	0.3373	0.0028	0.0002
0.04	0.0007	0.0019	0.1663	0.0596
0.06	0.0003	0.0011	0.3698	0.1156
0.08	0.0003	0.0005	0.3726	0.2646
0.10	0.0003	0.0004	0.3787	0.3310
0.12	0.0003	0.0003	0.3875	0.4075
0.14	0.0003	0.0002	0.4045	0.4543
0.16	0.0003	0.0002	0.4181	0.5399
0.18	0.0002	0.0001	0.5064	0.6130

Table A.2: sprobe

	CE			CP		
method ratio	bfs	merge	prune	bfs	merge	prune
0.14	0.2956	0.2449	0.2776	0.0001	0.0002	0.0001
0.16	0.2019	0.1296	0.1702	0.0002	0.0003	0.0003
0.18	0.1189	0.0166	0.0846	0.0004	0.0027	0.0005
0.20	0.0143	0.0077	0.0154	0.0031	0.0060	0.0030
0.22	0.0096	0.0023	0.0055	0.0047	0.0204	0.0084
0.24	0.0079	0.0005	0.0023	0.0057	0.0902	0.0204
0.26	0.0071	0.0001	0.0005	0.0064	0.2836	0.0902
0.28	0.0031	0.0001	0.0001	0.0150	0.3239	0.2836
0.30	0.0031	0.0001	0.0001	0.0151	0.3252	0.2837

Table A.3: backdoor.rules

	CE			CP		
method	bfs	merge	prune	bfs	merge	prune
ratio						
0.14	0.5349	0.0747	0.0769	0.0007	0.0185	0.0180
0.16	0.0450	0.0419	0.0693	0.0319	0.0339	0.0201
0.18	0.0396	0.0220	0.0454	0.0363	0.0643	0.0313
0.20	0.0219	0.0078	0.0281	0.0657	0.1651	0.0506
0.22	0.0179	0.0055	0.0085	0.0793	0.2276	0.1531
0.24	0.0175	0.0051	0.0017	0.0825	0.2398	0.4901
0.26	0.0169	0.0050	0.0005	0.0855	0.2442	0.7690
0.28	0.0148	0.0000	0.0002	0.0963	0.9750	0.8898
0.30	0.0144	0.0000	0.0001	0.0988	0.9793	0.9613

Table A.4: spyware-put.rules

	CE		CP	
method	merge	prune	merge	prune
ratio				
0.14	0.0626	0.0626	0.2786	0.2786
0.16	0.0595	0.0626	0.2952	0.2786
0.18	0.0572	0.0595	0.3042	0.2952
0.20	0.0403	0.0595	0.3891	0.2952
0.22	0.0235	0.0572	0.5321	0.3042
0.24	0.0092	0.0572	0.7513	0.3042
0.26	0.0045	0.0572	0.8631	0.3042
0.28	0.0020	0.0491	0.9368	0.3405
0.30	0.0009	0.0292	0.9719	0.4732

Table A.5: l7-all

	CE		CP	
method	merge	prune	merge	prune
ratio				
0.26	0.2809	0.3909	0.3243	0.2377
0.29	0.2809	0.3360	0.3243	0.2813
0.32	0.2809	0.2819	0.3243	0.3228
0.35	0.2809	0.2305	0.3243	0.3780
0.38	0.2769	0.1810	0.3341	0.4506
0.41	0.2769	0.1296	0.3341	0.5375
0.44	0.2769	0.0799	0.3341	0.6571
0.47	0.2769	0.0268	0.3341	0.8574

Appendix B

CD Content

The following contents can be found on the attached CD:

- **bt_xsemri00.pdf** – this thesis in PDF format
- **tex/** – source files of this thesis
- **reduction/** – source files of reduction tools