VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

OPTIMALIZACE ALGORITMŮ A DATOVÝCH STRUKTUR PRO VYHLEDÁVÁNÍ REGULÁRNÍCH VÝRAZŮ S VYUŽITÍM TECHNOLOGIE FPGA

OPTIMIZATION OF ALGORITHMS AND DATA STRUCTURES FOR REGULAR EXPRESSION MATCHING USING FPGA TECHNOLOGY

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. JAN KAŠTIL
AUTHOR

VEDOUCÍ PRÁCE Doc. Ing. ZDENĚK KOTÁSEK, CSc.
SUPERVISOR

BRNO 2015

Abstrakt

Disertační práce se zabývá rychlým vyhledáváním regulárních výrazů v síťovém provozu s použitím technologie FPGA. Vyhledávání regulárních výrazů v síťovém provozu je výpočetně náročnou operací využívanou převážně v oblasti síťové bezpečnosti a v oblasti monitorování provozu vysokorychlostních počítačových sítí. Současná řešení neumožňují dosáhnout požadovaných multigigabitových propustností při dodržení všech požadavků, které jsou na vyhledávací jednotky kladeny. Nejvyšších propustností dosahují implementace založené na využití inovativních hardwarových architektur implementovaných v FPGA případně v ASIC. Tato disertační práce popisuje nové architektury vyhledávací jednotky, které jsou vhodné pro implementaci jak v FPGA tak v ASIC. Základní myšlenkou navržených architektur je využití perfektní hashovací funkce pro implementaci přechodové tabulky konečného automatu. Dále byla navržena architektura, která umožňuje uživateli zanést malou pravděpodobnost chyby při vyhledávání a tím snížit paměťové nároky vyhledávací jednotky. Disertační práce analyzuje vliv pravděpodobnosti této chyby na celkovou spolehlivost systému a srovnává ji s řešením používaným v současnosti. V rámci disertační práce byla provedena měření vlastností regulárních výrazů používaných při analýze provozu moderních počítačových sítí. Z provedené analýzy vyplývá, že velká část regulárních výrazů je vhodná pro implementaci pomocí navržených architektur. Pro dosažení vysoké propustnosti vyhledávací jednotky práce navrhuje nový algoritmus transformace abecedy, který umožňuje, aby vyhledávací jednotka zpracovala více znaků v jednom kroku. Na rozdíl od současných metod, navržený algoritmus umožňuje konstrukci automatu zpracovávajícího libovolný počet symbolů v jednom taktu. Implementované architektury dosahují v porovnání se současnými metodami úspory paměti zlepšení až 200MB.

Abstract

This thesis deals with fast regular expression matching using FPGA. Regular expression matching in high speed computer networks is computationally intensive operation used mostly in the field of the computer network security and in the field of monitoring of the network traffic. Current solutions do not achieve throughput required by modern networks with respect to all requirements placed on the matching unit. Innovative hardware architectures implemented in FPGA or ASIC have the highest throughput. This thesis describes two new architectures suitable for the FPGA and ASIC implementation. The basic idea of these architectures is to use perfect hash function to implement transitional function of deterministic finite automaton. Also, architecture that allows the user to introduce small probability of errors into the matching process in order to reduce memory requirement of the matching unit was introduced. The thesis contains analysis of the effect of these errors to overall reliability of the system and compares it to the reliability of currently used approach. The measurement of properties of regular expressions used in analysis of the traffic in modern computer networks was performed in the thesis. The analysis implies that most of the used regular expressions are suitable for the implementation by proposed architectures. To guarantee high throughput of the matching unit new algorithms for alphabet transformation is proposed. The algorithm allows to transform the automaton to accept several input characters per one transition. The main advantage of the proposed algorithm over currently used solutions is that it does not have any limitation over the number of characters that are accepted at once. Implemented architectures were compared with the current state of the art algorithm and 200MB memory reduction was achieved.

Klíčová slova

Regulární výraz, vyhledávání, deterministický konečný automat, FPGA, perfektní hashovací funkce

Keywords

Regular expression, searching, deterministic finite automaton, FPGA, perfect hash function

Citace

Jan Kaštil: Optimization of algorithms and data structures for regular expression matching using FPGA technology, disertační práce, Brno, FIT VUT v Brně, 2015

Optimization of algorithms and data structures for regular expression matching using FPGA technology

Prohlášení

Prohlašuji, že jsem tuto doktorskou práci vypracoval samostatně pod vedením pana Doc. Ing. Zdeňka Kotáska CSc. a Ing. Jana Kořenka PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Jan Kaštil
October 15, 2015

Poděkování

Tato disertační práce vznikla na Ústavu počítačových systémů Fakulty informačních technologií Vysokého učení technického v Brně. Zejména bych chtěl poděkovat panu doc. Zdeňkovi Kotáskovi CSc. za jeho velkou podporu po celou dobu mého studia. Také bych rád poděkoval Ing. Janu Kořenkovi Ph.D. za odborné vedení práce. Současně bych také chtěl poděkovat kolegům z Ústavu počítačových systémů za řadu cenných rad a podnětů. Tato práce by ale nikdy nevznikla bez podpory mých rodinných příslušníků, rodičů, manželky Lucie i malého syna Jana, kteří se mnou měli velkou trpělivost a uskromnili se, abych mohl vytvořit tuto práci. Za to jim patří velké poděkování.

Contents

Chapter 1

Introduction

The rapid development of the Internet [5] and the increase of the speed of the network connectivity in general brings many enhancements to daily life. However, together with the usage of modern computer networks amount of malicious traffic also increases. Moreover, as more people and industries rely on the correct function of computer networks, the cost of possible damage by a malicious user increases. Therefore, the protection of the modern computer systems is a very important issue today.

The intrusion detection systems (IDS) [109] are one of many layers of protection developed to ensure correct function of the computer networks. Their main purpose is to analyse network traffic and detect a malicious content in the payload. If the inspected traffic contains malicious data, the alert is generated and the data transfer can be stopped or any other action to protect the network can be applied. Therefore, IDS have to analyze traffic in real time.

There exists many methods that can be used to analyse network traffic. One of the most simple methods is to look at the packet headers and trust obtained information. The TCP/IP packet header [93] contains a receiver's and sender's IP address together with port numbers and TCP setting called flags. The IANA [10, 84] organization assigns the specific port numbers to specific network protocols. For example, protocol number *80* is reserved for the HTTP communication. Therefore, if the IDS needs to know the application protocol used in the network communication, it can read the port number.

However, malicious traffic is often hidden. The malicious user may use a different application protocol at the standard port in order to avoid detection and to gain access into the network through firewall. Therefore, IDS may not rely on the information from the packet header, such as a port number, to detect any type of traffic and more advanced and more complex method needs to be used.

There exists many methods [78] that may be used by IDS to reliably analyse network traffic. The statistical analysis of network traffic is one of these methods. It measures distributions of many network parameters, such as the time between consequent packets, number of packets in one communication session, average number of packets per session, etc. Different methods are applied to analyse gathered data. The anomaly detection [108, 75, 70, 50] or machine learning [74, 44, 92, 112] are examples of these analytical methods. The statistical method do not need to analyse the actual content of the network, and therefore, they may work even with encrypted traffic. They are very effective against Denial of service attack, such as SynFlood [56], or port scanning [76].

However, the fact that statistical methods do not analyse the content of the payload is also their main disadvantage because they are not able to detect localised threads, when the attacker targets a specific vulnerability of a specific network device. For example, the packet designed to cause buffer overflow in the target machine will appear the same as any other network packet until its content is analysed.

Deep packet inspection methods are specialized on the analysis of the content of network communication. Therefore, they are able to detect attacks that may be invisible or very hard to detect by statistical based methods. The core operation of the deep packet inspection is pattern matching. For every vulnerability, a pattern is created that describes the content of the network communication that exploits a given vulnerability. The main disadvantage of the deep packet inspection is its dependence on the availability of the network content and high requirement of the processing power. The deep packet inspection has to process every symbol that is transported through the network. Therefore, the complexity of the used pattern matching algorithm is the real issue of the deep packet inspection methods.

There exists several methods how to hide malicious traffic from pattern based IDS. The most obvious one is to encrypt the whole communication, but encryption requires cooperation from the attacked system. It is up to system administrator to put IDS in the location, where the traffic is not encrypted. The second most often used method to hide the attack is to split data among several consequent packets [49]. Therefore, IDS have to be able to reconstruct the network stream correctly and perform analysis not on the every packet, but on the reconstructed data stream [104].

Vern Paxton [115] noted that the string pattern is not enough to describe a more complex attack. Therefore, regular expression matching together with additional analysis of the detected pattern is used in modern IDS.

In order to meet throughput required by modern computer networks, the regular expressions matching is often accelerated by the FPGA or ASIC coprocessors. For high speed networks, even the recent coprocessor are not „powerful“ enough to guarantee correct processing of all traffic with required reliability. Moreover, coprocessors may not be available too.

In such cases, it is often better to relax requirements on the IDS than to not have the IDS at all. For example, IDS often drop packets if they do not have enough computation power to properly analyse them. As a result, IDS will miss the attack that is in the dropped packets. Another example of the relaxed requirements is the IDS performing packet level pattern matching. These IDS analyse every packet independently from each other. Therefore, if the attack is divided into two packets it will be missed. But if the attack is placed in one packet only, IDS is able to identify it which is a better option than allowing every attack into the network.

The regular expression matching in the high speed network is mostly implemented by the finite automata. The nondeterministic finite automaton allows us to achieve very high speed by direct implementation into the FPGA [32, 33, 31, 121, 110] and they have generally less state and transitions than their deterministic counterparts. However they are not suitable for ASIC implementation and often have large state representation. The more detailed comparison between nondeterministic and deterministic approaches is presented later in this thesis in chapters 2 and 4.

This thesis focuses on deterministic automata and proposes two hardware architectures for fast regular expression matching. The experiments performed in this thesis shows that the increase of the size of automata during its determinization is not critical for a large part of the pattern set used in modern network tools. More information about these experiments are in chapter 7. Moreover, state representation of the deterministic automaton can be stored in a compact form in one register which allows easy implementation of context switching by a simple change of the value of the active state. Lastly, the deterministic automaton guarantees processing one symbol in a constant number of steps with one processor which allows us to store the transition table of automaton in memory and change the automaton by reloading the memory.

The first architecture proposed in this thesis performs exact regular expression matching. The architecture uses a perfect hash function to implement the transition table to guarantee one memory lookup per transition at most. The architecture is suitable for a regular expression whose automata have low saturation of transition table (see chapter 7). The measurement performed in chapter 9 shows that the architecture is able to achieve multigigabit throughput if combined with a suitable alphabet transformation.

The main limitation of the proposed architecture is the memory required to implement transition table. The second architecture presented in this thesis allows us to reduce the memory requirement of the transition table by introducing a small probability of a failure into the matching process. Every transition in the transition table is represented only by its hash value instead of a 2-tuple of an active state and input symbol. The trade off between memory requirements and the reliability of a matching unit can be done by changing the size of a stored hash value.

There are several main contributions in this work. The compatibility with many approaches on deterministic finite automata published in the literature [71, 16, 72, 114] is the biggest one. Moreover, the use of DFA as a basis for the pattern matching allows for the fast change of a regular expression that is searched for. The speed of the matching can be easily increased by accepting more symbols per one step of the automaton. Finally, the proposed architecture allows us to select a trade off between memory consumption and the reliability of the matching process. While the experiments focus on the FPGA implementation, the proposed architectures can be implemented in ASIC to support a higher clock frequency.

1.1 Goals of the work

The thesis has several important goals. This section summarises and formulates these goals as follows.

1. **Regular expression analysis** – Properties of regular expressions are well studied. However, regular expressions used in modern intrusion detection systems are only subset of all possible regular expressions and therefore their properties may differ. The goal is to identify properties of regular expressions that can be used to develop efficient implementation of regular expression matching component.

2. **Formulate requirements for the fast regular expression matching unit** – The requirements for the regular expression matching units depends on the use cases. The goal is to select the most important requirements for the regular expression matching units used in the network applications.

3. **Propose new hardware architecture for fast pattern matching** – The goal of the thesis is to use the result of the analysis of regular expressions to design new hardware architectures for the fast regular expression matching which will meet requirements of the modern network applications.

4. **Propose algorithms for preprocessing regular expressions** – The goal is to proposed a suitable algorithm for preparing the configuration data for the hardware acceleration unit.

1.2 Structure of the work

The first chapter describes the basic principles of the Intrusion Detection Systems (IDS) and serves as motivation for the presented research. The chapter is concluded by a description of the structure of the work. Chapter 2 is an introduction into the regular expressions theory. This chapter also contains a description of several structurse used in modern pattern matching languages that are not part of regular languages. The analysis, whenever the specific feature is regular, is done in this chapter. Chapter 3 continues the introduction and compares the pattern matching languages and their use in modern network applications. Chapter 4 contains descriptions of state of the art methods in fast regular expression matching and in their hardware acceleration. The theory of hash functions and perfect hashing is described in chapter 5. Chapter 6 describes the construction of the alphabet decoder which is responsible for transformation of the network stream into the alphabet of the matching unit. An analysis of the regular expression used in modern IDS is performed in chapter 7. Chapter 8 introduces new hardware architectures for pattern matching based on deterministic finite automata and a perfect hash function. The concept of the probabilistic pattern matching unit together with its hardware implementation is described and analysed in this chapter. Chapter 9 contains a measurement of the proposed architectures together with an evaluation of their effectiveness.

Chapter 2

Regular Expression

The purpose of this chapter is to introduce the basic formalism of the regular languages, such as regular expressions and finite automata in order to provide the formal background. The theoretical problems of described models are mentioned only briefly as the detailed description is outside the scope of this work. Interested readers may look into computer science lectures such as [86, 82, 111].

2.1 Definition of basic terminology

This section contains definition of basic terminology used further on in this thesis.

Definition 1. ***Function:*** *A function is a relation that uniquely associates members of one set with members of another set.*

Definition 2. ***Domain of definition:*** *The set of values of the independent variable(s) for which a function or relation is defined. Domain of definition of the function f is denoted $Dom(f)$.*

Definition 3. ***Codomain of function:*** *A set within which the values of a function lie.*

Definition 4. ***Alphabet:*** *Alphabet Σ is a nonempty set of symbols.*

Definition 5. ***String:*** *String u over Σ is a finite sequence of symbols from Σ and u_i denotes i-th symbol of the sequence. Set of all strings over Σ is denoted as Σ^**

Definition 6. ***Length of string:*** *Let's have string s. Then the $|s|$ is called length of string s equals the number of symbols in the string s.*

Definition 7. ***Equality of the string:*** *Let's have strings u and v over alphabet Σ, such that $u, v \in \Sigma^*$. String u is equal to v if and only if $|u| = |v| \wedge \forall_{0<i<|u|} : u_i = v_i$*

Definition 8. ***Empty string:*** *Empty sequence of symbols, i.e. the sequence with no symbol is called empty string. Empty sting is denoted ϵ.*

Definition 9. ***Concatenation of strings:*** *Let's have strings u and v over alphabet Σ. Concatenation of u and v is a new string $z = uv$, so that string v is connected at the end of string u.*

Definition 10. ***Language:*** *Language L is subset of Σ^*.*

Definition 11. ***Strings of language:*** *We say that string $x \in \Sigma^*$ belongs to language $L \subset \Sigma^*$ if and only if $x \in L$.*

Definition 12. ***Prefix:*** *Let s, u, v be strings over alphabet Σ such as $s = uv$. The string u is called prefix of s and is denoted by $u \sqsubseteq s$. If $v \neq \epsilon$, then u is called proper prefix of s. If u is proper prefix of s, it is denoted by $u \sqsubset s$.*

Definition 13. ***Suffix:*** *Let s, u, v be strings over alphabet Σ such as $s = uv$. The string v is called suffix of s and is denoted by $v \sqsupseteq s$. If $u \neq \epsilon$, then v is called proper suffix of s. If v is proper suffix of s, it is denoted by $v \sqsupset s$.*

Definition 14. ***Infix:*** *Let s, u, v, w be strings over alphabet Σ such as $s = uvw$. The string v is called infix of s. If $u \neq \epsilon$, then v is called proper infix of s.*

Definition 15. ***Prefix set:*** *Prefix set of string $w \in \Sigma^*$ is set of all strings u such that $u \sqsubseteq w$.*

Definition 16. ***Prefix free set:*** *Prefix free set X is set of such strings, so that if string x belongs to X, no proper prefix of x is member of X.*

Definition 17. ***Hamming distance:*** *Let's have strings $x, y \in \Sigma^*$ such that $|x| = |y|$. The hamming distance $D(x, y)$ is*

$$D(x, y) = |\{i : i > 0 \land i < |x| \land x[i] \neq y[i]\}|$$

The distance defined by definition 17 is called Hamming distance after the R.W. Hamming, who introduced it in [55] together with the proof, that Hamming distance is actually metric. Informally, Hamming distance is the number of positions i, so that the symbols at the position i is different in each sequence.

2.2 Definition of Regular Expression

The definition 18 from [86] defines a regular expression and regular languages in the way used in the information theory.

Definition 18. ***Regular Expressions:*** *Let Σ be an alphabet. The regular expressions over Σ and the languages that these expressions denote are defined recursively as follows:*

1. *$\emptyset$ is a regular expression denoting the empty set.*
2. *ϵ is a regular expression denoting $\{\epsilon\}$*
3. *a, where $a \in \Sigma$, is a regular expression denoting $\{a\}$*
4. *If r and s are regular expressions denoting the languages R and S, respectively, then*
   (a) $(r \bullet s)$ is a regular expression denoting RS
   (b) $(r + s)$ is a regular expression denoting $R \cup S$
   (c) (r^*) is a regular expression denoting R^*.

It is common practice to denote $(r + s)$ as a $(r|s)$ in the practical regular expressions. This work uses both notations. $L(r)$ means the language denoted by the regular expression r.

Definition 19. ***Regular languages:*** *Let L be language over an alphabet Σ. L is a regular language if $L = L(r)$ for some regular expression r over Σ.*

The regular expression used for descriptions of the network patterns contains many extensions. Some of these extensions increase descriptive power while others just simplify writing.

Definition 20. ***Character classes:*** *Let a, b, c, d be symbols from the alphabet Σ, then $[a - d]$ is an extension of the regular expression called character class denoting $a + b + c + d$.*

There exist several others descriptions of the character class in the practically used regular expressions. Character classes do not increase the description power of the regular expressions, but increases the succinctness of the expression.

2.3 Finite Automata

Regular expressions offer very compact and easy to read descriptions of regular language. However, the decision if the given string belongs into the regular language is not simple. Finite automata offers algorithmically simple and computationally effective method for solving the problem if the given string belong to the given regular language.

Definition 21. ***Nondeterministic Finite Automaton:*** *The nondeterministic finite automaton (NFA) is 5-tuple $M(Q, \Sigma, \delta, s, F)$, where*

- *Q is a finite set of states*
- *Σ is an input alphabet such as $\Sigma \cap Q = \emptyset$*
- *$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is a transition function*
- *$s \in Q$ is the start state*
- *$F \subseteq Q$ is a set of final states.*

The automaton processes the first symbol if the input string by changing its states according to its transition table. At the beginning, the automaton is in its start state s. The ϵ represents the empty string. Since definition 21 adds the empty string into the alphabet of the NFA, it is possible for the automaton to operate without accepting the symbols from the input string, or more precisely, to accept the empty substring of the input string.

2.3.1 Construction of the nondeterministic finite automaton from regular expression

There exists several methods used for the construction of the nondeterministic finite automaton from the given regular expression. The most commonly used method is called the Thompson method[120]. However, the original method constructed program for the implementation of the automaton, while we are interested in the abstract representation of the automaton. The Thompson method is based on the decomposition of the regular expression into its basic components according to definition 18. Every component of the regular expression is transformed into the simple nondeterministic finite automaton and automata are merged together in the same way as the regular expressions.

Figure 2.1 shows all structures required for the Thompson method. Figure 2.1a demonstrates the automaton for accepting one symbol of the alphabet while Figure 2.1b shows automaton accepting empty string. Figure 2.1c contains the automaton accepting the language that is a concatenation of languages of two other automata (Automaton1 and Automaton2). Both original automata are painted in black, while new states and transition added to perform concatenation are blue. Only the starting state and final state of original automata are drowned. It can be seen that the concatenation is performed by connecting the final state of the first automaton to the starting state of the second automaton by ϵ-transition. Figure 2.1d shows the automaton accepting the union of language of Automaton1 and Automaton2. The blue part of the Figure are new states and epsilon transition added by a construction algorithm the while black part are original automata. The construction algorithm added a new starting and new final state and connect starting states of two original automata with the new starting state by the ϵ-transitions. The final states of the original automata are connected to the newly created final state by ϵ-transitions. Figure 2.1f shows the construction of automaton accepting the language generated by the iteration of original automaton. The basic idea of this construction is to connect the final state to the starting state by the ϵ-transition. Figure 2.1e shows the automaton accepting the empty language. It is important to keep in mind that the size of the automaton created by this algorithm can be further reduced by removing ϵ-transitions or by other algorithms mentioned in the previous section.

The number of states and transitions in the automaton generated by the described algorithm is linear with the number of symbols in the input regular expression. In addition, the algorithm requires liner time to construct the automaton.

2.3.2 Properties of finite automaton

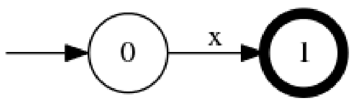
The previous section described the way to build NFA from an arbitrary regular expression. This section continues with a more detailed description of the NFA and its properties.

Definition 22. ***Automaton configuration:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be the nondeterministic finite automaton. The configuration of the automaton is the word χ*

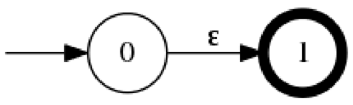
$$\chi = 2^Q\Sigma^*$$

Informally, the configuration of the automaton is a combination of its active states and string that remains to be processed in the future. Active states represent current status of the automaton. The next transitions have to originate from an active state and the active state is either the starting state of the automaton or the end point of some previous transition.

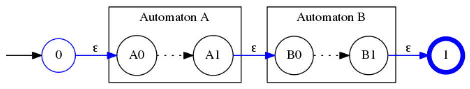
Definition 23. ***Active states:*** *Let's have finite automaton $M = (Q, \Sigma, \delta, s, F)$ and its configuration $\chi = Sx$, where $S \subset Q$ is the set of states and $x \subset \Sigma^*$. The states in the S are called active states.*



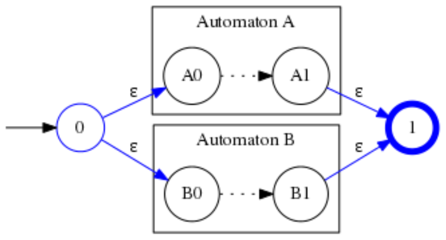
(a) Automaton accepting symbol x



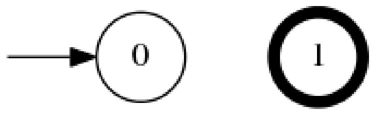
(b) Automaton accepting empty string



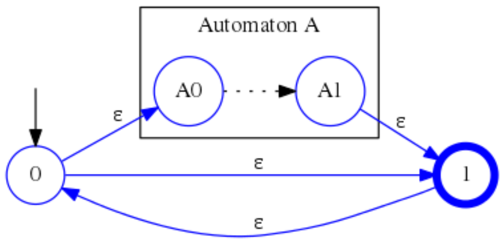
(c) Automaton accepting concatenation of languages



(d) Automaton accepting alternation of languages



(e) Automaton accepting empty language



(f) Automaton accepting iteration

Figure 2.1: Automata construction by Thompson algorithm

Definition 24. ***Step of automaton:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be the finite automaton and $\chi_1 = Q_1xy$ and $\chi_2 = Q_2y$ are configurations of the M, where $x \in \Sigma \cup \epsilon$, $Q_1 \subseteq Q$, $Q_2 \subseteq Q$ and $y \in \Sigma^*$. If $Q_2 = \cup_{s \in Q_1}\delta(s, x)$, then transition from χ_1 to χ_2 is called step of the automaton M and x is called accepted symbol.*

Let's have input string x and the automaton $M = (Q, \Sigma, \delta, s, F)$. The automaton starts its operation with one active state, which is the starting state. More formally, automaton starts in configuration $\{s\}\, x$. The automaton performs a step for every symbol in the input string. If there are ϵ-transitions in the automaton, after accepting every symbol from x, a step from symbol ϵ has to be performed as well. If there is at least one active final state after accepting all symbols in the input string, we say that the automaton accepts the input string and, therefore, the input string belongs to the language specified by the automaton M.

The transition function δ of the given automaton represents transitions that can be performed from a given active state and the input symbol. However, it is possible to extend the transition function in such way so that it will take the string instead of one symbol.Such an extension is useful for the simple formal definition of the language accepted by the automaton.

Definition 25. ***Extended transition function:*** *Let's have the nondeterministic finite automaton $M = (Q, \Sigma, \delta, s, F)$. The extended transition function δ^* of automaton M is* *mapping* $\delta^*_M : 2^Q \times \Sigma^* \to 2^Q$ *defined as follows:*

$$\delta^*(S,x) = \begin{cases} \cup_{s\in S}\delta(s,x) & \text{if } |x| \leq 1 \\ \delta^*(\delta^*(S,y),z) \wedge yz = x \wedge |z| = 1 & \text{if } |x| > 1 \end{cases}$$

The extended transition function takes as its parameter the whole string and the set of active states. If the input string is of a length smaller or equal to one, the extended transition function computes the transition function for every active state and the input symbol and returns the union of these results. It is important to note, that the definition of the transition function allows x to be an empty string. If the input string has a length larger than one, the extended transition function is defined by the recursion. It splits the input string into two parts by separating the last symbol of the string into a separate string. The remaining part of the string is used as an input of the recursive call of the extended transition function. The recursion repeats as many times as the length of the input string and since within each step one symbol is separated, the function is decomposed into several calls of the transition function itself.

Definition 26. ***Accepting string:*** *Let's have an finite automaton* $M = (Q, \Sigma, \delta, s, F)$ *and its extended transition function* δ^**. We then say that automaton* M *accepts string* x *if and only if*

$$F \cap \delta^*(s,x) \neq \emptyset$$

Definition 27. ***Language of the automaton:*** *Let's have the finite automaton* $M = (Q, \Sigma, \delta, s, F)$*. The automaton accepts language L, so, that*

$$L_M = \{x|x \subset \Sigma^* \wedge x \textit{ is accepted by } M\}$$

The NFA may be described by a simple enumeration of all its five components. However, such enumeration can be hard to read and interpret for the human reader.

In order to avoid confusion, two standard representations of automata are used throughout this thesis. The first is the table representation of the transition function. This representation describes the Q, Σ and δ in one table. Table 2.1 is an example of a table representation of the transition function. The first row of the table contains a list of all symbols in alphabet Σ, while the first column of the table contains a list of all states in Q. Every combination of the input symbol and state has its own unique cell in the table. The cell contains a set of target states of the transition. For example, let's take active state q_1 and input symbol b. The q_1 is found in the second row of the table and b is seen in the third column. The cell at this position contains state q_2, which means that the automaton contains a transition from q_1 to q_2 by the symbol b. It is possible that there is no transition for the given combination of the state and input symbol. This case is demonstrated by a combination (q_3, b) or (q_3, c). It is important to remember, that the table does not represent the information about the starting state or final states of automaton. However, there is only one starting state and very few final states and, therefore, this information can be stored in a text representation after the table.

The second representation of automaton $M = (Q, \Sigma, \delta, s, F)$ is based on a directed graph $G(Q, E, l)$. The Q is a set of states of the automaton, E is set of edges, so that

$$(q_1, q_2) \in E \iff \exists s \in \Sigma : q_2 = \delta(q_1, s)$$

State / Symbols	a	b	c
q_1	$\{q_1\}$	$\{q_2\}$	$\{q_3\}$
q_2	$\{q_2\}$	$\{q_1\}$	$\{q_3, q_1\}$
q_3	$\{q_3\}$	$\emptyset$	$\emptyset$

Table 2.1: Example of the table representation of NFA

and l is an edge labeling function so that $l\left(\left(q_1, q_2\right)\right) = \left\{s : s \in \Sigma \wedge q_2 = \delta\left(q_1, s\right)\right\}$. Starting and final states are graphically highlighted in the graph. The example of this representation can be seen in Figure 2.2. The represented automaton is the same as the automaton from Table 2.1. States are represented by circles and transitions are represented by edges of the graph. The label of the edge represents the input symbol, and the direction of the edge determines the direction of the transition. There is one edge that does not have any source state. This edge does not represent transition but serves as an indication of the starting state. Final states are represented by the bold circles in the graph.

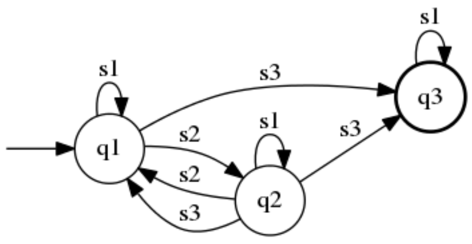


Figure 2.2: The example of graph representation of NFA

This thesis focusses on pattern matching in the high speed networks. The majority of patterns used in this field describe regular language. The main motivation for the introduction of the finite automata in this work is to be able to effectively decide whenever given string belongs to the given regular language. It can be seen that while one transition in the automaton is a fairly simple operation, one step of the nondeterministic finite automata may require to perform several transitions, since the transition is performed for every active state and the number of the active states may be as large as the number of states in the automaton. In order to be able to evaluate possible efficiency of automata based approaches, it is necessary to define some notations of the size of the automaton.

Definition 28. ***Size of the automata:*** *The size of the automaton $M = (Q, \Sigma, \delta, s, F)$ is defined as the number of states in the state set Q. The size of the automaton M is denoted by $|M|$.*

Definition 29. ***Transition size of the automata:*** *The transition size of the automaton $M = (Q, \Sigma, \delta, s, F)$ is defined as the number of transitions defined as the size of the domain of definition of δ. The transition size of the automaton M is denoted by $|M|_T$.*

The transition size of the automaton corresponds with the number of transitions in the automaton. This number can be smaller than $|Q| \times |\Sigma|$ since not all combinations of active state and input symbols must describe valid transition in the nondeterministic finite automaton.

Theorem 1. ***Time complexity of nondeterministic finite automata:*** *Let the nondeterministic finite automaton be $M = (Q, \Sigma, \delta, s, F)$ and input string x. Then, the decision if x belongs to the language accepted by the M can be done in $O(|Q|^2 \times |x|))$.*

The intuition behind this theorem is that in the worst case scenario, all states of the automaton M may become active for every symbol in the input string x and due to nondeterminism it is possible to perform a transition to every state. Therefore, $|M|^2$ transitions are performed in the worst case for every symbol in the input string. The algorithm working with this time complexitycan be found in [101].

It can be easily seen, that many different automata may accept the same language L. According to theorem 1, the number of state of the automaton plays a crucial role in the time complexity of the matching. Therefore, reducing the number of state in the automata will reduce the upper bound of time required for detection if the string belongs to the language accepted by the automaton.

Definition 30. ***Minimal nondeterministic finite automata:*** *The minimal nondeterministic finite automata $M_{min} = (Q_{min}, \Sigma_{min}, \delta_{min}, s_{min}, F_{min})$ for language L is the automaton for which the following condition holds:*

$$\forall M : L_M = L_{M_{min}} \implies (|M_{min}| \leq |M|)$$

Definition 31. ***Transition minimal nondeterministic finite automata:*** *The minimal nondeterministic finite automata $M_{min} = (Q_{min}, \Sigma_{min}, \delta_{min}, s_{min}, F_{min})$ for language L is the automaton, for which the following condition holds:*

$$\forall M : L_M = L_{M_{min}} \implies (|M_{min}|_T \leq |M|_T)$$

The minimal NFA according to definition 31 is defined as such an automaton that there is no smaller automaton. It is important to keep in mind that according to the given definition, there may be several minimal nondeterministic automaton, as long as they are all the same size.

Theorem 2. ***Uniqueness of minimal nondeterministic finite automata:*** *There exists language L accepted by automata M_1 and M_2, which both are minimal nondeterministic finite automata and $M_1 \neq M_2$.*

The language L_{00} described by the regular expression 00^* is an example of language with more than one minimal nondeterministic finite automata. Figure 2.3 shows two NFA accepting language L_{00}. Both show automata are minimal NFA.

The problem of finding the minimal nondeterministic finite automaton for the given NFA is PSPACE complete [53] and the size of the NFA cannot be efficiently approximated within an approximation factor $o(m)$ where m is the number of states unless $P = PSPACE$. The complexity of NFA related problems are elaborated in [53, 52, 57, 88].

It is possible to reduce the size of the NFA by heuristics. The removing of the epsilon transition is one of the methods that is often used for the reduction of NFA size.

Definition 32. ***ϵ-free nondeterministic finite automaton:*** *The ϵ-free nondeterministic finite automaton (ϵ-free NFA) is 5-tuple $M(Q, \Sigma, \delta, s, F)$, where*

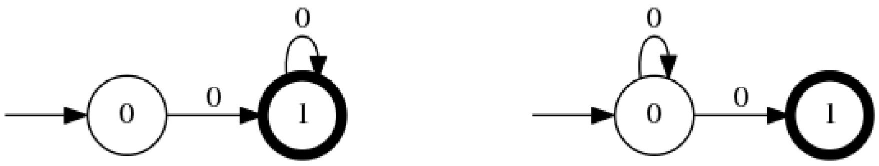


Figure 2.3: NFA for language L_{00}

- *Q is a finite set of states*
- *Σ is an input alphabet such as $\Sigma \cap Q = \emptyset$*
- *$\delta : Q \times \Sigma \to 2^Q$ is a transition function*
- *$s \in Q$ is the start state*
- *$F \subseteq Q$ is a set of final states.*

The only difference between definition 32 and 21 is the removing of the ϵ from the definition domain of the transition function. It is important to note that all previous definitions,theorems and observations discussed for the nondeterministic finite automata also holds for ϵ-free automaton.

Theorem 3. ***Equivalence of description power of NFA and ϵ-free NFA:*** *For every nondeterministic finite automaton M, there exists ϵ-free nondeterministic finite automaton M_e such that $L(M) = L(M_e)$*

Theorem 3 states that every NFA can be transformed into an ϵ-free NFA that accepts the same language. To prove the validity of the claim, algorithm 1 for transformation of NFA into ϵ-free NFA is presented. The core operation of the algorithm is ϵ-closure. Informally, ϵ-closure of a set of states is a function for computation of the set of all states, which can be reached by the sequence of the ϵ-transitions from given states. The ϵ-closure of one state is the ϵ-closure of a singleton set containing the state.

Definition 33. ***ϵ-closure:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be a nondeterministic finite automaton. The ϵ-closure is defined as*

$$closure_M(S) = \{q \in Q | q = \delta^*(S, \epsilon)\} \cup S$$

It should be kept in mind that for every automaton M every state s belongs to the $closure_M(S)$ of any set S that contains state s.

Algorithm 1 first creates relation δ_r that contains all transitions from the ϵ-closure of the given state. It is important to realize that δ_r cannot be the transition function, because the mapping from the domain of δ_r to the set of states of the automaton M_e is not unique. Moreover, δ_r is mapping from (Q, Σ) to Q, while the transition function of the NFA should be mapping from (Q, Σ) to 2^Q. Step 5 of the algorithm defines δ_e in such way that it contains correct transitions and fulfills all formal requirements of the transition function of NFA. The last step of the algorithm is a construction of the set of final states. The final state is every state that has some final state in its ϵ-closure in the original automaton. The set of states, starting state and the alphabet of the automaton remain the same.

Algorithm 1 Construction of ϵ-free NFA

```
Input: Nondeterministic Finite Automaton $M = (Q, \Sigma, \delta, s, F)$
Output: $\epsilon$-free Nondeterministic Finite Automaton $M_e = (Q, \Sigma, \delta_e, s, F_e)$
1: $\delta_r \leftarrow \emptyset$
2: for $p \in Q$ do
3:     $\delta_r \leftarrow \delta_r \cup \{(p, a) \to q | a \in \Sigma \wedge q \in Q \wedge \exists p^o \in closure_M(p) : q \in \delta(p^o, a)\}$
4: end for
5: $\delta_e = \{(q, a) \to S | (q, a) \in Dom(\delta_r) \wedge S = \{s | \delta_r(q, a) \to s\}\}$
6: $F_e = \{p | p \in Q \wedge closure_M(p) \cap F \neq \emptyset\}$
```

The removing of the ϵ-transitions is often followed by the other methods for the reduction of the size of the nondeterministic finite automata, such as prefix sharing [58]. There is a large number of methods designed to reduce the size of the nondeterministic finite automaton, however, their introduction is outside the scope of this work.

2.3.3 Deterministic Finite Automaton

The previous section described the nondeterministic finite automaton and stated the time complexity of the decision, whenever a given string belongs to the language accepted by the given NFA. According to theorem 1, the time complexity upper bound depends on the number of states in the automaton because all states of NFA can be active at once in the worst case scenario. As the automaton grows, the upper bound time complexity of accepting or rejecting a given string also grows. One way to reduce time complexity upper bound of the problem is to reduce the maximal number of active states in the automaton. For this reason, the deterministic finite automaton is defined. The deterministic automaton ensures that there is at most one next state for any combination of input symbol and a given state.

Definition 34. ***Deterministic Finite Automaton:*** *The deterministic finite automaton (DFA) is 5-tuple $M(Q, \Sigma, \delta, s, F)$, where*

- *Q is a finite set of states*
- *Σ is an input alphabet such as $\Sigma \cap Q = \emptyset$*
- *$\delta : Q \times \Sigma \to Q$ is a transition function*
- *$s \in Q$ is the start state*
- *$F \subseteq Q$ is a set of final states.*

The only difference between nondeterministic finite automaton and deterministic finite automaton is the definition of the transition function. The codomain of a transition function of NFA is a set of all possible subsets of Q (i.e. set of set of states), while the codomain of transition function of deterministic finite automaton is a set of states. It is still possible that there is no transition for the given state and input symbol in the deterministic finite automaton.

Due to the difference in the transition function, it is important to redefine the extended transition function for the deterministic finite automaton.

Definition 35. ***Extended transition function for deterministic finite automaton:*** *Let the deterministic finite automaton be $M = (Q, \Sigma, \delta, s, F)$. The extended transition function δ^* of automaton M is mapping $\delta^*_M : Q \times \Sigma^* \to Q$ defined as follows:*

$$\delta^*(s, x) = \begin{cases} \delta(s, x) & \textit{if } |x| \leq 1 \\ \delta^*(\delta^*(s, y), z) \wedge yz = x \wedge |z| = 1 & \textit{if } |x| > 1 \end{cases}$$

With the extended transition function, we may define the acceptance of the string and language exactly the same way as it was defined for the nondeterministic finite automaton.

Theorem 4. ***Equivalence of DFA and NFA:*** *For every nondeterministic finite automaton $M = (Q, \Sigma, \delta, s, F)$ there exists a deterministic finite automaton $M_D = (Q_D, \Sigma, \delta_D, s_D, F_D)$ so that*

$$L(M) = L(M_D)$$

Theorem 5. ***Equivalence of DFA and NFA:*** *For every deterministic finite automaton $M_D = (Q_D, \Sigma, \delta_D, s_D, F_D)$ there exists a nondeterministic automaton $M = (Q, \Sigma, \delta, s, F)$ so that*

$$L(M) = L(M_D)$$

We prove theorem 4 by providing algorithm 2 for the construction of the DFA from the given NFA which is called determinization of the NFA. Theorem 5 holds because the DFA is a special example of NFA. The NFA and DFA accept the same class of languages because theorem 4 and theorem 5 both hold.

Algorithm 2 Determinization of NFA

Input: ϵ-free Nondeterministic Finite Automaton $M = (Q, \Sigma, \delta, s, F)$
Output: Deterministic Finite Automaton $M_D = (Q_D, \Sigma, \delta_D, s_D, F_D)$

```
 1: $Q_D = 2^Q \setminus \{\emptyset\}$
 2: $s_D = \{s\}$
 3: for $s \in Q_D$ do
 4:     for $a \in \Sigma$ do
 5:         $T = \cup_{q \in s} \delta(q, a)$
 6:         if $T \neq \emptyset$ then $\delta_D(s, a) = T$
 7:         end if
 8:     end for
 9: end for
10: $F_D = \{S | S \in Q_D \wedge S \cap F \neq \emptyset\}$
```

Algorithm 2 produces the exponentially larger set of states. The exponential increase of the number of state is caused by a step 1 of algorithm 2, which defines the set of states of deterministic automaton as a set of all possible subsets of Q. However, not all states have to be useful for accepting the language of the automaton.

Definition 36. ***Accessible and unaccessible states*** *Let $M = (Q, \Sigma, \delta, s, F)$ be the finite automaton and δ^* its extended transition function. Let's have a set $S \subset Q$, so*

$$S = \{t | t \in Q \land \exists x \in \Sigma^* : \delta^*(s, x) = t\}$$

States belonging to the set S are called accessible while other states are unaccessible.

Unaccessible states are states that cannot became active for any given string. Removing unaccessible states from the automaton reduces the size of the automaton without affecting the language of the automaton.

Definition 37. ***Terminating states:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be the finite automaton and δ^* its extended transition function. The set $S \subset Q$ of terminating states is defined as follows*

$$S = \{t | t \in Q \land \exists x \in \Sigma^* : \delta^*(t, x) \in F\}$$

States in the set S are called terminating states and states not present in set S are called nonterminating states.

Definition 37 divides the states of the automaton into two groups. The terminating states are states from which it is possible to activate some final state and, therefore, it is still possible that the input string belongs to the language accepted by the automaton. However, for the nonterminating states, there is no string that will result in the activating of any of the final states of the automaton and when this state is reached, the input string can never be accepted. Therefore, it is clear that the given input string does not belong to the language accepted by the automaton. Removing of nonterminating states from the automaton will not affect the language accepted by the automaton, but reduce the size of the automaton.

It is possible to have combination of the input symbol and active state that do not belong to the domain of the definition of the transition function of the automaton. This situation indicates that the input string does not belong to the language defined by the automaton. However, it also prevents the automaton from accepting the whole input string which is good from the point of view of the time complexity but may present a problem from a theoretical standpoint. In order to mitigate this problem, the definition of the complete deterministic finite automaton 38 is present.

Definition 38. ***Complete finite automaton:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be the finite automaton. The M is complete if and only if $Dom(\delta) = Q \times \Sigma$.*

The automaton is called complete if for all possible combinations of state and input symbol, there exists the transition in the automaton. It is clear that any finite automaton may be transformed into a complete automaton by adding a new nonterminating state and every nondefined transition will lead to this state. It is important to keep in mind that some other works, such as [82], define the transition function of the DFA as a total function and, therefore, the DFA from their standpoint is always complete.

Definition 39. ***Well-specified automaton:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be a complete deterministic automaton. M is called well-specified if the following condition holds*

- *All states in Q are accessible*
- *There is maximally one nonterminating state in Q*

The well-specified automaton is a complete deterministic automaton and the only nonterminating state is used to have the complete automaton. This automaton is compromise between automaton without nonterminating states and the requirement of having a complete DFA.

Definition 40. ***Indistinguishable states:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be the finite automaton and δ^* is its extended transition function. Two states p and q are called indistinguishable if and only if*

$$\forall w \in \Sigma^* : \delta^*(p, w) \in F \implies \delta^*(q, w) \in F \wedge \delta^*(p, w) \notin F \implies \delta^*(q, w) \notin F$$

States p and q are called in distinguishable if and only if

$$\exists w \in \Sigma^* : \delta^*(p, w) \in F \wedge \delta^*(q, w) \notin F$$

The relation defined by the indistinguishable states on the well-specified automata is its equivalence. Therefore, it is possible to divide the set of states of the given well-specified automata into the equivalence classes defined by the indistinguishable states. Then it is possible to define a new automaton, whose states are the equivalence classes of the well-specified automaton.

Definition 41. ***Reduced deterministic finite automaton:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be the well-specified automaton. M is called reduced deterministic finite automaton if it does not contain indistinguishable states.*

Any well-specified automaton can be transformed into the reduced deterministic finite automaton by algorithm 3.

Algorithm 3 Minimization of DFA

```
Input: Well-specified Automaton $M = (Q, \Sigma, \delta, s, F)$
Output: Reduced Deterministic Finite Automaton $M_R = (Q_R, \Sigma, \delta_R, s_R, F_R)$
 1: Compute relation $R$: $R = \{(s,t) | s, t \in Q \wedge s, t$ are indistinguishable$\}$
 2: Compute quotient set of $Q$ by $R$: $Q_R = Q/R$
 3: $\delta_R = \emptyset$
 4: for $r \in Q$ do
 5:     Find $q_r$ such that $q_r \in Q_R \wedge r \in q_r$
 6:     for $a \in \Sigma$ do
 7:         $t = \delta(r, a)$
 8:         Find $q_t$ such that $q_t \in Q_R \wedge t \in q_t$
 9:         if $(q_r, a, q_t) \notin \delta_R$ then $\delta_R = \delta_R \cup (q_r, a, q_t)$
10:         end if
11:     end for
12: end for
13: Find $s_R$ such that $s_R \in Q_R \wedge s \in s_R$
14: $F_R = \{f | f \in Q_r \wedge \exists k \in F : k \in f\}$
```

The first step of 3 is the computation of the relation of all indistinguishable states. This relation is used in the second step to partition the set of states of the input automaton M.

The equivalence classes from this partition are states of the new automaton M_R. Therefore, states of the reduced automaton are in fact equivalence classes of the state set of automaton M. The δ_R starts as an empty relation and is filled in steps from 4 to 12. Step 6 finds the state q_r of the newly constructed reduced automaton. Therefore, $q_r \in Q_R$ condition. The second part of the condition specifies that q_r is the equivalence class that contains state r of the original automaton. The same notation is also used in step 8. In step 9, the new transition between the states determined in the previous steps are added into the transition function. The starting state is determined in step 13 as a class containing the starting state of M. The last step of the algorithm is to compute the set of final states. The final state set consists of all states that contain any final state of automaton M.

Theorem 6. ***Minimal deterministic automaton:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be the well-specified deterministic finite automaton and $M_R = (Q_R, \Sigma, \delta_R, s_R, F_R)$ be the reduced deterministic finite automaton created by algorithm 3. Then $L(M) = L(M_R)$ and M_R is the minimal deterministic automaton and there is no DFA that accepts $L(M)$ with fewer states.*

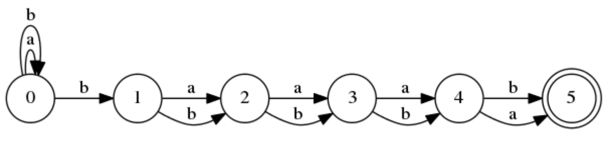
The proof of the 6 can be found in [82] as proof of theorem 2.4.

Theorem 7. ***Uniqueness of minimal DFA:*** *There is only one unique minimal deterministic finite automata.*

Algorithm 2 generates deterministic finite automata with up to exponentially more states than the given NFA, which may be prohibitive for practical use of the deterministic finite automata. According to previous results, it is possible to construct a unique minimal deterministic automaton from the given nondeterministic finite automaton. The important question is how large a reduction of the number of states can be achieved by the minimization algorithm 3. Theorem 8 shows the worst case situation. However, algorithm 3 can introduce significant state reduction in many real world situations. Therefore, it is always necessary to evaluate the efficiency of the minimization technique with the specific automaton in mind.

Theorem 8. ***Correspondence between the size of NFA and DFA:*** *There exists regular language L, so that if nondeterministic finite automata requires n states to be able to accept L, then minimal deterministic automaton accepting L will require at least 2^n states.*

The language accepted by the automaton from Figure 2.4 is an example of the language from theorem 8.



$$\begin{tikzcd}
0 \arrow[loop above, "a,b"] \arrow[r, "b"] & 1 \arrow[r, "a"] \arrow[r, bend right, "b"'] & 2 \arrow[r, "a"] \arrow[r, bend right, "b"'] & 3 \arrow[r, "a"] \arrow[r, bend right, "b"'] & 4 \arrow[r, "b"] \arrow[r, bend right, "a"'] & 5
\end{tikzcd}$$

Figure 2.4: NFA causing exponential state blow up during the determinization

The starting state of the automaton contains a loop by both symbols of its alphabet which results in the fact, that it is always active. When the start state is active and the input symbol is b, the second state becomes active and automaton starts counting the remaining symbols. Every b in the input string activates the counting of the number of symbols b.

The minimal deterministic automaton for this language is shown in Figure A.1 in Appendix A. While the NFA has only 6 states, the minimal DFA for the same language has 32 states. The reason for this blow up is the fact that DFA has to have a special state for every combination of activities in the counting part of the NFA, because before the last symbol is processed, it is not known which, if any, active state will result in the acceptance of the input string.

The state blow up represents the main disadvantage for the application of the DFA in realworld problems. However, unlike the NFA, the number of states do not affect the time complexity of the acceptance of the input string.

Theorem 9. ***Time complexity of DFA:*** *Let $M = (Q, \Sigma, \delta, s, F)$ be deterministic automaton and x be an input string. M is able to accept or reject x by performing at most $|x|$ steps.*

The time complexity of the deterministic automaton is linear with a number of symbols in the input string, because there is always at most one active state in the automaton and there is only one transition that can be performed by the given input symbol. The algorithm working with this time complexitycan be found in [101].

2.4 Pumping lemma for regular languages

The models for describing regular languages were provided in the previous section. It is clear that language described by the regular expression or finite automaton is regular language. However, the patterns used in the deep packet inspection often contain extensions in order to make them more simpler and understandable for the human user. It is possible that these extensions increase the descriptive power of the pattern beyond regular languages. Therefore, it is necessary to analyse the regularity of these extensions. Pumping lemma is the tool which can be used in this analysis.

If the language is described by different means of expression, it may be difficult to established its regularity. The claim that the language is regular can be proven by the construction of the regular expression or finite automaton that accepts a given language. Since there is no algorithm for the construction of a regular expression or finite automata from an arbitrary description of the language, the failure to construct finite automaton does not prove, that language is not regular because there may exist some undiscovered way to build the regular expression or finite automaton. In order to prove that a given language is not regular, it is necessary to prove that it is not possible to construct the finite automaton accepting the language.

The pumping lemma is an important tool in proving that the given language is not regular. It was first introduced in [99]. Pumping lemma is defined for the infinite language, because all finite languages are regular (we can simply enumerate all strings in the language). If the language does not meet the condition in the pumping lemma, it cannot be regular a language.

Theorem 10. ***Pumping lemma for regular languages:*** *Let L be an infinite regular language. Then there exists some positive integer m so that any $w \in L$ with $|w| \geq m$ can be decomposed as $w = xyz$, with $|xy| \leq m$ and $|y| \geq 1$, so that $\forall i \in \mathbb{Z}^* : xy^iz \in L$.*

The formulation of theorem 10 is taken from [82] where it is accompanied with proof. The basic idea behind the lemma is that if the string is longer than the number of states in the automaton then there has to be a loop in the sequence of transitions used to accept the string. Since the loop ends in the same state where it started, it is possible to „pump“ the loop as many times as necessary. Hence pumping lemma.

Theorem 11. ***Regularity of*** a^nba^n***:*** *The language* $L = a^nba^n|n \geq 0$ *is not regular language.*

Proof. Let's assume that language L is regular. Therefore, pumping lemma must hold. The string $s = a^mba^m$ belongs to L. If string s is divided into xyz according to the conditions specified by pumping lemma, the string y must be equal to a^k where $0 < k < m$ and $x = a^{m-k}$ According to pumping lemma, if $s \in L$, then also $xz \in L$ and $xz = a^{m-k}ba^m$. However, $m - k \neq m$ and, therefore, $xz \notin L$ which is a contradiction and, therefore, original assumption that L is regular is not true. □

The presented proof is a typical example of the use of pumping lemma. It is important to keep in mind that the pumping lemma cannot be used to prove that the given language is regular.

2.5 Beyond regular expressions

The regular expressions have been widely used in modern computer science. However, the practical application of the regular expression often requires us to implement new extensions. Some of these extensions are only for the increasing the comfort of the usage. An example of such extension is the character classes, since every character class may be described by the alternations. It is important to keep in mind that even if the description power of the language is not increased by the extensions, it still can partially affect some of the results from the theory of regular languages. For example, the character class allows us to significantly reduce the length of regular expressions. Therefore, any theoretical result which depends on the size of the regular expression will become invalid if the regular expression is not transformed into the form without character classes. Examples of such results may be found in [26]. It is important to realize that the impact of extensions may be extremely large, since it is possible to denote the whole alphabet by five symbols in the expression no matter how large the alphabet is. Therefore, it is possible that the regular expression with five symbols will be transformed into the automaton with hundreds of transitions.

However, there are some other extensions of regular expressions that increases the description power of regular expressions. The most common example of such an extension is the backreference. The backreference allows us to match the same string as was matched by the specified capturing group earlier in the matching process. The regular languages with these extensions are sometimes called „practical regular expressions“ or „regexps“ in theoretical work. The programmers and ordinary users often call them just regular expressions.

This work focuses mainly on the regular expressions in the theoretical sense and their syntactic extensions that do not increase the description power of the language beyond the power of regular expressions, such as character classes and constrain repetition. Therefore, in this work all patterns are called regular expressions. The term *theoretical regular expression* or *clasical regular expression* is used to emphasize that no extensions are considered.

If a specific extension increases the descriptive power beyond the regular languages, it will be discussed specifically with this extensions. Moreover, it is important to keep in mind that not all patterns which are described with these powerful extensions are describing language, which is not regular. For example, back reference is often used to describe a one layer parenthesis. The described situation is shown in the regular expression 1. For an easier orientation, the regular expression was broken into three lines. From the point of the regular expression matching, the first line matches a string. The last symbol of this string can be an apostrophe or double quote symbol or nothing. This alternation is contained in the brackets that is used as a target for backreference at the end of the second line. The second line matches such a string so that its last symbol is the same as the symbol in the brackets at the end of the first line. However, the described language can be represented by a much larger regular expression using only alternations as is shown in the regular expression 2.

Regular expression 1 Backreference is used for correct handling of apostrophes and double quotes

```
/expression1 (\x22|\x27|)
 expression2 \1
/si
```

The regular expression 2 consists of three copies of the same regular expression that differs only in the symbol that is used to encapsulate the string matched by the second line. It is important to realize that while this version of the regular expression does not use any structure that can describe nonregular language, it is three times longer. This example also illustrates the trade off between the complexity of the description language and the size of the description. Simply put, the more powerful model often allows a more succinct representation. The analysis of these structures and their affect on the efficiency of the regular expression matching is one of the contributions of this work.

Regular expression 2 Alternation used for correct handling of apostrophes and double quotas

```
/ expression1 \x22 expression2 \x22|
 expression1 \x27 expression2 \x27|
 expression1 expression2
/si
```

2.5.1 Perl Compatible Regular Expression

The Perl Compatible Regular Expressions (PCRE) [7] are used to describe advanced context matching. The complete description of the PCRE grammar is outside the scope of this work. However, this section will describe extensions of PCRE that are most commonly used in IDS in the comparison with the theoretical regular expression. See [48] [7] for a better understanding of the PCRE. The content of this section is also inspired by the [9], which contains the description of regexpses used in modern pattern languages together with the discussion on how every construction works and affects the searching engine.

Character classes

One of the most often used extensions in the PCRE are character classes. The character class is the set of symbols and every symbol from the set can be accepted at the position of the character class.

2.5.2 Dot-star construction

The symbol of dot */./* is used to denote any possible symbol from the alphabet. The dot-star construction */.*/* is the combination of the dot and unconstrained repetition. It is the method used to describe that any possible string can be accepted in the part of regular expression. Unfortunately, the dot-star construction causes state blow-up during the determinization of the automaton. It is important to note that any large character class combined with the repetition will have a similar impact.

Capturing group

The capturing group is a part of the regular expression that is denoted by brackets. Therefore, it is possible to apply the repetition on the whole subexpression instead of just one symbol. Moreover, the capturing group can be indexed in different parts of the expression by a more advanced concept, such as backreferences or subroutines. The capturing group can be named. The named capturing group can be referenced by its name or by the index of its parentheses.

Beginning and End of string

It is important to understand the meaning of the metacharacter ˆ at the beginning of the regular expression. This metacharacter means the beginning of the line. If this metacharacter is not present at the beginning of the regular expression, the string described by this expression can be found anywhere in the input stream. Therefore, the regular expression has to be modified to accept any possible string that occurs before the string it searches for. The dot star construction is used to describe this situation. For example, if the user writes regular expression */Hello/*, the IDS understood it as */.*Hello.*/*. But if */ˆHello/* is used, IDS interprets it only as */Hello.*/*. Therefore, adding the ˆ removes one dot star construction from the regular expression.

The end of the line is denoted by the metacharacter $ in the regular expression. The end of line metacharacter is matched after the last symbol of the input stream was accepted. Therefore, it allows us to anchor the pattern to the end of the data.

Options

The PCRE matching engine supports options, which influences the behaviour of the matching unit for the specific regular expressions. Therefore, two of the same regular expressions may describe different languages depending on the setting of these options. The options may be set inside of the regular expression or after the text of the expression. Table 2.2 summarises the options and their meanings.

It can be seen from the description in Table 2.2 that the options can drastically change the complexity of the resulting regular expressions. For example, option i significantly reduces the size of the alphabet. Option m is an example of the option that increases the complexity of the matching. The presence of option m changes the meaning of the beginning

Option	Option Description
i	This option allows case insensitive matching.
m	This option change the behaviour at the beginning and end of the line. If the option is set, new line characters are considered to be the end of line. Otherwise, the whole input string is considered to be one line.
s	Dot metacharacter matches also newline characters in the input stream.
x	Whitespace data characters in the pattern are ignored except when escaped or inside a character class. This option allows a more readable notation of patterns.
O	Override the configured PCRE match limit and PCRE match limit recursion for this expression.

Table 2.2: Options for PCRE matching engine

of the string to actually mean the beginning of the line. Therefore, the string described by the regular expression may start after any newline symbol such as \n. Therefore, even if the metacharacter ˆ is present, the regular expression has to start with a dot star type construction.

Repetition

The classical regular expressions contain two types of repetition, called star and plus. These symbols allow for the unlimited number of repetitions of the given regular expression or its part. However, the PCRE support other types of the repetition. One is denoted by the *?* and means zero or one time. This construction does not affect the complexity of the matching and it can be easily described by alternation in the framework of classical regular expressions. However, the second extension is called constraint repetition and can produce significant problems for the matching systems based on finite automata. Constraint repetitions are denoted by the $\{m,n\}$ and this means that the preceding capturing group has to be repeated at least m times and at most n times. It can be shown that the constraint repetition does not increase the description power of the regular expression languages because every constraint repetition can be rewritten by manually repeating the regular expression. However, it is easy to construct a very complex expression by writing a few characters. The regular expression 3 is an example of a simple expression that results in large automaton. The automaton created from this expression has 4000 states, whereas the expression has only 12 characters. The expression from the example is very simple and, therefore, it is easy to guess the correct number of states in the resulting automaton. However, if the expression is more complex, the resulting number of states in automaton may quickly grow out of any reasonable bounds. It is important to keep this information in mind while interpreting the analysis about the regular expressions sets, because most of such analysis report the number of symbols and meta-symbols in the pattern.

Regular expression 3 Simple regular expression resulting in large automaton

```
(test){1000}
```

Moreover, the constrained repetition is often used in combination with other RE constructions that are responsible for state explosion during the determinization and, therefore, they add another dimension to this problem.

Backreference

As mentioned earlier, the backreference allows us to match the same string as was matched by the specified capturing group earlier in the matching process. If the specified capturing group was processed more than once during the matching, the backreference refers to the string that was accepted by the last successful matching of the capturing group. The capturing group can be specified by its index in the regular expression or by its name. The regular expression 4 is an example of the named backreference. Almost every regular expression matching system has its own syntax. Fortunately, PCRE allows us to use the syntax of most common matching systems. The python syntax was used in the example because python module *re* was the first matching system which introduced named backreferences.

Regular expression 4 Example of named backreference written in Python syntax

```
(?P<named>[A-Z]*).*/(?P=named)
```

Regular expression 4 starts with the capturing group that matches an arbitrary long sequence of upper letters. The name *named* was assigned to the capturing group. Therefore, this capturing group can be accessed by its index or name. After the capturing group, the regular expression accepts an arbitrary string finished by a slash. The *(?P=* is the actual backreference, therefore, the string that was matched by the first capturing group has to be matched again after the slash. Due to the fact that backreferences are referenced by name or their index in the regular expression, it is possible to backreference the capturing group that was not accepted. This situation is shown by regular expression 5.

Regular expression 5 The regular expression with backreference to capturing group that can be skipped during the matching

```
(q)?b\1
```

Regular expression 5 describes the language containing the string *qbq*. The string *b* does not belong to this language, because to accept *b*, the capturing group with *q* is not matched. Therefore, the content of the backreference cannot be matched. In order to accept the string *qbq* together with *b*, the regular expression has to be modified as shown in regular expression 6.

Regular expression 6 Regular expression accepting strings *qbq* and *b*

```
(q?)b\1
```

The capturing group in regular expression 6 is always matched against the string. If the input string contains the symbol *q*, it is accepted by the capturing group. In the case of symbol *b*, the capturing group accepts only the empty string. The backreference has to be accepted after accepting symbol *b*.

Theorem 12. ***Regularity of languages with backreferences:*** *There exists languages described by the backreferences that are not regular languages.*

Proof. According to theorem 11, the language a^nba^n is not regular. The practical regular expression (a)*b\1 accepts this language. Therefore, the backreference allows us to describe nonregular language and theorem 12 holds. $\square$

Subroutine

The subroutine extension of regular languages allows for a simpler and more compact description of the regular expressions by allowing us to reuse part of the regular expression already written. If the regular expression contains several equal subexpressions, it is possible to denote the first occurrence of the subexpression as a subroutine and use this subroutine instead of writing the whole subexpression at the next occurrence of the subexpression. Example 7 shows the regular expression from 2 rewritten with the use of subroutines. The purpose of the regular expression is to ensure that *expression2* is surrounded by correctly formed apostrophes, quotation marks or nothing. Regular expression 2 achieves this goal by containing *expression1* and *expression2* three times. If these expressions are large and complex the orientation in the whole regular expression is troublesome. The regular expression 7 has the same first line. The only exception is that *expression1* and *expression2* are placed into capturing groups in order to be accessible later in the regular expression. The second and third line of the regular expression contains the use of subroutines. Instead of copying the whole *expression1* at the beginning of the second line, the structure *(?1)* forces the searching engine to use the expression that is written in the first capturing group. Therefore, *expression1* is matched. The first subroutine is followed by the character *\x27* and subroutine indexing *expression2*. The same situation is in line three. It is important to note that the subroutine copies the regular expression and not a string that was matched by this expression. Therefore, it can index part of the expression that was not matched yet. It can be seen that while the complexity of the matching process remains exactly the same, the regular expression is much simpler.

Regular expression 7 Alternation used for correct handling of apostrophes and double quotes with the subroutine extension

```
/( expression1 )\x22 ( expression2 )\x22|
(?1)\x27(?2)\x27|
(?1)(?2)/si
```

However, the subroutines can be used to increase description power beyond the framework of the classical regular expression. This increase can be achieved by the use of the recursion. The subroutine can be called inside its own definition, which results in the recursion of the regular expressions. This recursion can be used to define languages which are not regular. Expression 8 is an example of PCRE expression that defines language a^nb^n. The pumping lemma for a regular expression can be used to show that from the point of computer science, this language is not a regular language.

Regular expression 8 PCRE accepting language a^nb^n for $n \geq 1$

```
a(?0)?b
```

The matching process of this expression can be demonstrated on the string *aabb*. In the first step, *a* is accepted because it is the first symbol of the regular expression. After accepting *a*, the match of the subroutine begins. Therefore, the second *a* is accepted by the matching engine. Then, the subroutine could start again. However, there is no *a* in the input stream. The subroutine call is followed by the question mark, which is qualifier meaning zero or one time repetition. Therefore, if there is no more *a*, the subroutine is not matched and the matching engine will continue to accept the next symbol, which is *b*. After accepting symbol *b*, the matching of the subroutine ends and the matching engine will continue by accepting the second symbol *b*.

Assertions

In order to understand assertions in regexpses, it is important to understand how regexpses are used. Snort, for example, may match several patterns against one input stream in a predefined order (i.e. second pattern may be matched only after the first one). Therefore, the exact beginning and end of the match is required.

Let's suppose that the user wants to detect string *Hello*, but only if it is followed by exclamation mark. It is possible to write a regular expression */Hello!/*. However, such a regular expression will end its matching after the exclamation mark, which may cause problems in higher layers of the IDS. The practical regular expression 9 uses an assertion to describe this situation. The structure *(?=!)* forces the matching engine to look at the next position of the string and if it is *!*, the current position is returned as the end of the matched string.

Regular expression 9 Regexp with assertion

```
Hello(?=!)
```

The assertion is part of the regular expression that is matched without accepting the symbols from the input. In practice, it means that the assertion is matched as any other part of the regular expression, but if it matches the input string, the symbols accepted by the assertions are not considered accepted by the matching unit. The assertion can contain an arbitrary complex regular expression. The use of the assertion may, therefore, require processing the same input symbols many times, which could lead to the slowing of the matching process. For example, time complexity of the deterministic finite automata is considered O(n), under the assumption that one symbol is processed in constant time. However, in the presence of assertions in the regular expression, an input symbol may be processed by several assertions, which will increase the time complexity of the matching.

Chapter 3

Requirements for Pattern Matching

Pattern matching is used in many areas of computer science. Linux users are familiar with programs like grep or sed, which are used to match patterns in text files. This work focusses on the use of pattern matching in the area of network security.

Unfortunately, almost every application uses a different formalism for describing patterns. The different formalisms have different descriptive powers. The differences in formalisms are dictated by the application needs and the decision of developers of the application. However, as a result of these decisions, every application requires a different approach and algorithms for pattern matching.

Every application is required to deal with trade-offs between the speed of pattern matching and complexity of patterns. Complex patterns are required due to their succinctness and high descriptive power. However, whenever an input string belongs to the language described by the pattern, any decission requires more computation resources for patterns with a higher descriptive power. While it is relatively easy to implement a fast matching unit for several thousands strings, it is impossible to implement a matching unit able to match arbitrary PCRE which works in linear time[13]. Therefore, it is very important to choose the right level of description for pattern matching with respect to requirements of the application.

Even while Paxson introduced [115] that string patterns are not powerful enough to describe all attacks in modern networks, they are still used together with other languages because algorithms for string matching are fast. For example, program Snort [11, 103] allows us to use both string patterns and PCRE patterns in a rule set.

Authors of anti-virus ClamAV [3] developed their own pattern language. The ClamAV pattern can contain an alternation but does not contain repetition. Therefore, it does not have the descriptive power of regular language. The only possible repetition is a repetition of *any* character. This structure allows us to place a „gap“ between two part of the ClamAV pattern. In this case it is possible to limit the length of the „gap“ by an exact number or by range. Patterns are created from string patterns which are connected by a gap or alternation. The described limitations are heavily used for the acceleration of ClamAV pattern matching. However, the ClamAV starts to support the PCRE pattern in version 0.99[2].

3.1 Regular Expression

Regular expressions offer a higher descriptive power than string patterns. There exists many algorithms which can be used to search for the strings described by the regular expression in the input string. According to theorem 9, the worst-case time complexity of a matching input string against a pattern described by a regular expression is $O(n)$, where n is the length of the input string.

Patterns described by regular expressions require more complex preprocessing than string patterns and their memory consumption is also higher in general.

The regular expression is used in the IDS Bro [94], because it is possible to match the regular expression in real time [1] by implementing Deterministic Finite Automata. Even if the IDS supports more powerful language, it is often possible to rewrite most of its rules to the regular expression in order to speed up the matching process.

3.2 Extended Regular Expression

Extended Regular Expressions were developed in the industry as an extension of regular expressions and, therefore, there are many standards and implementations which differ. Although most of the extensions are on the syntactical level, for example, character classes, some extensions increases the descriptive power of their language. The Perl Compatible Regular Expression (PCRE) [7] and POSIX Extended Regular Expressions [12] are examples of the extensions which increase descriptive power.

The theoretical properties of regexpses are studied in [39, 40]. The most important result is proof that the matching of regexp is an NP problem in general. Therefore, systems which are using PCRE for describing their patterns cannot guarantee a fast enough response time to mitigate the threat or arbitrary rule. It is the responsibility of the author of the rule that the matching will not slow down the system.

3.3 Application of Pattern Matching Languages

Pattern matching is the core operation of many antivirus softwares, Intrusion detection systems or network filtering mechanisms. Another possible usage of pattern matching is, for example, in the field of natural language processing or in any analysis of the text data. This work is focused on the pattern matching applications used in network security, mostly IDS. This section describes software that leverage the pattern matching system and which uses patterns used further on in this work as test sets.

3.3.1 Snort

Snort [11] is the open source Intrusion Detection System which performs deep packet inspection in order to detect the threats in network traffic. The incoming network traffic is matched against a set of rules, where every known threat is described by one or more rules. The rules are described by the Snort proprietary language.

Figure 3.1 shows the basic architecture of the Snort and how it processes network traffic. The first step is called *packet decoder* and is responsible for the correct capturing of network

[1] The IDS has to detect a threat fast enough that the system is able to mitigate it.

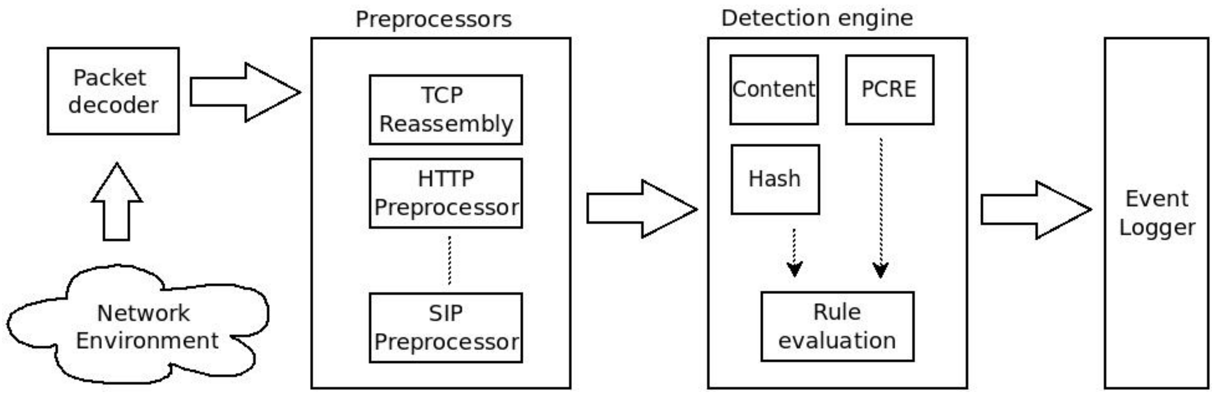


Figure 3.1: Architecture of Snort

traffic. The output of this step is a sequence of network packets that are sent to the selected preprocessor. Several preprocessors are active in a typical Snort installation.

The active preprocessor receives all input traffic and it is up to the preprocessor if it will perform any computation on the received traffic. Therefore, the first step of the preprocessor is the evaluation on whether the input traffic belongs to the monitored protocol. For example, the HTTP preprocessor performs an analysis and extraction of the HTTP headers but still receives all input traffic. The non-HTTP traffic is recognized as such by the HTTP preprocessor. The FRAG3 preprocessor is responsible for TCP reassembly, which is required to detect evasion type attacks[97].

The output of preprocessors is sent into the *Detection engine*, which is responsible for the matching of Snort rules. The *Detection engine* performs the matching of the content of the transfer against a predefined set of patterns as part of the rule matching. If no rule is matched by the *Detection engine*, the packet is dropped. But if one or more rules match the input packet (or flow in case of TCP reassembly), the event associated with each such rule is generated. The event is logged by the *Event Logger component*.

Every rule written in the Snort language consists of a header and body. The header of the rule describes the IP,port number and protocol number of the traffic that should be tested by the rule. The body of the rule consists of predicates. Every predicate has to be true in order for the rule to be matched. The predicates are evaluated in order from left to right. The predicates are forms of pattern matching. It is possible to specify if the matching should start after the end of a successful match of the previous pattern or if it should be performed against the whole payload. Several types of patterns are supported in the Snort rules. The first type is of predicate is byte comparison. This predicate is true if the specified byte is located on the specified position. The second type of predicate is described by keyword *content* and refers to the string matching. The last and most powerful pattern matching predicate is described by *PCRE* and it refers to the PCRE matching unit. For example, the predicate *pcre='(.*)example\1'* is matched if the payload contains a string *example*, which is predeceased and succeeded by the same string. Snort is able to stop the matching of the rule as soon as it is clear that some of the predicates will never be matched. Therefore, it is important to write the most discriminative and fastest predicates at the start of the body.

The Snort rule 1 describes the situation when the corrupted computer in the protected network tries to send a modified *cmd.exe* file to spread a malware infection and is selected as an example of the snort rule. The rule was divided into several lines in order to increase the legibility of the example. The first line indicates that the rule should be applied to any traffic coming to ports reserved for HTTP protocol in an external network from a protected network. It should be kept in mind that *$HOME_NET*, *$EXTERNAL_PORT* and *$HTTP_PORTS* are variables and their actual value can be set and changed through Snort configuration files. If the rule matches the traffic, an alert message specified on the second line of the example is generated. The third line verifies that the traffic goes to the server in an already established connection. The fourth line contains the first inspection of the payload. The string *malware* has to appear in the HTTP Header field. The *fast_pattern* modifier specifies that this content matching should be performed before any other content matching specified in the rule. The fifth line contains the pcre expression which specifies the vulnerability signature in a more detailed manner. The remaining lines specify additional information about the rule and are not used during the matching.

The example rule contains three predicates which all have to match. The predicates are evaluated from the easiest (flow) to the most time consuming (pcre) and while the first one fails, the processing of all remaining predicates can be stopped. The content matching is performed on the flow level. Moreover, the preprocessor has to identify HTTP headers for the content matching unit.

Snort rule 1 Example of Snort rule

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS
(msg:"BLACKLIST User-Agent known malicious user agent - malware";
 flow:to_server,established;
 content:"malware"; fast_pattern:only; http_header;
 pcre:"/^User-Agent\x3A[^\r\n]*malware/miH";
 metadata:policy balanced-ips drop, policy security-ips drop, service http;
 reference:url, <url to vulnerability description>
 classtype:trojan-activity;
 sid:16551;
 rev:8;)
```

The TCP reassembly is a necessary part of IDS detection. Snort rule 1 demonstrates that the correct function of the IDS requires to understood analysed traffic and to be able to correctly decode application level protocols such as HTTP. This task cannot be achieved without access to the data transferred in all packets of the given flow. Moreover, incorrect TCP reassembly allows a malicious user to hide its attack from the IDS. The Snort can perform the TCP reassembly, which results in a large amount of data being sent into the matching unit all at once, which further increases the amount of requirements on the pattern matching unit.

It is important to realize that even though the purpose of the IDS is to detect the network attack, the IDS itself may be a target of the attack. One of the obvious motivations for the attack on the IDS may be an attempt to mask the attack on the protected network. A possible attack on the IDS is the Denial of Service. If the IDS is overwhelmed by the incoming traffic, then it may miss the packets that contains an attack targeted on other network devices. The IDS is designed to operate on a wire speed. However, Snort, together with many other IDS, is susceptible to algorithm complexity attacks such as [113, 127]. These attacks often target the pattern matching unit because it consumes a large amount of processing resources. The basic idea of these attacks lie in the observation that even if the pattern matching unit operates on a speed in the order of gigabits per second for the average traffic, its worst case performance is significantly lower. It is necessary to specifically craft the malicious traffic with a knowledge of the rule set so as to achieve the worst case performance. The [113] works on the Snort languages, forcing the Snort evaluation engine to perform pattern matching multiple times. It is important to note that the attack is possible only with same set of rules and the quality of the rule may have an effect on the efficiency of the attack. The authors of [113] reported that with carefully crafted input traffic, it is possible to slow down the Snort processing in the order of million and was able to slow Snort to the speed of 4kbps.

The lesson learned from these attacks is that the implementation of the pattern matching has to be able to guarantee the adequate speed of the matching. The pattern which requires us to perform an operation that will slow down the system can be and will be used for the attack on the system and, therefore, such a pattern should be eliminated. It is reasonable for the pattern matching component to refuse to match such patterns.

3.3.2 L7 Filter

L7-filter is a content based packet classifier. Its purpose is to recognize application level protocols by analysing the packet content instead of relying on port numbers. L7-filter contains rules in the form of regular expressions, that are matched against the payload of the network packet. The rules are generated by the community and, therefore, their quality is very diverse.

The L7-filter is implemented as a filter in the IP-Tables[6]. Therefore, it is possible to quickly change its settings and add or remove rules for the new protocols.

The L7-filter contains two matching subsystems, one is referred to as *kernel space* since it runs in the kernel space of the OS, and the second one is called *user space*. The *kernel space* has the advantage of having faster access to the payload of the traffic since there is no need to copy data; but it also has the limitations that it has to run as a kernel module. The *user space* is slower than the kernel space but it allows us to run a more complex regular expression matching since the used matching library does not need to meet requirements of the kernel module. As the result, *kernel space* and *user space* have slightly different semantics of regular expressions and, therefore, every protocol is described by two different types of regular expressions. One regular expression is used in *kernel space* module, while the second one is used in *user space*. Since the *kernel space* and *user space* modules use a different regular expression to match the same protocol, the user space may be faster, even with the disadvantage of the copying payload data. Both types of regular expressions are written with speed in mind. However, while the speed of the matching is sufficient for use in SOHO[2] networks, it is a limiting factor for deployment on faster networks. For the use of L7 in backbone networks, the acceleration of regular expression matching is necessary.

3.4 Requirements of the Pattern Matching Unit

The main purpose of this thesis is to design a new architecture for the pattern matching unit suitable for use in the field of high speed networks together with the algorithms necessary for generation of the configuration of the matching unit. The different types of usage

[2]Small Office Home Office

of the pattern matching unit have different requirements. The previous sections briefly described the environment and type of usage of the pattern matching unit for high speed networks. According to these observations, the pattern matching unit should meet the following criteria.

1. **Speed** – architecture has to be able to match the input stream with the throughput in order of gigabits per second in the worst case. The matching speed has to be independent of the structure of the input stream or it will be susceptible to algorithm complexity attacks.

2. **Flow based** – the Snort preprocessors perform TCP reassembly and, therefore, HW acceleration of the pattern matching has to be able to match on the level of the whole flow to achieve same behaviour as a software implementation.

3. **Fast change of rules** – fast change of the matching rules is required to be able to adapt to the new threads and new attack that appear. While software based tools have enough memory to store thousands of unused rules in the memory, a hardware unit needs to use its memory more effectively. The fast change of rules allows the unit to store only active rules in its memory. Moreover, current IDS stores its rules in configuration files, where they can be easily commented, uncommented or modified. Only the restart of the IDS is required for such a modification to take place. The hardware acceleration unit has to be able to change its rules at least in comparable time.

Chapter 4

Algorithms and Architectures for Fast Regular Expression Matching

Previous chapters described the theory behind regular expressions and its usage in the field of high speed networks. This chapter focuses on the algorithms and architectures used to match regular expressions in the high speed networks. The most currently used approaches are based on finite automata and, therefore, they basically have the same advantages and disadvantages as their theoretical counterparts.

4.1 Architectures Based on Nondeterministic Finite Automata

The first pattern matching architecture based on nondeterministic finite automata was introduced by Ullman in [121, 61]. Every state of automata is transformed into one register and the transition function is realised by the connection between state registers. If the state becomes active during the operation, the register is set. When the state later becomes inactive, the register is reset. The register is set if the previous register was set and the input symbol is equal to the symbol for the given transition. Therefore, for every transition, one logical AND function and one comparator is necessary. The same technique optimized for FPGA was presented by Sidhu and Prasana [110].

The NFA generated into FPGA is created as a union of NFA generated for every rule. Hutchings et al. [58] proposed to share the prefix of these NFA to reduce the number of states in the final automaton. This idea is further extended by Lin in [81]. The paper introduces the reduction of a number of states in NFA by also sharing infixes and suffixes. The modified NFA remembers which state activates a shared component and later uses this knowledge to activate the correct output of the shared part of the automaton. The paper formalizes when it is possible to implement proposed sharing techniques and when not to. If sharing is not possible, the algorithm automatically builda an original NFA. Therefore, there is always a correct matching unit generated. The authors reported a reduction in the number of states up to 70%.

The problem of the architecture is the high resource requirements of the comparators, since every transition in the automaton requires an 8-bit comparator. Clark [33] tackles this problem by introducing the shared alphabet decoder. Every symbol of the alphabet used in the automaton has only one comparator and all transitions used the same comparator. This modification simplifies routing and reduces resource requirements. In [32], Clark extended the shared decoder to process more than 8-bits in one step, which effectively increases the throughput of the matching unit.

Kosar et al. [69] noted that only a reduction in the size of the nondeterministic automaton is done by the synthesis tool during the optimisation phase. Kosar proposed to apply the reduction techniques to reduce the size of the nondeterministic finite automata before constructing the pattern matching unit. The achieved reduction in the number of states of the automaton was up to 66%.

Sourdis et al. [116] proposed the use of shift registers and counters to further reduce the size of implementation of the NFA by an effective implementation of counting constraints. There exists three types of counting constraints. The first type describes *exactly* N times repetition. The use of the shift register primitive to implement this type of counting constraint does not reduce number of states of the NFA, but reduces the number of register primitives used in the FPGA and, therefore, simplifies and speeds up the place and route phase. The second type of counting constraint is *at least* N times. This type of constraint is implemented by the counter and, therefore, instead of using N bits only $log_2 N$ bits is required. The third type is *between* N and M times. This block is implemented by a combination of the shift register and counter.

The paper [34] presents PUREM methodology. PUREM is the extension of the processor pipeline with the regular expression matching engine. The engine is tightly coupled with the processor pipeline which is the main difference against previous approaches. The processor instruction set in extended by instructions controlling the Reconfigurable Function Unit (RFU), which performs partial matching. The processor is responsible for the correct combination of many partial matches produced by RFUs. The proposed implementation accepts 32 bytes of the text data per RFU instruction. The main advantage is that the methodology does not require any changes in the software and supports the PCRE library. The speed of CPU is a limiting factor in the matching. Therefore, the approach is not suitable for a soft core CPU implemented in FPGA.

Wang [122] focus on the possibility of the fast change of regular expressions. The input PCRE expression is parsed and the parsing tree is mapped on the set of CCR[1] and the connection between them. Every CCR can become active if the previous CCR is active and the input symbol meets the conditions associated with the CCR. The CCR can stay active as long as incoming input symbols meet CCR's activity conditions. The CCR then not only reports its activity, but also the number of symbols which it accepted. In fact, every CCR represents a subexpression of the original regular expression. The [122] uses partial dynamic reconfiguration for fast changing of matched regular expressions. The disadvantage of this approach is that the FPGA bitstream generation has to be performed before every change of the matching rules.

Namjoshi and Narlikar [90] describe an extension for the NFA to accept backreferences. The basic idea is to extend the state with additional information. Therefore, instead of a set of active states, the algorithm has a set of configurations. The size of the configuration set is exponential with the number of backreferences and, therefore, the runtime of the algorithm is also exponential. Another contribution of the work is introducing the liveness analysis into the backref-NFA. The authors test if two or more backreferences can be active together and use these results to tighten the time bond on the algorithm and to predict the probability of the DoS attack. The result of this analysis can also be used to reduce memory consumption of the matching algorithm by merging configurations that differ only in dead backreferences.

[1]CCR stands for Character Class with Repetition

The NFA based implementations rely on massive parallelism to achieve a high throughput because the simulation of many possible transitions at once is required. The paper [30] describes iNFAnt architecture, which uses NFA on GPU for regular expression matching. It obtains multi-gigabit throughput by simulating every NFA transition in an independent thread. Therefore, the NFA can achieve the same throughput as a DFA but requires less memory. The authors compare their approach with Hybrid automaton from the regex tool [15] and report superior results. The GPUs are more common than FPGA accelerators. The main disadvantage of GPU based acceleration of NFA is the high latency. The FPGA base accelerators are often situated in the network subsystem, while the GPU based acceleration requires that data are transported through PCIExpress bus into the GPU and then back to the processor.

The main disadvantage of NFA based approaches implemented into FPGA is that the reconfiguration of FPGA is required to change the matching rules. Therefore, such approaches cannot be easily applied in ASIC technology. The FPGA specific tools are necessary to prepare FPGA bitstream. The preparation of the FPGA bitstream can take several hours depending on the complexity of the regular expression set and used mapping. Moreover, the FPGA software is the third party tool and, therefore, licensing requirement may further complicate their usage. The next disadvantage of NFA based approaches is their large state representation. The state of NFA is the combination of the activity of all states. Such state representation can be thousands bits large. Therefore, if the context switching of the unit is required, then all these bits have to be read from the design and stored in the memory. The context switch is necessary if the matching unit is to work on the level of the network flow instead of on the packet level. For these reasons this thesis focuses on the DFA based matching architectures.

4.2 Architecture Based on Deterministic Finite Automata

The main problem of approaches based on deterministic finite automata is the possibility of exponential state blow-up during determinization of the automaton and the large transition table of DFA. While this blow up can ot be prevented in theory, it is often caused by a few specific constructions in the rules. Different approaches tried to identify these specific constructions and introduce extensions into finite automata in order to reduce a state explosion.

The Asynchronous Parallel Finite Automaton (APFA) [77] was introduced in 2009 by Yang Li et al.. The basic idea in this approach comes from observation that most of the state explosion in DFA is caused by length restriction in the regular expressions. Moreover, they argue that most of these length restrictions are used in waiting for the new line character. An example of such a regular expression in the Snort rule set is regular expression 10. In order to solve this problem, the authors divide the matching into two asynchronous parts. The first part is responsible for preprocessing input data, while the second part does the actual matching. The preprocessing part counts the length from the active position to the first occurrence of the \n symbol. When the matching part enters the repetition section it is able to accept or decline a pattern in one step by comparing a given number with the length of the repetition. While APFA effectively solves one particular cause of the growth of DFA, it does not provide a generally applicable solution. Moreover, the authors do not present an actual implementation of the DFA.

The hierarchical automaton [80] is designed in order to reduce a state blow-up caused by the dot star construction at the beginning of the sub-patterns in one regular expression.

Regular expression 10 Regular expression with waiting for new line symbol

```
AUTH\s[^\n]{100}
```

The regular expression 11 is an example of such a regular expression.

Regular expression 11 Regular expression with many dot-star constructions

```
.*AB.*CD|.*MN.*PQ
```

The main idea of the hierarchical automaton is the same as the idea of APFA. The original automaton is divided into two automata, master and slave. The master automaton matches only sub-patterns, which is AB,CD,MN,PQ in the case of regular expression 11 and slave automaton connects these sub-patterns together. The master automaton may be viewed as an alphabet transformation technique transforming the character stream into a stream of sub-patterns. The main disadvantage of this approach is that as sub-patterns become more complex, it is possible, that one or more sub-patterns will overlap. Two sub-patterns overlap if the suffix of the first sub-pattern is the prefix of the second. The hierarchical automaton may produce a false positive in the presence of overlapping sub-patterns. Moreover, the authors of [80] do not provide a universal approach on how to select sub-patterns. In the presented experiments, the authors considered every dot-star construction as a border between two sub-patterns and were able to achieve up to 98% memory savings for the subset of Snort rules.

The absence of the clearly described and generally applicable algorithm for selecting non-overlapping sub-patterns is the main disadvantage of the hierarchical automaton since its efficiency depends solely on the efficiency of the sub-pattern selection.

The Luchaup et al. [83] focused on the increasing of the throughput of the matching unit by accepting more input symbols in one step. Luchaup noted that most of the network traffic does not contain the attack and, therefore, the automaton circles in its starting state most of the time. In order to utilise this knowledge, Luchaup divides the input stream into several chunks and runs the matching for every chunk in parallel with the speculative assumption that the matching automaton should be in its starting state at the beginning of the chunk. If the entire attack string is located in one chunk, the matching will be successful. However, if one attack string spans through more than one chunk, it may be missed.

In order to avoid these types of mistakes, the algorithm produces a sequence of active state visited for every chunk. If processing of previous chunk reveals that the original assumption was wrong and the automaton should not be in the starting state at the beginning of the chunk, the chunk is processed again, this time with the correct active state at the beginning of the processing. The second processing does not need to process the whole chunk, but it may finish when the active state is the same as the active state of the first match.

The importance of speculative matching is that it achieves higher throughput without the increase of the size of the automaton, which is the problem of the approaches using alphabet transformation. The main disadvantage of the approach is the fact that there has to be the same amount of matching unit as the number of simultaneously processed chunks and every unit needs it own access to memory with a transition table. This simultaneous memory access can be solved by having as many independent memories as there are matching units.

Brodie [23] describes a new architecture for the fast regular expression matching together with the algorithm for increasing the number of characters accepted by one transition of the automaton. The algorithm for increasing the number of accepted characters is described in chapter 6 and requires storing two additional bits per transition.

The authors reduced memory requirements by the run length compression of the transition table. For every possible input symbol, there is a sequence of the next state so, that first value corresponds to the first state, the second value to the second state, etc. These sequences are encoded by the run length compression to minimize memory consumption. In order to obtain fast memory access, the transition table is addressed indirectly. Every symbol is used as a pointer to the indirection table, where there is an index of the first and last position of the symbol's sequence. The whole sequence is loaded from the memory into the system and decoded.

The efficiency of the described approach depends on the compactness of the transition table. It is possible to rename states so as to achieve a more compact representation of the state sequences. The main disadvantage of the approach is the necessity to decode the data of a variable size for every input symbol.

The Kumar [71] presented the improvement of the deterministic finite automata for the regular expression matching in high speed networks. The improvement is based on the observation that the high amount of transitions in an automaton is labelled by the same symbol and that they also lead to the same state. The left side of Figure 4.1 shows an example of the described situation. The automaton in the figure is designed in a such way that each symbol always leads to one state.

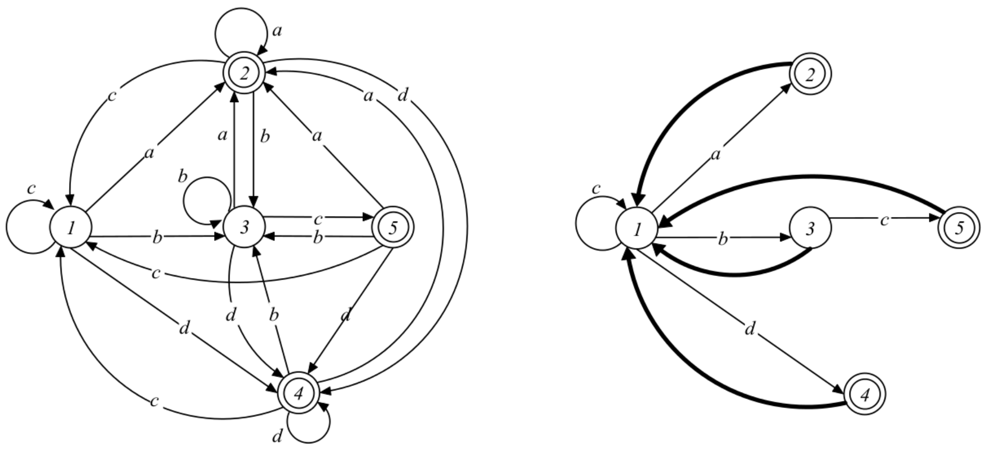


Figure 4.1: Principle of Delay DFA [71]

The Delayed Input DFA (DDFA) introduces transitions without any label called delayed transitions. The right side of Figure 4.1 shows the default transitions in bold. Only state 1 has a transition for every input symbol. Other states simply contain default transition into state 1.

Default transitions may be used only if it is not possible to accept the input symbol in the given state. Therefore, if two states share a transition, it is possible to store this transition only once. One of these states is source state of the transition while the second state has only delayed transition leading into the first one. The use of the delay transition reduces the size of the memory required to store the whole transition table, but it also increases time complexity of the matching process since the delayed transition does not accept the input character.

Kumar's approach for the construction of the DDFA does not allow for cycles of the delayed transitions. This restriction is designed to ensure that the automaton has to accept the input symbol.

The first step in the construction of the DDFA from DFA is to build a space reduction graph. The space reduction graph has the same set of vertices as the DFA. There are edges between all possible combination of states and every edge is labelled by the number of transitions shared by these two states. The delayed transition in the DDFA connect states that are connected by the edges with the highest label in the space reduction graph. In order to achieve maximal compression, the minimal spanning tree of the space reduction graph can be used to indicate the position of all delayed transitions in the DDFA. However, the minimal spanning tree tends to produce long paths of delayed transitions and, therefore, limits the matching throughput. Kumar describes the heuristic approach based on Kruskal's algorithm to solve this problem.

The second solution for the problem of long paths of delayed transition was introduced by Becchi in a paper[16]. The paper defines the term of *depth* of the state. The depth of the state s is defined as the minimal number of states visited when moving from the starting state into state s. The depth of all states in the automaton can be computed with the linear time complexity. The main idea of [16] comes from an observation that if the delayed transition leads to the state with the lower depth, every string of the length N will require at most $2N$ traversals. The limitation of delayed transitions only in the direction towards states with a lower depth produces the space reduction graph that is directed. Moreover, the space reduction graph is also acyclic because the delayed transition cannot connect the states with the same depth. These properties allows for faster construction of the DDFA. The best possible delayed transition for the given state is selected independently for each state. It is even possible to generate the DDFA directly from the NFA and, therefore, completely avoid the construction of the large DFA. The degree of compression of the proposed algorithms is up to 99% according to the measurements published in [16].

The content addressing DDFA (CDDFA), presented in [72], is an improvement of DDFA designed to reduce the worst case time complexity of the matching process. The main disadvantage of DDFA is the need of several memory access in order to accept one character. The [16] were able to reduce time complexity to $2N$ with the same amount of compression as the original approach. However, the worst case throughput of the matching unit based on DDFA is still only half of the throughput of the unit based on DFA. CDDFA achieves the same worst case throughput as the original DFA solution with the memory compression being similar to the DDFA.

The main idea of content addressable DDFA is to label states according to their outgoing transitions instead of by numbers. Therefore, it is possible to decide, if the target state of the given delay transition contains the outgoing transition for the input symbol. If another default transition has to be performed, the name of the state allows us to devise the next target state without any memory lookup. The only limit of the length of default transition string is the available memory. The name of the state has to contain a part of the transition table to allow for described functionality. The size of the name is proportional to the number of outgoing transitions and the length of the maximal path of delayed transitions from a given state.

Kumar reported in [72] that the CDDFA created for the regular expression of Snort IDS requires only 10% of memory required by DFA while maintaining the same speed of the matching.

The basic idea of the approach [123] is the utilisation of the bitmap transition table together with transition table reordering. The authors then compared their approach with the DDFA. The main point of their argument lies in the observation that the DDFA is too complex to be implemented in the hardware and the number of memory access is unpredictable, which has a direct impact on the matching throughput.

The Reorganized and Compact DFA(RCDFA) is designed with respect to the observation that a lot of states in the DFA are similar. However, these similar states are not placed in the same parts of the transition table, which influences the level of compression.

RCDFA works with the full transition table defined as 2 dimensional array. The construction consists of several steps. First, similar states are moved to the same location of the transition table by a sequence of matrix transformation. Then bitmap compression is used to reduce the size of the table.

The proposed hardware implementation is described as a three stage pipeline. According to the authors, the design was able to operate on 156MHz and achieved a throughput of 2.4Gbps per engine. These values suggest some form of the alphabet transformation[2], however, this paper does not discuss it.

The Delta Deterministic Finite Automaton (δDFA) [45] is designed primarily for use in a processor based system. These systems are often equipped with very fast, but small, cache memory and large, but slower, DRAM memory. If the transition table is stored in the fast cache memory, the pattern matching system can obtain a very high throughput. However, if it has to be placed into the DRAM memory, the speed of the matching is drastically reduced. The authors of δDFA propose to use the cache to store only one line of transition table. The stored line should corresponds to the active state of the automaton. Therefore, it is possible to perform a transition by only looking in the fast memory. Since the active state changes, it is necessary to update the line from the slow DRAM memory. The analysis of automata used in high speed networks shows that in many cases only a few values in the transition line have to be changed. The δDFA introduces the data structure supporting such partial updates. The partial updates speed up the reading from the DRAM memory because less amount of data is transferred. The second benefit of this approach is the reduction of the memory consumption of the whole automaton.

While δDFA provides an effective means of reducing memory requirements of automata and still maintains a high throughput, it cannot guarantee that the transition will be finished with only one memory access. The actual number of required access depends on the similarity between transition tables of consequent states. Moreover, data read from the DRAM memory has to be decoded, which requires processing time. The decoding does not present problem for processors because they are typically much faster than the memories, but may have a significant effect in the FPGA based solution.

The efficiency of DFA based architecture depends on the speed of one transition. The speed of the transition is decided by the actual implementation. The many described methods focused on the reduction of the size of the DFA, but did not provide the implementation of it. If the implementation is provided, it often requires several consequential steps to determine the next state, which linearly slows the matching. The number of steps required for the performing transition may depend on the size of the alphabet, such as in [23] or [45].

[2]Alphabet transformation is described in chapter 6 of this work.

If the alphabet transformation is used to increase throughput of the matching, the increase in the size of the automaton's alphabet may present problems and limit scalability of the solution. Therefore, in order for the architecture to guarantee that it can perform every transition in time independent of the size and structure of the automaton, its alphabet or the structure of input stream is necessary. None of the presented architecture meets all of the criteria.

Chapter 5

Hash functions

The hash functions play an important role in modern computer science. Every application has slightly different requirements on the hash functions and, therefore, slightly different approaches are used for every application. This chapter does not try to summarise the whole field of hashing functions. A thorough discussion of hash functions is beyond the scope of this work. The purpose of this chapter is to prepare the reader for the use of hash functions further on in this work and to ensure that all readers and researchers have the same understanding of the meaning of the used terms.

5.1 Universal hash functions

The universal hash functions are used in many areas of computer science for many different purposes. Every purpose has slightly different requirements for the hash functions. In this thesis, the term hash function refers to a function that performs seemingly random mapping from a domain of definition into an image and can be computed in constant time.

Let us suppose that we have set of possible keys S and the output interval M. The key is a bit vector of arbitrary meaning. The hash function is a function that maps S to M. The size of M is usually much smaller than the size of S.

There exists $|S|^{|M|}$ functions mapping S to M. However, not every such function offers good results in specific applications. Since every application has different requirements, the selecting of the correct function to be used as hash function is a complicated problem. The notation of a „good“ hash function is often used. The „good“ hash function is such a hash function that behaves correctly in the given application. The most common requirements for the hash function are:

- **Speed** – the hash value should be computed as fast as possible.
- **Universality** – the distribution of the hash values should be uniform over the whole output interval M
- **Randomness** – the small change in the input should guarantee a large change on the output of the hash function

The randomness property of the hash function is not always required or guaranteed because of a very complicated implementation or because it is not required by the target application. Perfect hash functions 5.5 or semantic hashing [105] are typical examples of these hash functions.

Moreover, users often consider a hash function to be any function that perform mapping from the input domain to output domain. An example of such a situation is the build-in hash function in the python language. Algorithm 4 shows snippet of python code, which is used to compute the hash value of an integer. It can be seen that with one exception, the hash function is an identity. In this work, the hash function is expected to fulfill all these requirements unless it is explicitly stated otherwise.

Algorithm 4 Python hash function for Integer type [8]

```
class int:
    def __hash__(self):
        value = self
        if value == -1:
            value == -2
        return value
```

The definition of universality property 42 is taken from [87]. The application requirements on the hash function are often in collision with each other. Some applications prefer to relax the requirement on randomness or universality for the sake of other requirements, such as the speed of the implementation.

Definition 42. *Universality of hash function: Let c be a positive constant. A family H of function S to M is called c-universal if any two distinct keys collide with a probability of at most c/m, i.e., for all x,y in S with $x \neq y$,*

$$\{h \in H : h(x) = h(y)\} \mid \leq \frac{c}{m}|H|$$

The universality of hash functions is connected with the notation of the collision of hash functions. Definition 43 defines hash collisions. The universality property of a hash function means that every hash value has a similar number of collisions. Therefore, we may be reasonably sure that the given input will not have too many collisions while others would be collision free. This is a welcomed assurance, since we do not generally know the input of the hash functions in advance.

Definition 43. ***Collision of hash function:*** *Let's have hash function H from S to M and values $x, y \in S$. If $H(x) = H(y) \wedge x \neq y$, it is said that x collides with y. The appearance of x and y is called collision of hash function.*

The size of set S is much bigger than the size of the output interval. Therefore, collisions must exist for any given hash function due to the pigeonhole principle. However, in many application not all members of S are used. Suppose, for example, that S is the set of all possible names and the application is the list of phone numbers in the given company. In this case, not all combinations of name and surname appears. Moreover, the company employs only a small part of all living humans and, therefore, only a very small part of S will ever be used in the application. Therefore, even if there are always collisions in general, there may not be collisions for the given set of used keys. A more detailed description of the probability of collisions may be found in section 5.3.

5.2 Algorithms for Computation of Universal Hash Functions

The previous sections discussed the requirements that a good hash function has to meet together with the behavior of a good hash function. This section discusses different ways on how to implement the families of hash functions that meet the requirements from previous sections. There exists many algorithms for the computation of hash functions. Cryptographic hash functions, such as MD-5 [102][1] or SHA [91], are often used instead of the universal hash function because of the belief that they produce a more uniform distribution of outputs. This belief stems from comparing cryptographic hash functions with functions that are called hash functions but do not meet requirements for good hash functions, such as the built-in hash function in python described in algorithm 4. However, the cryptographic hash functions have to meet additional requirements and require more computational resources than most universal hash functions. Therefore, the use of cryptographic hash functions for non-cryptographic applications may significantly harm performance without adding any benefit to the application. This section introduces algorithms for computations of the universal hash function that are fast and meet uniformity and randomness requirements for the hash functions.

Jenkins Hash

The hash function has been developed by Bob Jenkins [60]. The Jenkins hash is famous for its speed[2] on modern CPUs. The hash function has three 32bits state registers whose values are mixed together by a mix function. The mix function is a combination of bit operations, such as bitwise XOR, shifts and rotations. The sequence of operations is specifically designed to allow pipelining on modern CPUs. At the beginning of the hash computation, the state registers are filled by a seed value. Then the registers are updated adding the corresponding part of key and are mixed together. The updating and mixing continues until the whole key is processed. According to the author, Jenkins hash requires $6n + 35$ instructions to compute the hash of the n bytes key.

H3 hash

The H3 hash function, sometimes called universal, was designed by Lawrence Carter and Mark Wegman [28]. The hash function only relies on XOR and logical AND operations. The hash does not support keys with a different length, but it is suitable for hardware implementations. Every output bit of hash function can be computed independently of other output bits. The first logical AND between the key and seed register is computed. All bits of the output of the previous operation are XORed together in order to produce a 1-bit value. Every output bit of H3 hash has its own seed register, which has the same length as the length of keys. Therefore, the memory requirement of the hash is larger than, for example, Jenkins hash, where only three 32 bit registers are necessary. Figure 5.1 shows a diagram of the H3 hash implementation taken from [100].

Due to its easy hardware implementation and relatively good quality of hashing function, H3 hash is used in many hardware applications, such as cache design [106] or transaction memory implementations [107].

[1]MD5 should not be considered safe for cryptographic purposes since successful attacks were published [125, 124]

[2]Faster variants, such as SpookyHash [59], were developed later

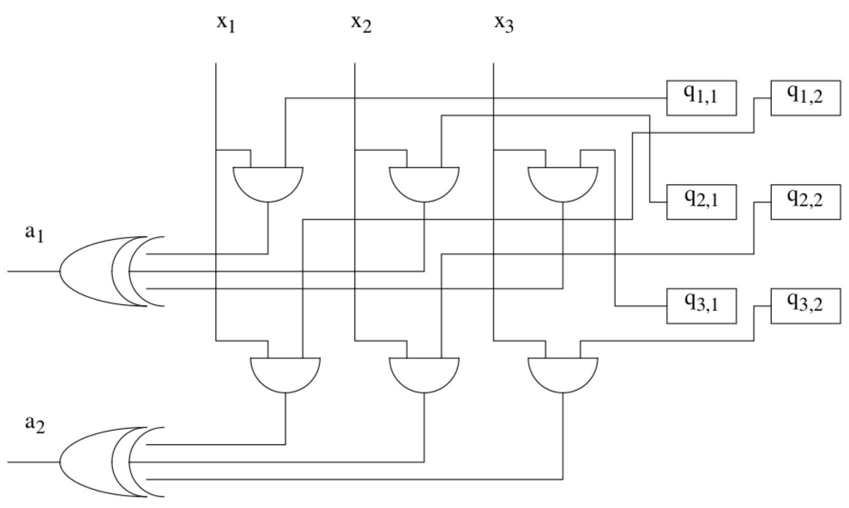


Figure 5.1: Hardware architecture for H3 hash function of the 3-bit input and 2-bit output

Pearson Hash

Peter Pearson [95] developed a hash function for fast hashing of strings with a different length. One of the main requirement for the Pearson hash is to distribute similar strings into distinct integers. The Pearson hash uses table T, which is a permutation of numbers in intervals from 0 to 255 as a seed. During the computation of the hash value, every character of the key is looked-up in table T and found values are XORed together by bitwise XOR. The wrong selection of T values could significantly reduce the quality of the hash function. Pearson states that he experimented with the random selection of T and found no outstanding good or bad arrangements. [95]. The Pearson hash is often used in the 8-bit microcontrollers since it requires only a memory lookup and XOR operation.

All described hash functions are in fact families of hash functions. This means that it is possible to randomly generate new instances of hash functions by random generation of the seed. It is important to keep in mind that an incorrectly generated seed may reduce the quality of the hash function. However, it is always possible to generate a new value of the seed if current the hash function behaves incorrectly.

5.3 Birthday paradox

The birthday paradox demonstrates a situation with an unexpectedly large probability of collisions. It was first formulated in the following way: *let's have twenty people in one room. What is the probability that two people have a birthday at the same day in the assumptions of uniform distributions of birthdays?* Surprisingly, the answer is 41%. The birthday paradox is not an actual paradox since it describes a strictly logical situation. However, the problem is often misunderstood which leads to different results and thus it looks like a paradox. The birthday paradox has important implications for the use of hash functions in modern computer science.

The correct answer to this problem requires an understanding when we ask two people arbitrarily. Therefore, we have to evaluate the probability for every possible pair. In order to know if there are people with the same birthday, we have to ask every person in the room for their date of birth. Let's assume that the date of birth is uniformly distributed through the whole year. As we ask people about their birthday, we will mark days in the calendar as used and we want to compute the probability that all people will have a birthday in the unused day. When we ask the first person, no birthday collision will be discovered since all days in the year are unused. The second person has a very high chance to succeed because only one day is filled. Therefore, the probability that the second person has a birthday on the unused day is $\frac{364}{365}$. The probability, that the next person has a birthday on the unused day decreases as more and more birthdays are marked in the calendar. Equation 5.1 computes the probability that all twenty people will have birthday on an unused day.

$$P = \frac{365}{365} \times \frac{364}{365} \times \cdots \times \frac{346}{365} \tag{5.1}$$

The probability that two people will have a birthday on the same day is the same as the probability that they fail in the previously described experiment. Therefore, the probability that two people have a birthday on the same day is defined by equation 5.2.

$$P = 1 - \left(\frac{365}{365} \times \frac{364}{365} \times \cdots \times \frac{346}{365} \right) P = 0.41 \tag{5.2}$$

The resulting probability that at least two of the randomly selected group of 20 people will have a birthday on the same day is 41%. There is a disproportion between the value of the probability and the ratio between the number of people and the number of days in one year.

The presented approach can be generalized to an arbitrary number of „people" and an arbitrary numbers of „days". Equation 5.3 describe the generalized birthday paradox.

$$P = 1 - \prod_{n=1}^{k} \frac{l - n + 1}{l} \tag{5.3}$$

The value of k is the number of *persons* and l is the number of *days*. The importance of the birthday paradox in computer science is that it describes the behavior of the collisions of the hash functions. In this case, k would be the number of keys for the hash function and l would be the output interval of the hash function.

Mathis [85] formulated a generalized version birthday paradox as stated in definition 44.

Definition 44. ***Generalized birthday paradox:*** *Suppose each member of a population independently receives a number randomly selected from $\{1, 2, 3, \ldots, x\}$, and a random sample of size n is to be taken. If $0 < p < 1$, what is the smallest value of n so that the probability that at least two of the sample have the same number is at least p?*

Mathis provides two approximation methods that allows easy computation of n. If the probability p is set to be 0.5, equation 5.4 provides the approximate value of n for any given x. For more detailed information about equation 5.4 readers should refer to [85].

$$n = 0.57280 + 1.17905\sqrt{x}, \text{ where } x \text{ is the size of interval.} \tag{5.4}$$

Equation 5.4 can be used to establish how many keys can be hashed into the interval of size x so that the probability of collision between any two keys is 50%.

$$x = \left(\frac{n - 0.57280}{1.17905}\right)^2 \tag{5.5}$$

In many cases, it is possible to select the size of the interval x according to the number of keys that are used. Equation 5.5 can be used to establish the size of the interval for the n keys with a probability of collision being 50%. Unfortunately, the equation contains a power of 2 of the number of samples, which indicates that in reality the output of the hash function would have to be prohibitively large for some applications. Therefore, collision is the real problem that has to be dealt with by any algorithm or data structure that relies on hash functions.

5.4 Hash tables

Hash tables are data structures that allow for storing addition data with a set of keys. The use of the hash function allows for quick computation of the position of data from the key and, therefore, reduces the number of memory operations required to read, store or modify data assigned to the key. Therefore, if the user wants to store or retrieve data from the hash table, the hash function has to be computed and the result is used as an address to the memory where data is located.

Unfortunately, as was noted in previous sections, the main problem of hash functions are collisions. This holds true for hash tables as well. One trivial way on how to solve collisions would be to allocate so much memory that the probability of collisions would be negligible. Equation 5.5 can be used to compute the required size of the memory for any given key in the hash table. However, even if we accept 50% probability of collision as negligible, the required memory would have to be in the order of a power of 2 larger than the number of keys. Use of such a large memory would be very ineffective.

The more practical option is to implement a mechanism for dealing with collisions when they occur. There exists many such mechanisms[87]. The chained hashing is an example of this mechanism. The hash table does not contain the value itself, but the linked list of all values that were placed in this position. The disadvantage of this method is the longer time complexity of the lookup operation, as when the input data is searched; the system must read all data in the linked list and check them. In the worst case, all data could be placed into the same linked list and the lookup would take linear time with the number of inserted keys. The universality property of the hash function limits the probability of this situation, but it is still possible if the data is unknown in advance.

The different mechanisms for handling collisions use different strategies to store the colliding data, but in the end all of them have to search through all colliding keys during the lookup operation and, therefore, all of them increase the time complexity of the look-up operation.

Another possibility on how to deal with collisions is to find hash functions which do not have any collision, but finding and maintaining hash functions that do not have collisions for the given set of key is a complex task. Hash functions without collisions for the given set of keys are called perfect hash functions and are discussed in section 5.5. The problem is that if new data is added into the hash table, it is necessary to rebuild the hash table at least in some interval, which may increase time complexity of the insert operation. Another disadvantage of a perfect hash function is the fact that memory requirements of a perfect hash function often depends on the size of the set of keys that do not have collisions.

5.5 Perfect hashing algorithms

The main disadvantage of the hash table is the need for a mechanism to solve collisions of hash functions. This section presents an algorithm for producing hash functions that does not have collisions for the given set of keys S.

Definition 45. ***Perfect hash function:*** *Let's have universe U, set of keys $S \subset U$ and an output interval of integers M. Hash function $f : U \to M$ is called the perfect hash function for S if and only if $\forall x, y \in S : x \neq y \implies f(x) \neq f(y)$.*

According to definition 45, a perfect hash does not have any collision for any two members of set S. If the set S does not change during the computation, the perfect hash for S is called static perfect hash. However, if the application requires for S to change, the perfect hash function for S is called the dynamic perfect hash function.

Another important parameter is the size of the output interval. If the perfect hash has an output interval of the same size as the size of the set S, the perfect hash is called minimal.

The theory of perfect hashes was summarized by Zbigniew Czech, George Havas and Bohdan Majewski in the paper Fundamental Study of Perfect Hashing [38]. This section contains only a brief introduction into the perfect hashing and a description of a few algorithms for generation of perfect hash functions.

The FCH algorithm [46] was one of the first algorithms that is very close to the theoretical lower bound of space requirements for minimal perfect hash functions. The algorithm requires about 2.5 bits per key.

The FCH algorithm is divided into three steps. The first step is called *mapping* and is responsible for transforming the input key to the integer value and assigns every key to its *Bucket.* The *bucket* is simply a set of keys. The mapping step uses three hash functions.

$$h_{10} : S \to \{0, \ldots, n-1\} \text{ where } n = |S| \tag{5.6}$$

$$h_{11} : \{0, \ldots, p_1 - 1\} \to \{0, \ldots, p_2 - 1\} \tag{5.7}$$

$$h_{12} : \{p_1, \ldots, n-1\} \to \{p_2, \ldots, b-1\} \tag{5.8}$$

The S is the set of keys, where a key may be any complex data structure. The h_0 only maps the input key into the integer so other hash functions may process it. Value b is the number of *buckets* that will be used by the perfect hash function. The p_1 and p_2 are parameters that will affect the distribution of keys into *buckets.*

The *mapping* step is defined by equation 5.9.

$$buckets(key) = \begin{cases} h_{11} \circ h_{10} & \text{if } h_{10}(key) < p_1 \\ h_{12} \circ h_{10} & \text{otherwise} \end{cases} \tag{5.9}$$

Equation 5.9 assigns every input key into one bucket. The parameter p_1 is used to decide on how many keys are assigned by the function h_{11}. If the parameter is set to $0.5n$ both hash functions will process exactly the same number of keys. However, output intervals of hash functions are different. The difference is caused by the parameter p_2. According to the [46], p_1 should be set to $0.6n$ and p_2 to $0.3b$. This setting means that 60% of all input keys will be uniformly distributed among 30% of the buckets and the remaining 40% of keys will be uniformly distributed into 70% of all buckets. As a result, there will be few large buckets and a lot of smaller ones.

The second step of the FCH algorithm is called the ordering step and its purpose is just to order the buckets by their size. The third step will process the buckets from the largest one to the smallest one.

The third step of the FCH algorithm constructs table g, which assigns $\log_2(n)$ bits to every bucket and table d, which assigns one bit to every bucket. Equation 5.10 shows how the perfect hash for the key is computed and how the values from g and d are used to avoid collisions.

$$h(key) = (h_{20}(key, d(B_{key})) + g(B_{key})) \mod n \tag{5.10}$$

The value B_{key} used in equation 5.10 is the index of the bucket that contains the given key. The index is computed by equation 5.9. The input key is extended by one bit value stored in table d and the random hash function h_{20} is computed. The result of h_{20} is incremented by the value from g to solve possible collisions with values from other buckets. The operation modulo is used to guarantee the minimalist of the found perfect hash function.

The values of d and g are selected by the state space search. The algorithm starts by randomly selecting the hash function h_{20} in such a way that there are no collisions in any bucket. The collisions between keys from different buckets are solved later in the algorithm by changing the value of g. One of the purposes of d is to change the hash function in a situation, where keys inside one bucket have the collision. The buckets are processed from the largest one. For every bucket, the value of g is selected in such a way that there is no collision with already processed buckets. If there is no possible value of g for the given bucket, the value of g of the previous one is changed. Due to this backtracking, the worst case of the searching step has exponential time complexity. However, experiments on the real data show that the FCH algorithm is able to construct the perfect hash function for the millions of keys in a matter of hours[46, 21].

The first perfect hash algorithm based on random graphs was presented in [37] and is often referred to as the CHM algorithm. The basic idea of the algorithm is that a set of keys $|S|$ is transformed into the graph $G = (V, E)$ by the hash functions. Every edge $e \in E$ of the graph G corresponds to one key in S. Then the algorithm finds function $g : V \to [0, |S| - 1]$, so that function $h : E \to [0, |S| - 1]$ defined by equation 5.11 is bijection.

$$h\big(e = (u, v) \in E\big) = \big(g(u) + g(v)\big) \mod |E| \tag{5.11}$$

Function h assigns an integer to every edge in the graph. The assigned integer is computed as a sum of numbers assigned to the vertices of the edge by function g. The operation modulo only ensures that the integer is not larger than the number of edges. Along with the fact that h is bijection, it means that function h assigns a unique value from interval $[0, |S| - 1]$ to every edge and, therefore, to every member of S.

The suitable function g has to be selected. Unfortunately, a suitable function g may not exist for some graphs. However, if the graph G is acyclic, function g will surely exist and it may be found in linear time with a number of vertices. For every connected component, the algorithm selects one vertex v and assigns value 0 to the selected vertex. Then, the algorithm traverses graph G from vertex v by a depth-first search and assign values to the visited vertices. If the vertex w is visited from vertex u, the value of g is set to $g(w) = \big(h(e) - g(u)\big) \mod |E|$. Since the graph was acyclic, every vertex is visited only once and, therefore, every value of g is correct at the end of the algorithm. Function g assigns an integer value to every vertex of graph G.

It is important now to describe how CHM constructs the graph G for the set of keys S. This problem is solved by two randomly selected hash functions, which will map every key to the edge. Let's have $V = \{0, \ldots, c|S| - 1\}$ and hash functions $h_1 : U \rightarrow V$ and $h_2 : U \rightarrow V$, where U is the universe of all possible keys and $S \subset U$. Then edges of graph $G(V, E)$ are defined by the equation 5.12.

$$E = \{(h_1(x), h_2(x))|x \in S\} \tag{5.12}$$

Graph G is considered a random graph because the hash functions h_1 and h_2 are randomly selected. The algorithm does not guarantee that the graph G is acyclic. However, as long as $|V|$ is at least $2 \times |S|$, there is a nonzero constant probability that the random graph will be acyclic. Therefore, if the random graph G contain cycle, the new hash functions are selected until the acyclic graph is found. According to the [37], if $|V| = 3 \times |S|$, the expected number of iteration required to find acyclic graph is $\sqrt{3}$. The probability that the random graph is acyclic depends on the ration between $|V|$ and $|S|$ and is selected by parameter c.

The memory consumption of the CHM algorithm is much larger than the FCH algorithm. The CHM requires $c \times \log_2 |S|$ bits per key. However, the CHM algorithm finds the perfect hash function much faster than FCH.

The CHM algorithm was improved by Botelho in [20] and published as the BMZ algorithm. The main difference was that the BMZ algorithm allows us to have a small number of cycles in the random graphs and, therefore, parameter c may be smaller. This can reduce the amount of required memory, but the amount of bits per key still depends on the size of S.

In [21], Botelho published a new algorithm based on the random hyper graph which reduces the memory requirements to about 2 bits per key, which is approximately the same result as FCH. The algorithm is called BDZ. The basic idea behind BDZ is the same as in CHM with the exception that the edge in the hyper graph is not assigned to the result of the PHF, but to the one hash function. The output of the selected hash function is used as an output of the perfect hash function.

The algorithm consists of two steps. The first step is called the *mapping* and the second step is called the *assigning* step. If the minimal perfect hash function is required, the third step called *ranking* is necessary. The mapping step constructs the random acyclic hyper graph G and list L containing the sequence of edges in the correct order for the *assigning* step. The assigning step takes the list L as an input and produces table g according to equation 5.13 to meet the requirements of the perfect hash function.

$$PHF(key) = \begin{cases} h_1(k), (g[h_1(k)] + g[h_2(k)] + g[h_3(k)]) & \mod 3 = 0 \\ h_2(k), (g[h_1(k)] + g[h_2(k)] + g[h_3(k)]) & \mod 3 = 1 \\ h_3(k), (g[h_1(k)] + g[h_2(k)] + g[h_3(k)]) & \mod 3 = 2 \end{cases} \tag{5.13}$$

The mapping step has the same principle as the mapping step of CHM. The probability that the given random hyper graph is acyclic depends on the ratio between $|V|$ and $|S|$ and the number of hash functions.

The theorem *6.5* from [38] provide the equation 5.14 to compute the smallest possible c_r, so that if $|V| = c_r \times |S|$, the probability that the random hyper graph is acyclic is a nonzero constant.

$$c_r = \begin{cases} 2+\epsilon, \epsilon > 0 & \text{for } r = 2 \\ r\left(\max\limits_{y>0}\left\{\dfrac{y}{(1-e^y)^{r-1}}\right\}\right)^{-1} & \text{for } r > 2 \end{cases} \tag{5.14}$$

The evaluation of equation 5.14 shows that setting r to 3 allows us to construct smallest hyper graph. The actual value of c_3 is around 1.23. The authors of [21] noted that there is a phase transition of probability that the random hyper graph is acyclic for the $r > 2$, while for $r = 2$ the probability changes continuously. While it was sensible to use a value larger than c_2 to increase the probability of the acyclicity of random hyper graph for $r = 2$, using higher values than c_r for $r > 2$ do not have any significant effect on the probability. The probability that a random graph is acyclic is near 1 if $|V| = c_r \times |E|$.

The second output of the *mapping* step is list L of edges ordered in such way that edge e_i contains vertex that is not incident with any edge e_j for $j > i$. List L can be generated during the test of the hyper graph acyclicity with time complexity O(n).

The *assigning* step generates table g from list L. Firstly, an algorithm initializes table g in such way that $\forall i \in V : g_i = r$ and creates array $Visited$ so, that $\forall i \in V : Visited_i = False$. The algorithm traverses all edges $e \in L$ from the tail to the head and selects vertex $u \in e$, so that $Visited_u = False$. Let j be the index of u in e or informally, let j be the index of hash function whose output points to the vertex u for the processed edge e. The value of g_u is computed by equation 5.15. The $Visited_u$ is set to *true* for every u incident with the edge e.

$$g_u = \left(j - \sum_{v \in e \wedge Visited_v = True} g_v\right) \mod r \tag{5.15}$$

Created data structure allows for computing a perfect hash function in constant time. The memory requirements are defined by table g. For $r = 3$, g is 1.23 times larger than a number of keys, which means that 2.46 bits have to be stored for each key. The [21] used compression techniques and different encodings in order to reduce the memory requirements and the authors were able to obtain memory consumption around 1.95 bits per key for the perfect hash function.

The construction of a minimal perfect hash function requires an additional *ranking* step and disallows some optimization used to reduce memory requirements. The basic idea in the construction of minimal perfect hash function from minimal hash function stems from the observation that table g has value 3 for every unused vertex. Therefore, it is possible to count the number of used vertices that are before the unique vertex found by equation 5.13 and return this count as a result. In order to achieve constant time complexity, additional data structure is necessary to speed up the counting of all previous *used* vertices. For more detailed information about this step, see [21].

The algorithm presented in [21] is simple to evaluate because the evaluation requires only three memory lookups and simple arithmetic operations sum and modulo. In [64] we proposed to change addition in equation 5.13 to XOR. This change allows us to remove the modulo operation since it is possible to set values of g in such way that they never exceed $r - 1$. The motivation for this change is to allow for faster implementation in FPGA. This thesis uses the variant used in [64].

5.6 Set membership problem

Previous sections introduced hash tables, which can be used to store values assigned to the given set of keys. However, due to possibilities of hash collisions, it is necessary to verify if the given input belongs to the set of keys that are contained in the hash table. This is one of the real world applications for the set membership problem discussed in this section.

Brodnik [24] defines the set membership problem in following way. Given that universal set $M = \{0, \dots, m-1\}$ and any subset $N = \{e_1, \dots, e_n\}$ the membership problem is to determine whether the given query element in M is an element of N. The two trivial methods can be used to solve this problem. The first is the use of the list of all elements of N. For example, the hash table can be used to store the members of N. The second method is to store all possible answers into the bitvector. For every member of M, there will be exactly one bit, which will be set to 1 if the corresponding member of M is in N or 0 instead.

Fredman, Komlós and Szemerédi [47] presented an approach base on hash table. Yao in [126] argues that hash tables have linear time complexity in the worst case due to collisions. Therefore, Yao focuses on the use of sorted tables instead of the hash table.

The Buhrman in [27] states that the information theoretic lower bound on the number of bits used for storing sets of size at most n is $\lceil \log_2 \left(\sum_{i \le n} \binom{m}{i} \right) \rceil$ which indicates that the data structure has to store $\Omega\left(n \log_2 m\right)$ bits. Brodnik [24] states that if $n \ll m$, the hash table with perfect hashing meets the theoretical bound while allowing queries in constant time.

Buhrman [27] and Ta-Shma [118] add the requirements that one memory access can read only one bit of information. Reading only one bit could simplify the hardware implementation of the problem. However, papers deal with theoretical analysis and only prove the existence of effective algorithms.

We can reduce memory requirements if we permit the errors set membership problem.

Bloom filters

The bloom filter is the probabilistic data structure used to represent sets. Bloom filters reduce the memory requirement by allowing for a small portion of false positives results in the look-up operation. Bloom filters were introduced in [18] and are currently used in many real world applications. The basic idea behind bloom filter is to have one memory array A and k universal hash functions $h_k : N \rightarrow \{0, \dots, |A| - 1\}$, which maps any possible key to the index into array A. The bloom filter allows us to add new members into the set during its operation. At the beginning of the operation, the bloom filter does not contain any members and all values in A are set to 0. When the new member is to be added into the set represented by the bloom filter, array A is updated according equation 5.16.

$$\forall i < k : A_i = A_i \vee 1 \tag{5.16}$$

Equation 5.16 states that to add a new member into the bloom filter, the new member has to be hashed by all k hash functions and the output of every hash function is used as an index into A. The value of A at the position given by the hash function is set to 1.

In order to answer a membership query, the bloom filter hashes the query by all hash functions and checks that the value of A is set at all positions. If at least one hash function points to zero in A, the query is not in the set represented by the bloom filter. Since the bloom filter does not store keys itself, but only a bit at the position specified by a randomly selected hash function, it is possible that the key that is not a member of the set represented by the bloom filter has set bits at all required positions. This situation is referred to as a false positive because the bloom filter incorrectly answers the membership query and states that the key belongs to the set.

The probability of a false positive depends on the size of array A and the number of hash functions k and the size of the set S represented by the bloom filter. Increasing the size of A decreases the probability of a false positive since the output range of hash functions is larger and, therefore, probability of collisions is lower. The number of hash functions k is more interesting. The larger k decreases the probability of the false positive since more positions are checked, but it increases the probability of a false positive by setting more bits during adding a new member into set S. Increasing the size of set S increases the probability of a false positive because more bits in A are set during a membership query.

Equation 5.17 computes the probability that a specific bit is still zero after adding n-th element.

$$p = \left(1 - \frac{1}{|A|}\right)^{kn} \approx e^{\frac{-kn}{|A|}} \tag{5.17}$$

The probability of a false positive for a random key is the probability that k randomly selected positions in A are set to 1 and can be computed by equation 5.18.

$$f = (1 - p)^k \tag{5.18}$$

The [22] contained a detailed analysis of the bloom filter theory together with some of its extensions.

Imperfect Hashing

The basic idea behind imperfect hashing is that the hash table will not store keys themselves, but only the hash value of the key. The number of bits used to store hash values is typically smaller than the number of bits required for storing an original key. However, since the hash function is used for a reducing the number of bits, there exists a possibility of collisions. The collision will result in a false positive error, since the key K_1 that is not in the hash table will have the same hash value as a key K_2 that already is in the hash table and, therefore, during lookup operation K_1 will be considered to be in the hash table.

The approach was first analyzed by [29]. The probability of a false positive depends on the number of bits used to store the hash values. If r bits is used, the probability of the false positive is 2^{-r}. It is important to realize that two hash functions are used. One hash function decides the position of „key“ in the hash function and, therefore, it randomly selects the hash value that the second hash function has to have. Since the second hash function has to collide with the previously selected value, the probability of a collision is the probability, that the hash function will return one specified value, which is 2^{-r}.

Chapter 6

Alphabet transformation

The purpose of this chapter is to show benefits of the alphabet transformation of the automaton and how it can be used to increase throughput or reduce memory requirements of automata. It will be shown further in this chapter that using the alphabet transformation to increase throughput of the matching process increases the size of the alphabet. However, the other techniques can be used for reduction of the size of the resulting alphabet to achieve both purposes.

Alphabet transformation methods divide pattern matching process into two computation steps. First step consists of the actual alphabet transformation. Second step is the simulation of the finite automaton that is done exactly in the same way as without alphabet transformation but on different alphabet.

Basic principles of these methods can be seen on the figure 6.1. The process of the alphabet transformation is independent on the finite automaton performing the actual matching process. Therefore it is possible to implement these processes as loosely coupled.

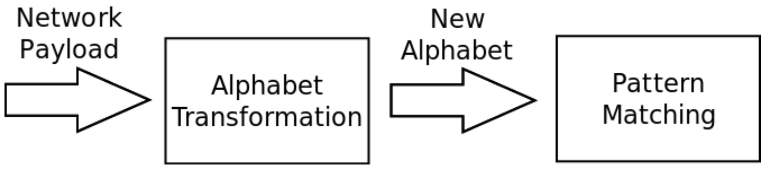


Figure 6.1: Basic principles of alphabet transformation

The previous work described several ad-hoc algorithms for the alphabet transformation but did not formally define the alphabet transformation itself or the properties of the generated alphabets.

6.1 Character class

The regular expressions can contain the character classes to simplify the notation. There are basically two methods to deal with character classes during the construction of the finite automata. The first method generates unique transition for every symbol in character class which is fairly simple but discards the positive effect of the character class. Second method labels the transition by character class instead of one symbol. The transition is performed if and only if the input symbol belong to the character class describing the transition. This method leads to the automaton with different alphabet than the alphabet of the input string, because while the input string consists of symbols the alphabet of the automaton consists of the set of symbols. Therefore it is necessary to perform alphabet transformation from the alphabet of the input string into the alphabet of the automaton.

Lets have an automaton $M(Q, \Sigma_{class}, \delta, s, F)$ and alphabet Σ, such that $\Sigma_{class} \subseteq 2^{\Sigma}$. The given problem is, whenever string $x \in \Sigma^*$ belongs to the language accepted by the automaton M. The alphabet transformation is the mapping $T : \Sigma^* \rightarrow \Sigma_{class}$ such that $T(s) = \{s_c | s_c \in \Sigma_{class} \wedge s \in s_c\}$ and every string from Σ^* has to be describable by the sequence of symbols from Σ_{class}. For character class, the mapping $T(s)$ maps every character of Σ into one or more character classes from Σ_{class}. The main problem of this alphabet transformation is the fact, that one symbol of Σ can possibly belong to several symbols of Σ_{class}. The figure 6.2 demonstrates such situation.

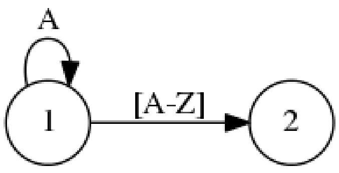


Figure 6.2: Overlapping symbols

If the automaton M is not deterministic, the same approach which handles nondeterminism in the automaton can be used to *select* the correct symbol from the $T(s)$. It may be backtracking or all symbols may be processed in parallel. However, if the automaton M is deterministic, the possibility of several input symbols presents large problem from the point of the time complexity of the matching. The DFA guarantees that every symbol is processed in the constant time. However, if the symbol from the input string is transformed into n possible symbols in the automaton's alphabet, the processing of these symbols will require at least n operations to determine, which of the possible symbol is the correct one. Another possible solution is to thread the deterministic automaton same as nondeterministic and allow existence of more active state. The alphabet transformation can significantly increase time complexity of the matching. Therefore, to guarantee the time complexity of the deterministic finite automata, it is required that every symbol is transformed into at most one symbol of the Σ_{class}.

Definition 46. ***Deterministic alphabet:*** *Lets have alphabets Σ and Σ_{class} and alphabet transformation $T : \Sigma^* \rightarrow \Sigma_{class}$. The Σ_{class} is deterministic alphabet if following condition holds*

$$\forall s \in Dom(T) : |T(s)| \leq 1$$

Definition 47. ***Overlapping symbols:*** *Lets have alphabets Σ and Σ_{class} and alphabet transformation $T : \Sigma^* \rightarrow \Sigma_{class}$. Than let $x, y \in \Sigma_{class}$ be different symbols. The x and y are called overlapping if following condition holds*

$$\exists s \in \Sigma^* : x \in T(s) \wedge y \in T(s)$$

Overlapping symbols are symbols that causes problems during the alphabet transformation because system is not able to deterministicaly decide which one of them should be returned. However, if additional information is available during the time of alphabet transformation it may be possible to determine the correct transformation. For example, it is possible, that all overlapping symbols describe transitions from different states in the automaton. Than it would be possible to select the correct symbol with the knowledge of the active state of the automaton.

Definition 48. ***1-deterministic alphabet:*** *Lets have alphabets Σ and Σ_{class} and alphabet transformation $T : \Sigma^* \to \Sigma_{class}$ and automaton $M = (Q, \Sigma_{class}, \delta, s, F)$. The alphabet Σ_{class} is 1-deterministic for the automaton M if and only if following condition holds*

$$\nexists k \in \Sigma^* : \exists s \in Q : \exists x, y \in \Sigma_{class} : x \in T(k) \wedge y \in T(k) \wedge (s, x) \in Dom(\delta) \wedge (s, y) \in Dom(\delta)$$

According to definition 48, the 1-deterministic alphabet for automaton M is the alphabet without overlapping symbols or with overlapping symbols that do not belong to the label of transitions outgoing from the same state of the automaton M. Therefore, it is possible, with the knowledge of the active state of the automata, always decide which symbol from Σ_{class} should be the result of the transformation to allow correct matching. It is obvious that every deterministic alphabet is also 1-deterministic for every automaton M. However, the 1-deterministic alphabet do not have to be deterministic. Moreover, the 1-deterministic alphabet for the automaton M_1 do not have to be 1-deterministic alphabet for the automaton M_2.

The importance of the 1-deterministic alphabet is that it is possible to achieve higher reduction of the number of transitions in the automaton than with purely deterministic alphabet. The 1-deterministic alphabet may be smaller than deterministic alphabet, because it can contain overlapping symbols as long as they are not used in the same state. This condition is more restrictive for the alphabet transformation because the alphabet transformation have to take active state into account. However, the process of alphabet transformation can transform one symbol in advance and effective implementation is still possible. The notation of 1-deterministic alphabet can be generalised to the k-deterministic alphabet in this sense to describe this situations.

Regular expressions currently used by the intrusion detection systems contain many character classes. However, these character classes are used mainly for the simplification of the creating the regular expressions and may not be optimal for the construction of the automaton. Therefore, this section describes few simple algorithms that are used to cope with character classes written by authors of the rules and produces automata with character classes that are more suitable for the matching process. It may be necessary to expand all character classes from the automaton to normalize it before using some algorithm to build the specific type of character classes that we are interested. The algorithm 5 is responsible for the removing character classes from the transition function of the automaton.

There are many different methods how to compute character classes for the given automaton. However, different methods produce different character classes. To be able to select the best method, it is necessary to have some techniques for comparison of different character classes. Since the main reason for introduction of character classes is the reduction of the number of transitions, we will prefer such alphabet transformations that allow construction of smaller automata.

Definition 49. ***Ordering of alphabet transformations:*** *Lets have automaton $M = (Q, \Sigma, \delta, s, F)$ and Σ, Σ_1 and Σ_2 are alphabets, such that there exists alphabet transformations $T_1 : \Sigma^* \to \Sigma_1$ and $T_2 : \Sigma^* \to \Sigma_2$. If there exists transition minimal automata $M_1 = (Q, \Sigma_1, \delta_1, s, F)$ and $M_2 = (Q, \Sigma_2, \delta_1, s, F)$, such that $L(M) = L(M_1) = L(M_2)$, we say that*

$$T_1 <_M T_2 \iff |M_1|_T < |M_2|_T$$

Algorithm 5 Expanding all character class

```
$\delta_{new} = \emptyset$
**for** $q \in Q$ **do**
    **for** $s_{class} \in \Sigma_{class}$ **do**
        **for** $s \in s_{class}$ **do**
            $\delta_{new} \leftarrow \delta_{new} \cup (q, s, \delta(q, s_{class}))$
        **end for**
    **end for**
**end for**
return $\delta_{new}$
```

It is possible to define the optimal alphabet transformation. Informally, the optimal alphabet transformation is such alphabet transformation, that allows to create the automaton as small as possible.

Definition 50. ***Minimality of alphabet transformation:*** *Lets have the automaton $M = (Q, \Sigma, \delta, s, F)$ and alphabet $\Sigma_{minimal}$ with the alphabet transformation $T_{minimal} : \Sigma^* \to \Sigma_{minimal}$. The alphabet transformation $T_{minimal}$ is called optimal if and only if following condition holds:*

$$\forall \Sigma_{class} \subseteq 2^{\Sigma} : \nexists T_{class} : T_{class} <_M T_{minimal}$$

The minimal alphabet transformation from 50 may be subjected by other conditions to allows better implementation. For example we may require minimal alphabet transformation generating deterministic character class alphabet.

The algorithm 6 generates the largest possible character classes to allow maximal reduction of transitions in the automaton without changing the number of states. The alphabet created by this algorithm is denoted by the optimal nondeterministic alphabet in the rest of this work because the algorithm does not guarantee that the resulting alphabet is deterministic. The resulting automaton will have as little transition as possible without changing the number of states.

As was discussed in earlier paragraphs, the nondeterministic alphabet transformation increases the time complexity of the matching process by introducing another source of nondeterminism. The algorithm 7 generates the alphabet whose transformation can be done in deterministic fashion.

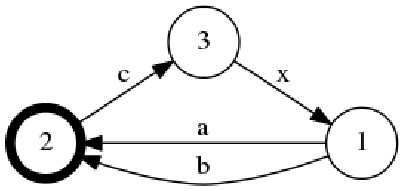
The first step in the algorithm 7 is to find mapping *Sym* which maps every symbol from the input alphabet into the set of transitions that are labeled by this symbol. in this case, every transition is defined by tuple of states. The second step of the algorithm is to define the relation R between symbols of the alphabet Σ. Two symbols are in the relation R if they are mapped by mapping *Sym* to the same set of transitions. In other words, two symbols are in relation R if and only if they label the same transitions. The relation R is equivalence relation. Therefore it is possible to use it to define the equivalence classes on the alphabet Σ. Each of these equivalence classes corresponds to one symbol of the alphabet of the new automaton. The last step of the algorithm consists of two nested for loops to ensure that every transition from the original automaton is transfered into the transition table of the new one. The description of the algorithm defines the transition function as an

Algorithm 6 Generation of optimal nondeterministic alphabet transformation

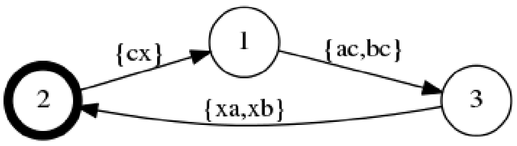
```
Input: Finite automaton $M = (Q, \Sigma, \delta, s, F)$
Output: Finite automaton $M = (Q, \Sigma_{class}, \delta_{class}, s, F)$
  $\Sigma_{class} \leftarrow \emptyset$
  $\delta_{class} \leftarrow \emptyset$
  for $s \in Q$ do
      for $t \in Q$ do
          $Class \leftarrow \{c | c \in \Sigma \wedge \delta(s, c) = t\}$
          $\Sigma_{class} \leftarrow \Sigma_{class} \cup \{Class\}$
          if $Class \neq \emptyset$ then
              $\delta_{class} \leftarrow \delta_{class} \cup (s, Class, t)$
          end if
      end for
  end for
```

Algorithm 7 Generation of optimal deterministic alphabet transformation

```
Input: Finite automaton $M = (Q, \Sigma, \delta, s, F)$
Output: Finite automaton $M = (Q, \Sigma_{class}, \delta_{class}, s, F)$
  $\forall s \in \Sigma : Sym(s) = \{(q_1, q_2) | q_1 \in Q \wedge q_2 \in Q \wedge q_2 = \delta(q_1, s)\}$
  Find relation $R \subset \Sigma \times \Sigma$ such $(s_1, s_2) \in R \leftrightarrow Sym(s_1) = Sym(s_2)$
  Compute quotient set of $\Sigma$ by $R$: $\Sigma_{class} = \Sigma / R$
  $\delta_{class} \leftarrow \emptyset$
  for $q \in Q$ do
      for $s \in \Sigma$ do
          Find $s_{class}$ such that $s_{class} \in \Sigma_{class} \wedge s \in s_{class}$
          $\delta_{class} \leftarrow \delta_{class} \cup (q, s_{class}, \delta(q, s))$
      end for
  end for
```



(a) Original deterministic finite automaton



(b) 2-strided version of the left automaton

relation δ_{class}. And since the relation is the set of tuples all transitions will be represented once, even if they are added into the relation many times. Actually, every transition will be added into the transition table as many times as is the size of the character class that describes this transition.

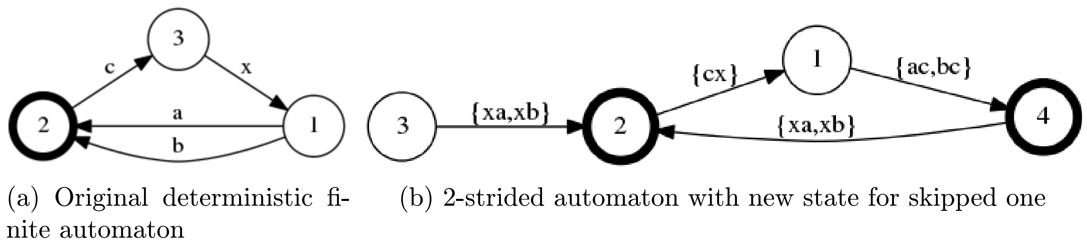
6.2 Multistriding Methods

The multistriding is the method used to increase throughput of the pattern matching by accepting several symbols at once. One of the first widespread approach was published by Brodie in [23]. The input of this method is deterministic finite automaton without character classes. The core of the multistriding is method called doublestride, which is able to concatenate two successive transitions in the automaton and produce one which will accept two symbols at once. The doublestride method can be applied as many times as necessary and each time it is used, the throughput of the automaton is double. However, the size of the alphabet increases exponentially while the number of symbols accepted by one transition increases linearly. To limit the exponential increase in the size of the alphabet [23] search equivalence classes over the multistrided symbols. Therefore, resulting symbol is set of strings.

Since the multistrided symbol accepts several characters at once, the transition has to skip over states in original automaton. Figures 6.3a and 6.3b demonstrate this situation. Suppose, that state *1* is active state and string *ac* is at the input. The automaton from the figure 6.3a accepts the input by moving from state *1* to state *2* and than into state *3*. However, the automaton from figure 6.3b moves immediately from state *1* into state *3* by accepting both symbols at once. The state *2* was skipped. This skipping do not present any problem in most cases. However, if the skipped state is final state, automaton fail to accept given string and continues the matching process. To deal with this problem, [23] stores additional two bits with every transitions. These bits represent information if the transition skipped final state or starting state and uses these bits to detect whenever given string belongs to the language of the automaton.

The main disadvantage of this approach is the need to store addition information with transition, which may significantly increase the memory requirements since the number of transition in the automaton can be very high. The multistriding method described in the [23] always doubles the number of accepted symbols. However, while accepting 4 symbols per step may not be enough for required throughput, accepting 8 symbols per step may require too much memory. This constrain is second disadvantage of [23].

Another of the most commonly used multistriding method was proposed by the Michela Becchi in [17]. The basic idea of this approach is a concatenation of the two following symbols to produce two-character alphabet. This concatenation is called recursively until a specified number of character per symbol is obtained. It can be seen, that this approach



(a) Original deterministic finite automaton (b) 2-strided automaton with new state for skipped one

Figure 6.4: The duplication of final state in multistriding

can produce only symbols of the size equal to power of two.

It is important to keep in mind, that the accepting more than one character per transition leads to the *skipping* states as was demonstrated in the example of the previous approach. The main difference of Becchi's approach against Broodi's is the handling of the skipped states. The paper [17] presents two algorithms. One for NFA and other for DFA. The main difference between these two algorithms is the way they handle final states. The NFA approaches just add transition leading to the skipped final state. However, the DFA approach generates the copy of the final state, which contains the same outgoing transition as the original target of the transition. It is important to keep in mind, that [17] uses delayed transition to compress the transition table. The exact method used to build default transitions was described in [16]. Since the new state has the same outgoing transition as another state in the automaton, all its outgoing transition can be replaced by one default transition. The example of duplication of the final state is shown at the figure 6.4. The final state *3* in the original DFA is skipped in the multistrided version. Therefore, multistrided version contain new state 4, which has same outgoing transitions as a state *3*, but is final. All transitions that are skipping a final state and lead should lead into state *3* will end in the new state *4*. Transitions, that did not skip any final state will end in the original state *3*. The presented example do not have any transition into state *3* that do not skip final state and therefore state *3* is inaccessible and could be removed from the automaton. Unfortunately, this is not the case in larger and more complex automata.

The handling of the exponential increase of the size of the alphabet of the automaton is the second difference between [23] and [17]. Becchi introduces character classes into the original automaton before the multistriding. After the multistriding, automaton is immediately transformed into DelayDFA form described in [16]. The last step of the process is alphabet reduction to produce new character classes. If more than two characters per step needs to be accepted, the output of the last alphabet reduction is used as an input for the whole process. These compression and reduction techniques allows to produce more compact automaton than approach described in [23]. Authors noted, that without these reductions, it would not be possible to build 4-striding automaton with 2K states on the computer with 4GB RAM.

The symbols generated by combination of several alphabet reduction steps together with multistriding are in form of recursive data structure, where each layer of recursion is set of 2-tuples of sets. Of course, this complex structure can be easily transformed into the set of strings.

The main disadvantage of this approach is the same as in [23]. The multistride automaton accept 2^k symbols per step, which do not allow finer tuning of trade of between throughput and memory requirements. Authors of [17] noted, that generating 4-stride automaton is impractical because of the size of resulting automata.

6.3 Expression Automaton

The previous section introduced concept of the alphabet transformation into the pattern matching. The main advantage of the approach is the minimal interaction between the automaton and the alphabet decoder. Every algorithm for pattern matching can use alphabet transformation methods to increase throughput of the matching at the cost of the increasing size of the alphabet. Unfortunately, the alphabet grows too quickly for practically used pattern sets.

Basic concept of expression automaton of generalized automaton was first introduced by Brzozovski and McCluskey in [25] to construct regular expression from state diagrams. Brzozovski and McCluckey iteratively removed states from the state diagram and connected the edges. The new edge was labeled by the regular expression which was combination of the expressions on the connected edges. They obtained state diagram with only two states and one edge labeled by the regular expression equivalent to the original state diagram.

In this work, we define Expression Automaton in the same way as [117].

Definition 51. *The expression automaton (EA) is 5-tuple $M(Q, \Sigma, \delta, s, f)$, where*

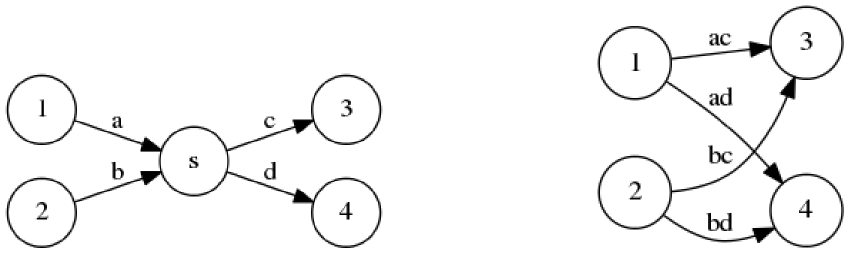
- *Q is a finite set of states*
- *Σ is an input alphabet such as $\Sigma \cap Q = \emptyset$*
- *$\delta \subseteq Q \times R_\Sigma \times Q$ is a finite set of transitions, where R_Σ is a set of all regular expression over Σ*
- *$s \in Q$ is the start state*
- *$F \in Q$ is the final states.*

The basic idea is that the Expression Automaton has transitions labeled by regular expression over the alphabet Σ. Expression Automaton can be used as a basic model for the multistriding regular expression matching unit. The regular expression describes set of strings that may appear in the input string. Each of these strings may consists of several characters and all its character are going to be accepted by the one transition of the Expression Automaton.

However, most implementation of the regular expression matching unit have some requirements over labels of transitions. To model these units, we have to define additional constrains over the language R_Σ. Therefore, we may say that every symbol is a string that belongs the language L, where $L \subset R_\Sigma$ while L has to comply with additional constraints given by the method. For example, in the approach presented in [23], the $L \subset \{s | s \in \Sigma^* \wedge |s| = 2^n\}$, where n reflects how many times was the multistriding applicate.

Generalised automaton was first introduced in [43] and it allows only strings as transitions labels. However, the length of the string is not limited. Therefore, the actual throughput of the generalised automaton depends on the input data. Giammarresi and Montalbano [51] defined Deterministic Generalised Automaton (DGA) as an generalised automaton, for which all outgoing symbol in every state are prefix free set[1]. The paper introduce superfluous states, that can be removed in order to create transitions labeled by strings.

[1]The original paper calls prefix free set just prefix set



(a) Original automaton before reduction (b) The new automaton after removing superfluous state s

Figure 6.5: Removing superfluous state

Definition 52. ***Superfluous states:*** *Let $M(Q, \Sigma, \delta, s, F)$ be DGA. The state $q \in Q$ is a superfluous state for M if following condition holds:*

$$q \notin F \wedge q \neq s \wedge \nexists x \in \Sigma : (q, x, q) \in \delta$$

Informally, the superfluous state can not be initial or final state of the automata and cannot have self loops. Therefore, the superfluous state can be removed from automaton and all its incoming transition can be connected with all of its outgoing transitions to create new transition that accepts concatenation of labels of original transitions. The process of removing superfluous state is demonstrated on the figure 6.5.

Definition 53. ***S-reduction:*** *Let $M(Q, \Sigma, \delta, s, F)$ be a DGA and $q \in Q$ be a superfluous state. Then $S(M, q) = (Q_q, \Sigma, \delta_q, s, F)$ is a generalized automaton where $Q_q = Q - \{q\}$ and $(r, u, v) \in \delta_q$ if $(r, u, v) \in \delta$ or $\exists(r, u_1, q) \in \delta, (q, u_2, v) \in \delta : u_1u_2 = u$*

The paper [51] proves, that application of S-reduction do not change the language accepted by the automaton.

Definition 54. ***Irreducible DGA:*** *DGA is irreducible if it do no contain any superfluous or indistinguishable states.*

To build irreducible DGA one has to remove all indistinguishable states together with all superfluous states. The removal of the indistinguishable states can be done by the algorithm 3. The S-reduction do not create any indistinguishable state by itself, however, it creates new automaton with new set of superfluous states. More precisely, some states that were originally superfluous may contain self loops after the operation. If the automaton contain cycle, the S-reduction will remove states from this cycle until the cycle is reduced into self loop of last state or until there is no superfluous states in the cycle. If the goal of the application of the S-reduction is to create DGA as small as possible, it is necessary to select the largest possible set of superfluous states that will not contain any cycle. Unfortunately, the Maximal Accyclic Subgraph problem is NP-Hard problem. Moreover, the analysis performed in [54] shows, that this problem can not be approximated in polynomial time.

The paper [51] shows, that if the S-reduction for all superfluous states that induce maximal acyclic subgraph in the automaton is applied to minimal deterministic automata, the resulting DGA is also minimal. However, there may be more than one minimal DGA.

6.4 Pattern Automaton

All previously described methods for generating multistriding automaton have serious limitation for practical use. DGA can guarantee only the worstcase throughput, where worst case is the repeated walk-through cycle of transitions with smallest sum of label lengths. The methods described by Becchi or Brodie can guarantee the desired throughput at the expense of the memory requirements. Moreover these methods allows only to accept power of two symbols which limits the possibility of the trade-of between speed and memory requirements.

Therefore we have designed pattern automaton, which allows the label transitions with n-tuples of set of characters where the n can be any positive integer. The pattern automaton is the combination of the character classes and multistriding. The main advantage of the pattern automaton over the previously used multistriding approaches is its finer trade-of between speed and memory requirements, since it can accepts 3 or 5 characters per transition. It is possible to say n-strided pattern automaton to specify the exact number of symbols that are accepted per transition of the given pattern automaton.

Definition 55. ***Pattern automaton:*** *Lets have alphabet Σ, set Σ_p and integer n such, that*

$$x \in \Sigma_p \implies x = x_1 x_2 x_3 \dots x_n \wedge \forall_{0<i<n+1} : x_i \subseteq \Sigma$$

than the finite automaton over alphabet Σ_p is called pattern automaton over alphabet Σ and every member of Σ_p is called pattern symbol.

Theorem 13. ***Expression power of pattern automaton:*** *For every automaton $M(Q, \Sigma, \delta, s, f)$ there exists alphabet Σ_p and automaton $M_p(Q, \Sigma_p, \delta, s, f)$ such that $L(M) = L(M_p)$ and for every $M_p(Q, \Sigma_p, \delta, s, f)$ there exists automaton $M(Q, \Sigma, \delta, s, f)$ such that $L(M_p) = L(M)\Sigma = \cup_{x \in \Sigma_p} x$.*

The theorem 13 states, that the pattern automaton have the same description power as an finite automaton. To prove this theorem, it is necessary to show, that every regular language can be represented by the pattern automaton and that language accepted by any pattern automaton is regular language. Every regular expression can be represented by the finite automaton M over some alphabet Σ. Then, it is possible to create new alphabet $\Sigma_p = \{\{x\} | x \in \Sigma\}$. Than, the pattern automaton M_p can be created from the original automaton M simply by changing the labels on transitions to be singleton set from Σ_p instead of original symbol from Σ. To prove that every pattern automaton describes regular language, the expression automaton defined by definition 51 can be used. Every language accepted by the expression automaton is regular language. Therefore, showing that every pattern automaton is also expression automaton would be sufficient to prove that every language accepted by pattern automaton is regular language. To prove this requirement, one has to show, that every member of Σ_p can be described by the regular expression over Σ. This statement is indeed true, because alphabet of pattern automaton consists of n-tuples of character classes. Every character class can be represented by the alternation of the appropriate symbols from Σ and the n-tuple is created by the concatenation of regular expressions describing the character classes. Therefore it can be shown, that the theorem 13 is true.

The n-strided pattern automaton is constructed from the finite automaton with character classes by recursively concatenating of the transition until the length n is achieved.

Definition 56. ***Visited states:*** *Lets have automaton $M(Q, \Sigma, \delta, s, F)$, state $q \in Q$ and string $s \in \Sigma^*$. The visited states of q by s is a set of states*

$$V(q, s) = \{u | u = \delta^*(q, s_p) \wedge s_p \sqsubset s \wedge s_p \neq \epsilon\}$$

The Visited states are states that were traversed during accepting string s from state q. It is important to note, that states q and the active state after accepting whole s are not visited states. The definition 56 defines the visited states as a set of states that become active if any proper prefix of string s is accepted from the active state q with the exception of the empty string. The defined notation of the visited states is very useful in the describing of the algorithm used for construction of pattern automaton.

It is important to realize, that there are a lot of optimisation in the IDS. One such optimisation deals with the end of matching. In the IDS, the packet or stream consider to match regular expression if part of the input stream matches. If the automaton reach its final state, the end of matching is signalised to the IDS. Sometimes, the whole inspection can be finished, in other cases matching continues to find other occurrences of the pattern. However, the automaton do not have to deal with accepting all remaining character in the input stream. This optimisation allows to simplify the construction of the automaton. For example, if the string *TEST* has to be found in the input stream, the pattern */TEST/* is given. Theoretically, this pattern should correspond to the regular expression *.*TEST.**. However, due to the optimisation described in this paragraph, the regular expression *.*TEST* is used. The demonstration of the situation can be seen in the figure 6.6.

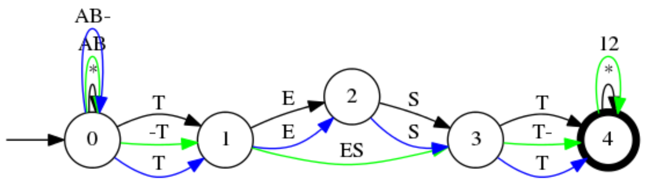


Figure 6.6: Accepting string „AB-TEST-12“ by pattern /TEST/

Black arcs of figure 6.6 shows the transitions the automaton for the pattern */TEST/*. Blue arcs demonstrates what happens when the string *AB-TEST-12* is sent into the IDS. The string *AB-* is accepted by cycling in the state *0*. The string TEXT is accepted by moving from state *0* to state *4*. Because the state *4* is final state the matching ends and text *-12* is not processed at all. The real world implementation of IDS would not need to store transitions cycling in state *4* and therefore it could save memory. Green arcs denote situation in which alphabet transformation was used to increase throughput of the matching by factor of two. The string *AB* is accepted by cycling in the state *0*. The string *-TES* moves automaton from state *0* to state *3*. The automaton has to accept two character to be able to perform one step but only one character of pattern remain to be accepted. The automaton has to accept part of the string *-12*. Therefore the cycling transition in state *4* has to be presented in the automaton before start of the construction of pattern automata.

To produce correct pattern automaton, the input automaton has to correctly describe required behaviour. Therefore, before construction of the pattern automaton, it is necessary to have automaton generated from the regular expression with both dotstar construction. The algorithm 8 describes one possible method how to generate such automaton directly from the automaton generated from the regular expression without second dotstar construction.

Algorithm 8 Duplicating end state with self-loops

```
Input: Finite automaton $M(Q, \Sigma_{class}, \delta, s, F)$
Output: Nondeterministic finite automaton $M_D(Q_D, \Sigma_{classD}, \delta_D, s, F_D)$
 1: $F_D = F$
 2: $Q_D = Q$
 3: $\Sigma_{classD} = \Sigma_{class} \cup \{s | \exists c \in \Sigma_{class} : s \in c\}$
 4: $D = \emptyset$
 5: $\delta_D = \delta$
 6: for $(q, s) \in Dom(\delta)$ do
 7:     if $\delta(q, s) \in F$ then
 8:         if $\nexists q_n : (\delta(q, s), q_n) \in D$ then
 9:             Generate new state $q_n$, such $q_n \notin Q_D$
10:             Add new state in state set $Q_D = Q_D \cup \{q_n\}$
11:             Add new state into set of final states $F_D = F_D \cup \{q_n\}$
12:             Add new state into duplicate set $D = D \cup \{(\delta(q, s), q_n)\}$
13:             Add transition $(q_n, \{s | \exists c \in \Sigma_{class} : s \in c\}, q_n)$ into the $\delta_D$
14:         end if
15:         Find $q_n$, such as $(\delta(q, s), q_n) \in D$
16:         Add transition $(q, s, q_n)$ into the $\delta_D$
17:     end if
18: end for
```

The algorithm 8 duplicates all final states and add self-loops into the duplicates. Due to the existence of character classes, the self-loop over all possible symbol can be realised by one transition. However, the existence of character classes containing all possible input characters is not guaranteed in the input automaton. The line 3 of the algorithm create such character class and add it into the alphabet of the new automaton. It is important to keep in mind, that the alphabet Σ_D can not be deterministic after this step for any automaton, which uses this character class together with any other symbol from the Σ_D. The algorithm tests all transitions and if the transition leads into the final state, the new transition is created and ends in the duplicated final state. The algorithm uses set D to store all duplicated states together with corresponding original state.

It is important to realize that the presented algorithm can produce unnecessary large automata. For example, if the final state do not contain any outgoing transition, it is not necessary to duplicate it. However, if there are any outgoing transitions from the final state, the duplication is necessary to ensure, that the added self-loop will not affect future matching.

The algorithm 9 takes finite automaton as the input and produces pattern automaton as the output. The input automaton is considered to be nondeterministic. The duplication of final states in the first step of the algorithm produces nondeterministic final automaton. Therefore, the pattern automaton produced by the algorithm has to be considered

Algorithm 9 Generation of pattern automata

```
Input: Finite Automaton with character classes $M(Q, \Sigma_{class}, \delta, s, F)$
Input: Number of characters accepted by one step $n$
Output: Pattern Automaton $M_p(Q_p, \Sigma_{classP}, \delta_p, s, F_p)$
 1: Duplicate final states by algorithm 8 into automaton $M_D(Q_D, \Sigma_{classD}, \delta_D, s, F_D)$
 2: $\Sigma_{classP} = \emptyset$
 3: $\delta_p = \emptyset$
 4: $F_p = F_D$
 5: $Q_p = Q_D$
 6: $D = \emptyset$
 7: $Q_{process} = Q_D$
 8: while $Q_{process} \neq \emptyset$ do
 9:     $Q_{new} = \emptyset$
10:     for $q \in Q_{process}$ do
11:         if $\nexists q_o : (q_o, q) \in D$ then
12:             $q_o = q$
13:         else
14:             Find $q_o$, such $(q_o, q) \in D$
15:         end if
16:         Compute all $n$-character symbols $q_S = \{s | s \in \Sigma^*_{class} \wedge |s| = n \wedge (q_o, s) \in Dom(\delta^*_D)\}$
17:         for $sym \in q_S$ do
18:             if $V(q_o, sym) \cap F = \emptyset \vee \delta^*(q_o, sym) \in F_D$ then
19:                 Add a new transition $\delta_p = \delta_p \cup (q, sym, \delta^*_D(q_o, sym))$
20:                 Add symbol into alphabet $\Sigma_{classP} = \Sigma_{classP} \cup \{sym\}$
21:             else if $\exists q_{td} : (\delta^*(q_o, sym), q_{td}) \in D$ then
22:                 Add new transition into duplicated state $\delta_p = \delta_p \cup (q, sym, q_{td})$
23:                 Add new symbol into alphabet $\Sigma_{classP} = \Sigma_{classP} \cup \{sym\}$
24:             else
25:                 Generate new state $q_n \notin Q_p$
26:                 Add new state into set of states $Q_p = Q_p \cup \{q_n\}$
27:                 Add new state into processing pipeline $Q_{new} = Q_{new} \cup \{q_n\}$
28:                 The new state is final state $F_p = F_p \cup \{q_n\}$
29:                 Store, which state is being duplicated $D = D \cup \{(\delta^*_D(q_o, sym), q_n)\}$
30:                 Add a new transition $\delta_p = \delta_p \cup (q, sym, q_n)$
31:                 Add a symbol into alphabet $\Sigma_{classP} = \Sigma_{classP} \cup \{sym\}$
32:             end if
33:         end for
34:     end for
35:     $Q_{process} = Q_{new}$
36: end while
```

nondeterministic, even if the input automaton was deterministic.

Since the input automaton do not contain loops in final states in general, the first step in generation of pattern automaton is to duplicate final state and add self-loops into duplicates.

The lines 2-7 of the algorithm are just initialization of the required data structures. The set D contains tuples that perform mapping from original states into the duplicated states. This set is empty at the beginning of the computation, because no state is duplicated yet. The set $Q_{process}$ contains the set of states that should be processed. The $Q_{process}$ is initialized to contain all states from the original automaton.

During the processing of states in $Q_{process}$, the new version of this set, called Q_{new} is generated. The algorithm is repeated as long as there are states to be processed. The actual processing of the specified state is done in the lines from 11 to 33. The state q is state to be processed.

The *if* command from line 11 to 15 is responsible to detect, if the state q is original state or duplicate of the original state q_o. The decision is made by the analysis of all duplications contained in the set D. Every duplication in this set is described by the tuple of original state and the new state. If there is a duplication, which have state q at the position of a new state, than original state is returned as q_o. Otherwise, q_o is set to be equal to q.

The line 16 contains the core operation of the algorithm. The set q_S will contain all n-character symbols that can describe the outgoing transition from the given state. Strings in the set q_S needs to satisfy three conditions. Firstly, they have to be string over symbols from the alphabet of the automaton. This condition does not have to be required since it is implied by the third condition but it is mentioned to increase eligibility of the algorithm. The second condition holds true if the string have requested length. The third condition is the most crucial one and holds true if it is guaranteed that it is possible to perform n transition in the automaton, such, that every transition will accept the appropriate symbol from s and the first transition will start from the state q_o. The description uses the notation of extended transition function defined in chapter 2. The domain of definition of extended transition function contains all possible string that can be accepted together with state from which they can be accepted.

The state q_o is used as source state for the operation because it is always the state from the original automaton, while state q could be a duplicate. It is important to keep in mind, that even if duplicated state has the same outgoing transitions as original states, the duplicated states are not known in the transition function of the original automaton.

The new transition is generated for every symbol from the set q_S as described from line 17 to line 31. The algorithm uses three ways how to create a new transition. The main difference is whenever the target state should be duplicated or not.

If the transition do not skip any final state, the duplication of the target state is not necessary and algorithm simply adds the new transition into the transition function of the generated automata together with adding the new symbol into its alphabet. The notation of visited states 56 is used to detect all states that are skipped by the new transition. This situation is described by the first part of the condition in the line 18. The second part of this condition states, that if the target state of the transition is final state, no duplication is necessary.

If condition from line 18 was not satisfied, the duplication of the target state is necessary, because processing the string corresponding to symbol s from state q_o would lead to the positive matching in the original automaton but the transition from q_o by s in the new automaton ends in nonfinal state. The condition on the line 21 checks, if the state q_o

do have duplicate in the new automaton by looking into set D. If the duplicate for the original target state exists, the new transition lead into the duplicate. If not, the duplicate is created.

The creation of the duplicated state is described by the lines from 25 to 29. Firstly, the new state has to be different from all states that already exists in the new automaton. It is important to note, that all states of the original automaton was copied into the new automaton in the initialization phase of the algorithm. The new state is called q_n and its creation is described on the line 25. The line 26 adds the new state to the state set of the new automaton while line 27 place state q_n into the set Q_{new}, which contain states that will be processed in the next iteration of the algorithm. The new state has to be final state, which is guaranteed by the line 28. The line 29 is responsible for the update of the set D by adding a new duplication record into it. When the duplicated state is created, the new transition that leads into it is added into the automaton by line 30 and its symbol is added into the alphabet of the automaton as depicted on the line 31.

Theorem 14. ***Equivalence of pattern automaton and finite automaton:*** *Lets have an automaton $M(Q, \Sigma, \delta, s, f)$. The algorithm 9 creates the automaton $M_p(Q_p, \Sigma_p, \delta_p, s_p, f_p)$ such that $L(M) = L(M_p)$ for every automaton M.*

To see that the theorem 14 holds for every automaton M, it would be necessary to show, that the algorithm 9 correctly handles all possible situations that can arise. The algorithm do not create character classes but relies on the correct input. The example on the figure 6.7 shows most of situations which arises during generation of the pattern automaton. The black states and transitions in the figure belongs to the original nondeterministic automaton used as an input to the algorithm 9. The input automaton has one final state denoted as *4*. This state is duplicated into state *5*. Transitions and states created by the final state duplication are drawn in blue color. The algorithm 9 is called to produce 2-strided pattern automaton. The output automaton consists of all states in the figure together with green transitions. During generation of the pattern automaton, three main situation can arise and figure 6.7 shows all of them.

First and most common situation is the multistriding over sequence of nonfinal states. The transition from state *2* is the example of such situation. The green transition from state *1* to state *3* is generated instead of transitions from *1* to *2* and from *2* to *3*. Since the state *2* is not final state, no duplication is performed.

Second situation is demonstrated by the processing of state *3*. By accepting string tT from state *3*, automaton skips state *4* which is final state and ends in state *1*, which is nonfinal state. Therefore, state *1* is duplicated into state *6*, which is new final state and has the same outgoing transitions as state *1*.

The third situation is represented by the state *5*, which is the replica of final state *4*. Lets suppose, that input string for the automaton is *Test!Test?*. The original automaton will accept this string up to second occurrence of symbol t and fail to accept symbol *?*. However, the regular expression in the IDS describes only part of the input stream, which means that if the final state is visited, the regular expression is marked as found. Since the original automaton will end in final state, it found the given regular expression. Unfortunately, if 2-striding without adding state *5* would be used, the automaton would accept *Test!Tes* and end in state *3*. While the original automaton visited state *4* two times, the 2-strided automaton visited state *4* only once and missed the fact, that *Test!Test* is also described by the input regular expression. By adding the state *5*, which contains the self loop over

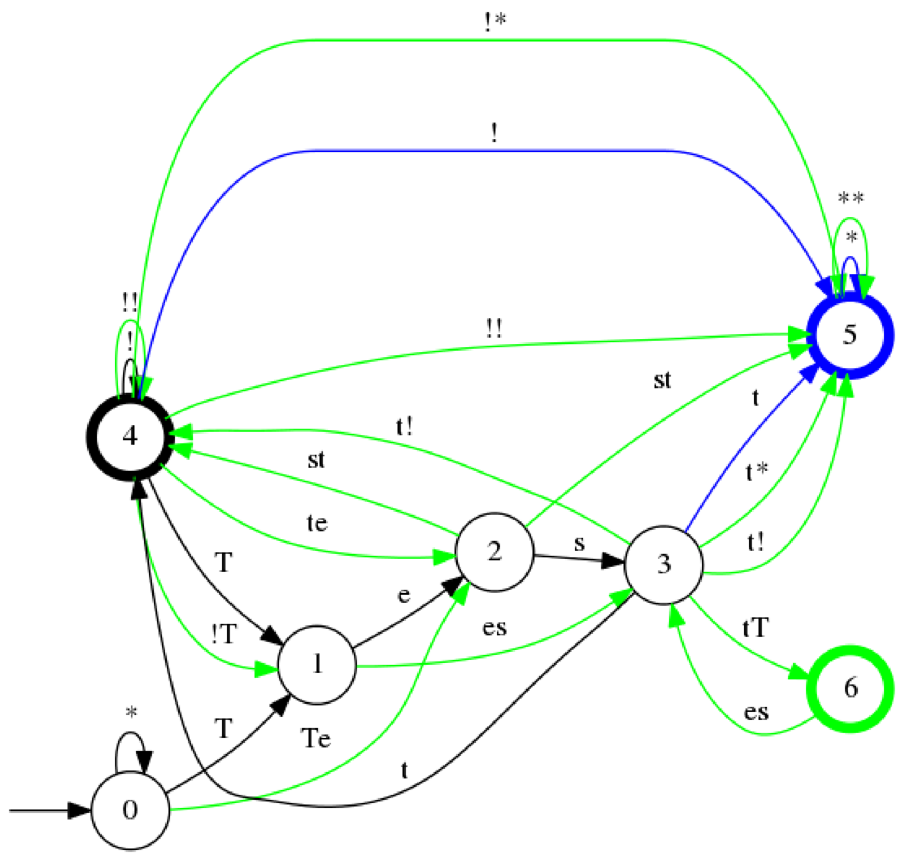


Figure 6.7: Example of pattern automaton generated by algorithm 9

all possible symbols, the automaton will be able to accept *t?* and finish the matching of the second *Test*.

Theorem 15. ***Time complexity of pattern automata generation:*** *The algorithm 9 works with time complexity $O(2 \times |Q| \times \Sigma^n)$, where n is the number of symbols that is accepted by one transition.*

The theorem 15 describes the time complexity of the algorithm 9. The algorithm process every state of the input automaton. If the state meets specified condition, it has to be duplicated and the duplicated state has to be process in the next iteration. Therefore, the algorithm will process all states at once and duplicate all nonfinal states in the worst case resulting in processing $2 \times |Q|$ states. The final states are duplicated in the first step of the algorithm to ensure, that there is self loop for every possible end of matching. For every processed state, the algorithm has to find the closure of transition function with the specified length. If the automaton is fully defined, it will require to process all possible combination with repetition of the specified length from input alphabet. Therefore, every state will be processed at worst case in Σ^n steps. Therefore, it can be seen that the theorem 15 holds true.

The time required for generation of the pattern automaton depends on the number of states in the automaton and on the structure of the automata. The introduction of character classes into the automaton significantly speeds up the process, since it reduces the base of the exponent function. Therefore, if the algorithm 6 would be used before, construction of the pattern automaton, the speed up would be in order of magnitude.[2] However alphabet generated by the algorithm 6 is generally nondeterministic and therefore the alphabet Σ_p of the newly generated pattern automaton will be also nondeterministic. To be able to use the advantage of deterministic pattern automaton, the alphabet determinisation function is necessary.

To be able to describe the algorithm for alphabet determinization over pattern symbols, several basic operation over pattern symbols need to be defined.

Definition 57. ***Length of pattern symbol:*** *Lets have pattern symbol $s = s_1 s_2 \dots s_n$. The n is called the length of pattern symbol s and is denoted by $len(s)$.*

Definition 58. ***Size of pattern symbol:*** *Lets have pattern symbol $s = s_1 s_2 \dots s_n$. The size of pattern symbol s is denoted by $|s|$ and is computed as follows*

$$|s| = \sum_{s_i \in s} |s_i|$$

Definition 59. ***Intersection of pattern symbols:*** *Lets have pattern symbol $s^1 = s^1_1 s^1_2 \dots s^1_n$ and pattern symbol $s^2 = s^2_1 s^2_2 \dots s^2_n$. The pattern symbol s^3 is intersection of pattern symbols s^1 and s^2 denoted as $s^1 \cap s^2$ and is defined as follows*

$$(s^3 = s^1 \cap s^2) \iff \forall 0 < i < (n+1) : s^3_i = s^2_i \cap s^1_i \wedge s^3_i \neq \emptyset$$

The definition 59 defines the intersection of two pattern symbols as a new pattern symbol, that consists of intersections of corresponding character classes. It is important to note, that if there are coresponding character classes that have empty intersection, the operation is not defined.

Definition 60. ***Diference of pattern symbol:*** *Lets have pattern symbol $s^1 = s^1_1 s^1_2 \dots s^1_n$ and pattern symbol $s^2 = s^2_1 s^2_2 \dots s^2_n$. The difference of pattern symbols s^1 and s^2 denoted as $s^1 - s^2$ is a set of pattern symbols D computed as follows*

$$D = \left\{ s : |s| = |s^1| = |s^2| \wedge \forall 0 < i < n+1 : s_i = s^1_i \setminus s^2_i \vee s_i = s^2_i \cap s^1_i \wedge s \neq s^1 \cap s^2 \right\}$$

The definition 60 describes the difference of two pattern symbol. Informally, the $s^1 - s^2$ describes strings, that corresponds to the symbol s^1 but not to the symbol s^2. Unfortunately, the language of pattern symbols is not closed to this operation, which means, that the strings that corresponds $s^1 - s^2$ can not be described by one pattern symbol in general. The definition 60 solves this problem by defining the set of pattern symbols that describes the difference. This set contains of pattern symbols that meets three conditions. The first condition states, that the pattern symbol has the same length as s^1 and s^2. If the pattern symbols s^1 and s^2 have different length, their difference is the empty set because there is

[2]It is important to keep in mind, that the actual speed up depends mostly on the structure of the automaton in question.

no pattern symbol that can satisfy this condition. The second condition states specifies the content of the pattern symbol. Every set in the pattern symbol is computed as a set difference of sets from corresponding positions or as an intersection of these sets. The third condition ensures, that every pattern symbol from the difference set has at least one position computed by the set difference operation.

Defined operations can be used to describe algorithm for generating deterministic pattern alphabet from the nondeterministic one. The algorithm 10 takes nondeterministic pattern alphabet Σ_N as input and produces deterministic pattern alphabet Σ_D together with mapping [3] $M : \Sigma_N \to \Sigma_D$. The mapping M shows which symbols from the alphabet Σ_D should be used to describe the same language as given symbol from Σ_N.

The basic idea of algorithm 10 is to start with empty alphabet Σ_D, which is deterministic for sure and adding new symbol to it in such way, that the alphabet will remain deterministic. The adding new symbol into the deterministic alphabet is implemented in function AddSymbol. This function takes four parameters. First parameter sym is the symbol that should be inserted. The deterministic alphabet to which the symbol should be inserted is given as a second parameter Σ_D. The third parameter is the mapping $M : \Sigma_N \to \Sigma_D$. The AddSymbol function changes this mapping during the computation to reflect actual state of the computation. The fourth parameter $M_{back} : \Sigma_D \to 2^{\Sigma_N}$ is reverse of mapping M. It is used to simplify the process of updating the mapping M during the changes of the alphabet.

The first step in the function AddSymbol is to update the mapping M_{back}. This update is done by line 2 to 4 of the algorithm 10. The condition in the line 2 checks the existence of record for the new symbol in the mapping. If the record is not found, it means that this is first level of recursion and the new symbol is the symbol from alphabet Σ_N and the identity is added to the mapping M_{back}. If the record exists in the mapping, it means that the symbol was created by the recursive calling of AddSymbol and the mapping M_{bakc} was update by the caller.

When the symbol is added into the deterministic alphabet, two situations may occur. First, symbol do not collide with any other symbol in the alphabet. In this situation it is possible to just add this symbol into the alphabet. This situation is described by lines from 5 to 7 of the algorithm 10.

The line 9 to 30 deals with the second, more interesting, case that the new symbol collides with some other symbol that is already in the alphabet. Actually, it is possible, that new symbol collides with more than one symbols but because the target alphabet is deterministic, every collision can be dealt with independently. Therefore, algorithm simply takes the first colliding symbol for the processing. Firstly, the colliding symbol p is removed from the alphabet Σ_D and the mapping M by lines 9 and 10. The line 11 computes the difference between symbol p and new symbol sym. This is the set of symbols, that describes as large part of the symbol p as possible without having collision with the symbol sym. It is important to note, that these symbols do not collide with any symbol from Σ_D, because they are created from the symbol p, which was taken from Σ_D. The line 11 computes the difference between symbol sym and the symbol p called sym_{new}. These are symbols that do not collide with p and describe as much as possible of the symbol sym. However, symbols from sym_{new} can collide with other symbols from Σ_D.

The symbols sym and p may not be the symbols from the original alphabet Σ_N but they could be created by previous call of the AddSymbol function. Therefore, mapping

[3]The term mapping is used in broadest way possible since the mapping M can map one symbol from Σ_N to several symbols from Σ_D

Algorithm 10 Alphabet determinization for pattern symbols

```
Input: Nondeterministic pattern alphabet $\Sigma_N$
Output: Deterministic pattern alphabet $\Sigma_D$
Output: Mapping $M : \Sigma_N \rightarrow \Sigma_D$
 1: function AddSymbol(Sym, $\Sigma_D$, $M : \Sigma_N \rightarrow \Sigma_D$, $M_{back} : \Sigma_D \rightarrow 2^{\Sigma_N}$)
 2:     if $sym \notin M_{back}$ then
 3:         $M_{back} = M_{back} \cup (sym \rightarrow \{sym\})$
 4:     end if
 5:     if $\nexists p \in \Sigma_D : p \cap sym$ then
 6:         $\Sigma_D = \Sigma_D \cup \{sym\}$
 7:         $M = M \cup (sym \rightarrow sym)$
 8:     else
 9:         Remove conflicting symbol from alphabet $\Sigma_D = \Sigma_D \setminus \{p\}$
10:         Remove conflicting symbol from mapping $\forall x \in \Sigma_N : M = M \setminus \{(x \rightarrow p)\}$
11:         Remove intersection from symbol p: $p_{new} = p - sym \cap p$
12:         Remove intersection from new symbol: $sym_{new} = sym - sym \cap p$
13:         Discover set of original symbols $p_o$ of p, such as $(p \rightarrow p_o) \in M_{back}$
14:         Discover set of original symbols of sym $sym_o$, such as $(sym \rightarrow sym_o) \in M_{back}$
15:         Add intersection to mapping $\forall p_i \in p_o : M = M \cup (p_i \rightarrow p \cap sym)$
16:         Add intersection to mapping $\forall sym_i \in sym_o : M = M \cup (sym_i \rightarrow p \cap sym)$
17:         Add intersection to mapping $M_{back} = M_{back} \cup (p \cap sym \rightarrow p_o \cup sym_o)$
18:         Add intersection into alphabet $\Sigma_D = \Sigma_D \cup \{p \cap sym\}$
19:         for $s \in p_{new}$ do
20:             Add a symbol from difference into $M$: $\forall p_i \in p_o : M = M \cup (p_i \rightarrow s)$
21:             Add a symbol from difference into $M_{back}$: $M_{back} = M_{back} \cup (s \rightarrow p_o)$
22:             $\Sigma_D = \Sigma_D \cup \{sym\}$
23:         end for
24:         Order $sym_{new}$ by size of symbol
25:         for $s \in sym_{new}$ do
26:             Add a symbol from difference into $M$: $\forall sym_i \in sym_o : M = M \cup (sym_i \rightarrow s)$
27:             Add a symbol from difference into $M_{back}$: $M_{back} = M_{back} \cup (s \rightarrow p_o)$
28:             Recursively call AddSymbol(s,$\Sigma_D$,$M$,$M_{back}$)
29:         end for
30:     end if
31: end function
32: Order $\Sigma_N$ by the size of pattern symbols
33: $\Sigma_D = \emptyset$
34: $M = \emptyset$
35: $M_{back} = \emptyset$
36: for $s \in \Sigma_N$ do
37:     AddSymbol(s,$\Sigma_D$,$M$,$M_{back}$)
38: end for
```

M_{back} is used to detect from which symbols of Σ_N are these two symbols derived. If the *sym* or p are from Σ_N, the M_{back} will return set containing *sym* or p respectively.

The sets returned from previous lines are used to update the mapping M in the lines 15 and 16. Mapping from every original symbol into the newly intersection of symbols p and *sym* is added. The M_{back} is updated in the line 17 to map the intersection into the union of original symbols of p and *sym*, both original symbols have to be rewrite to this symbol. This line is the actual reason, why there may be more than one original symbol for the newly created symbols. The line 18 add the intersection of these symbols into the alphabet Σ_D. No checking for the existence of the intersection of this new symbol with symbols from Σ_D is necessary, because this checking was performed when the original symbol p was inserted in the Σ_D at some previous step.

The next step is to add symbols returned by the difference between p and a $p \cap sym$. These symbols do not have any collision with symbols that are already in $Sigma_D$ for exactly the same reason as the intersecting symbol. Therefore line from 19 to 23 updates mapping M and M_{back} together with adding symbols from the difference into the Σ_D.

The lines 24 to 29 are responsible for inserting the symbols created from *sym* into the Σ_D. The first step is to order these symbols by size. This step is not absolutely necessary, but it allows the replicability of the approach. Moreover, the size and structure of the resulting alphabet depends on the order in which the symbols are added into it. The lines 26 and 27 upates mapping M and M_{back}. The line 28 is the recursive call of AddSymbols to insert new symbols into the Σ_D. The recursive call is necessary, because the new symbols may collide with other symbols that are already in Σ_D. This is possible due to the fact, that the original symbols may have intersection with several different symbols in Σ_D (checked in line 5) and the algorithm processed just the first one. The other collisions are dealt with by this recursive calls.

Lines from 32 to 38 implements the main loop of the algorithm. Firstly, in the line 32, the input alphabet Σ_N is ordered. Then, new alphabet Σ_D together with both mappings are initialized and emptied. Lines 36 to 38 depicts the walk through the ordered version of Σ_N and adding symbols from it into the deterministic Σ_D. The function AddSymbols ensures that the alphabet Σ_D is always deterministic and mappings M and M_{back} are up to date. The alphabet Σ_D and mapping M are main output of the algorithm.

The algorithm 11 takes the pattern automaton as its input and generates the pattern automaton with the deterministic alphabet as its output. If the alphabet of the input automaton is already deterministic, the algorithm will produce copy of the input automaton.

Algorithm 11 Generating the pattern automaton with deterministic alphabet

```
Input: Pattern automaton with nondeterministic alphabet $M_p(Q_p, \Sigma_{classP}, \delta_p, s, F_p)$
Output: Pattern automaton with deterministic alphabet $M_D(Q_p, \Sigma_D, \delta_D, s, F_p)$
1: Use algorithm 10 to generate $\Sigma_D$ and $M$ from $\Sigma_{classP}$
2: $\delta_D = \emptyset$
3: for $(q, s) \in Dom(\delta)$ do
4:     Find a new symbols $S$, such $S = \{x | (s \to x) \in M\}$
5:     for $sym \in S$ do
6:         Add new transition into $\delta_D$: $\delta_D = \delta_D \cup \{(q, sym, \delta(q, s))\}$
7:     end for
8: end for
```

The first step of algorithm 11 is the construction of deterministic alphabet for the new automaton. After the construction of deterministic alphabet, the algorithm walks through the transition function of the input automaton by the for cycle on the line 3. For every transition, the mapping M is used to determine the set of symbols that corresponds to the label of actual transition. The new transition is inserted into δ_D for every such symbol. It is important to keep in mind, that mapping M allows to map one symbol from original alphabet into many symbol from the deterministic alphabet.

The output of the algorithm 11 is the pattern automaton with the deterministic alphabet. If the pattern automaton is constructed by the algorithm 9, it is nondeterministic. Therefore, the pattern automaton should be determinised and minimised by the algorithms given in chapter 2.

The pattern automaton constructed by described algorithms have higher throughput than finite state automaton. The main advantage of the pattern automaton is its ability for fine tuning of the trade off between speed and the memory requirements by allowing to accept arbitrary number of symbols per transition. Due to its definition, pattern automaton can be determinised and minimised in the same way as the finite state automaton.

6.5 Reducing memory requirements

The transition table of the deterministic finite automata contains large amount of redundant information. There exists a lot of methods that can exploit this redundancy to reduce overall memory requirements of the automaton.

6.5.1 Default state

The default state optimisation exploits the idea that most of the transitions in the automaton lead into one state. The basic principle of default state optimisation is described in the Dragon book [14] in the chapter 3.9.8. For example, most of the transitions in the automaton generated from the regular expression */.*HELLO/* leads into its starting state. Storing all these transition in transition table is ineffective. The introduction of the default state allows to remove all transition leading to the one specific state. The optimization changes the behaviour of the transition function of the automaton. Lets suppose that the input automaton is fully defined. One of the states of the automaton is selected to be default state and all transitions leading into the default state are removed from the transition function. If the transition is not find in the transition table during the lookup, transition into the default state is performed instead. The transition into the default state process always one input character. Therefore default state optimisation do not affect the speed of the matching process. This is the main difference between default state approach and Delayed DFA [71], which do not accept any symbol by the Delayed transitions.

It is possible to further improve the optimisation by allowing multiple defaults states. In this extension, every state has its own default state. If the transition is not found in the transition table during the matching process, correct default state has to be determined from the active state. Therefore, another table containing the record for every state in the automaton has to be accessed to perform transition into the defaults state. Depending on the implementation, this additional memory access may affect the speed of the matching.

Chapter 7

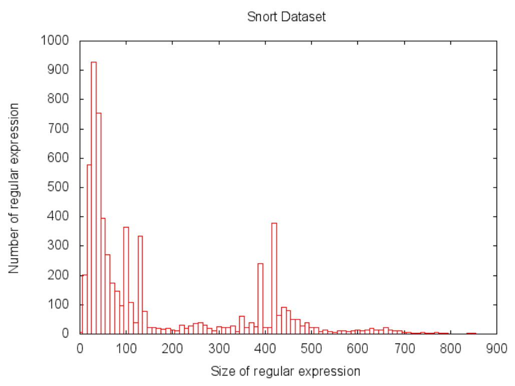
Analysis of regular expressions used in Intrusion Detection Systems

The chapter 4 described complex algorithms used for processing of the regular expression. Many modern IDS rely on the regular expression matching to detect the possible threat in the network payload. For many IDS, the quality of the regular expression matching determines the quality of the whole Intrussion Detection System. Therefore, almost every IDS uses different implementation of the regular expression engine which leads to different syntax of the language used to describe the regular expression for different threats. Moreover, many system implements their own extensions of the regular expression languages, such as character classes, conditional regular expressions or back references. Unfortunately, these extensions often differs across different systems in both syntax and semantics. Some of the extensions introduced by the regular expression matching engines increases the descriptive power of the language far beyond the scope of regular languages. The example of such extension is the backreference in PCRE which is used in Snort IDS.

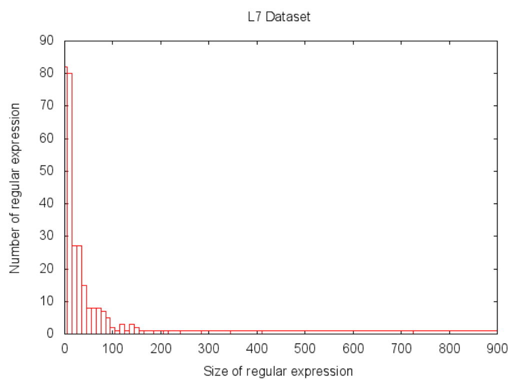
This chapter focusses on the experimental analysis of the „regular expressions“ used by the existing network security systems. The preliminary results of this analysis was published in [66] and in [65]. The presented experiments were performed for the L7 (3.3.2) and Snort (3.3.1) ruleset, because of these two systems are syntactically compatible with available RE parser.

7.1 Properties of the regular expressions

Experiments presented in this chapter were performed on the Snort [11] dataset and the ruleset from L7 netfilter [1]. The Snort dataset contains 6441 unique PCRE expressions divided into the several groups while L7 contains only 260 regular expression describing different protocols. The lengths of the regular expressions vary in range from few bytes to almost 1KB. The distribution of lengths of regular expression can be seen at the Figures 7.1a and 7.1b. The lengths of the regular expressions are rounded to nearest ten. It can be seen that Snort rules are clustered around several most common lengths. Therefore, it would not be sensible to measure average length for this dataset. The most of regular expression in the L7 set has the length up to 50 bytes. However, the longest regular expression in the set has the length of 6320 bytes. The regular expressions with length larger than 100 bytes



(a) Analysis of lengths of expressions from Snort dataset



(b) Analysis of lengths of expressions from L7 dataset

are rare in the L7 rules.

The computation complexity of the PCRE matching process is greatly affected by the complexity of the regular expression itself. The table 7.1 shows how many constructions of the PCRE grammar are used in the given rule set. It can be seen that the Snort contains much more complex rules than the L7 filter and that Snort rules often use construction that are beyond the classical regular expressions.

In general, whole network stream has to be checked. The beginning of the line (BOL) symbol in the regular expression can be used to simplify the matching process, because if the regular expression containing the BOL symbol is not at the beginning of the flow, it is not necessary to search for it further in the stream. It can be seen, from the table, that 1260 snort rules contain the BOL symbol. However, the multiline modifier (m), which changes the behaviour of the BOL, is presented in the 1285 snort rules. The performed experiments show, that the 773 rules contain both BOL symbol and multiline modifier. This means, that only 487 rules out of 6641 could benefit from stopping the matching if the rule is not found at the beginning of the network stream. The rest of the rules has to be searched in the whole payload.

It is important to note, that Limmer and Dresler in [79] observed, that most of the current attack on the network appears up to 2000B after the change of the direction of the communication. In most cases, it means first 2000B of the payload. This observation allows to speed up pattern matching by discarding uninteresting part of the traffic at the cost of slight increase of false negatives during the matching. However, it is not possible to predict the amount of the missed attacks by the analysing of the rule without the additional knowledge about the attack and the internet traffic itself.

Another important structure is the caseless modifier, which effectively reduces the size of the alphabet, because matching process do not have to differentiate between lowercase and uppercase symbols. Unfortunately, L7 rules do not use this modifier at all.

It can be seen, that the Snort rules often used the iteration in the form of zero or more iterator (star iterator) and counting constrains which can significantly increase the complexity of the regular expression and the size of the corresponding finite automaton.

Snort also use backreferences which are beyond classical regular expressions. It is interesting to see, that the subroutines are currently not used in Snort rules. We used RE parser [98] which is able to build finite automaton for 3195 PCRE Snort rules. The remaining rules contained some form of the unsupported structure. Only 13 rules from the L7 ruleset

Type of structure	Number of structures in Snort	Number of structures in L7
Assertions	5	0
Backreferences	1416	0
BOL	1260	86
Capturing	10848	76
Caseless	2784	0
Counting constraints	1996	0
stats/dotall.txt	1925	0
Symbols	196695	5905
End of line symbol	240	21
Modifier s	1	0
Character class	27627	305
Lazy iterator	2036	0
Modifier B	3	0
Modifier O	16	0
Modifier P	119	0
Modifier R	298	0
Modifier m - Multiline pattern	1285	0
Setting options inside regexp	1	0
Unsietting options inside regexp	1	0
Once or more iterations	4384	37
Zero or more iterations	20454	87
Subroutines	0	0
Modifier U - ungreedy matching	643	0
Zero or once	1820	524

Table 7.1: Numbers of different types of regular expression

could not be transformed into finite automaton.

It is possible to conclude, that the Snort rules contains much more complicated patterns than L7. L7 supports only classical regular expressions. Snort uses in contrary many extensions provided by the PCRE.

7.2 Saturation of transition table

The main purpose of this chapter is to analyse the properties of the regular expression used in the intrusion detection systems. Therefore it is important to define the measurable properties. This work uses finite automaton for the matching and therefore the analysis will focus on properties of the finite automaton. The most commonly measured properties are size of the automaton and transition size of the automaton as defined in chapter 2. These two properties together reflects the size of the memory required to store the automaton.

However, the finite automaton is a theoretical concept, which can be represented by many different implementations. Every implementation then require some form of mapping, which transform the theoretical model of the automaton into specific data structures usually stored in the memory. The chapter 4 summarizes the state of the art implementations of the regular expression matching algorithms and architectures. The efficiency of the mapping from the finite automaton into the corresponding data structure of the pattern matching unit plays crucial role to determine memory requirements for the given automaton. The more effective mapping, the less memory is required to store the automaton.

One of the properties, that affects the efficiency of the mapping is the saturation of transition table. Therefore we define saturation of the transition table to describe the memory efficiency.

Definition 61. ***Saturation of transition table:*** *Saturation of transition table of automaton M is $Sat(M) = \frac{|M|_T}{|S||\Sigma|}$, where S is the set of states in M and Σ is the alphabet of M.*

The saturation of the transition table indicate how much of the transition table is filled with the relevant information. If the saturation is 1, every possible combination of a state and an input symbol has transition in the automaton. The saturation 0 actually means, that the automaton do not contain any transition at all.

The saturation of the transition table plays important role in efficient selection of implementation methods that maps the given finite automaton into specific hardware implementation. The mapping techniques has various efficiency of memory or FPGA logic utilization with respect to the saturation of the transition table. For example, if saturation is 1, very efficient mapping is to simply implement transition table as memory array and assign one memory location to every combination of active state and input symbol. However, such datastructure has high memory requirements (number of states times number of symbols) and if the saturation of the transition table is going lower, more and more memory cells are unused and efficiency of such mappings decreases.

The saturation of the fully defined deterministic automaton is always one. However, the implementation tricks and optimization of the modern IDS, together with the specific types of patterns may result in automata that are not fully defined.

The first and obvious reason for the low saturation is the fact, that the IDS do not require automaton with the sink state because if automaton should perform nonexisting transition, than it is obvious that the given string is not accepted by the automaton and therefore

State	[ˆA-Z]	[ˆA]	[A-Z]	A
1	1	∅	2	∅
2	∅	1	∅	3
3	∅	∅	∅	∅

Table 7.2: Transition table with character classes for automaton from fig. 7.2

IDS can finish the matching. Another advantage of this implementation is that neither computation time nor any computation resources are spend to process transition into sink state or cycling in it. For example, regular expression starting with BOL symbol and not using the multiline operator could have low saturation because these regular expressions are typically strings and therefore most combination of the state and input symbol do not describe transition of the automaton.

The important source of the low saturation of transition table is the introduction of the default states, which are described in the section 6.5.1 of this work. Since the default state is the state that becomes active if there is not transition from the current active state and the given input symbol, all transitions that lead into it may be removed from the transition function. It is easily to see, that while the main aim of this optimisation is to reduce number of transitions, it also reduces the saturation.

The use of character classes is another possible reason for the low saturation of the transition table. The one transition described by the character class holds the same information as several transition described by the symbols of the alphabet. Therefore there is no need to have a transition for every symbol in the alphabet and the saturation of transition table is decreased. The description of character classes and their use for the reduction of number of transition was presented in the previous chapters. The different methods of construction and handling character classes have different effects on the saturation of the transition table.

The example of the automaton with saturation no more than 50% can be seen at the figure 7.2. It is important to keep in mind, that this automaton can accept any input character in any active state but it is not fully defined.

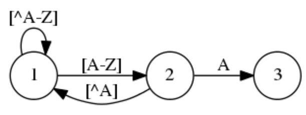
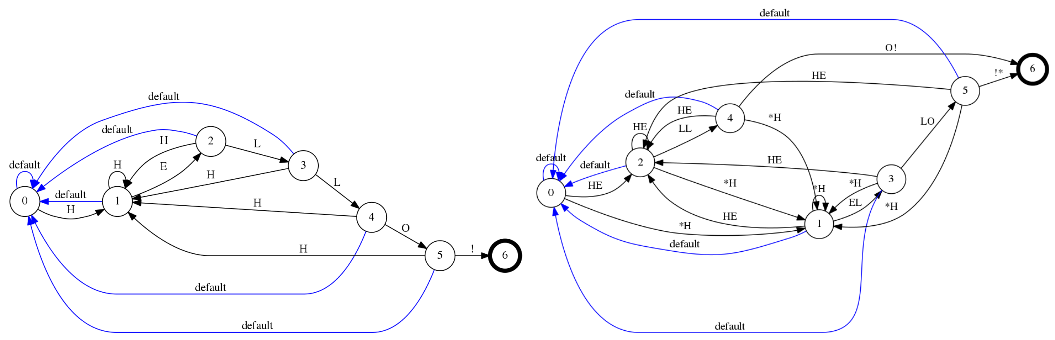


Figure 7.2: Automaton with saturation no more than 50%

The example in Figure 7.2 has three states and for every ascii symbol, there exists transition in the automaton. The transition table for this automaton can be seen in the table 7.2. Transitions, that do not exist in the automaton are denoted by the ∅ symbol in the table. It can be easily seen from the table, that the saturation of the automaton is $\frac{4}{12}$.Lets analyse behaviour of the automaton for states 1 and 2. It can be seen, that the automaton accepts all symbols from the uppercase of english alphabet. The saturation in the transition table is low, because symbols such *A* fits the description of several transitions. For example, in state one, it is not necessary to define transition by the symbol *A*, because if it appears in the input, the transition described by symbol *[A-Z]* will be used. Therefore, the cell in transition table that corresponds to the transition from 1 by symbol *A* will



(a) Automaton for regular expression *.*HELLO!*

(b) Multistride automaton for regular expression *.*HELLO!*

Figure 7.3: Automata for regular expression *.*HELLO!*

become empty and the saturation of the transition table will be lowered. It is important to note, that the alphabet from the example is 1-deterministic for the automaton showed in the example. Therefore, it is possible to determine the correct transition. It can be seen that all transition in the automaton overlap with each other. Due to this overlapping, it is necessary to know the active state of the automaton during the alphabet transformation process.

Lets analyse situation at the state 3. There are no transitions from this state, which means that no matter what input symbol appears, the transition into the default state will be performed. Similar situation would appear, if the state 3 would be final state of the automaton. The exact behaviour of the IDS after reaching the final state depends on its setting but if IDS is interested only in the first match, it is possible to end matching process immediately after the final state become active. In such case, there is no need to have outgoing transitions from final state, which will be another cause for lower saturation of transition table by placing another empty raw into the transition table.

The multistriding of the automaton may further decrease saturation of transition table. The figure 7.3a shows the automaton accepting language defined by the regular expression *.*HELLO!*. The automaton uses default state to decrease level of saturation of transition table. Character classes are not applicable due to the simplicity of the automaton. The figure 7.3b shows the 2 strided automaton accepting the same language. This automaton uses the default state together with character classes. The concept of character classes is represented by symbol * in the label. This symbol means, that any input symbol may appear on this position. For example, transition description **H* means any symbol followed by *H*. In the classical definition of character classes, all symbols from Σ needs to be placed at the position of *. The number of symbols in the automaton would by significantly higher without this optimization. Furthermore, since transition with label **H* is outgoing transition for all states of the automaton the size of transition table and the saturation would be much higher without utilisation of character classes. The default state optimization is used mainly for the simplification of the example and because different multistriding methods have different transitions back to the starting state. This difference is caused by various descriptions of multistrided symbols by methods. See chapter 6 for clear explanation. However, it can be seen in the Figure 7.3a that each default transition could be replaced by one transition labeled with proper character class. This do not hold for the Figure 7.3b. Lets look for example at state 1. There are several outgoing transitions labeled by *EL,*H,HE*. The default state optimization add a single transition for all other combinations of inputs. Of course, the exact method how to specify behaviour of automaton for remaining inputs depends on the symbols used by the automaton. This usually requires more complex description than simple set of characters. If the multistride symbol was defined as a pattern symbol (sequence of character classes), the default state behaviour would have to be simulated by four transitions labeled by *[ˆH]E, [ˆEH][ˆHE], HL*. Some of these symbols would be used only in one state of the automaton and therefore they would be be represented by empty cell in the transition table for all other states. This can further reduce the saturation of the transition table. It can be seen, that the combination of the multistriding with character classes or default states has significant effect on the structure of the transition table.

The multistriding itself can decrease saturation of transition table simply by introducing large set of more specific symbols. The original automaton has alphabet consisting of 8-bit ASCII symbols. Therefore the maximal size of the alphabet is 256 symbols. If the automaton contains 257 transitions, at least one symbol has to be used twice due to the pigeon principle. The saturation of transition table correlates with the amount of repetitive usage of the symbol. When one symbol is used many times it contribute to high saturation of transition table but when the symbol is used only once, it contribute to the low saturation of transition table. The multistriding methods create more specific symbols that describe larger part of input string to speed up the matching process. However, as a side effect, the alphabet of multistrided automaton is larger that alphabet of the original automaton. Therefore, chance of repeated use of the same symbol is decreased which contributes to low saturation of transition table.

It can be concluded, that while saturation of deterministic fully defined automaton is always 1, practically used optimizations can generate transition table with various level of saturation. This section described how different optimisations affect the saturation of transition table and provided the examples. However, the actual value of the saturation depends on the regular expression itself and on the exact combination of optimizations used for the implementation. It is important to keep in mind, that the main purpose of all these optimizations is to reduce the size of the transition table and not to change the saturation of the transition table. The saturation of the transition table is simply the property of the automaton which can be utilized for mapping the automaton to the data structure or hardware architecture.

7.2.1 Experiments with saturation

The previous section summarised some methods for the reduction of the size of the automaton and the impact of these methods on the saturation of transition table. We discussed that also the combination of various optimization methods has direct influence on th saturation of transition table. Therefore, this section describes measurements performed on automata generated from regular expressions used in high speed networks. For experiments, we use the Snort[11] dataset and L7 rules [1].

In the first experiment, the minimal deterministic finite automaton was created for every regular expression in the set. Regular expressions with the exponential state blowup during the determinisation were omitted from the experiment because because they are not suitable for the methods based on DFA. The limit for the maximal memory usage and computation time was introduced to detect these regular expressions. If the computation reaches one of these limits it is terminated and the regular expression is not covered in the

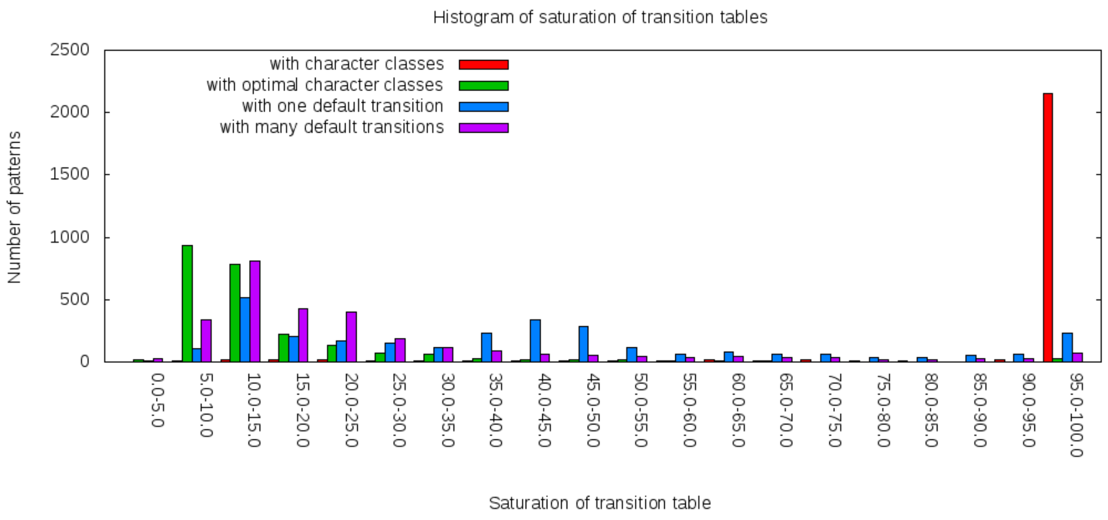


Figure 7.4: Saturation of transition table of DFA created from Snort regular expressions. Histogram shows how many patterns are with saturation presented at the x-axe. The x-axe is annotated in percentages. The level of saturation is shown for various optimisation methods.

results.

The FSM tool [89] was used for the determinisation and minimization of automata created from regular expressions. The FSM tools do not have a support for interrupting the determinisation if the size limit of the resulting automaton is exceeded. Therefore the limit on the virtual memory of the determinisation and minimization process was set to 1GB of memory by the ulimit command. The saturation of transition table of remaining automata together with its size were measured. Since modern computers have several CPU cores and large amount of memory, it is possible to use the parallelization to speed up the experiment. The GNU parallel [119] command was used to achieve effective parallelization.

Two properties of the automaton are measured. First is the number of transitions which corresponds to the memory requirements to represent the automaton. The second measured property is saturation of transition table. The results are presented in two histograms for each dataset. Histograms in figures 7.4 and 7.5 shows the distribution of the saturation in the snort and L7 dataset respectively. The number of transitions are reported in the figure 7.6 for rules in the Snort dataset and figure 7.7 represents the distribution of the number of transitions in automata for the L7 dataset.

The histograms are presented for all optimisations discussed in previous sections and compared to standard representation without character classes:

without character classes This setting do not contain any character class in the automaton. If the regular expression contained character classes, transitions with one character as a label was generated from these character classes by algorithm 5.

optimal character classes The optimal character classes was generated automatically to generate as large character classes as possible. As a result, every two state in the automaton are connected by at most one transition in each direction. The alphabet generated in this way is nondeterministic. The main purpose of this experiment is to show maximal possible efficiency of the character classes in the reduction of the

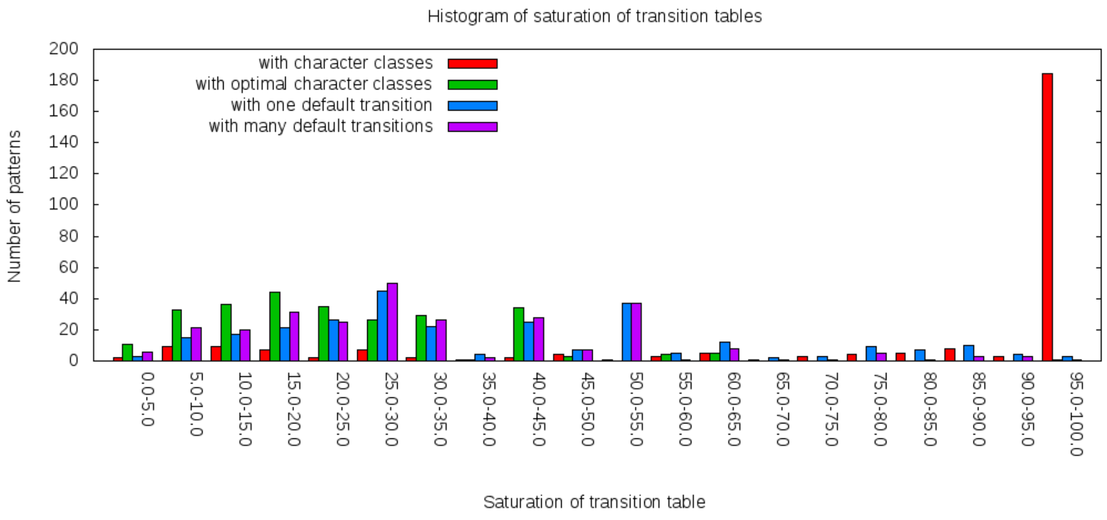


Figure 7.5: Saturation of transition table of DFA created from L7 regular expressions. Histogram shows how many patterns are with saturation presented at the x-axe. The x-axe is annotated in percentages. The level of saturation is shown for various optimisation methods.

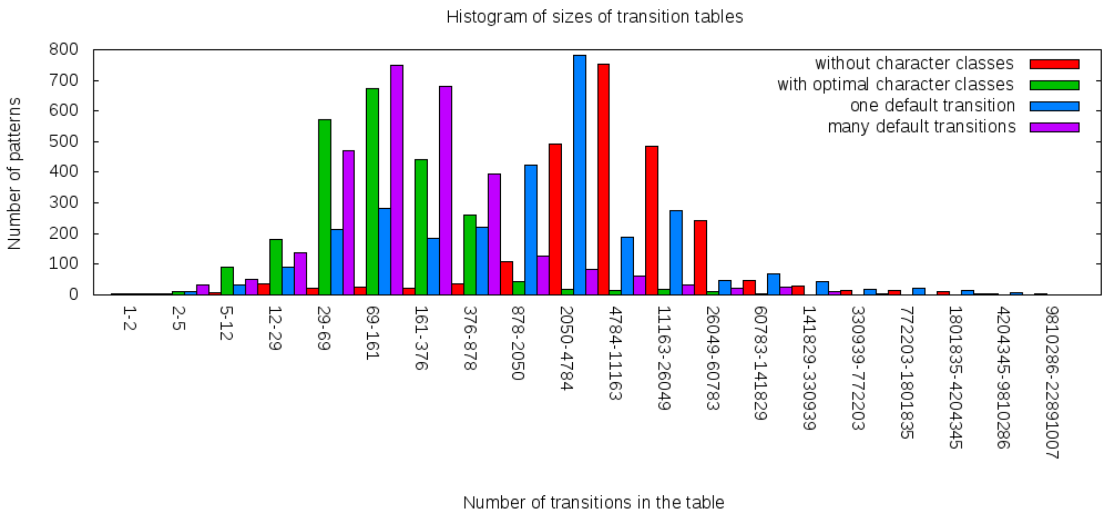


Figure 7.6: The number of transitions in DFAs created from the regular expressions in Snort dataset. The histogram presents for various optimisation methods the distribution of patterns according to the amount of transitions in the transition table of the automaton.

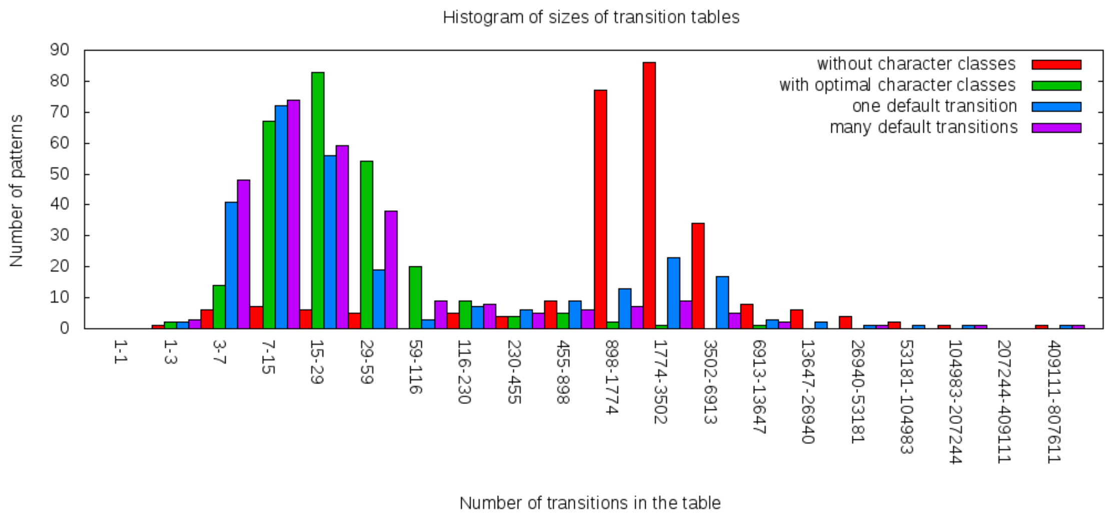


Figure 7.7: The number of transitions in DFAs created from the regular expressions in L7 dataset. The histogram presents for various optimisation methods the distribution of patterns according to the amount of transitions in the transition table of the automaton.

transition size of the automaton and how its affect saturation of the transition table. The algorithm 6 was used for the generation of character classes in this settings.

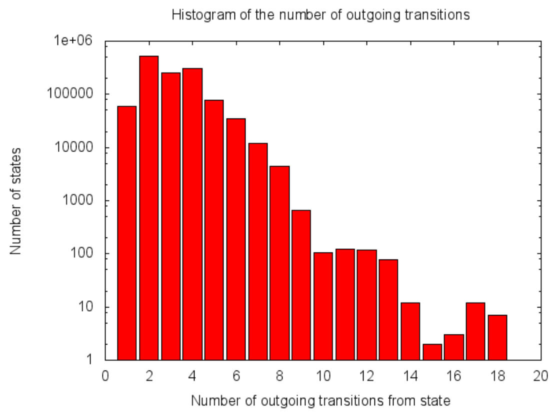
one default state The one default state setting shows the histogram with the saturation of transition table of automata with one default state. The default state is the state that had the highest number of incoming transition in the fully defined variant of the automaton. The transition that originally lead to the default state were removed from the automaton to reduce memory consumption. The principle of the default state optimisations is described in the section 6.5.1 of this work.

many default states This settings represents situation where every state has its own default state. The principle of selection of default states is the same as for one default state, but only transitions from one state are counted at one time. The principle of the default state optimisations is described in the section 6.5.1 of this work.

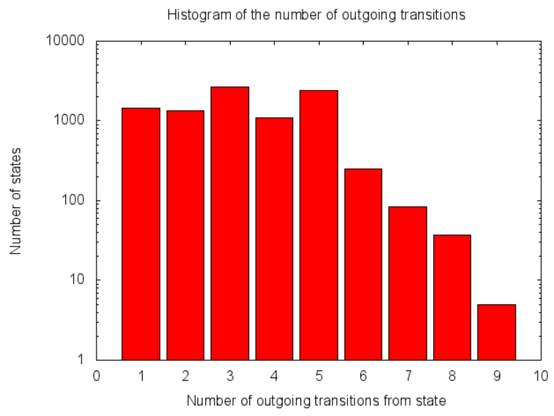
It can be seen, that the automaton *without character classes* has saturation in the interval $(0.9, 1\rangle$ for most of the patterns. By analysing the distribution of the patterns according to the number of transitions, it can be seen, that this settings produced the largest automata.

The figures 7.6 and 7.7 show that the most of the regular expressions are clustered around one or two peaks in the histograms. It is possible to compare two optimisations by comparing their peaks. The optimisation with lower number of transitions is considered more efficient because it can pack more regular expressions into smaller automata. According to this comparisson, the *one default state* is slighty more efficient than the method *without character classes* for the Snort dataset. The histograms for the L7 dataset present significant reduction of the automata *with one default state* optimisation. For this optimisation, automata are distributed in the whole interval. It means that the optimisation reduces the size of automata and produces automata with lower saturation.

The *many dafault states* shows further reduction of the automata in both dataset. It



(a) Snort ruleset



(b) L7 ruleset

Figure 7.8: Histogram of outgoing transitions

can be seen from the figure 7.4 that most of automata created from Snort dataset have saturation between 10 and 15 percent if this optimisation is used.

The last optimisation is *optimal character class*. Automata reduceed by this optimisation are the smallest ones for the Snort dataset. For the L7 dataset, the distribution of the sizes of automata with this optimisation is similar in comparisson to the *many default state* optimisation.

The low saturation for the *many default states* indicates, that for every state s in the Snort ruleset, there is one other state q that is target of large amount of transitions that are outgoing from s. The experiment *with character classes* do not show lower saturation or significantly lower number of transitions than *many default states*. This fact indicates, that the most of the transitions from every state s in Snort ruleset lead to one state q. It can be seen, that state q is different of most of the states s by comparing *many dafault states* and *default state* optimization. These three observations indicates, that most of the states in automata have only small number of successors.

The Figure 7.8 shows the histogram of number of different outgoing transitions per state in the snort and L7 rulesets. Two transitions are considered different, if they lead to the different target state. If two transitions lead to the same state by different symbols they are considered the same in this experiment. The number of different transitions were counted for every state of every automata. The histogram 7.8 presents results of this analysis. It is important to keep in mind, that the y axis of 7.8 is logarithmic.

The experiment results presented in 7.8 proves the intuition from the above paragraph that there are only few states accessible for any given active state. The most common number of next accessible states in Snort ruleset is 2, which corresponds to the situation, where the automaton accepts just one character, or one set of characters, while all other symbols reset or stop the matching process. The typical source of this situation is accepting strings. In this case, automaton can move forward by accepting one symbol or reset the matching by accepting any other symbol. Therefore this experiments shows, that the rules in Snort ruleset contains large amount of strings. The high number of states that has only one successor is surprising because it reflect situation in which the symbol is always accepted. This situation can be created by counting. If finite automaton is used for counting, it has to have one state for every value it can count. Therefore, counting creates

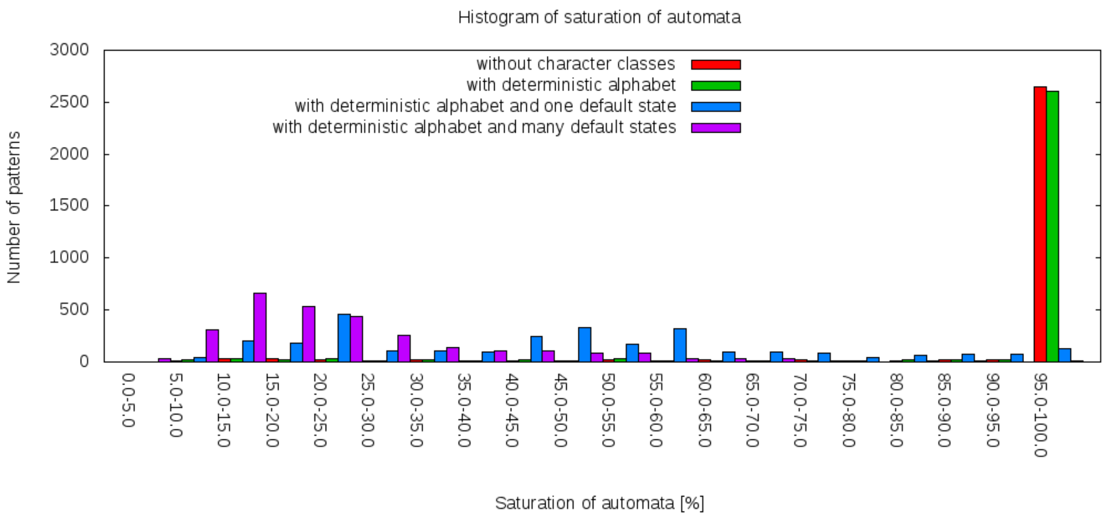


Figure 7.9: Saturation of transition table of DFA with deterministic alphabet created from Snort regular expressions. Histogram show how many patterns are with saturation presented at the x-axis. The x-axis is annotated in percentage. The level of saturation is shown for various optimisation methods.

large and inefficient automata.

The most common number of successors is 2 for Snort ruleset and 3 for L7 ruleset. Together with the number of transitions known from previous experiments, it indicates, that Snort ruleset contains large sequences of states that only counts or accepts strings. Moreover, the Snort ruleset contains states with 18 successors while L7 ruleset has state with at most 10 successors. Experiments show that the Snort contains more complex structures. However, Snort rules often contain large sections that generates very simple automata.

The histograms 7.4 and 7.5 present that the introduction of the *optimal character classes* significantly reduces the number of transitions and saturation of the automaton. However, this optimisation creates automata with nondeterministic alphabet which may result in complicated implementation. The deterministic alphabet transformation can be used to avoid tis issue. However, automata using the deterministic alphabet may be larger in general. The next analysis are focuses on the size and the saturation of automata with the optimal deterministic alphabet. The analysis was performed for the Snort and L7 rulesets. The figure 7.9 reports the saturation of automata generated from the Snort ruleset and figure 7.10reports results for L7 ruleset. The saturation of automata without character classes and default transitions is presented only for comparison and the results are exactly the same as in previous experiments. The automata with deterministic alphabet was created by algorithm 7. The histogram presents automata with three different iptimisations.

with deterministic alphabet Automata have deterministic alphabet and do not contain any default states. It can be seen, that the introduction of deterministic alphabet transformation do not affect the saturation for most of automata. The high saturation of automata is nothing unexpected because most of the original automata have high saturation, which indicates that they are fully defined or at least most of their states have defined outgoing transitions for every possible symbol. The deterministic alphabet do not have overlapping symbols therefore all symbols must appear in every

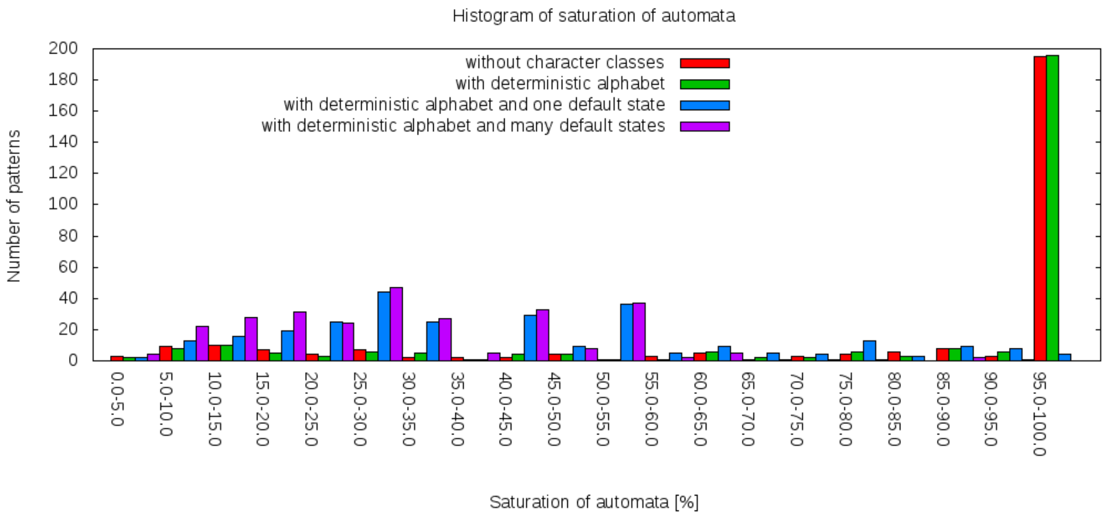


Figure 7.10: Saturation of transition table of DFA with deterministic alphabet created from L7 regular expressions. Histogram show how many patterns are with saturation presented at the x-axis. The x-axis is annotated in percentage. The level of saturation is shown for various optimisation methods.

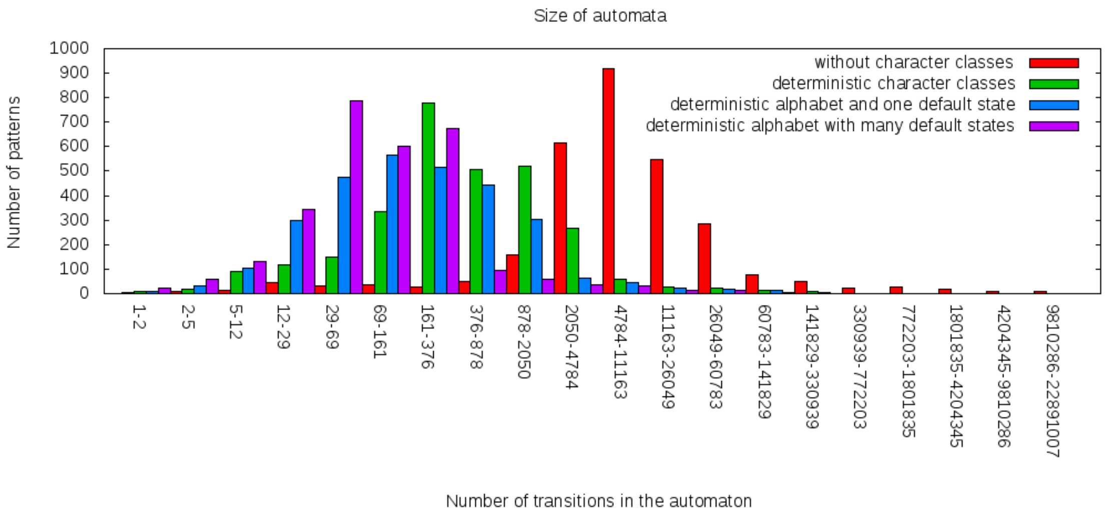


Figure 7.11: Transition size of DFA with deterministic alphabet created from Snort regular expressions. Histogram show how many patterns creates automaton with transition size presented at the x-axis. The transition size is shown for various optimisation methods.

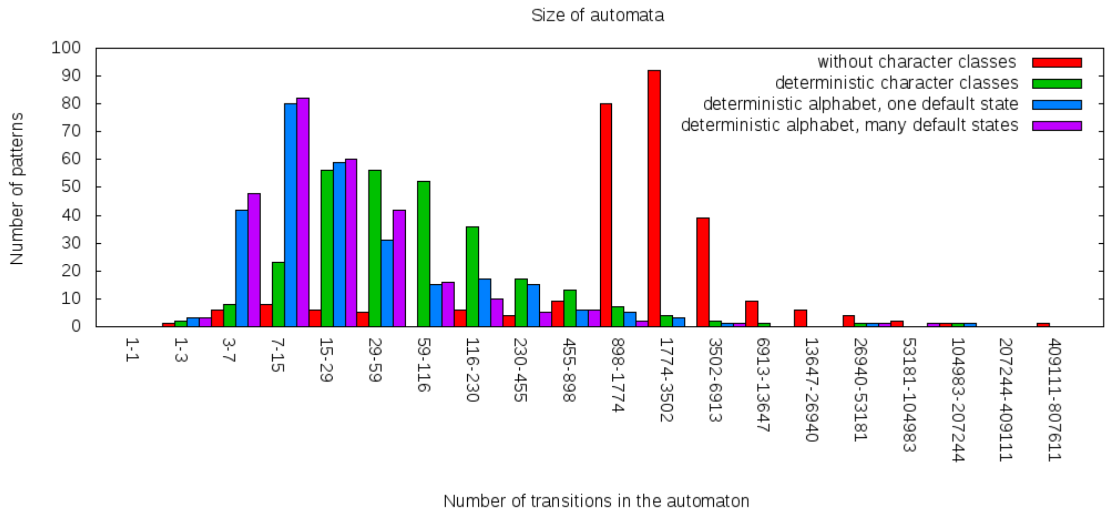


Figure 7.12: Transition size of DFA with deterministic alphabet created from L7 regular expressions. Histogram show how many patterns creates automaton with transition size presented at the x-axis. The transition size is shown for various optimisation methods.

state of fully defined automata. For the Snort ruleset, the deterministic alphabet helps to decrease only slightly the amount of automata with saturation higher than 95%. This can be seen in Figure 7.9. L7 do not indicate any reduction of the saturation of transition table by the deterministic alphabet as can be seen in Figure 7.10. However, the figures 7.11 and 7.12 shows that introduction of deterministic character classes into the automaton generates automata with much smaller number of transitions. It means that reduces the memory requirements to store transition tables for automata of both rulesets.

deterministic alphabet, one default state The introduction of the default state reduces the size of automata and produces automata with low saturation. By comparing figures 7.9 with 7.4, it can be seen, that the combination of deterministic alphabet and one default state produces automata with larger saturation then automata without character classes and with default state. However, the introduction of the deterministic character classes reduces the size of automata even with combination with default state as can be seen by comparing figures 7.11 and 7.6.

deterministic alphabet, many default states The comparison of this optimisation with only *one default state* indicates, that the *many default states* optimisation significantly reduce the saturation for the Snort ruleset. It is shown in Figure 7.9. The comparing these two optimisations according to the size of automata indicates that *many default state* optimisation reduces the size of the automaton. The results of the same experiment performed for the L7 ruleset are less convincing, due to the fact that L7 contains less automata and automata of L7 are much smaller. However, the figure 7.10 shows that there is more automata with many default states and saturation in interval from 5 to 25 percent.

According to experiments performed in this subsection, the introduction of the optimal nondeterministic alphabet produced small automata with the low saturation of transition table. Since the implementation of the optimal nondeterministic alphabet is not straightforward and is an outgoing issue, the experiments with the deterministic alphabet and default states was also performed. It was shown that, even with deterministic alphabet the optimisations can be used to produce small automata with the low saturation.

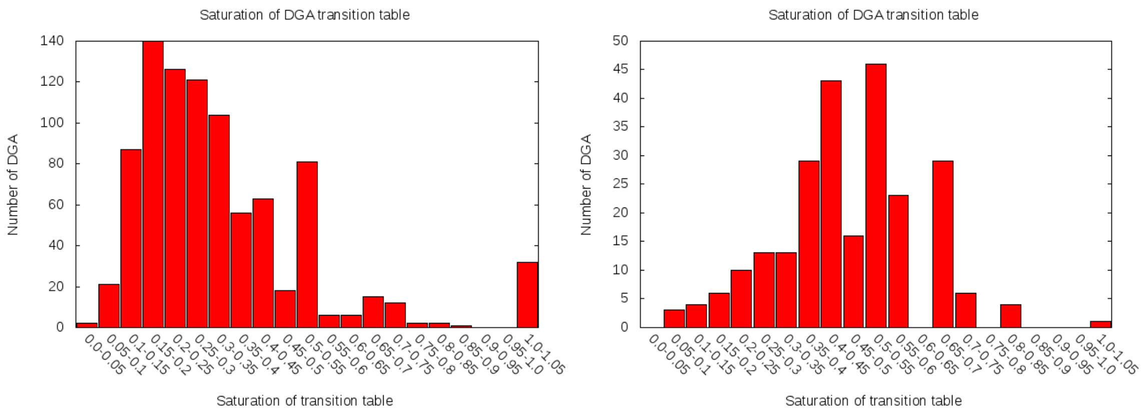
7.2.2 Saturation of multistriding automata

The previous section measured the finite automata accepting one symbol per transition. However, to achieve throughput of 10Gbps, such automaton would have to perform 1250 transitions per microsecond while every transition corresponds to the read of the memory from random address. Such speed is very difficult to obtain with modern memories. Multistriding automata offer solution to this problem because it allows to accept several symbols per transition and therefore it reduces frequency of memory accesses. For example, 4-stride automaton will require to perform memory access only once per every 4 bytes, because it performs only about 312 transitions per microsecond.

The crucial question is, whenever multistriding automata have the similar distribution of the saturation of transition table as the original automara or if the multistriding produces denser transition tables. Moreover, the behaviour of default state optimisation and many default states optimisation may be different from the behaviour seen with the finite automata accepting only one symbol per transition. This chapter focusses on experiments with analysis of the saturation table of multistriding automata.

Saturation of Deterministic Generalised Automata

Many multistriding techniques were presented in the literature, some of them described in chapter 6. The DGA is known theoretical concept but up to our knowledge, it was not applied to the field of pattern matching of high speed networks up to now. Therefore, the first experiments in this section is focussed on the DGA.

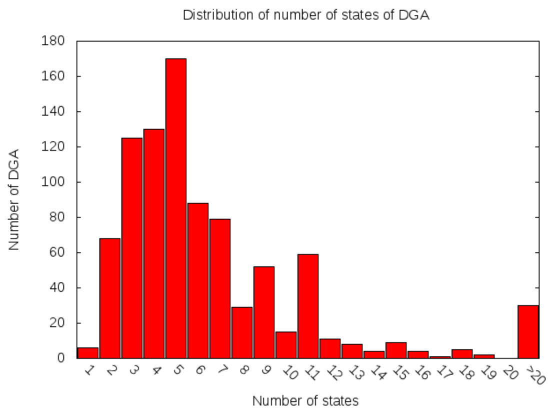


(a) Snort ruleset

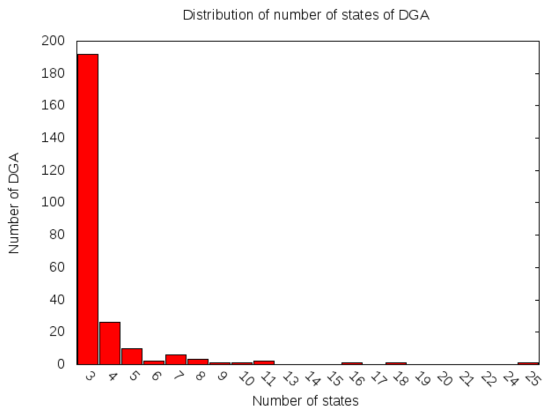
(b) L7 ruleset

Figure 7.13: Histogram of saturation of transition table of DGA

The Netbench library [98] contains the implementation of DGA construction in Python. Since it is required to perform determinimisation and minimisation of finite automata during the construction of DGA, FSM library [89] was used to speed up these operation. However,



(a) Snort ruleset



(b) L7 ruleset

Figure 7.14: Histogram of distribution of number of states of DGA

since the DGA construction requires to solve maximal acyclic subgraph problem, the time limit was set during the experiment. Processing of every regular expression was limited by 3 hour timeout. The regular expressions that were not processed in this time were considered to fail in this experiment. Another limitation given to the tool was maximal memory available. The available memory was limited to 1GB by ulimit command to allow processing of several regular expression in parallel.

The rulesets from the previous section was used also in this experiment. The figure 7.13 shows the distribution of saturation of the transition table for L7 ruleset and Snort ruleset.

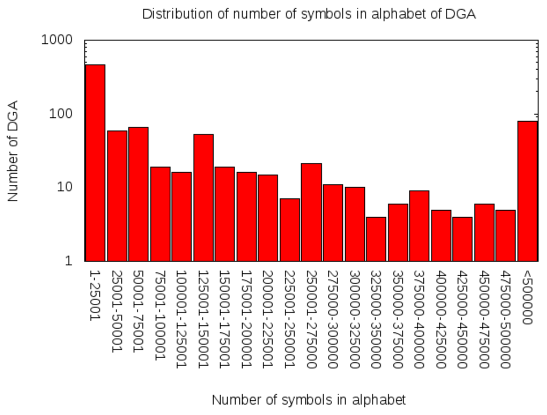
The transition in DGA can be labeled by the arbitrary string. However, it is important to note, that the alphabet of DGA do not contains strings but only the symbols that are concatenated to create string labeling the transition. If the saturation defined by definition 61 was used, it could have value much over 1. Therefore, we redefine saturation of transition table for this experiment according to the definition 62.

Definition 62. ***Saturation of transition table:*** *Saturation of transition table of DGA* $M = (Q, \Sigma, \delta, s, F)$ *is* $Sat(M) = \frac{|M|_T}{|S||\{x|x\in\Sigma^* \wedge \exists q\in Q:(q,x)\in Dom(\delta)\}|}$, *where* S *is the set of states in* M *and* Σ *is the alphabet of* M.

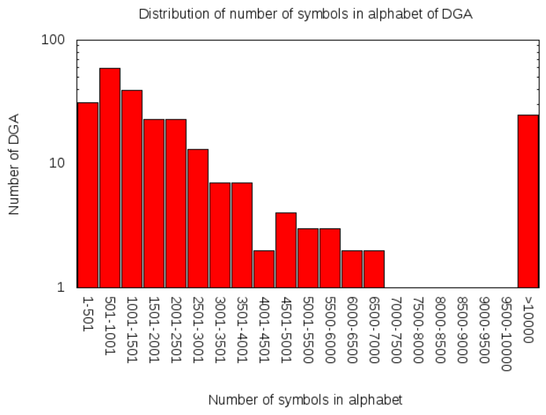
The definition 62 uses count of all unique strings used to describe transitions in the automaton instead of the size of the alphabet in the computation of the saturation. Therefore, it is possible to have meaningful comparison of the saturation of transition table of DGA with other approaches.

The distribution of saturations of transition table for L7 ruleset shows, that most of the DGA has saturation of transition table around 50%. Only few regular expression produces DGA with saturation higher than 75%.

The distribution of the saturation of transition table for Snort ruleset shows, that most successfully build DGA has transition table with saturation around 20%. The difference between the Snort and L7 rulesets lays in the complexity of regular expressions in different sets. The Snort ruleset are more complex and therefore the DGA created from these expressions has more states and every state has different set of symbols which leads to the low values of saturation.



(a) Snort ruleset



(b) L7 ruleset

Figure 7.15: Histogram of distribution of size of alphabet of DGA

The distribution of the number of states in DGA is shown in the figure 7.14. It can be seen, that the most of the DGA generated for the L7 ruleset have only three states. This indicates, that the DGA reduces the regular expression into several string consequent string matching problems.

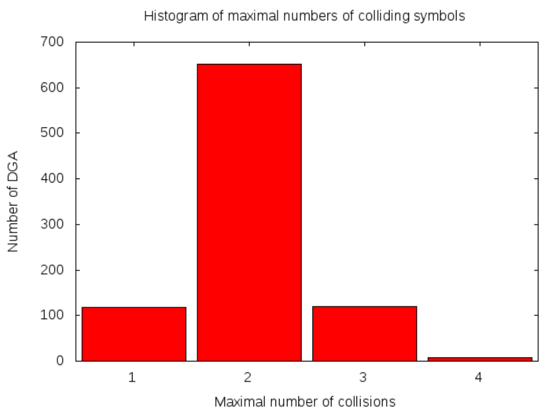
The most of the DGA created from the Snort ruleset have 5 states. However,one DGA generated from the Snort ruleset had more than 4000 states. To be able to effectively show the histogram in the figure 7.14, all DGA with more than 20 states were merged into one histogram bucket denoted as > 20.

The measurement of the number of transitions were previously used to establish the size of automata. However, for DGA we decided to measure size of the alphabet instead. The size of the alphabet allows as to analyse not only size of automata but also complexity of the alphabet transformation. For DGA, alphabet transformation actually means string matching at the required speed where we want to match all symbols of the alphabet at once. The distribution of the size of the alphabet of the DGA for Snort and L7 ruleset is shown in the figure 7.15.

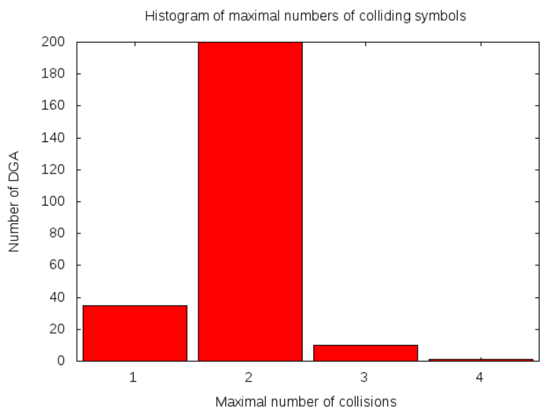
Most of DGA created from the rules in L7 ruleset have up to 1000 symbols in the alphabet and as the size of the alphabet grows, the number of DGA decreases. However, there are few automata, which have alphabet much larger. The largest alphabet has more than one million symbols.

The situation is more complicated for the Snort ruleset. Most of DGA created from regular expressions in Snort rulesets have alphabet size up to 25000 symbols. As the size of the alphabet increases, the number of automata with such large alphabet decreases, however by much slower rate than for L7 ruleset. As was the case with L7, there are also several regular expressions that generates DGA with alphabet much larger that others. All ocucrences of these regular expressions were counted together in bucket denoted as > 500000.

Described experiments indicates, that the DGA can achieve high reduction in the number of states in the original automata but the number of transition in DGA is larger than the number of transitions in the original automata. However, the DGA process more than one character per transition and therefore it achieves higher throughput. Unfortunately, it is not possible to guarante or even guess the throughput of the DGA without the knowledge



(a) Snort ruleset



(b) L7 ruleset

Figure 7.16: Histogram of collisions between symbols in the DGA alphabet

of the structure of the specific DGA and the input traffic for this DGA. However, the main advantage of DGA is that it transforms the regular expression matching into the string matching problem and the string matching in highspeed networks can be done much more effectively than matching for whole regular expressions.

The DGA guarantees that for all outgoing symbols from the specified state, maximally on symbol will accept input string. However, this guarantee do not hold for the whole DGA. Therefore, if the string matching for the DGA symbols is performed without the knowledge of the DGA state, more than one symbol can be found. The figure 7.16 shows how many DGA contains the up to given number colliding symbols. The X axis of the figure, denoted as the Maximal number of collisions, contain maximal number of symbols that collide with each other in one DGA. For example, value 2 indicates, that no matter which string is send to the input of DGA, the string matching will find at most two different symbols at once. The Y axis of the figure shows the number of DGA. It can be seen, that most of the DGA may have only two colliding symbols and that there is no DGA that would have alphabet with more than four colliding symbols. Therefore it is possible to implement DGA based matching with the string matching without any addition knowledge about the state of DGA. This allow to use any string matching technique described in the literature so far.

The experiments provided in this section indicates that the DGA may be suitable alternative for matching some of the regular expression used in the modern intrusion detection system. Unfortunately, construction of the DGA is time consuming and since the Netbench implementation is written in Python, many regular expressions from the Snort ruleset were not able to generate DGA in the given time. The overall conclusion from presented experiments is that the DGA is suitable alternative to implement pattern matching unit. However, the effective implementation of the DGA construction is necessary for further evaluation.

Saturation of transition table of pattern automaton

Experiments with pattern automaton was published by Kastil and collective [63, 67] or in czech language [68]. However, these papers dealt with the overall structure of the pattern matching unit and did not provide detailed description of algorithms used to construct the

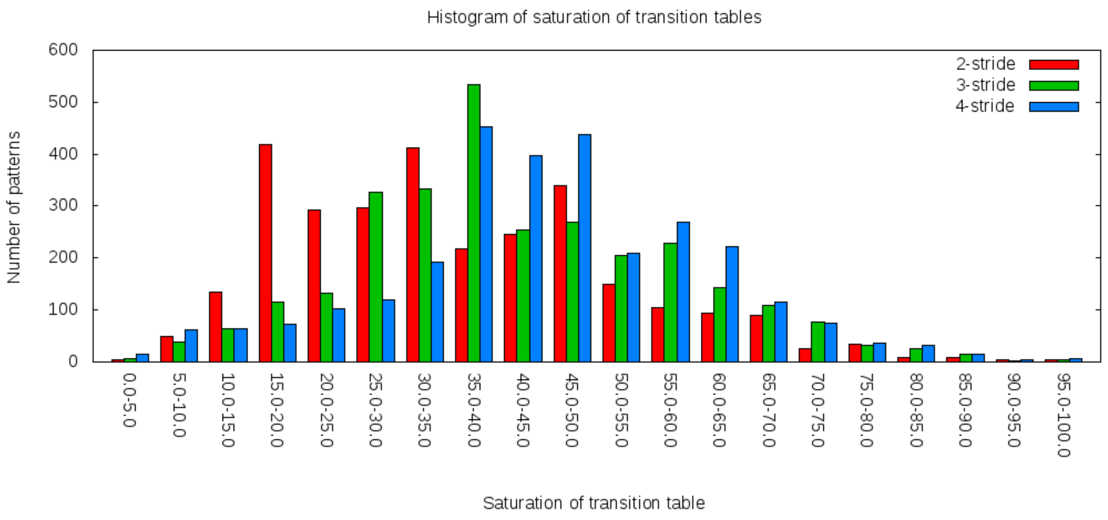


Figure 7.17: Saturation of transition table of pattern automata created from Snort regular expressions. Histogram shows how many pattern creates automata with saturation of transition table presented at the x-axis. The axis is annotated in percentages. The level of saturation is shown for various number of character accepted by one transition of pattern automaton.

pattern automaton. Also the name pattern automaton did not appear in the paper, instead more general name multichar automaton was used. Moreover, the experiments in the paper focused on small number of regular expression which were merged together.

The implementation used in this thesis is the same as the implementation used for the original papers and is available together with this thesis. The C/C++ code loads automata described in FSM file format [89]. The Netbench library [98] is used to construct non-deterministic finite automata with as large character classes as possible for every regular expression from the given ruleset and for storing them into FSM file. The next step is construction of the pattern automaton by algorithm 9. The resulting pattern automaton may have nondeterministic alphabet. The algorithm 11 is used to ensure, that the pattern automaton has deterministic alphabet. After the pattern automaton is created, the FSM library [89] is used to perform determinisation and minimisation of the automaton. Automata with *default state* optimisation and *many default state* optimisation are also created.

The process of construction of pattern automata was limited by ulimit command to use 1GB memory at max and to run no more than 10 minutes. It would be possible to increase these limits to be able to process more regular expression at the cost of increasing time required to perform whole experiment.

Pattern automata was constructed to accept from 2 to 4 bytes per transition. Figure 7.17 describes the distribution of saturations of transition table for pattern automata with *many default states* generated from Snort ruleset while figure 7.18 shows the result of the same experiment for L7 ruleset. For the results of experiments without any optimisation or with *one default state* optimisation see appendix B. It may be seen, that increasing number of accepted symbols per transition slightly increases the saturation of transition table. The saturation of transition table for Snort rules is lower than 50% for most of rules. However,

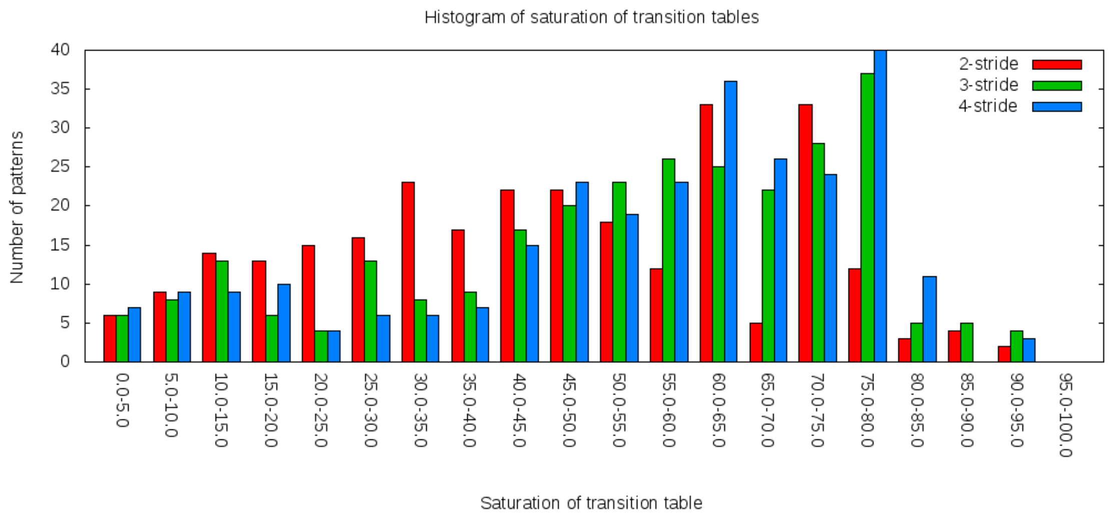


Figure 7.18: Saturation of transition table of pattern automata created from L7 regular expressions. Histogram shows how many pattern creates automata with saturation of transition table presented at the x-axis. The axis is annotated in percentages. The level of saturation is shown for various number of character accepted by one transition of pattern automaton.

for L7 rules, the saturation of transition table is higher than 50% for most of rules. This difference is caused by the different complexity of regular expression in both sets.

The second important measure about automata is the number of transitions. It is expected that the multistriding will increase the number of transitions in the automaton.

The figure 7.19 shows how the different number of symbols accepted per transition affects the size of automata generated from L7 ruleset. It can be seen, that with increasing number of accepted symbols the number of large automata also increases. It is important to note, that the runtime and available memory for every experiment was limited and therefore, several largest 4-strided pattern automata were not generated in time. While the general distributions of the size of the automaton are similar, the largest 3-strided pattern automaton is 5 times larger than largest 2-strided pattern automaton.

The figure 7.20 shows results of the same experiment performed for the Snort ruleset. Regular expressions used in Snort are more complex than regular expressions used in L7 filter. Therefore the generated pattern automata are larger for Snort ruleset. The figure indicates, that with the increase of number of symbols accepted per transition the size of the automaton also increases. The whole distribution of transition size of automata is slightly shifted to the right which indicates that the most of automata increases its transition size only by small amount of transitions and was moved by one or two buckets in the histogram. It is important to keep in mind, that due to the time and memory limitations, some regular expressions were not converted into pattern automata in time and therefore their transition size was not shown in the histogram.

Presented experiments show, that while accepting more symbols per transition may produce very large automata in worst cases, the increase of the transition size of the automaton is acceptable in most cases. Presented experiments did not measure and evaluate some of the large automata that were generated due to the time and memory restrictions.

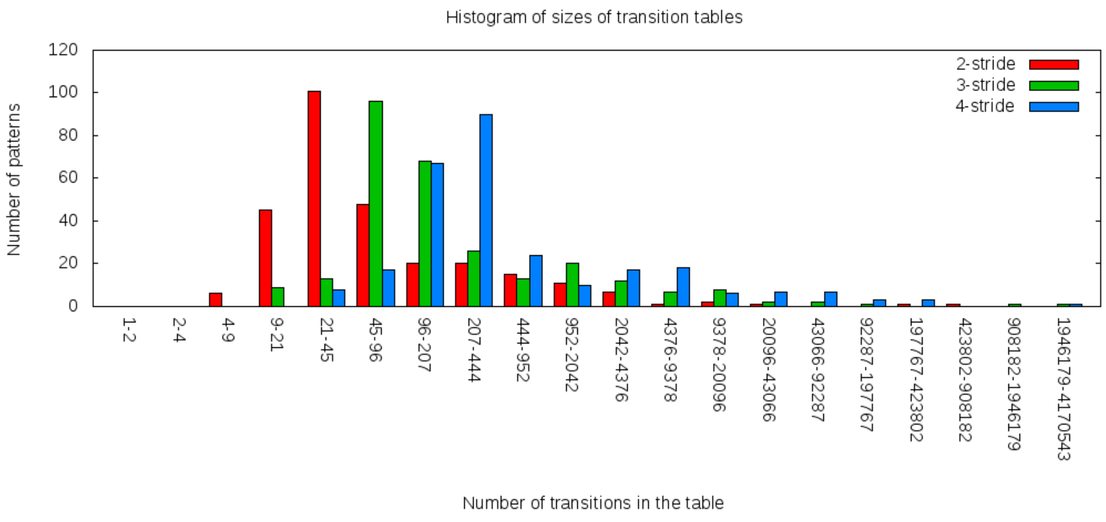


Figure 7.19: Transition size of pattern automata created from L7 regular expressions. Histogram shows how many pattern creates automata with transition size presented at the x-axis. The transition size is shown for various number of character accepted by one transition of pattern automaton.

However, there are only few of such automata[1] and these automata are not suitable for the methods based on multistrided DFA, such as pattern automaton.

Experiments also showed, that automata generated from regular expressions used in the modern highspeed networks varies in the size and saturation of the transition table. However, experiments with saturation revealed, that most of the regular expression generate automata with low saturation of transition table. These results hold true even for the pattern automata that accepts several symbols per transition.

[1]Their actual count depends on the number of accepted symbols per transition.

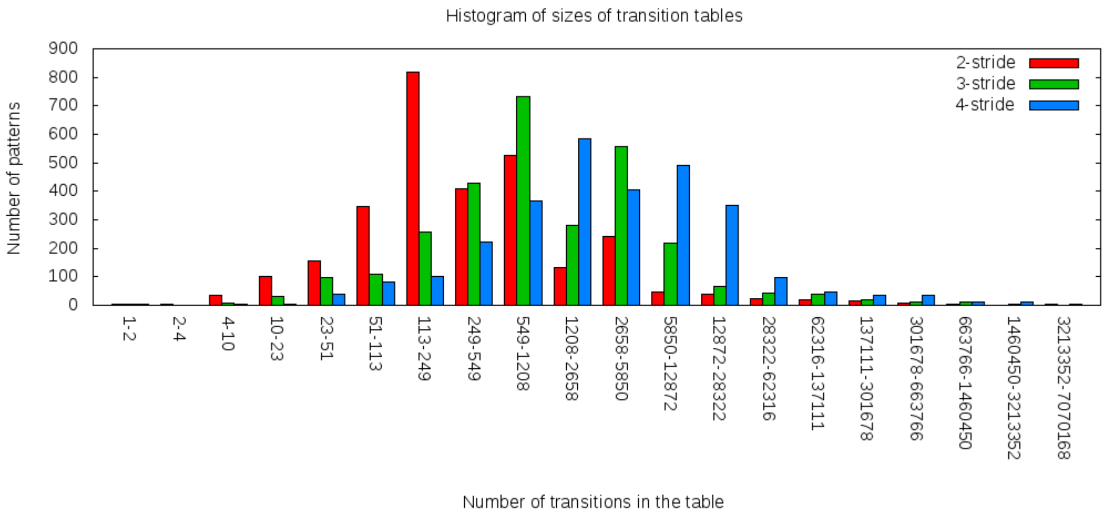


Figure 7.20: Transition size of pattern automata created from Snort regular expressions. Histogram shows how many pattern creates automata with transition size presented at the x-axis. The transition size is shown for various number of character accepted by one transition of pattern automaton.

Chapter 8

Perfect Hashing based Deterministic Automata

The analysis of the regular expression from the previous chapter showed, that most of the regular expressions used in the modern IDS generates deterministic Finite Automata with sparse transition table. Moreover, there exist other techniques that increase sparsity of the transition table. For example, DelayDFA [71] increases sparsity of the transition table of complex regular expression by removing „duplicate“ transitions. Another example of approach that may increases sparsity of the table is multistriding.

This chapter presents a novel architecture for implementation of the DFA with sparse transition table. Presented architecture is based on the use of the perfect hashing function to implement a transition table.

It is important to keep in mind that the memory access is the most time consuming operation performed by the pattern matching algorithm. The DFA based implementation of pattern matching minimizes the number of required memory access per character.

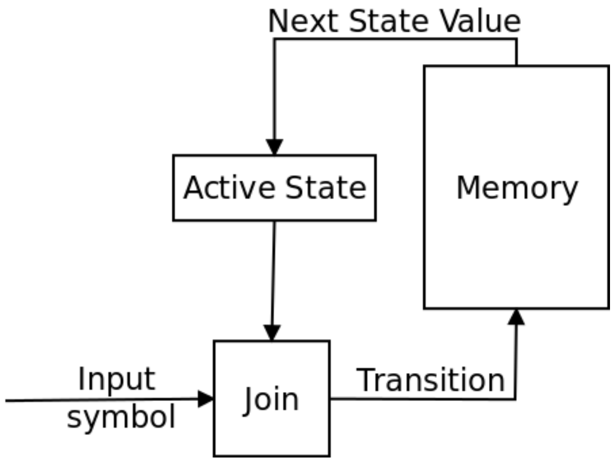


Figure 8.1: Principle of DFA implementations

The figure 8.1 shows the basic principle of every DFA based pattern matching method. When a new symbol arrives at the input of the pattern matching unit it is joined with active state to form description of the transition. This description is used for the memory look up in the transitional table to find a next state. The different implementation of DFA differs in the way of handling the memory look-up operation. The choice of the memory look up algorithm determines the speed of the most timely critical operation of all pattern matching and throughput of the whole pattern matching operation.

The problem of the fast regular expression matching by deterministic finite automata can be reduced to the problem of effective implementation of transition table. Two basic approaches for the implementation of the transition table are described in the literature.

- 2D matrix of all possible combination
- Linear array of transitions

The 2D matrix representation achieves the best results for the fully defined random transition tables. For the fully defined transition table, there is virtually no overhead. The one dimension of the matrix is described by the alphabet symbols and the second by the state number. Therefore every cell in the matrix corresponds exactly to one transition. The cell contains only the number of the next state without any additional information. The operation of joining the actual state together with an input symbol is the computation of memory index, where the cell is stored. The efficiency of the matrix representation decreases with the decrease of saturation of the automaton because cells for the nonexistent transitions are still represented in the matrix. The size of the memory required for matrix representation of the deterministic finite automaton can be computed according to the equation 8.1.

$$M_{matrix} = |S| \times |\Sigma| \times log_2(|S|) \tag{8.1}$$

For example, automaton with 10000 symbols and 10000 states requires transition table with 100000000 next state values and whole solution would require $175MB$ of memory since each next state value had to be represented at least by $log_2(10000)$ bits.

The linear array of transitions can be used for effective implementation of the automaton with low saturation. The transition table of the automaton is stored as an linear array containing only existing transitions. The linear array have to contain the next state value and additional information for identification of the transition but do not have to store any information about nonexistent transitions. Most often, the additional information is in the form of starting state and the symbol of the transition. Therefore, the linear array stores entire transitions in the form of three tuple. The memory requirements of this approach can be evaluated according to the equation 8.2.

$$M_{list} = |\delta| \times (2log_2(|S|) + log_2(|\Sigma|)) \tag{8.2}$$

Lets consider example from above. We need to specify the number of transitions in the automaton to be able to compute the memory requirements of the linear array implementation of the transition table. For example, automaton with 20000 transitions will require $105KB$ of information stored in transition table. However, automaton with 60000000 transitions will require $315MB$ of memory. It can be seen that linear array implementation of transition table can have save large amount of memory for the automata with small number of transition but large alphabet.

The correct choice of the method depends on the number of transitions in the automaton and saturation of its transition table. The number of transitions can be computed from the saturation of transition table and the size of the universe of all possible transitions in the automaton. Therefore, the equation 8.2 can be rewritten to use saturation instead of the size of transition table. The rewritten form is the equation 8.3.

$$M_{list} = (|S| \times |\Sigma|) \times Sat \times (2log_2(|S|) + log_2(|\Sigma|)) \tag{8.3}$$

It is possible to compute the saturation where both methods require the same amount of memory. If the acual automaton has lower saturation than the linear array implementation will achieve higher efficiency. However, for higher saturation it is reasonable to use matrix representation. The equation 8.6 combines equations 8.3 and 8.1 to compute the turning point, where the memory required by matrix representation equals the memory required by list preresentation. If the saturation of the given automaton is lower than the computed value, it is advisable to use the list representation of the transition table.

$$\begin{align} M_{matrix} &= M_{list} \tag{8.4} \\ |S| \times |\Sigma| \times log_2(|S|) &= (|S| \times |\Sigma|) \times Sat \times (2log_2(|S|) + log_2(|\Sigma|)) \tag{8.5} \\ Sat &= \frac{log_2(|S|)}{2log_2(|S|) + log_2(|\Sigma|)} \tag{8.6} \end{align}$$

The figure 8.2 shows results of equation 8.6 for different combination of the number of states and the size of the alphabet. The color of the figure corresponds to the saturation in percentage for which both methods require the same amount of memory. If the saturation of the transition table of the automaton is lower than the value in the figure, the linear array of transition have better memory efficiency than matrix representation.

The figure indicates, that if the alphabet of automata is small, the saturation for which is the list implementation suitable can reach almost 50%. However, as the size of the alphabet increases, the required saturation decreases. The reason for this behaviour lays in the fact, that the list implementation has to store the representation of the symbol with every transition and the smaller alphabet is represented on lower amount of bits. The increasing the state size of automata increases value of saturation for which is the list representation preferable.

The figure 8.2 shows that there exists a lot of automata that can be efficiently implemented by the linear array of transitions. However, the transition lookup in the array can be very slow since it is not possible to directly compute the position of transition without any additional information. It is possible to speed up the lookup by the ordering the array. In such case, the lookup can be done in logarithmic time complexity with the number of all transition. However, even the logarithmic complexity of the transition look up may require a lot of memory access, which would reduce the throughput. The transition look up have to be performed with the constant time complexity to guarantee throughput required to process modern networks.

Hash tables can be used to speed up the search in the array. The theoretical principles of the hash functions and hash tables was described in the chapter 5.3. The look up in the hash table have expected constant time complexity. However, if the collision (see definition 43) occurs in the hash table, the time complexity will increase. In the worst case, time complexity of the hash table look up is the same as time complexity of the look up in the linear array since all transitions can map into the same position in hash table. Therefore, the use of the hash table will reduce time required for the look up but since there is a nonzero probability of the collision it is not able to guarantee the constant time complexity in all cases.

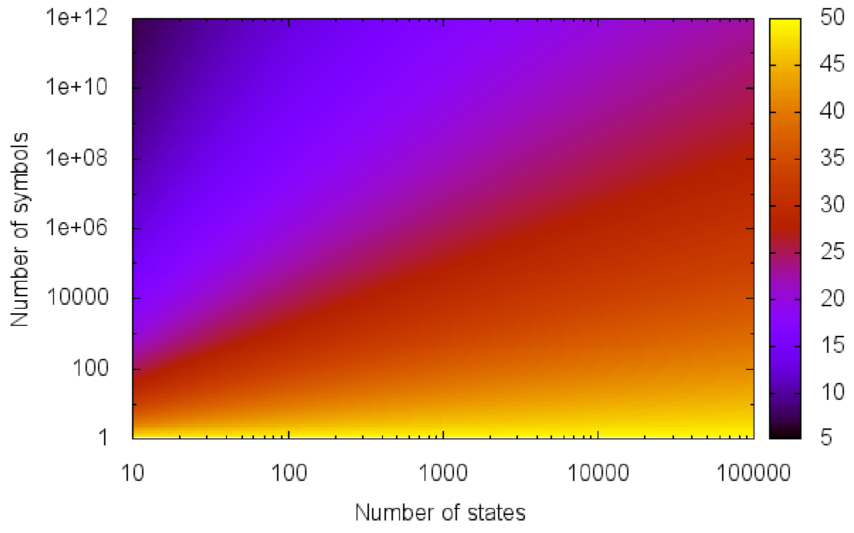


Figure 8.2: The saturation turnpoint

The probability of the collision can be surprisingly high. It is described by the Birthday paradox 5.3. Even one collision in the transition table will reduce the throughput of the matching unit since the traffic may iterate through the colliding transitions.

This work proposes to use perfect hashing functions (see section 5.5) instead of universal hash functions (see section 5.1) in the implementation of transition table. The use of the perfect hash function will guarantee

- Constant time complexity of transition lookup
- Only existing transitions need to be stored in transition table

The basic principle of the deterministic finite automaton based on perfect hashing is shown at the figure 8.3. It can be seen, that the computation of the next state can be divided into three steps.

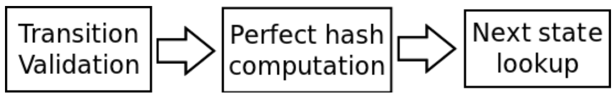


Figure 8.3: Principle of next state lookup

First step is called the Transition Validation and is designed to detect nonexistent transitions. It is important to realize that all possible input symbols can appear during the matching process. However, only several of these symbols will describe transition from the active state. All other symbols do not describe any transition and are filtered out in Transition Validation step. Transition Validation step can be reformulated as a set membership problem because we are asking if the given combination of active state and input symbol belongs to the set of transition. The correct algorithm for the set membership problem depends on properties of the set. Since this work focusses on the automata with sparse transition table, it can be assumed that the set membership problem is going to be solved on sparse sets. The discussion about the state of the art of the set membership problems can be found in the chapter 5.6. Set membership query can be performed with constant time complexity.

The second step in the figure 8.3 is the computation of the perfect hash function for the combination of active state and input symbol. Perfect hash will assign unique number to every legal combination and because transition is already validated, illegal combination cannot appear at the input of perfect hash function. Moreover, the size of assigned number is only reasonably larger[1] than the number of transitions in the automaton. The size of assigned number can be limited to the size of the transition table if the minimal perfect hash function is used. However, the minimal perfect hash function requires more resources. The current state of the art in the area of the perfect hashing can be found in the chapter 5.5. It is important to realize that the perfect hash function can be computed in the constant time complexity. The computation of the perfect hash function requires several memory lookup. The actual number of the lookups depends on the selected algorithms. The BDZ algorithm used in this work was first introduced in [21] and the details of its behaviour are summarised in the chapter 5.5. The selected algorithm requires three memory access. However, it is possible to do all these memory access in parallel.

The third and last step is the actual look up in the transition table. The position of the transition in the transition table is known from the previous step. Therefore, only one memory access is required to found the value of next state.

The proposed system has the constant time complexity since every step has only constant time complexity. Moreover it is possible to perform transition validation and perfect hash computation in parallel to further increase processing speed. In this case, the perfect hash may return undefined results for the nonexistent transitions. Such results are discarded after the transition validation detect the illegal transition.

8.1 Hardware architecture

This section focuses on the description of the innovative hardware architecture of the pattern matching unit. The previous section defined three basic steps that has to be performed. First step is the transition validation which can be reformulated as a set membership query. Since the transition has to be found with constant time complexity, the set membership problem has to be also solved in the constant time complexity. One of the best approaches for the set membership problem in constant time was published by Brodnik and Munro in [24]. They point out that the hash table with the perfect hash function is the most effective method for sparse sets. The transition validation can be done in parallel with other two steps if the incorrect results are removed afterwards. The figure 8.4 shows more detailed version of the architecture.

It can be seen from figure 8.4, that there are two block that both perform computation of the perfect hash function. There are many functions that can be used as a perfect hash function for the given set. However, it is not important which function is used as long as the requirements of the perfect hashing are fulfilled. Therefore, it is possible to merge these

[1]The actual size of the output interval depends on the algorithm used for construction of PHF and its setting.

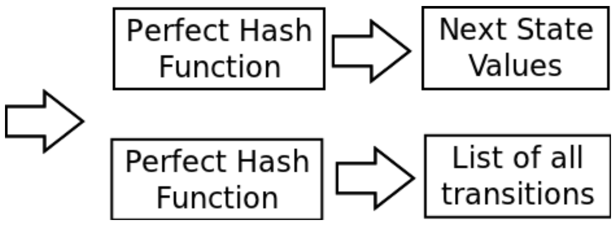


Figure 8.4: Perfect Hashing Automaton

two blocks into one perfect hashing function. The same is possible for the two memories that store the next state value and the set of transitions. The transition store in the list of transitions contain only the starting state and the symbol. Therefore there is no duplicity in memories.

The BDZ algorithm is based on the assumption that for every key it is possible to select one hash function, that will point to its location in the hash table. The problem is how to select the correct hash function out of three. To solve this problem, algorithm stores two bit information to every key. It is possible to compute the index of the *correct* hash function these bits. The actual value of these bits is selected during the preprocessing step and the hardware architecture do not change these bits. It performs only the computation to select correct hash function.

The figure 8.5 shows one line in the memory of the transition table. The line contains three fields described above. The whole line can be loaded in one clock cycle in FPGA implementations. However, even if the FPGA implementation of the memory may work in one cycle, the pipelining may be preferable to achieve higher clock frequency.

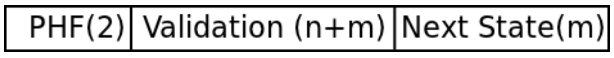


Figure 8.5: Structure of memory line

Numbers in the bracket in the figure 8.5 represents the size of the field in bits. The field for computation of perfect hash function requires always only two bits. The size of next two field depend on the configuration of the matching unit. The value of m depends on the number of state in the maximal automaton that can be implemented by the unit with specified configuration and n depends on the number of symbols in the alphabet of this automaton.

Basic principle of the perfect hashing automaton can be seen from the figure 8.6. Active state is concatenated with the input symbol in join block and used as the key for three indepent universal hash functions. Results of hash functions are used as addresses into transition memories. It is important to realize, that every hash function has its own part of the memory and therefore there are no memory access collisions. Therefore, three memory lines are obtained simultaneously. PHF field from every memory line is used as an input to the PHF block which select correct memory line from the three obtained candidates. The validation information for the transition is stored in the second field of selected memory line and it is used as an input to the validation block. Validation block compares the validation information with the inputs of the universal hash functions. If values are equal, selected memory line contain specified transition. Otherwise, the transition is not defined in the automaton. There are two possible actions for the non defined transition. If the automaton contains default state, the default state will became a new active state. However, if the default state is not present in the automaton, the matching process is terminated as there is not longer any possibility to accept input string. The selection of the correct next state is the responsibility of the next state multiplexor, which select the actual value of the next state according to the result of validation process.

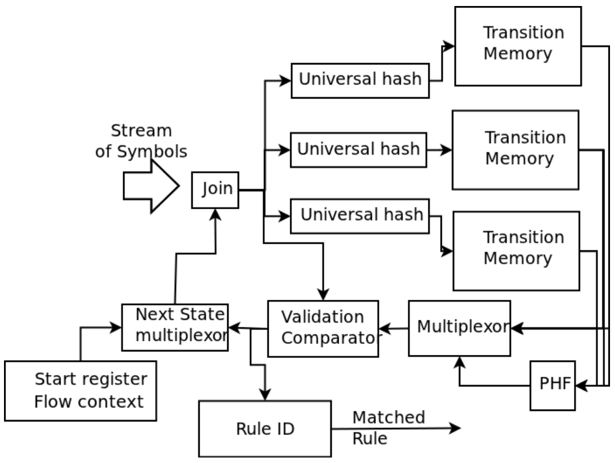


Figure 8.6: Perfect Hashing Automaton

Hardware implementation of the proposed architecture is straightforward. It is important to keep in mind that the throughput of the matching unit depends on the frequency of the perfect hashing and the memory look-ups.

The pipelined implementation of the matching process is hard due to the sequential nature of the deterministic automaton. The current memory look-up depends on the results of previously performed memory operation. The parallelization of the Deterministic Finite Automata still remains to be solved. However, it is possible to run several matching units in parallel on the different part of the network data stream. The section 8.2 describes one possible way to implement parallel network flow processing.

It can be seen that computation of hashes is the most complex operation in the matching process. In software implementation of the perfect hashing the output interval of universal hash functions are limited according to the number of keys to reduce memory usage. The limitation is often implemented by the operation modulo by an arbitrary number which depends on the number of transition in the automaton. The modulo operation is relatively cheap at the modern CPUs. For example, the Intel Core architecture can perform 32bit division by IDIV instruction [36] in time of 11 - 21 clock cycles [35]. The divider in FPGA requires a lot of resources and time. It is possible to use specialised algorithm to compute only modulo operation. Even if the implementation of modulo operation is simpler than division, it still requires considerable amount of resources and time. The comparison of four approaches for implementation of a modulo reduction was presented in [41]. It is important to keep in mind the difference between the frequency of FPGA and CPU. The modern CPUs works at the frequency of 4GHz, while the 200Mhz is considered very fast design in the world of FPGA. Therefore, increasing latency by few clock cycles of the matching unit in FPGA has much bigger impact on performance than adding few steps to the computation on the modern CPU.

The transition table will be implemented by dedicated memory block. Even if the transition table is smaller than size of the dedicated memory block, it will not be possible to use remaining part of the memory block for other purposes without the implementation of complex memory management unit. Implementation of this unit would significantly slow down the system. Therefore, the unit for computation of the modulo may compute only modulo by constant operation, where the constant is the size of available memory and not the number of transitions in automaton. It could be possible to use saturating arithmetics [42] instead of the modulo operation. In this case, if the value of hash function exceeds maximal allowed value, maximal value is returned instead of original value. This method is simple to implement but it invalidates uniformity of the hash function.

It can be seen from the figure 8.6, that very one of the three universal hash functions has its own dedicated memory block. Therefore, result of every hash function has to be reduced by its own modulo operation. If the size of the memory is 2^k, the modulo may be implemented simply by ignoring higher bits of the hash value. Therefore it may be reasonable to set size of all memory block to be 2^k.

This implementation of the given algorithm decreases of the memory utilisation by increasing the constant c described in 5.5. The increasing of constant c has positive effect on the probability of finding perfect hash function. It is important to keep in mind that the additional memory would be wasted in hardware, since only pattern matching unit is connected to this memory block.

8.1.1 Universal hash function

Throughput of proposed perfect hashing automaton is determined by the time required to compute universal hash value and the time required to perform memory access. The time for memory access is the property of the given memory and can not be changed by the design. Therefore, reducing time required for the hash computation is necessary to increase the throughput of the system. There is a lot of implementations of the universal hashing which differs both in their quality and in the time they require to compute the result. The more detailed description of the different families of universal hash function is in the chapter 5 of this work.

The selected perfect hashing algorithms relies on the universality of hash functions to guarantee that the perfect hash function will be found in a reasonable time. The universality of hash function can be seen as a statistical property. In reality, there will alway be a set of keys, that will produce non-uniform distribution of hash results due to the collisions. Therefore, the algorithm for finding perfect hash function has to be able to cope with the hash functions which are not „fully“ universal. The selected perfect hashing algorithm have a probabilistic nature. There is a given probability that one iteration will find correct perfect hash function. This probability was computed under an assumption, that used hash functions are universal. If the non universal hash function are used, the probability of finding perfect hash function will be reduced and the algorithm will require more iterations to find perfect hash functions. Since the perfect hash function has to be found in the preprocessing step, it may be reasonable to spend more time in the finding perfect hash function in exchange for faster matching. This trade-off can be addressed by selecting the universal hash functions from figure 8.6. The more complex hash functions will required longer processing time but will return more uniform output distribution. The more simpler implementation of hash function will enable faster matching process at the expense of the more iteration during the preprocessing step.

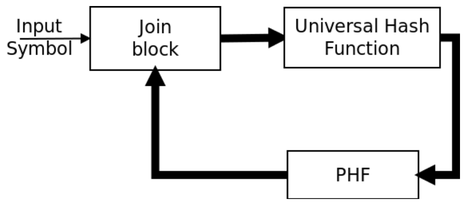


Figure 8.7: Pipelining in Perfect hashing

The figure 8.7 describes simplified version of the architecture. The bold line symbolises loop, that has to be performed for every input symbol. It is not possible to increase throughput of this loop by pipelining because the join block requires the output of the loop from previous iteration as its input. Therefore this loop is called critical loop further in this work.

Composed Perfect Hashing

Composed hashing is the method designed to reduce the number of operation performed in the critical loop. The previously described architecture computed universal hash functions for the whole combination of the active state and the input symbol. Therefore computation of the hash function have to be started after the end of previous transition. However, if the solution uses multistriding method based on the alphabet transformation, the input symbol is represented on more bits than the state. It is possible to start computation of the universal hash function only with the input symbol and add state information in later steps of hash function. In this scenario, only the end part of the hash function is in the time critical loop. The principle of this pipelining is shown in the figure 8.8

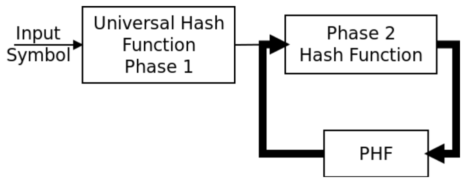


Figure 8.8: Pipelining in Perfect hashing

The bold lines in the figure 8.8 symbolises the critical loop. The computation of hash function is divided into two phases. The first phase is computed only on the input symbol and therefore it can be computed before the time critical loop. The second phase is computed on the result of the first phase and the active state. This phase is in the time critical path. However, second phase can be much simpler than whole universal hash function. The input symbol is represented by higher amount of bits in the multistriding automata and therefore the computation of the first phase is often more complex than the second phase.

Experiments done in this work suggest, that simple xor between the active state and the output of the first stage can be used as a second stage of the hash function. Several bits of the state representation had to be used twice since the output of the first stage of the hash function has more bits then state information. The xor function is implemented by the LUT in the FPGA or even by specialized fast logic. Therefore if the xor function would not be sufficient, it could be replaced by any other combination function that can be generated by one layer of LUT without the any additional time requirements.

The non-uniformity of the composed hash function may prolong run of the algorithm searching for the perfect hash function but it will not affect the actual matching itself. The experiments comparing the implementation of the composed hash function with ordinary jenkins hash function is presented in 9.1

8.2 Concurrent flow processing

Together with the multistriding, proposed architecture achieves multigigabit throughput per flow. Fast processing per flow is necessary to be able to check large data streams for the virus traffic. However most of the traffic on the core networks consists of high number of slower flows. It is required to process all of these slower flows concurrently and with minimal additional delay. It is impossible to implement thousands of pattern matching units to search in thousands of flows. Moreover, most of matching units would not be used since the most of searched flows are much slower than matching units.

The switching of context of the matching unit is an elegant solution for the problem. The proposed solution is based on the deterministic finite automaton and therefore it is possible to switch context just by changing the active state of the unit. The pattern matching system has to cooperate with the flow context unit since state of the unit has to be stored together with other information about flows.

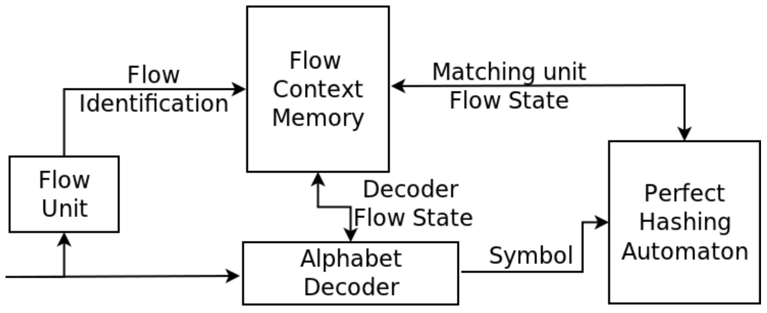


Figure 8.9: Basic principle of context switching

The figure 8.9 describes the basic principle of the switching of context of the pattern matching unit. When new packet arrive, the flow ID is computed from its header. The flow ID is used to retrieve informations associated with the flow. Flow information contain the state of the alphabet decoder and the active state of the automaton. These states are downloaded to the alphabet decoder and the automaton and the packets payload is processed. After the end of packet processing, the state of the alphabet decoder and pattern matching automat are stored back in the flow context memory.

Core networks typically consist of the traffic of many users and therefore they contain mix on hundreds and thousands flows [73]. Therefore the effective context switching is one of the most important requirements on the modern pattern matching units. Moreover, switching has to be performed in one clock cycle without introducing any additional latency to minimize the effect of the switching to the throughput of the solution. The context switching can be performed after every packet in the worst case. The worst case may appear on the backbone networks, which aggregates many different network communications or it may be caused by malicious user who tries to hide from IDS by denial of service type attack.

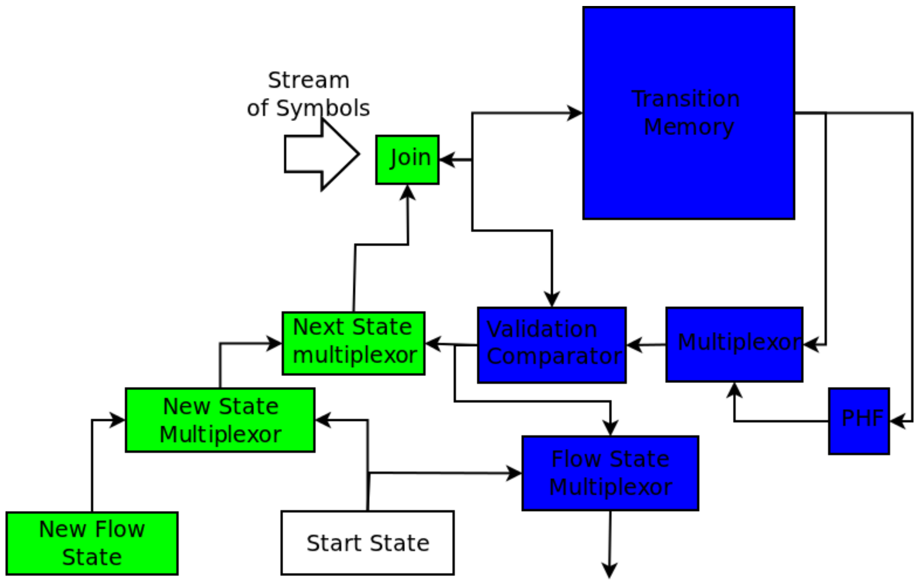


Figure 8.10: Two flows during context switching

It can be seen from the figure 8.10 that the next state value has to go through next state multiplexor for the purpose of the implementation of resetting the automata. When the next state value is not found in the transition table the next state multiplexor uses default value in the start register. Implementation of the context switching is done by the extension of this multiplexor. If the signal for the context switch appears, the next state multiplexor will set the next state value of the another flow into the active state. Thank to this technique the context switching will not add any latency to the matching process since the next state multiplexor was already in the time critical loop. To be able to switch back to the current flow the pattern matching unit has to be able to store its state just before context switching command.

The state returned by the pattern matching unit during context switching has to be value returned by the next state multiplexor. Returning directly the output of the transition table is wrong because the actual transition does not have to be in the transition table. There have to be two next state multiplexors be able to return the output of the next state multiplexor and concurrently perform context switching. Fortunately, the second next state multiplexor is outside of the critical loop.

The storing of the state of the pattern matching unit has to be performed by the external flow management which is also responsible for the TCP reordering. In this work we assume that input packets will arrive in the correct order from the network. This work do not discuss the packet reordering but any reordering algorithm can be used by the flow management without affecting the pattern matching ability of the presented unit.

It is important to keep in mind, that the alphabet decoder may have its own state and it may be necessary to store this state in the flow memory as well.

8.3 External memory

The presented approach utilises on-chip memory because its low latency and high throughput. The extremely high throughput of the on-chip memory is obtained by the massive parallelism where one FPGA can contain hundreds of the small memories. For example, largest Virtex6 FPGA contains 1064 BlockRam components[4]. However, actual size of the on-chip memory is limited and the size of transition table depends on the size of available memory. This limitation negatively affect the size of the set of regular expressions that can be downloaded into the unit. Sets of regular expressions used in modern intrusion detection systems have several thousands regular expressions. It is possible to solve the memory problem by the utilising large off chip memories, such as QDR. However, accessing the off chip memory introduces long latencies and therefore reduces the throughput of the matching process.

As described in the previous section modern network traffic consists of the large number of flows that have to be matched simultaneously. Therefore it is possible to hide the memory latency by the switching between flows. The switching mechanism described in the previous section is suitable for this purpose since it does not introduce any additional latency to the matching process.

The number of concurrently processed flows is the same as the latency of the off chip memory. Matching unit switch the context every clock cycle just after computing the universal hash functions pointing into the transition memory. When memory look-up finishes the result of the memory look up is stored into the actual state register by the context switching mechanism.

The main drawback of the proposed solution is the reduction of the throughput per flow. Even if the overall throughput of the unit is still high some packets of extremely fast flow may be dropped. Second disadvantage of the method is more complex control mechanism. The matching unit has to contain fifo of symbols that are actually processed to be able to perform validation of the transition. The size of fifo depend on the latency to the off chip memory since fifo contain exactly one record for every waiting memory transaction. Moreover, every concurrently processed flow has to have its own input buffer.

It may be reasonable to combine off-chip and on-chip memory to increase throughput. The most of the traffic on the network is not malicious and therefore it will not match any of IDS patterns. As the result, the active state of the automaton will be one of the first states during most of the time. Therefore the several of the most frequently used states may be stored in the onchip memory and other states in the off-chip memory. It is possible to see the on-chip memory as a simple cache. The only difference between this on-chip memory and classical cache is that the on-chip memory does not have any replacement policy and it is filled before the matching begins. The absence of the matching policy allows simpler and faster implementation.

The transition table indexed by the perfect hashing function requires random access and therefore it is not possible to place all interesting transitions into one continuous memory block. As the results, the on-chip memory requires complex mechanism for detection cache hits. This mechanism requires resource and time. Therefore the throughput of this solution is lower than the solution with only on-chip memory even if all used transitions are stored in on-chip memory.

The word size of off chip memories are often much larger than required by the pattern matching unit. Most of the off chip memory will be unused if the transition table is just stored in off chip memory instead of on chip memory. For example, four 36 bit wide words are associated with every address in the QDRII used on the NetFPGA board [128]. However, even the largest automata do not have transition tables wider than 40 bits which means than at least 104 bits will be unused. To deal with this problem we propose to store several lines of the transition table in one word in off chip memory. The final selection of correct word can be performed inside the FPGA. It is important to keep in mind that the FPGA logic will have to perform some form of the additional address translation between position in the transition table and in the off chip memory.

Simple implementation of the address transformation is division. Lets suppose that the width of transition table is 40 bits and the width of the QDRII memory is 144 bits. The constant $c = \frac{144}{40}$ is the number of transitions that can be stored in one line of the off chip memory. If the p_1 is the position of the line in the transition table than $p_2 = p_1/c$ is the address of the line in off chip memory which contain the transition. Since the line p_2 contains several transitions, correct one has to be selected. Index $i = p_1 mod c$ is the index of the transition in the returned line. The basic problem of this solution is complexity of the division by the arbitrary constant. If the $c = 2^k$ where k is natural number that the division and modulo can be implemented as a simple bit shifts. However, c has value of 3 in our example and therefore it would require full implementation of divider.

The presented example filled the off chip memories by line. If the memory is going to be filled by columns the c would be equal to the size of the column which is often power of two and therefore it could be implemented without the need for the divider. However, since the size of one memory line does not have to be divisible by the columns width the last column will be smaller. The position in this column has to be computed by multiplication by the constant factor.

The figure 8.11 demonstrates the difference in the memory usage of described approach. However, if the transition lays at the border of two memory lines than both approaches requires two memory operations per transition which reduces the throughput. It is possible to use probability automaton described in the section 8.4 to reduce the width of transitions fit them into one memory line.

Another approach would be to use l-perfect hash function which allows to have at most l collision for the given address. The implementation based on l-perfect hash function would require large changes in the pattern matching unit. If the perfect hashing is used the matching unit does not have to have any knowledge where the transition table is stored and it is possible to switch between several memories during the matching process.

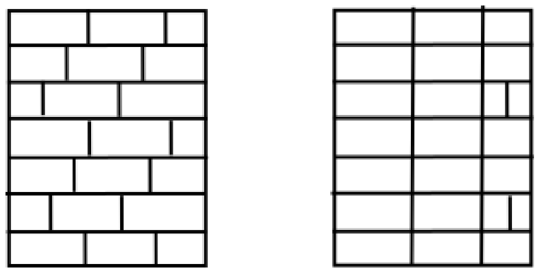


Figure 8.11: Ordering of transition in large memories

8.4 Probabilistic Automaton

The basic problem of the described architecture is the size of required memory. More than $\frac{2}{3}$ are used to store informations required to validate existence of the transition in the automaton. Validation of the transition is the set membership query in constant time. Brodnik [24] noted that constant time set membership queries can be efficiently implemented by the perfect hashing. Our architecture allows to use one perfect hash function for validation purpose together with obtaining the next state which reduce overhead traditionally induced by perfect hash function. Therefore, to reduce memory requirements we have to lessen our requirements on the matching unit. There are two basic requirements on the validation process. Constant validation time is the first requirement on the validation process. Moreover, it is required, that transition is validated in one period of system clock because speed of the validation limits throughput of entire matching unit. Second requirement is to validate without any false positives or negatives. Probabilistic automaton described in this section lessens the second requirement and allow small probability of false positives in the validation process to reduce memory requirements of the whole pattern matching unit. The idea of perfect hashing automaton with faulty transition table was first published in [62].

8.4.1 Problem statement

A Probability Automaton trades small probability of the failure in the matching process for reduction of the size of the memory required to store transition table. Therefore, every transition from active state has small probability that it will be realised even if it is not labeled by the input symbol. If it is possible to realise an existing transition in the automaton then the correct transition is realised. However, if no transition from active state is labeled by the input character, automaton may perform any other transition. Probability automaton is not able to realize more than one transition per symbol and therefore, there will be always only one active state.

The probabilistic automaton is defined as follows:

Definition 63. *The probabilistic finite automaton is 8-tuple $M(Q, \Sigma, \delta, D, P, p, s, F)$, where*

- *Q is a finite set of states*
- *Σ is an input alphabet such as $\Sigma \cap Q = \emptyset$*
- *δ is a function $\Sigma \times Q \to Q$*
- *D is a default transition function $Q \to Q$*
- *P is a transition function of the automaton $\Sigma \times Q \times R \to Q$ such as:*

$$P(\sigma, q, p_r) = \begin{cases} \delta(\sigma, q) & (\sigma, q) \in dom(\delta) \\ D(q) & rand(0-1) > p_r \wedge (\sigma, q) \notin dom(\delta) \\ rand(q) & otherwise \end{cases}$$

- *p is a probability of failure*
- *$s \in Q$ is the start state*
- *$F \subseteq Q$ is a set of final states.*

This definition of the probabilistic automaton is the extended version of the definition of the deterministic finite automaton. The function D defines the default transitions as described in section 6.5.1 If the transition is not in the definition domain of the δ then the default transition is performed. The main extension is that the transition function P is probabilistic function. There is a nonzero probability that the random state will be returned instead of returning value of D. This probability is the parameter of the automaton and is called failure probability. The function *rand* is the random number generator with uniform distribution. The *rand* function generates the outputs from the set given as its parameter. It is important to keep in mind that if there is a transition in δ then it will be always realized.

The process of the accepting characters from the input stream is the same as for the deterministic finite automata.

The transition function P can be implemented as a transition function of the deterministic finite automaton with the presence of the faults. It is shown in previous chapter that the implementation of the next state logic of DFA is divided into two subproblems, next state selection and transition validation. The transition validation detects, if the next state returned from the next state selection should be used or if the default state should be loaded instead. If the validation step makes false positive[2] mistake, the wrong next state is selected as a new active state. Since the perfect hash do not have any guarantees on which line is returned for the nonexistent keys, the returned next state can be any state from the automaton. Therefore, the behaviour of the probabilistic automaton can be implemented as a deterministic finite automaton with faulty transition validation. The faulty implementation of the validation process can be used to reduce memory requirements of the validation unit and therefore reducing overall memory consumption of the pattern matching.

8.4.2 Imperfect hashing vs Bloom filters

The best known implementation of validation block for sparse automata is the hash table with perfect hash function storing the list of all transitions. However, it is well known that allowing faults in the search can reduce memory requirements. Bloom filters [18] are structure that is often use for the set membership queries with faults. The bloom filters allows only false positives during the search. For more detailed description, see section 5.6.

The imperfect hashing described in section 5.6 is other method often used for the implementation of the set membership queries with presence of faults. The basic principle of this method is to hash the key before storing it into the list of all keys. Hashing the key allows to reduce the amount of the memory required to store the key. This section compares bloom filters and imperfect hashing according to their efficiency and implementation complexity.

The input parameters for the comparison is the number of bits used to represent the key and the number of all valid keys together with the required probability of fault. From these values it is possible to compute the size of the most effective bloom filter representing this set.

Bloom filter is the structure of m bits array and k hash functions used to represent sets. If the new key is inserted into the set represented by the bloom filter, k bits is set to value 1. Positions of bits are decided by the k hash functions. Therefore the probability, that given bit is set to zero after inserting n keys can be computed according to the equation 8.7 as was stated in [22].

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} \tag{8.7}$$

When the bloom filter is used to perform set membership query, it computes all k hashes for the given key and test if all of bits on the specified positions are set to 1. If any of

[2]Detect nonexistent transition as transition from the automaton.

hash functions return an index to bit with zero value, it is sure, that this key is not in the set. However, if it is possible that all k bits are set to 1 for the key that do not belong into the set due to the fact that the given k bits were set by several different keys in the set. Therefore, bloom filter may have false positive. The probability of the false positive depend on the probability that the specific bit is set to 1 and the number of hash functions an it can be computed according to the equation 8.11

$$fp = (1-p)^k \tag{8.8}$$

$$fp = \left(1 - e^{\frac{-kn}{m}}\right)^k \tag{8.9}$$

$$fp = e^{\ln\left(\left(1-e^{\frac{-kn}{m}}\right)^k\right)} \tag{8.10}$$

$$fp = e^{k\ln\left(1-e^{\frac{-kn}{m}}\right)} \tag{8.11}$$

The probability of the faulty transition is the parameter in the faulty automaton. Therefore, the efficiency of the bloom filter is not measure by achieved false positive rate but rather by the memory and the number of hash functions that are required for specified false positive probability. It is possible to reformulate the equation 8.11 to compute the memory size required to achieve false probability given as a parameter. The resulting equation is 8.17.

$$\ln(fp) = k\ln\left(1 - e^{\frac{-kn}{m}}\right) \tag{8.12}$$

$$\frac{\ln(fp)}{k} = \ln\left(1 - e^{\frac{-kn}{m}}\right) \tag{8.13}$$

$$e^{\frac{\ln(fp)}{k}} = 1 - e^{\frac{-kn}{m}} \tag{8.14}$$

$$e^{\frac{-kn}{m}} = 1 - e^{\frac{\ln(fp)}{k}} \tag{8.15}$$

$$\frac{-kn}{m} = \ln\left(1 - e^{\frac{\ln(fp)}{k}}\right) \tag{8.16}$$

$$m = \frac{-kn}{\ln\left(1 - e^{\frac{ln(fp)}{k}}\right)} \tag{8.17}$$

From the equation 8.17 can be seen that the size of the bloom filter depends on the number of keys in the set, number of hash functions and required probability of the false positive. The number of keys in the bloom filter is the same as the number of transition in the automaton and the probability is of false positive is the given parameter. The only remaining unknown variable is the number of hash function. The number of hash function is limited by the available number of resources.

If the minimal perfect hash table is used to store all members of the set, the required memory is given as a memory used for the implementation of minimal perfect hash function and the size of the set. It is possible to reduce memory for the representation of the set by storing only hashes of members. If this memory reduction scheme is used, it is possible that false positives appear in the set. The false negatives are not possible. The probability of false positive is given by equation 8.18.

$$f_p = \frac{1}{2^s} \tag{8.18}$$

The variable s in equation 8.18 represents number of bits that are used to store hash of the member of the set.

The minimal perfect hash function divide whole universe of possible members into as much buckets as there are members of the set in such way, that every bucket contains exactly one member of the set. Every bucket contain one member an several nonmembers of the set. We assume, that minimal perfect hash function is uniform and therefore every bucket has the same size.

When the membership query is performed, minimal perfect hash select one bucket, that could contain the given key. Selected bucket stores the hash value of the original key. The hash of the key is computed and compared with the value assigned to the bucket. If these values are equal, the key is considered to be the member of the set. Since the hash function always assign the same value to the same key, there may be no false negative. The false positive is generated by collisions in the hash function for selected bucket. The equation 8.18 is the probability of the collision of the hash value with the hash of the original key stored in the bucket.

To compare the this methods with bloom filters, the size of the whole set representation for the given set and probability requirements is necessary. The equation 8.19 computes the number of bits that is required to represent the hash value assigned to the every bucket in order to achieve false positive with probability f_p.

$$s = \lceil \log_2 \left(\frac{1}{f_p} \right) \rceil \tag{8.19}$$

To compute whole memory requirements of the hash table with perfect hashing, it is necessary to sum the memory requirements of the perfect hash function together with the memory required to store the presentation of all the members of the set. Equation 8.20 computes the whole memory requirement of the hash table with the minimal perfect hashing and possible false positives.

$$m = n \times \left(h + \lceil \log_2 \frac{1}{f_p} \rceil \right) \tag{8.20}$$

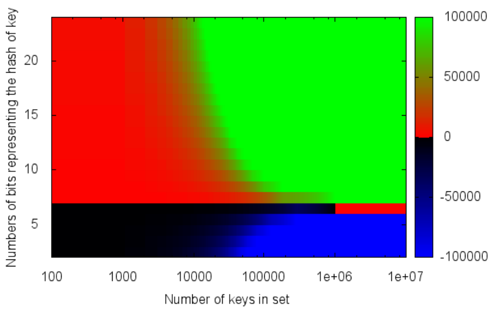
The n in the equation 8.20 represents the number of the keys, which corresponds to the number of transitions in the automaton in context of this work. The h represents number of bits per key that is required by implementation of the minimal perfect hash function itself.

The equations 8.18,8.19 and 8.20 were designed under the assumption, that used perfect hash function is minimal. If the used perfect hash function is not minimal, there is nonzero probability, that the tested input is placed into the bucket that do not belong to any member of the set. The probability of the false positive is than computed by the equation 8.21.

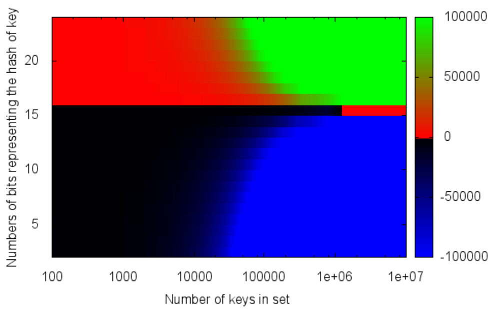
$$f_p = \frac{n}{k} \times \frac{1}{2^s} \tag{8.21}$$

The first part of equation 8.21 is the probability that the perfect hash function will select used bucket. The n is the number of members of the set and the k is the size of the output of the perfect hash function. The k can be also called the number of all buckets. The second part of the equation is the probability, that the selected bucket will have the collision with the input.

The equation 8.22 computes the number of bits that has to be used to store a hash value for every bucket to achieve the probability of false positive f_p. If compared with 8.19,



(a) Comparison of hash tables with minimal perfect hash and bloom filters



(b) Comparison of hash tables with nonminimal perfect hash and bloom filters

it can be seen that nonminimal hash function requires slightly less bit to achieve the same false probability in one bucket, because $k > n$ and therefore the value in logarithm will always be slightly lower that for the minimal perfect hash function.

$$s = \lceil \log_2 \left(\frac{n}{k f_p} \right) \rceil \tag{8.22}$$

Unfortunately, the nonminimal perfect hash function has have memory for the nonused buckets. Therefore, the whole size of the hash table is computed by the equation 8.23. It is important to keep in mind, that parameter h from equation 8.23 is different than the one from equation 8.20 because the minimal PHF may require some additional memory to ensure its minimality. The memory required for the hash table with nonminimal perfect hashing is therefore larger that the memory required for the table with minimal perfect hash function.

$$m = k \times \left(h + \lceil \log_2 \frac{n}{k f_p} \rceil \right) \tag{8.23}$$

Figures 8.12a and 8.12b shows how much more bits of memory are required for the bloom filer than for the hash table with perfect hash function. From memory requirements for the bloom filter computed by equation 8.17 is substracted memory requirement of the hash table computed by equation 8.20 or 8.23 respectively. The required probability of failure is shown on the y axes and is measured by the number of bits require to store the hash key in the hash table. This method of measurement may slightly prefer the hash table because hash table can not provide any other probabilities of false positives, while bloom filter allows arbitrary probability of false positive. The x axis of figures shows the number of the members in the given set. It can be seen, that the expected probability of the false positive is the main criteria for selecting correct approach. If the required probability of the false positive is very small, the hash table seems to be more suitable implementation method. The size of the set is affect the selection only for one specific probability of false positive lookups.

The comparison between 8.12a and 8.12b reveals, that the use of the nonminimal hash function requires more memory. Therefore, the border between area where bloom filter is the more effective implementation and area of the hash table is shifted from 6 bit per hash value to 16 bit per hash key.

The analysis in this section of the work shows, that hash table is more memory effective approach if the very low probability of the false positive is required.

8.4.3 Hardware architecture of probability automaton

The previous section compared the efficiency of the imperfect hashing and the bloom filter and concluded, that the imperfect hashing is better option for the low probability of the false positive. Moreover, the bloom filter may require many hash function to achieve optimal performance which translates into many memory access for every validation. Imperfect hashing requires only computation of perfect hash function and one memory access for the validation. It is important to keep in mind, that it is possible to use the same perfect hash function which is used for next state selection, which further reduces resource requirements of imperfect hashing. Therefore, the validation block is implemented by the imperfect hashing New hash function computes the hash image of the key of each transition. Computed hash is stored in the transition table instead of the transition key. When accepting a new symbol, automaton computes the hash value of the concatenation of the active state and the input symbol. Computed value is compared with the stored hash and transition is made if the values are equal. This implementation ensures behaviour described by the formal model presented in section 8.4.1.

However, it is possible to predict behaviour of the automaton with the knowledge of the hash function used to implement validation block. Thus, with knowledge of the hash function, selecting nonexistent transition is deterministic process. The randomness of the automaton is represented only by the random generation of the hash function.

Only small adaptations are required to change the implementation of the PHF_DFA to the probabilistic automaton. The new universal hash function has to be computed in the hardware. The quality of this function directly influence the quality of the validation process and therefore the quality of whole pattern matching process. It is important to keep in mind that the validation unit have more time for its computation then the universal hash functions used for the perfect hashing. Validation process composes of the computation of the hash value and comparing it with the result obtained by the perfect hashing. Since the comparison does not start before the end of the next state look up, the computation of validation hash can take as long as a computation of universal hash function together with one memory lookup and selection of the next state value. The figure 8.13 illustrates the architecture of the perfect hash automaton with the validation hash. The larger size of the validation block represents, that more time is available for the computation of validation hash. The only requirement is, that the output is available at the same time as the output of the next state selection.

It is possible to further improve speed of the pattern matching unit by the parallelizing the presented approach. If the validation hash is computed faster than the whole perfect hash function than it is reasonable to implement three comparator to validate all three candidates for the next state. After the perfect hashing computation is finished the real next state value is known and other two validation results are discarded.

8.4.4 Reliability of probability Automata

The probability of the false positive during the transition validation process is the parameter of the automaton. If the imperfect hashing is used than it depends on the size of the hash value stored instead of the key and can be evaluated by the equation 8.24

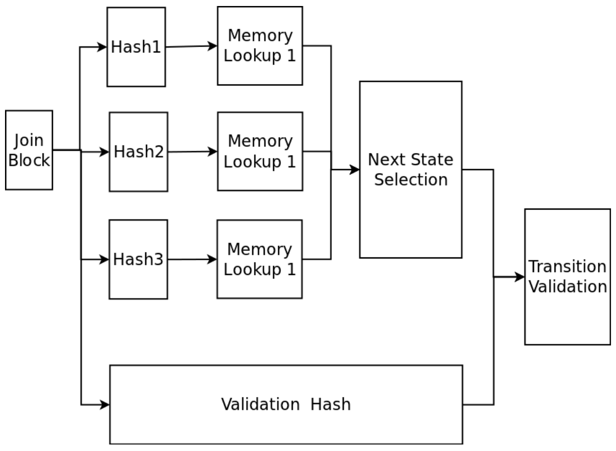


Figure 8.13: Time for computation of validation hash

$$P_{fail} = \frac{1}{2^n} * \frac{|R|}{m} \tag{8.24}$$

The value n is the number of bits used to store validation information, $|R|$ is the number of all transition in the automaton and m is the number of line in the transition table.

In is important to keep in mind that the proposed architecture does not use minimal perfect hashing function. Therefore $|R|$ will always be smaller than m. If the [21] is used to implement perfect hashing then the ration $\frac{m}{|R|}$ should be at least 1.23. Software implementation of the perfect hashing function tries to minimize this ration in order to increasi efficiency of memory usage. However in hardware implementation, the size of memory for the pattern matching unit is determined before the number of transition is known. The unused memory is still reserved for the matching unit and can be used to decrease chance of the failure of the unit. Therefore, the ratio depends on the regular expression configured into the matching unit and can be higher.

The equation 8.24 is based on assumption that the automaton is trying to perform nonexistent transition. The probability of this action is based on the used regular expression set and the network traffic. However, it can be assumed, that this probability is close to one, since the regular expression contains description of the attacks and the most of the data on the production networks will useful traffic. If this probability lowers, the reliability of the solution can only increase, since validation block can not made false negative error. But since it is not possible to evaluate the probability of nonexistent transitions in advance we consider it 1.

When the new symbol arrives it is concatenated with active state and used as an input to perfect hash function, which returns position of the transition in transition table. However, if the generated combination is not a valid transition, perfect hashing function returns any number in its output interval. Since transition table is not full there is a nonzero probability that the result from PHF will not point into occupied place in the transition table. In that case, system will be able to recognise, that input combination is not correct transition and therefore there will not be any chance of failure in validation block. The $\frac{|R|}{m}$ is the probability that nonexistent transition will be mapped to the existing transition in the transition table. Second part of the equation is the probability that this non existing transition has the same hash value as the transition stored in this specific line in the transition table. We assumes

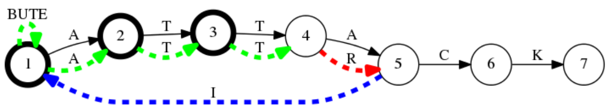


Figure 8.14: Validation failure versus matching failure

that the hash used for the validation purposes is universal. Therefore, the probability that given key will have one specific hash value is $\frac{1}{k}$, where k is the size of the output interval of the hash function. The size of the output interval is always power of two, therefore in equation 8.24 we measure the size in the number of bits used to store the hash value.

It is important to note, that failure in the validation process does not have to cause failure in the matching process. The example of the situation when failure in the validation process does not cause error in the matching process can be seen on figure 8.14. The pattern matching unit was configured to find pattern *ATTACK*. However, string *ATTRIBUTE* appeared in the payload. The green arrows represents the correct function of the validation block and the matching unit. The red arrow represents step where validation block made a mistake. The automaton should returned to its starting state when the symbol R appeared at the input. However, R was accepted by the automaton due to false positive in the validation block. The blue arrow represents correct function of the validation block but incorrect function of the matching unit. The symbol $I$is not recognised as part of the pattern and the unit is reseted into its starting state. All of the remaining symbols are accepted in the starting state while the unit waits for new possible occurrence of the searched pattern. The normal traffic on the network is different from the attack patterns and therefore automaton will correct its state after performing several steps.

The probability automaton may produce both false positives and false negatives. First we will focus on the false positive. The probability of the false positive result depends on the hamming distance between searched string and the closest pattern in the pattern matching unit. Equation 8.25 computes probability of the false positive for the given hamming distance k.

$$P_{fpos} = P_{fail}^{k} \tag{8.25}$$

$$P_{fpos} = \left(\frac{1}{2^n} * \frac{|R|}{m}\right)^k \tag{8.26}$$

It is important to keep in mind that if there is hamming distance k between string and the pattern, than symbols on all other positions are equal. Therefore there is a transition for all symbols except the k and the probability automaton can not made mistake if there is a valid transition. This is the reason, why the probability of false positives equals to the probability of k mistakes in validation process.

The figure 8.15 shows example of the situation, where probability automaton accepts wrong string and generates false positive.

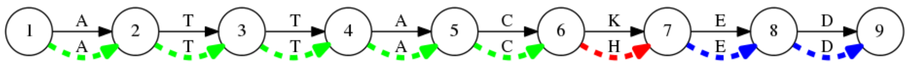


Figure 8.15: Accepting false positive

The pattern matching unit from the figure 8.15 is configured to search for the pattern *ATTACKED*. However, the string in the network payload is *ATTACHED*. These two strings differs in only one character at the sixth position in the word. The green arrows symbolise correct transitions. The red arrow symbolises position where validation block should reject the transition, however it validate wrong transition due to the mistake. Since this position is the only difference between the searched string and the pattern, the probability automaton goes to the final state by the blue arrows symbolising correct function of validation block but incorrect active state.

The false negative mistake can present much more problems than false positive since it represents the attack that went into the network without being recognised by IDS. The figure 8.16 presents situation that will generate false negative result.

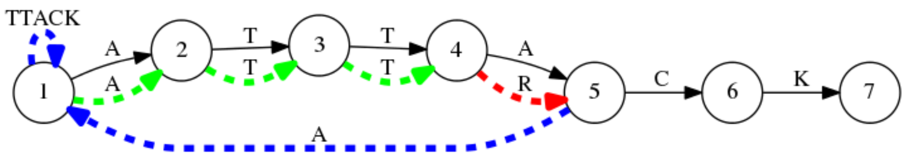


Figure 8.16: Accepting false negative

The pattern matching unit from the figure 8.16 is configured to search for pattern *ATTACK* and the input string is *ATTRATTACK*. The input string contain the pattern however the matching unit will not detect it. The red arrow represents error in the validation block. The pattern starts at the position 5. The probability automaton should be in starting state when accepting fifth position but due to an mistake that occurred exactly one step before the pattern it is in wrong state and the pattern is missed. It can be seen that mistake exactly one step before beginning of the pattern will cause false negative.

Of course, it is possible to have more than one cooperating validation mistakes. First mistake cooperates with the second one if the second mistake is done because of the wrong state caused by the first mistake. After the first mistake automaton needs several steps to return to its correct state. Every validation mistake that appear during these steps is considered cooperating mistake. The cooperating mistakes have to be considered for correct computation of the probability of failure for the pattern matching unit. If mistakes do not cooperate, then they affect failure probability as a single mistake because the automaton will return to the correct state between them. Cooperating mistakes have to be on the specific position in the input stream to be able cooperate.

The probability of the false negative depends on the actual input text and the structure of the automaton, because the position of the validation mistakes depends on the regular expression that is searched and the input text. Moreover, not all mistakes at the required positions can cooperate. The time required for automaton to achieve correct state depends on the input string and on the mistake itself. Therefore the fact that two specific validation mistakes cooperates do not necessary means that different validation mistakes on the specific position would also cooperate. As the result, to compute probability of the false negative match, it is necessary to take into account the representation of the input string and the pattern itself.

The previous paragraphs showed, that the probability of the failure of the whole matching process is smaller than the probability of the failure of the validation process. However, it was also discussed, that this probability depends on the input stream itself which is unknown during the analysis[3]. Therefore, further in this work we will assume, that the probability of the false match is the same as the probability of the failure of the validation process. The assumption will guarantee, that the probability automaton will be always at least as reliable as our analysis predicts and it will be much more reliable most of the time.

Network traffic consists of flows and every flow consists of packets. The currently used hardware accelerating units often process every packet independently. The independent processing of packets allows simple increasing throughput by parallel implementation of the unit. The attacker may divide its attack between several packets to hide it from matching units. Therefore it is important to be able to process every packet of the flow by the same matching unit. It is not possible to divide the attack described by one regular expression into several flows. Therefore it is possible to process flows concurrently and error in the one flow does not have any effect on the processing of other flows. Every flow can be considered as an independent statistical experiment.

The division of the data stream into the packet depends on the network settings. Therefore it is possible that the attack is placed at the border of two packets by accident. It can be seen that even if the pattern matching unit works correctly on the packet level, there is still probability of the failure at the flow level. The matching unit that process every packet independently makes error on every packet border. Therefore, it is possible to say, that mean time between errors of this type of matching unit is the same as the mean length of network packets.

It is possible to use the mean time between errors to compare probability automaton and packet level matching units. The probability of the failure of the probability automaton is estimated as the probability of the failing of transition validation. The equation 8.27 computes $f_r(k)$, which is probability, that the automaton did not make a mistake in k steps. The P_{fail} value is the probability that the validation block of the probability automaton fails to correctly validate the transition.

$$f_r(k) = (1 - P_{fail})^k \tag{8.27}$$

The mean number of symbols between errors of the probability automaton t_c can be computed from the f_r by the equation 8.28.

$$t_c = \sum_{\forall i \in \mathbb{N}_1} i \times f_r(i) \tag{8.28}$$

The t_c depends on the number of bits that are used in the validation block of the probability automaton. Probability of the validation failure decreases with the increasing of the memory available for the transition validation. The figure 8.17 shows the dependency of mean number of symbols between errors and the amount of memory available for transition validation.

The Y axes of figure 8.17 has logarithmic scale. The expected mean number of symbols between errors increases very fast until 25 validation bits is used. After this value the increase of the expected mean number of symbols is slow. The expected mean number of symbols between errors for 25 validation bits is 40×10^{12}. It is important to keep in mind, that the use of probability automata is often connected with the multistriding and therefore one symbol corresponds to the processing of several bytes of network traffic.

[3]The different networks may have different network streams. Therefore it would be impossible to know this information before analysing of the traffic of the target network

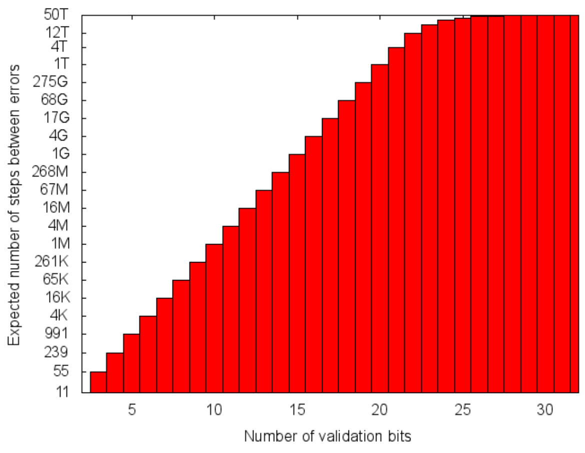


Figure 8.17: The expected mean number of symbols between errors of probability automata in number of processed symbols

The maximal size of IPv4 datagram is 2^{16} bytes [96], while IPv6 increases this limit up to $2^{32} - 1$ bytes [19]. The average size of the IP packet is often smaller than the maximal possible size due to other limitation of the network and the effectivity of the network layers. It can be seen that the mean number of symbols between errors is the same as the maximal size of IP packet if 8 bits is used for transition validation and 16 bits is required to achieve mean number of symbols between errors roughly the same as the size of IPv6 jumbograms.

The problem with the packet level pattern matching is the attack, which is hidden by being intentionally placed on the border of two packets. The question is, if the probability automaton is susceptible to similar type of the attack. Since the position of the failures depends on the randomly generated hash functions, attacker can not determine, where and when errors appears without the knowledge of the used seeds of these hash functions. Moreover, it is possible to generate seeds during the process of the configuration of the probability automaton in the hardware unit. In this case, attacker can not determine the seed of hash functions without direct access to the hardware of probability automaton or the configuring unit. Therefore it is impossible for attacker to hide the attack by intentionally placing it near the appearance of the error.

It is important to realize, that the behaviour of the probability automaton is different than behaviour of reliable matching unit working on packet level. Mainly because the probability automaton may perform error one very position in the network stream while the packet level matching will fail only on the border between two packets. The detailed comparison of reliability of these two approaches would require extensive simulations on many different types of network payload.

However, the comparison provided in this section indicates, that using validation bits in interval from 16 to 25 bits will guarantee the reliability that is higher than the reliability of the pattern matching performed on the packet level.

Chapter 9

Experimental Results

The experimental results in chapter 7 focussed on the structure and properties of the regular expression used in the modern IDS and how are these properties affected by the multistriding techniques. According to the results of these measurement, chapter 8 described perfect hashing automaton and probability automaton pattern matching unit. This chapter reports the results of experiments performed with these two hardware architectures.

9.1 Perfect hash evaluation

The perfect hashing function used to implement the perfect hashing automaton's transition table relies on the universal hash functions. These universal hash functions must be computed for every symbol from the input stream. This section reports how universal hash functions impacts the construction of perfect hash functions. The goal of the experiment is to find simplest hash function that can be used for the generation of the perfect hash function. The selected algorithm for perfect hash construction is a probabilistic algorithms. It randomly generates seeds for universal hash function and attempts construction of perfect hash function. If the construction fails, new seeds are generated and construction phase is repeated. One iteration contains the generation of seeds and attempt at construction of perfect hash function. If the algorithm do not find perfect hash function during 10 iterations, it fails. The required number of iterations of perfect hash generation algorithm is used as the main criterion of the experiment.

The experiment was performed on the pattern automata constructed from the L7 and Snort datasets. Every pattern automaton accepted 2 characters per transition. The table 9.1 shows the obtained results for three selected hash functions. If compared by the average number of iterations, all hash functions are similar. However, according to the number of successfully created automata, the jenkins hash outperforms others for both L7 and Snort ruleset. The H3hash is the second best for the Snort ruleset.

For the practical reasons, the time limit for the creation of the perfect hashing was set to one hour. Since the whole perfect hash generation together with the hash functions were implemented in python, some of the automata were not generated in time.

The conclusion of this experiment is that all three hash functions can be used for the construction of perfect hash automaton. Therefore, the implementation requirements should decide which of them will be used in the hardware unit.

Hash function	L7		Snort	
	Average number of Iteration per rule	Number of Successful rules	Average number of Iteration per rule	Number of Successful rules
Jenkins hash [60]	2.71	273	1.98	2843
Composed hash 8.1.1	2.77	264	2.00	2742
H3hash [28]	2.66	253	2.17	2802

Table 9.1: Comparison of universal hash functions

9.2 Resource utilisation

The previous experiment established several hash function that may be used for the generation of the perfect hash automaton. This section describes the results of the experimental evaluation of the resource utilisation of the proposed pattern matching architecture.

9.2.1 Perfect hashing automaton

The perfect hashing automaton uses the BLOCKRAM of the FPGA to store the whole transition table. The experiment is focussed on the resource requirements and the speed of the matching of the VHDL implementation designed during writing of this thesis.

The table 9.2 shows several measurement for different size of the transition table of the perfect hashing automaton. The experiment focusses on the hardware implementation of the PHFDFA architecture. First three columns of the table specifies the important parameters of the architecture. The state size and symbols size are numbers of bits that are used to represent every transition. From the implementation standpoint they specifies the size of one line of the memory. Number of transition specifies the capacity of the transitional memory. Any automaton that do not exceed any of these three limits may be downloaded into the unit. Columns Number of BRAM and Occupied slices contains the resources utilised by the implemented design. Frequency contains maximal frequency of the clock signal reported after place and route phase. The last column corresponds to the throughput of the network that can be processed by the unit under the assumption that alphabet transformation is used. The assumed alphabet transformation transforms four input characters into one symbol of the automaton. The perfect hashing automaton in this measurement has latency of one clock cycle and uses composed hash function.

The table 9.2 shows that number of slices required for the implementation increases very slowly with the increasing size of transition table. The amount of required BlockRAM memory increases linearly with the size of transition table. The main issue indicated by the experiment is the decrease of operation frequency with the size of transition table. It turns out, that the memory block generated with coregen tool is slower when larger memories are configured. However timing analysis of routed design reveals that critical paths starts at the output of BlockRAM and finishes at the BlockRAM input.

To remove the dependency of the frequency on the size of transition table, registers at outputs of the memory cores were activated. These register increased latency of the pattern

State size [bits]	Symbol size [bits]	Number of transitions	Number of BRAM	Occupied Slices	Frequency [MHz]	Throughput [Gbps]
10	10	12288	9	728	191	3.0
10	10	24576	18	802	166	2.6
10	10	49152	42	640	158	2.5
10	10	98304	87	825	121	1.9
10	10	196608	174	950	67	1.0
10	15	12288	12	656	181	2.9
10	15	24576	24	750	163	2.6
10	15	49152	48	791	142	2.3
10	15	98304	102	794	96	1.5
10	15	196608	204	970	70	1.1

Table 9.2: Properties of hardware implementation of PHFDFA with 1 cycle latency

matching to 3 clock cycles. Table 9.3 shows results obtained by the measurement of this architecture. The target frequency was set to 200MHz for all settings. It can be seen, that up to 100 000 lines in transition table, the target frequency was met. Actually, the maximal frequency of the design is higher for the smaller tables. However, since the pattern matching unit hash higher latency of processing every symbol, the actual throughput of the design is lower.

The increasing latency of the pattern matching unit slows the matching performance for one input stream. However, it is possible to process 3 input streams simultaneously due to the pipelining. In that case, the throughput in the table 9.3 should be multiplied by 3 i.e. number of input streams. It can be seen, that while frequency of the design with latency 1 clock cycle decreases with the increasing size of the transition memory, the clock frequency of design with higher latency is more independent on the size of transition memory. Therefore overall throughput of the matching unit with the higher latency may be higher than the throughput of the unit with lower latency, especially for larger automata. If the pattern matching unit should be performed on many network flows, the second design may be preferable.

The limitation of the pattern matching unit is the memory available in the FPGA. Presented architecture may be connected to the larger external memory to allow matching of larger regular expressions.

9.2.2 Probability automaton

Previous experiments indicate that the size of the transition table is one of the main limitation of the perfect hashing automata. The probability automaton described in section 8.4 reduces the memory requirements for storing transition table by allowing very small portion of error into the validation process. This is achieved by storing hash of transitions instead of transitions themselves. However, to perform transition validation, the transition has to be hashed by validation hash function and the result has to be compared with the data stored in transitional table. The validation hash function has to be computed in the critical loop and therefore it may reduce frequency and slow down the matching process.

The H3 hash function was used as a validation function during this experiment because

State size [bits]	Symbol size [bits]	Number of transitions	Number of BRAM	Occupied Slices	Frequency [MHz]	Throughput [Gbps]
10	10	12288	9	666	215	1.1
10	10	24576	18	760	213	1.1
10	10	49152	42	734	206	1.1
10	10	98304	87	885	201	1.1
10	10	196608	174	960	189	1.0
10	15	12288	12	721	212	1.1
10	15	24576	24	767	227	1.2
10	15	49152	48	839	201	1.1
10	15	98304	102	885	204	1.1
10	15	196608	204	978	168	0.9

Table 9.3: Properties of hardware implementation of PHFDFA with 3 cycle latency

it can be computed in one clock cycle at the required frequency. The Probability automaton was created for different settings and the resource utilisation together with achieved frequency after place and route is reported in table 9.4. The throughput reported in the table is computed from achieved frequency and assumption, that four characters are processed per transition[1].

The table 9.4 indicates, that the maximal possible clock frequency of the design decreases with the increase of the size of transition table for larger transition tables. The same results were obtained from previous experiments with perfect hashing automaton. The comparison with table 9.3 shows, that while the clock frequency of the probability automaton is slightly lower than for perfect hashing automaton. No fine tuning of the synthesis and place&route parameters were performed during experiments. It can be concluded, that even if use of H3 as validation function introduce slight delay into the design, it is still possible to achieve 200MHz clock frequency for the most experiments. Moreover, if external memory would be used to store transition table, the delay of validation hash function would be negligible.

The memory requirements of the probability automaton do not depend on the number of states and size of the alphabet of the automata. Instead it depends on the required reliability of the pattern automaton and number of transitions in the automaton. The probabilistic automaton used in the experiment has 16bits for validation function. By comparing table 9.4 and table 9.3, it can be seen, that probability automaton saves significant number of BlockRAMs for larger automata. However, the probability automaton do not offer any savings for the automata with small transition table.

The experiments proved, that it is reasonable to implement probability automaton in systems that requires larger transition table and ideally uses external memory for storing transition table. The perfect hashing automaton unit is preferable for small automata or in situation where extreme reliability is necessary.

[1]See experiments in chapter 7 for the requirements on transition memory size for different multistriding settings.

State size [bits]	Symbol size [bits]	Number of transitions	Number of BRAM	Occupied Slices	Frequency [MHz]	Throughput [Gbps]
10	10	12288	9	687	203	1.0
10	10	24576	18	762	201	1.0
10	10	49152	36	835	202	1.0
10	10	98304	78	852	188	1.0
10	10	196608	156	981	155	0.8
10	15	12288	9	729	204	1.0
10	15	24576	18	792	203	1.0
10	15	49152	36	877	199	1.0
10	15	98304	78	939	178	0.9
10	15	196608	156	981	156	0.8

Table 9.4: Properties of HW implementation of Probabilistic Automaton with 3 cycle latency

9.2.3 Comparison of memory requirements

The memory requirement is one of the most important metric because it indicates how many patterns can be matched by the given unit. This section focus on measurement and comparison of PHFDFA with Probability Automaton and δDFA.

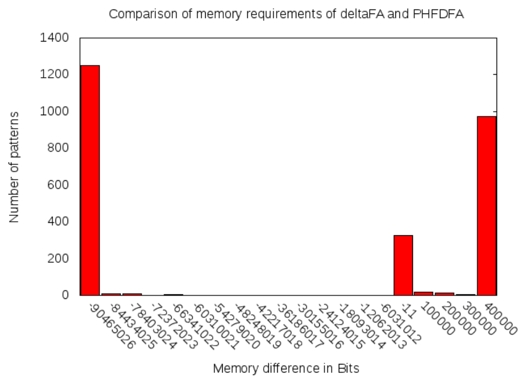
The δDFA accepts only one symbol per transition which limits its throughput. To increase throughput, several parallel instances of δDFA has to be run at once which increases the amount of memory required for pattern matching. The PHFDFA and Probability Automaton may accept arbitrary number of symbols per transition to increase the throughput of the matching process.

The comparison provided in this section focus on the automata that accepts 4 character per every transition and therefore achieves multigigabit throughput. Data structures for all three architectures are prepared for every pattern from selected ruleset. The difference of sizes of two data structures is computed to establish which architecture is better for a given pattern. Results of this experiment are showed in Figure 9.1. The negative value on the x-axis means, that the original data structure is smaller than proposed one while the positive number on the axis means that the proposed data structure is more efficient. The absolute value of the number on x-axis indicates the size of the difference.

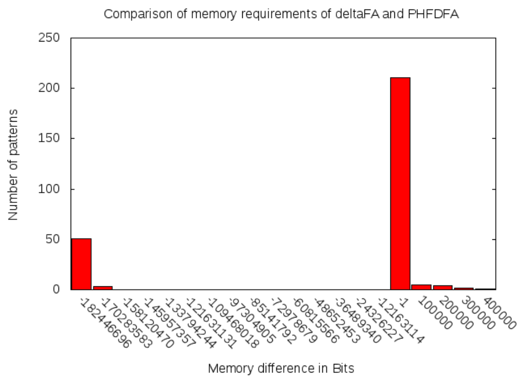
Figure 9.1a compares the PHFDFA with δDFA for Snort ruleset. The negative value on the x-axis means that the δDFA is more efficient while positive value indicates that these patterns should be implemented by PHFDFA. It can be seen, that patterns are divided into three groups. The first group contain patterns that can not be effectively implemented by the PHFDFA architecture. This group contains about 1250 patterns. The second group contains about 500 patterns which can benefit by from the PHFDFA architecture. The third group contains remaining pattern which requires same amount of memory for both approaches. The histogram demonstrates that the PHFDFA architecture can achieve significant reduction of the memory requirement for the suitable patterns. The memory savings for the whole rule set was also measured during this experiment by simply summing up the saving for every pattern which was put into histogram. The use of the PHFDFA for all patterns required about 200MB less memory.

Figure 9.1b compares PHFDFA with δDFA for L7 ruleset. The negative value on x-axis indicates that the δDFA is better option for a given pattern while positive value indicates otherwise. L7 ruleset contain much simpler rules and therefore the saving available by using PHFDFA was not demonstrated.

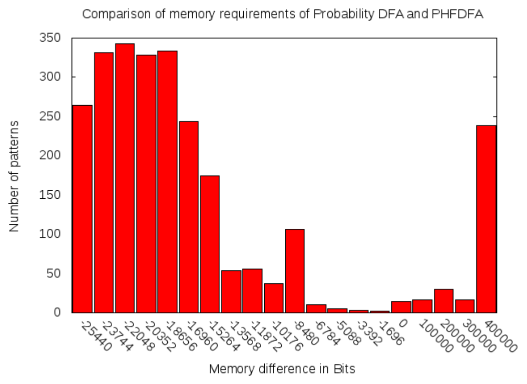
The comparison between PHFDFA and probability automaton for Snort ruleset is shown in Figure 9.1c. The negative value on the x-axis indicates, that PHFDFA has lower memory requirements while positive value indicates that probability automaton is able to reduce memory requirements of a given pattern. The probability automaton in this experiment uses 16 bit to represent validation hash which correspond to $1.5e10^{-3}\%$ probability of faulty transition. The histogram shows that there are significant number of patterns that may benefit from the probability automaton. However, the division between groups is not as large as in Figure 9.1a. Moreover, negative values on x-axis aren't smaller than 30000 bit, which indicates that even if the probability automaton is not better option, it will not cause large increase of the required memory. Implementing whole Snort ruleset by probability automata will cause additional reduction of required memory by 23MB.



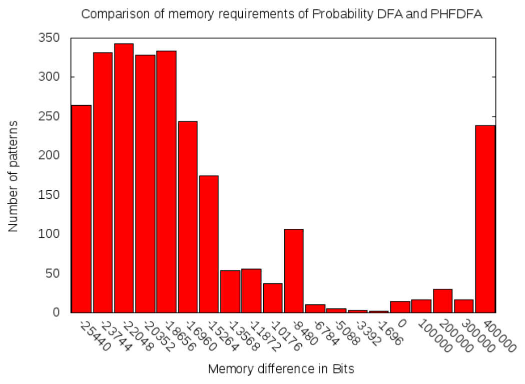
(a) The difference in number of bits required to store transition table for δDFA a PHFDFA. Histogram shows how many pattern in the Snort ruleset has at least memory savings presented on x-axis if implemented by PHFDFA instead of δDFA.



(b) The difference in number of bits required to store transition table for δDFA a PHFDFA. Histogram shows how many pattern in the L7 ruleset has at least memory savings presented on x-axis if implemented by PHFDFA instead of δDFA.



(c) The difference in number of bits required to store transition table for PHFDFA and Probability Automaton. Histogram shows how many pattern in the Snort ruleset has at least memory savings presented on x-axis if implemented by Probability Automaton instead of PHFDFA.



(d) The difference in number of bits required to store transition table for PHFDFA and Probability Automaton. Histogram shows how many pattern in the L7 ruleset has at least memory savings presented on x-axis if implemented by Probability Automaton instead of PHFDFA.

Figure 9.1: Comparison of memory requirements of different architectures

Chapter 10

Conclusion

The thesis deals with the hardware acceleration of pattern matching. The pattern matching is the core operation for many modern intrusion detection systems. The thesis focus on analysis of rules used in real world instead of targeting theoretical limitations and on designing new hardware architectures for the fast pattern matching at multi gigabit speed according to results of the analysis.

The field of pattern matching offers many algorithms with different properties. The chapter 4 provides quick excursion through the most interesting ones. Algorithms based on deterministic finite automata often requires large and slow memories while algorithms based on nondeterministic finite automata may have problems with context switching between severals input streams due to the large internal state.

To achieve multi gigabit matching speed, it is necessary to process several input symbol per one step of the matching unit. The chapter 6 describes current state of the art methods that are used to increase throughput or reduce memory requirements by means of changing the input alphabets. The new algorithm [63] for alphabet transformation is described. The proposed algorithm is able to change alphabet in such way, that the resulting pattern matching will process any number of symbols per transition which provides better opportunity to trade off between speed and memory requirements than current solutions.

The chapter 7 contain detailed analysis of regular expression used in modern computer networks. The analysis focused on the measurement of the saturation of transition table and effect of different optimization of transition table on the saturation [65]. The analysis showed existence of large set of regular expressions with relatively low saturation of transition table.

Two new hardware architectures are presented in this thesis. The first architecture is called *perfect hashing automaton* and performs exact regular expression matching [63, 68, 67] while the second architecture, called *Probability automaton*, allows small portion of errors in the matching process to reduce memory requirements of the system [62]. The analysis performed in chapter 8.4.4 shows that expected number of symbols between error rises with the number of bits used for the validation purposes. If 16 bits is used, the expected number of symbols between two errors is more than 4 billion. This is much higher reliability than exact regular expression matching performed on packet level.

Both proposed architectures were implemented and their resource utilisation were measured [64]. The memory used for implementation of transition table is the limiting factor of the clock frequency of both designs. The measurement performed in chapter 9 shows that the perfect hashing automaton implemented in current FPGA can accommodate transition table with up 200k transitions, which is enough to store most of the regular expression used in modern intrusion detection systems. The use of probability automaton can lower memory requirements. The measurement also indicates, that the probability automaton should be used for the larger automata, since the savings are much larger for large automata. The comparison of the memory required to implement perfect hashing automaton and δDFA were performed to established the efficiency of the proposed architecture. The use of perfect hashing automaton caused the memory savings in order of hundreds MB for Snort rule. Automata causing exponential blowup during determinisation can not be implemented by methods described in this thesis.

10.1 Contributions

1. Analysis of regular expressions used in modern intrusion detection systems and the measurement of the effect of different optimisation techniques on automata generated from given regular expressions
2. Introduction of the alphabet transformation algorithms able to produce n-striding automaton for every positive integer n.
3. New algorithm for determinisation of the alphabet
4. Introduction of new hardware architecture for the fast regular expression matching based on deterministic finite automaton and perfect hash function
5. Introduction of the probabilistic automaton for the regular expression matching and analysis of the effect of failure probability on the reliability of the whole matching process.

Bibliography

[1] Application layer packet classifier for linux.

[2] Clamav 0.99b meets yara!

[3] Clamav homepage. `<www.clamav.net/lang/en>`.

[4] Ds-150: Virtex6 family overview.

[5] Internet world stats. `<www.internetworldstats.com/stats.htm>`.

[6] Ip tables web page.
`<http://www.netfilter.org/projects/iptables/index.html>`.

[7] Perl compatible regular expression. `<www.pcre.org>`.

[8] Python hash algorithms. `<effbot.org/zone/python-hash.htm>`.

[9] Regular expressions info web page. `<http://www.regular-expressions.info/>`.

[10] Service name and transport protocol port number registry.
`<www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>`.

[11] Snort homepage. `<www.snort.org>`.

[12] Standard for information technologyportable operating system interface (posix(r)) base specifications, issue 7. *IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013)*, pages 1–3906, April 2013.

[13] Alfred V Aho. Algorithms for finding patterns in strings, handbook of theoretical computer science (vol. a): algorithms and complexity, 1991.

[14] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. *Compilers: principles, techniques, and tools.* Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007.

[15] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 79 –89, sept. 2008.

[16] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pages 145–154, New York, NY, USA, 2007. ACM.

[17] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 50–59, New York, NY, USA, 2008. ACM.

[18] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[19] D Borman, S Deering, and R Hinden. Rfc 2675: Ipv6 jumbograms, 1999.

[20] Fabiano C Botelho, Yoshiharu Kohayakawa, and Nivio Ziviani. A practical minimal perfect hashing method. In *Experimental and Efficient Algorithms*, pages 488–500. Springer, 2005.

[21] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and Space-efficient Minimal Perfect Hash Functions. In *In Proc. of the 10th Intl. Workshop on Data Structures and Algorithms*, pages 139–150. Springer LNCS, 2007.

[22] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[23] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *Computer Architecture, International Symposium on*, 0:191–202, 2006.

[24] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

[25] J. A. Brzozowski and E. J. McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *Electronic Computers, IEEE Transactions on*, EC-12(2):67 –76, april 1963.

[26] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:87–98, 1996.

[27] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 449–458, New York, NY, USA, 2000. ACM.

[28] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 106–112, New York, NY, USA, 1977. ACM.

[29] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65. ACM, 1978.

[30] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, October 2010.

[31] Christopher R. Clark and David E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *In Proceedings of 13th International Conference on Field Program*, pages 956–959, 2003.

[32] Christopher R Clark and David E Schimmel. Scalable pattern matching for high speed networks. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 249–257. IEEE, 2004.

[33] C.R. Clark and D.E. Schimmel. A pattern-matching co-processor for network intrusion detection systems. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pages 68–74, Dec 2003.

[34] Ming Cong, Hong An, Lu Cao, Yuan Liu, Peng Li, Tao Wang, Zhi-hong Yu, and Dong Liu. Pattern-unit based regular expression matching with reconfigurable function unit. In David Taniar, Osvaldo Gervasi, Beniamino Murgante, Eric Pardede, and BernadyO. Apduhan, editors, *Computational Science and Its Applications – ICCSA 2010*, volume 6019 of *Lecture Notes in Computer Science*, pages 427–440. Springer Berlin Heidelberg, 2010.

[35] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*

[36] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* `<http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-1-2a-2b-3a-3b-manual.pdf>`.

[37] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An Optimal Algorithm for Generating Minimal Perfect Hash Functions. *Information Processing Letters*, 43:257–264, 1992.

[38] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1–2):1 – 143, 1997.

[39] Cezar Câmpeanu and Nicolae Santean. On the closure of pattern expressions languages under intersection with regular languages. *Acta Informatica*, 46(3):193–207, 2009.

[40] Cezar Câmpeanu and Sheng Yu. Pattern expressions and pattern automata. *Information Processing Letters*, 92(6):267 – 274, 2004.

[41] J-P. Deschamps and G. Sutter. Comparison of FPGA implementation of the mod M reduction. *Latin American applied research*, 37:93 – 97, 01 2007.

[42] PM Ebert, James E Mazo, and Michael G Taylor. Overflow oscillations in digital filters. *Bell System Technical Journal*, 48(9):2999–3020, 1969.

[43] Samuel Eilenberg. *Automata, Languages, and Machines.* Academic Press, Inc., Orlando, FL, USA, 1974.

[44] Jeffrey Erman, Anirban Mahanti, Martin Arlitt, Ira Cohen, and Carey Williamson. Offline/realtime traffic classification using semi-supervised learning. *Performance Evaluation*, 64(9):1194–1213, 2007.

[45] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved dfa for fast regular expression matching. *ACM SIGCOMM Computer Communication Review*, 38(5):29–40, 2008.

[46] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A Faster Algorithm for Constructing Minimal Perfect Hash Functions. In *SIGIR '92: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 266–273, New York, NY, USA, 1992. ACM.

[47] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984.

[48] J. Friedl. *Mastering Regular Expressions.* Oreilly Series. O'Reilly Media, Incorporated, 2006.

[49] Ziemba G. et al. Rfc 1858: Security considerations for ip fragment filtering. 1995.

[50] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1):18–28, 2009.

[51] Dora Giammarresi and Rosa Montalbano. Deterministic generalized automata. *Theor. Comput. Sci.*, 215:191–208, February 1999.

[52] Gregor Gramlich and Georg Schnitger. Minimizing nfa's and regular expressions. In *STACS 2005*, pages 399–411. Springer, 2005.

[53] Hermann Gruber and Markus Holzer. Inapproximability of nondeterministic state and transition complexity assuming p ¡¿ np. In *IN: PROC. DLT. VOLUME 4588 OF LNCS*, pages 205–216. Springer, 2007.

[54] Venkatesan Guruswami, Rajsekar Manokaran, and Prasad Raghavendra. Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 573–582. IEEE, 2008.

[55] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[56] S.H.C. Haris, R.B. Ahmad, and M.A.H.A. Ghani. Detecting tcp syn flood attack based on anomaly detection. In *Network Applications Protocols and Services (NETAPPS), 2010 Second International Conference on*, pages 240–244, Sept 2010.

[57] Markus Holzer and Martin Kutrib. Descriptional and computational complexity of finite automata - a survey. *Inf. Comput.*, 209(3):456–470, 2011.

[58] Brad L Hutchings, Rob Franklin, and Daniel Carver. Assisting network intrusion detection with reconfigurable hardware. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 111–120. IEEE, 2002.

[59] Bob Jenkins. Spookyhash.

[60] Bob Jenkins. Hash functions. *Dr Dobbs Journal*, 22(9):107–+, 1997.

[61] AnnaR. Karlin, HowardW. Trickey, and JeffreyD. Ullman. Algorithms for the compilation of regular expressions into plas. *Algorithmica*, 2(1-4):283–314, 1987.

[62] J. Kastil and J. Korenek. High speed pattern matching algorithm based on deterministic finite automata with faulty transition table. In *Architectures for Networking and Communications Systems (ANCS), 2010 ACM/IEEE Symposium on*, pages 1–2, Oct 2010.

[63] J. Kastil, J. Korenek, and O. Lengal. Methodology for fast pattern matching by deterministic finite automaton with perfect hashing. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 823–829, Aug 2009.

[64] J. Kastil, V. Kosar, and J. Korenek. Hardware architecture for the fast pattern matching. *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 0:120–123, 2013.

[65] Jan Kaštil and Jan Kořenek. Hardware accelerated pattern matching based on deterministic finite automata with perfect hashing. In *Proceedings of the 13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems DDECS 2010*, pages 149–152. IEEE Computer Society, 2010.

[66] Jan Kaštil. Vyhledávání regulárních výrazů ve vysokorychlostním síťovém provozu. In *Počítačové architektury a diagnostika 2009*, pages 89–94. Tomas Bata University in Zlín, 2009.

[67] Jan Kaštil and Jan Kořenek. Deterministic finite automaton with perfect hashing for fast pattern matching. In *Proceedings of Junior Scientist Conference 2008*, pages 103–104. Technical University Wien, 2008.

[68] Jan Kaštil and Jan Kořenek. Deterministický konečný automat pro vyhledání vzorů ve vysokorychlostních sítích. In *Proceedings of the 14th Conference STUDENT EEICT 2008*, Volume 2, pages 227–229. Brno University of Technology, 2008.

[69] V. Kosar, M. Zadnik, and J. Korenek. Nfa reduction for regular expressions matching using fpga. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 338–341, Dec 2013.

[70] Christopher Krügel, Thomas Toth, and Engin Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 201–208. ACM, 2002.

[71] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, ANCS '06, pages 81–92, New York, NY, USA, 2006. ACM.

[72] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, ANCS '06, pages 81–92, New York, NY, USA, 2006. ACM.

[73] Kun-chan Lan and John Heidemann. A measurement study of correlations of internet flow characteristics. *Comput. Netw.*, 50:46–62, January 2006.

[74] Pavel Laskov, Patrick Düssel, Christin Schäfer, and Konrad Rieck. Learning intrusion detection: supervised or unsupervised? In *Image Analysis and Processing–ICIAP 2005*, pages 50–57. Springer, 2005.

[75] Aleksandar Lazarevic, Levent Ertöz, Vipin Kumar, Aysel Ozgur, and Jaideep Srivastava. A comparative study of anomaly detection schemes in network intrusion detection. In *SDM*, pages 25–36. SIAM, 2003.

[76] Cynthia Bailey Lee, Chris Roedel, and Elena Silenok. Detection and characterization of port scan attacks, 2003.

[77] Yang Li, Zheng Li, Nenghai Yu, and Ke Ma. Apfa: Asynchronous parallel finite automaton for deep packet inspection in cloud computing. In Martin Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 529–540. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-10665-1_48.

[78] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

[79] T. Limmer and F. Dressler. Improving the performance of intrusion detection using dialog-based payload aggregation. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pages 822–827, April 2011.

[80] Cheng-Hung Lin and Hsien-Sheng Hsiao. Hierarchical state machine architecture for regular expression pattern matching. In *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*, GLSVLSI '09, pages 133–136, New York, NY, USA, 2009. ACM.

[81] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of pattern matching circuits for regular expression on fpga. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(12):1303–1310, Dec 2007.

[82] Peter Linz. *An Introduction to Formal Language and Automata.* Jones and Bartlett Publishers, Inc., USA, 2006.

[83] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte regular expression matching with speculation. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 284–303, Berlin, Heidelberg, 2009. Springer-Verlag.

[84] Cotton M. et al. Rfc 6335: Internet assigned numbers authority (iana) procedures for the management of the service name and transport protocol port number registry. 2011.

[85] Frank H. Mathis. A generalized birthday problem. *SIAM Review*, 33(2):265–270, 1991.

[86] Alexander Meduna. *Automata and languages: theory and applications.* Springer-Verlag, London, UK, 2000.

[87] K. Mehlhorn and P. Sanders. *Algorithms and data structures: the basic toolbox.* Springer, 2008.

[88] AR. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Switching and Automata Theory, 1972., IEEE Conference Record of 13th Annual Symposium on*, pages 125–129, Oct 1972.

[89] Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. A rational design for a weighted finite-state transducer library. In *Revised Papers from the Second International Workshop on Implementing Automata*, WIA '97, pages 144–158, London, UK, UK, 1998. Springer-Verlag.

[90] K. Namjoshi and G. Narlikar. Robust and fast pattern matching for intrusion detection. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, March 2010.

[91] National Institute of Standards and Technology. *FIPS PUB 180-4: Secure Hash Standard.* 2012.

[92] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys & Tutorials, IEEE*, 10(4):56–76, 2008.

[93] University of Southern California. Rfc 793:transmission control protocol. 1981.

[94] Vern Paxson. Bro: a system for detecting network intruders in real time. In *In Computer Networks*, 1999.

[95] Peter K Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, 1990.

[96] Jon Postel et al. Rfc 791: Internet protocol. 1981.

[97] Thomas H Ptacek and Timothy N Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, DTIC Document, 1998.

[98] Viktor Pus, Jiri Tobola, Vlastimil Kosar, Jan Kastil, and Jan Korenek. Netbench: Framework for evaluation of packet processing algorithms. *Symposium On Architecture For Networking And Communications Systems*, pages 95–96, 2011.

[99] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

[100] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. A performance study of hashing functions for hardware applications. In *In Proc. of Int. Conf. on Computing and Information*, pages 1621–1636, 1994.

[101] Elaine Rich. *Automata, computability and complexity: theory and applications.* Pearson Prentice Hall Upper Saddle River, 2008.

[102] R. Rivest. Rfc-1321: The md5 message-digest algorithm, 1992.

[103] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[104] K. Rowett and S. Sikdar. Intrusion detection system, September 29 2005. US Patent App. 11/125,956.

[105] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969 – 978, 2009. Special Section on Graphical Models and Information Retrieval.

[106] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 187–198, Dec 2010.

[107] D. Sanchez, L. Yen, M.D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 123–133, Dec 2007.

[108] Arif Sari. A review of anomaly detection systems in cloud networks and survey of cloud security measures in cloud storage applications. *Journal of Information Security*, 6(02):142, 2015.

[109] Abhijit Sarmah. Intrusion detection systems: Definition, need and challenges. October 2001.

[110] Reetinder Sidhu and Viktor K Prasanna. Fast regular expression matching using fpgas. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 227–238. IEEE, 2001.

[111] Dan A. Simovici and Richard L. Tenney. *Theory of formal languages with applications. With a preface by Philippe Flajolet.* Singapore: World Scientific Publishing. xii, 629 p. $ 86.00 , 1999.

[112] Chris Sinclair, Lyn Pierce, and Sara Matzner. An application of machine learning to network intrusion detection. In *Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual*, pages 371–377. IEEE, 1999.

[113] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking algorithmic complexity attacks against a nids. In *IN ANNUAL COMPUTER SECURITY APPLICATIONS CONFERENCE*, 2006.

[114] Randy Smith, Cristian Estan, and Somesh Jha. Xfa: Faster signature matching with extended automata. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 187–201. IEEE, 2008.

[115] Robin Sommer and Vern Paxson. Enhancing Byte-level Network Intrusion Detection Signatures with Context. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 262–271, New York, NY, USA, 2003. ACM.

[116] Ioannis Sourdis, João Bispo, João M.P. Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51(1):99–121, 2008.

[117] Yo sub Han and Derick Wood. The generalization of generalized automata: Expression automata. *International Journal of Foundations of Computer Science*, 16:510, 2005.

[118] Amnon Ta-Shma. Storing information with extractors. *Inf. Process. Lett.*, 83(5):267–274, September 2002.

[119] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[120] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[121] Jeffrey D. Ullman. Combining state machines and regular expressions for automatic synthesis of vlsi circuits. Technical report, Stanford, CA, USA, 1982.

[122] Hao Wang, Shi Pu, Gabriel Knezek, and Jyh-Charn Liu. A modular nfa architecture for regular expression matching. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 209–218, New York, NY, USA, 2010. ACM.

[123] Kai Wang, Yaxuan Qi, Yibo Xue, and Jun Li. Reorganized and Compact DFA for Efficient Regular Expression Matching. pages 1–5, 2011.

[124] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005*, pages 19–35. Springer, 2005.

[125] Tao Xie, Fanbao Liu, and Dengguo Feng. Fast collision attack on md5.

[126] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM (JACM)*, 28(3):615–628, 1981.

[127] Yu Zhang, Ping Liu, Yanbing Liu, Aiping Li, Cuilan Du, and Dongjin Fan. Attacking pattern matching algorithms based on the gap between average-case and worst-case complexity. *Journal of Advances in Computer Networks*, 1:228–233, September 2013.

[128] N. Zilberman, Y. Audzevich, G.A. Covington, and A.W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *Micro, IEEE*, 34(5):32–41, Sept 2014.

List of Figures

List of Tables

Appendix A

Minimal DFA

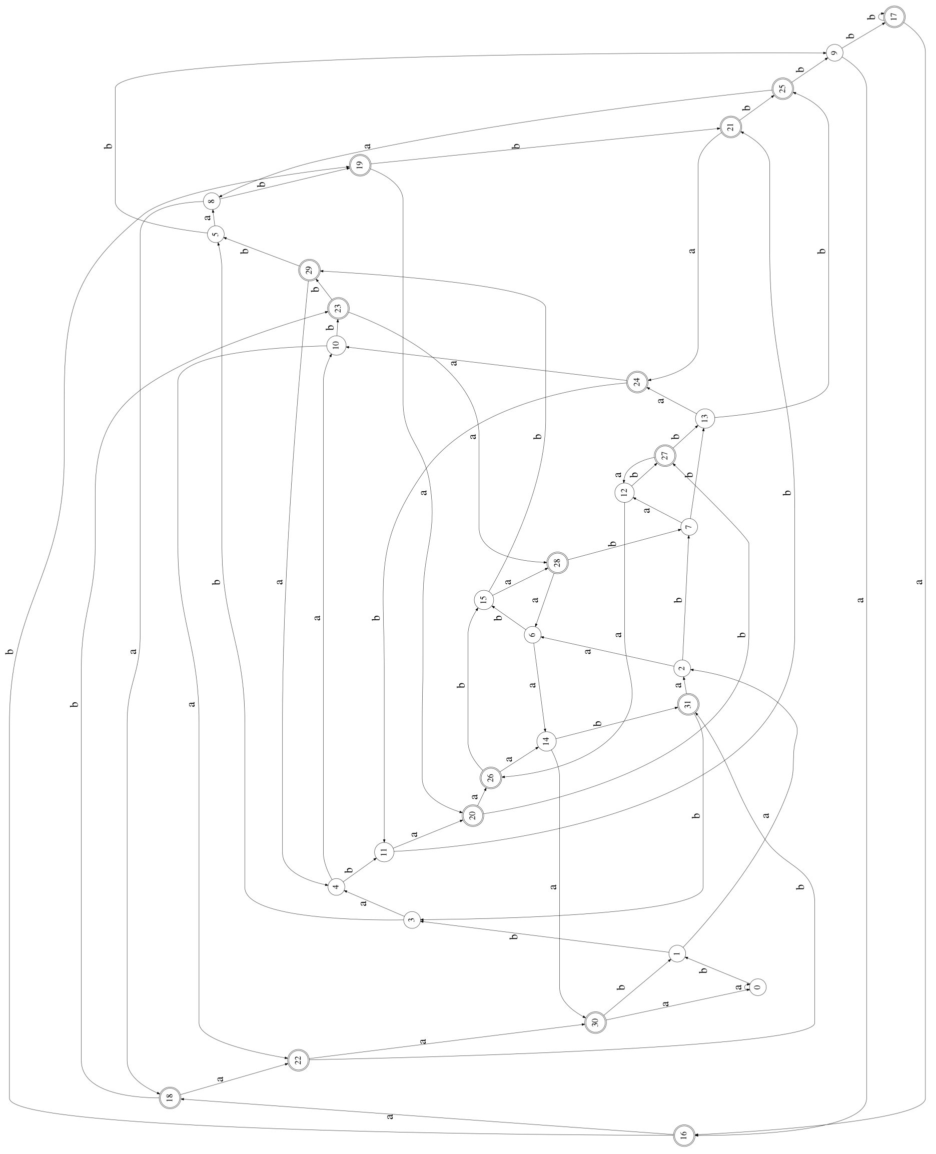


Figure A.1: Minimal DFA accepting the same language as the NFA in figure 2.4

Appendix B

Saturation of transition table of pattern automata

The chapter 7.2.2 described experiments for measurement of saturation of transition tables of pattern automata. The chapter presented the result for the mamny default states optimisation. This appendix contain the results for other experiments as well.

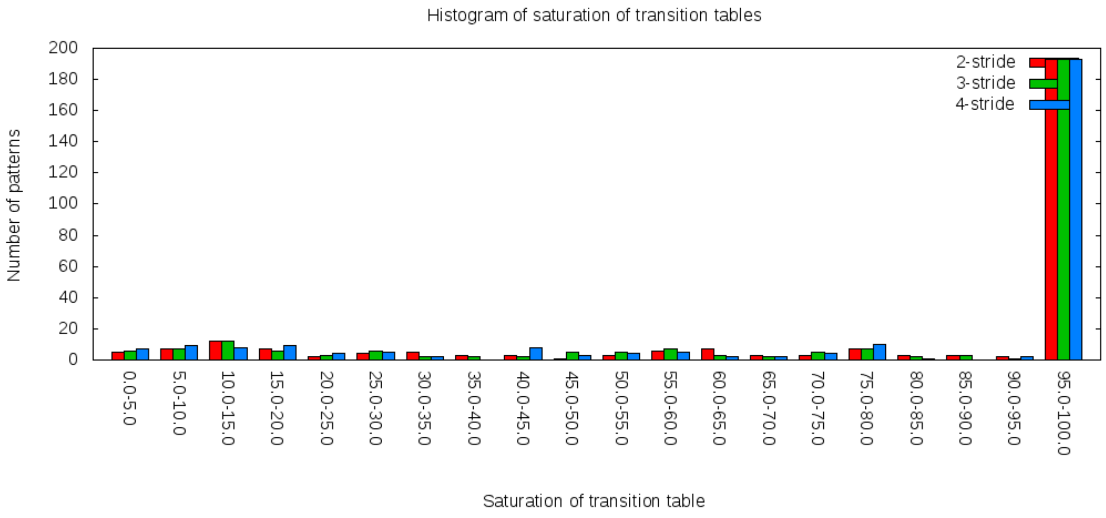


Figure B.1: Saturation of transition table of pattern automata created from L7 regular expression. Histograms shows how many pattern creates automata with saturation of transition table at the x-axis. The axis is annotated in percentages. The level of saturation is show for various numbers of character accepted by one transition. No default state is allowed in the pattern automaton.

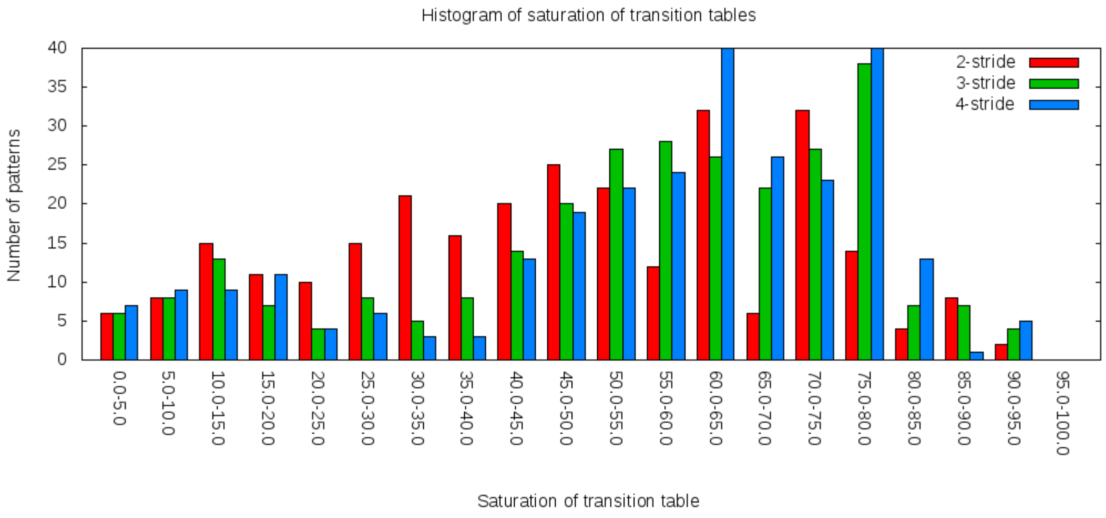


Figure B.2: Saturation of transition table of pattern automata created from L7 regular expression. Histograms shows how many pattern creates automata with saturation of transition table at the x-axis. The axis is annotated in percentages. The level of saturation is show for various numbers of character accepted by one transition. One default state is allowed in the pattern automaton.

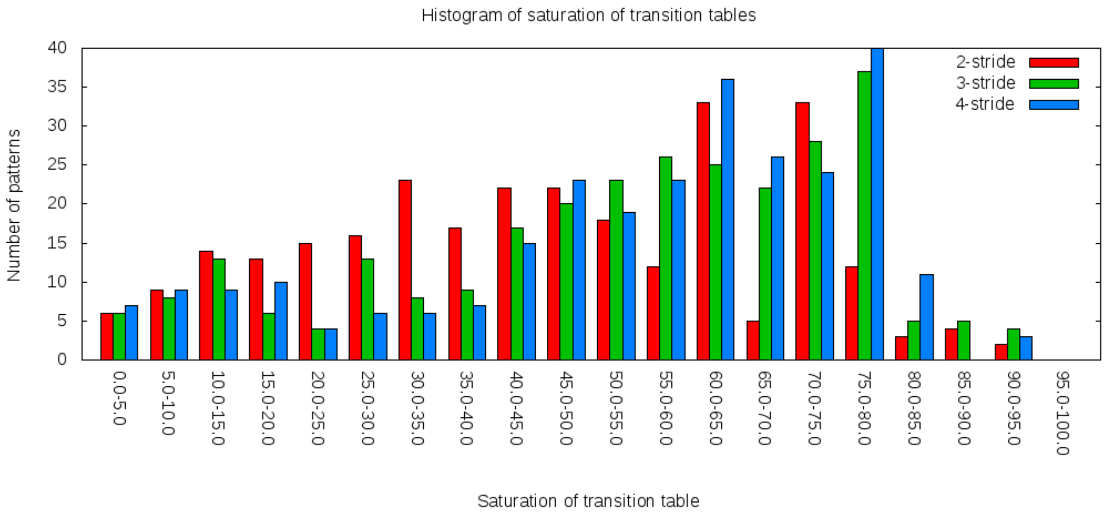


Figure B.3: Saturation of transition table of pattern automata created from L7 regular expression. Histograms shows how many pattern creates automata with saturation of transition table at the x-axis. The axis is annotated in percentages. The level of saturation is show for various numbers of character accepted by one transition. Many default states are allowed in the pattern automaton.

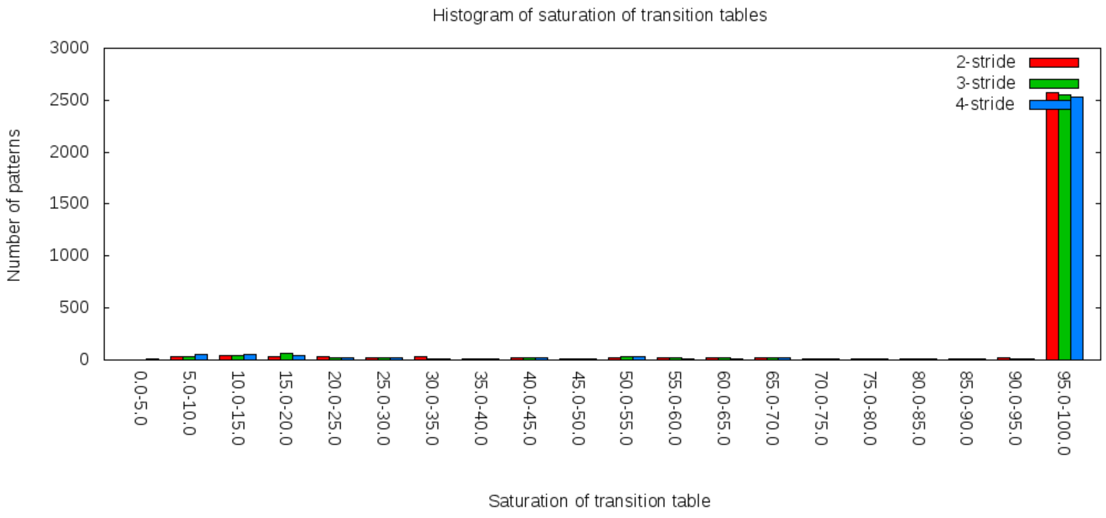


Figure B.4: Saturation of transition table of pattern automata created from Snort regular expression. Histograms shows how many pattern creates automata with saturation of transition table at the x-axis. The axis is annotated in percentages. The level of saturation is show for various numbers of character accepted by one transition. No default state is allowed in the pattern automaton.

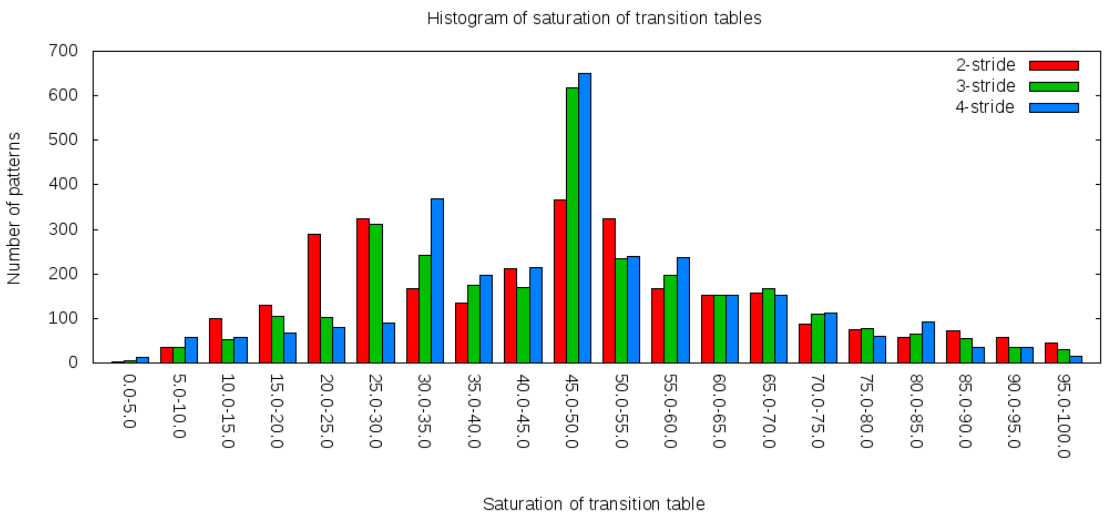


Figure B.5: Saturation of transition table of pattern automata created from Snort regular expression. Histograms shows how many pattern creates automata with saturation of transition table at the x-axis. The axis is annotated in percentages. The level of saturation is show for various numbers of character accepted by one transition. One default state is allowed in the pattern automaton.

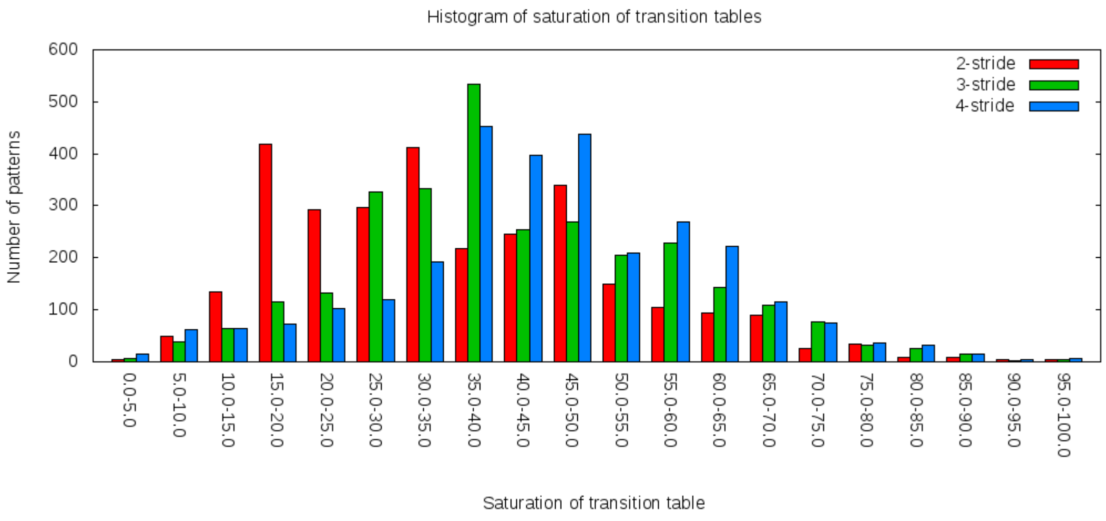


Figure B.6: Saturation of transition table of pattern automata created from Snort regular expression. Histograms shows how many pattern creates automata with saturation of transition table at the x-axis. The axis is annotated in percentages. The level of saturation is show for various numbers of character accepted by one transition. Many default states are allowed in the pattern automaton.

Appendix C

Transition size of pattern automata

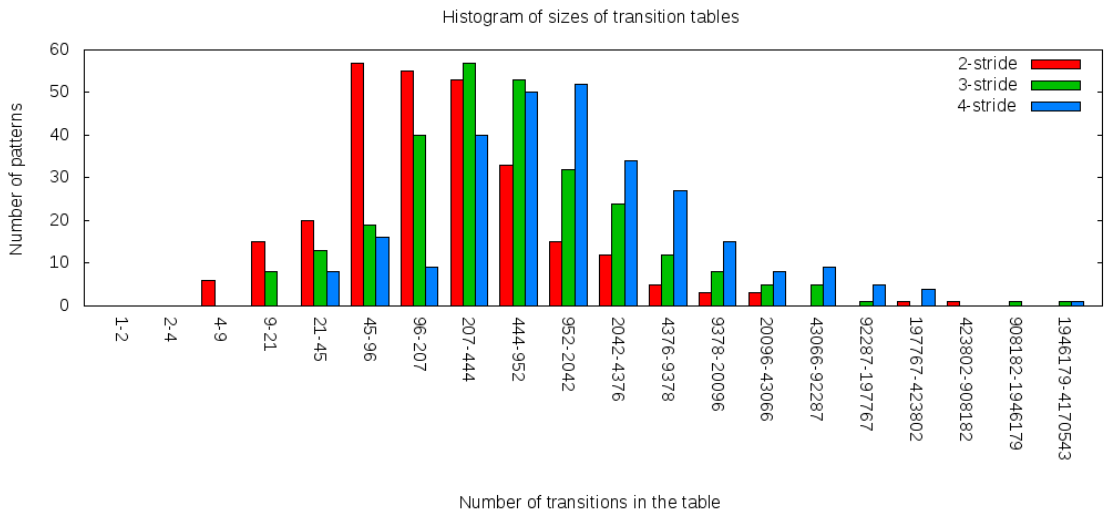


Figure C.1: Transition size of pattern automata for created for L7 regular expressions. Histogram shows how many pattern creates pattern automata with transition size presented at the x-axis. The transition size is showed for various number of character accepted by one transition of a pattern automaton. No default state is allowed in an automaton.

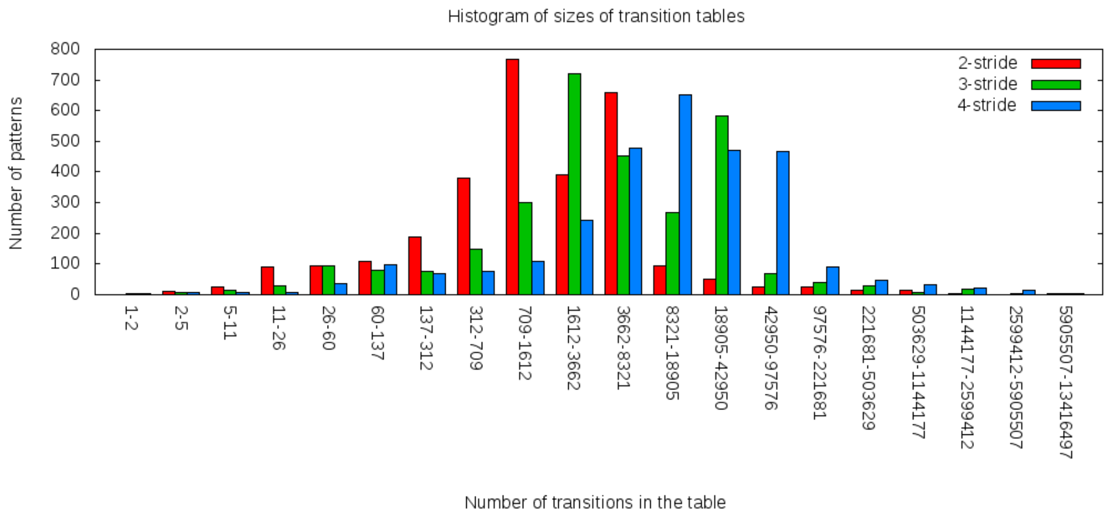


Figure C.2: Transition size of pattern automata for created for Snort regular expressions. Histogram shows how many pattern creates pattern automata with transition size presented at the x-axis. The transition size is showed for various number of character accepted by one transition of a pattern automaton. No default state is allowed in an automaton.

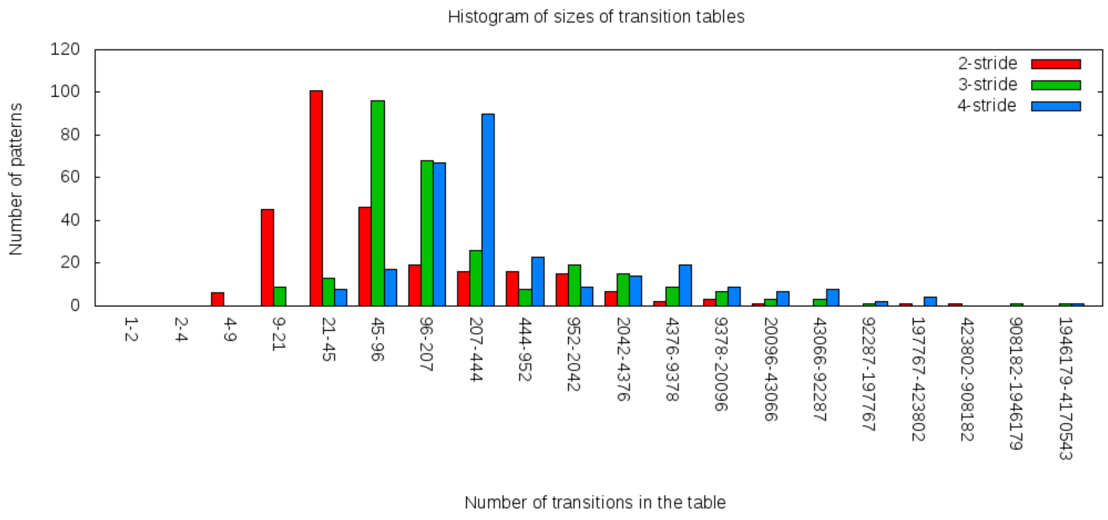


Figure C.3: Transition size of pattern automata for created for L7 regular expressions. Histogram shows how many pattern creates pattern automata with transition size presented at the x-axis. The transition size is showed for various number of character accepted by one transition of a pattern automaton. One default state is allowed in an automaton.

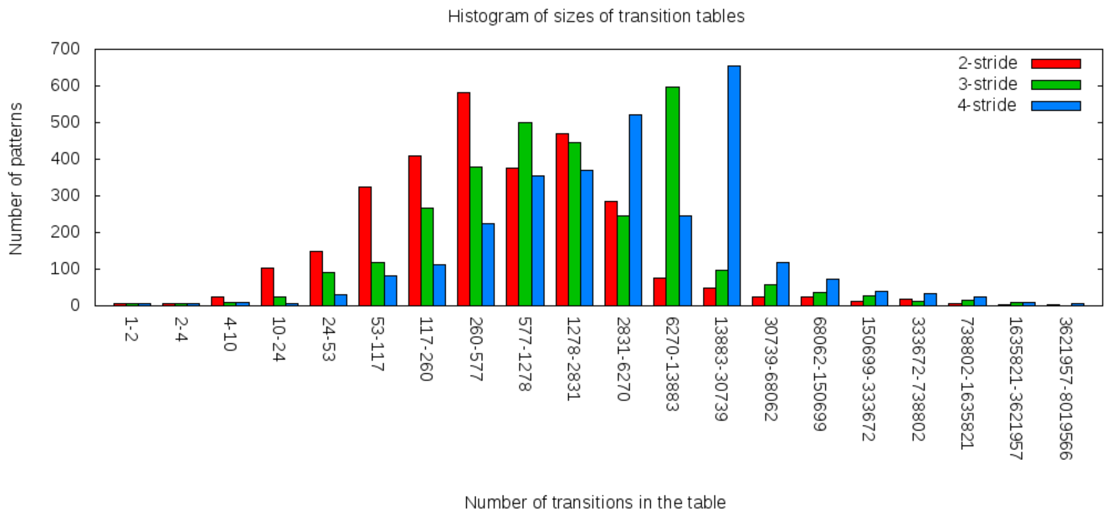


Figure C.4: Transition size of pattern automata for created for Snort regular expressions. Histogram shows how many pattern creates pattern automata with transition size presented at the x-axis. The transition size is showed for various number of character accepted by one transition of a pattern automaton. One default state is allowed in an automaton.

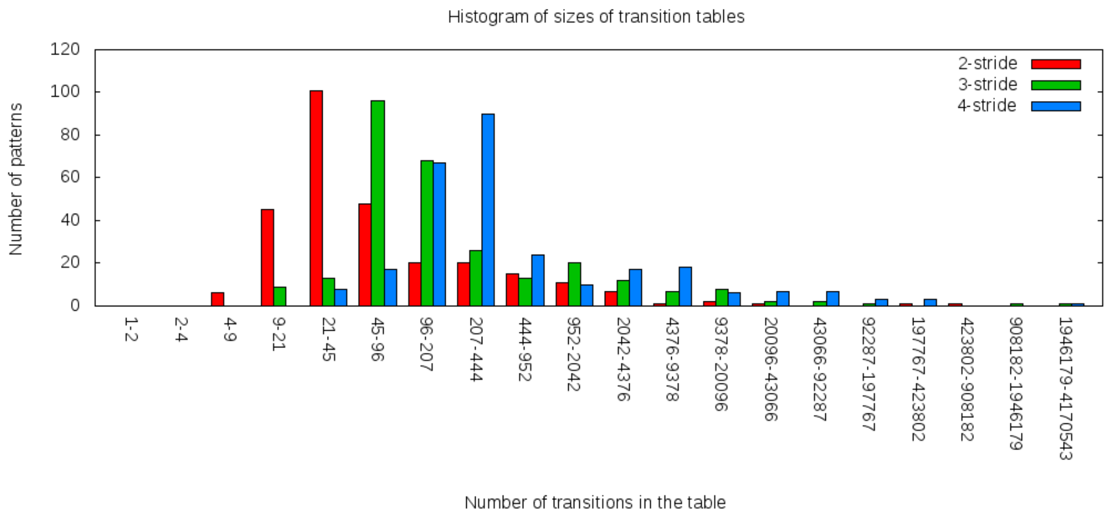


Figure C.5: Transition size of pattern automata for created for L7 regular expressions. Histogram shows how many pattern creates pattern automata with transition size presented at the x-axis. The transition size is showed for various number of character accepted by one transition of a pattern automaton. Many default states are allowed in an automaton.

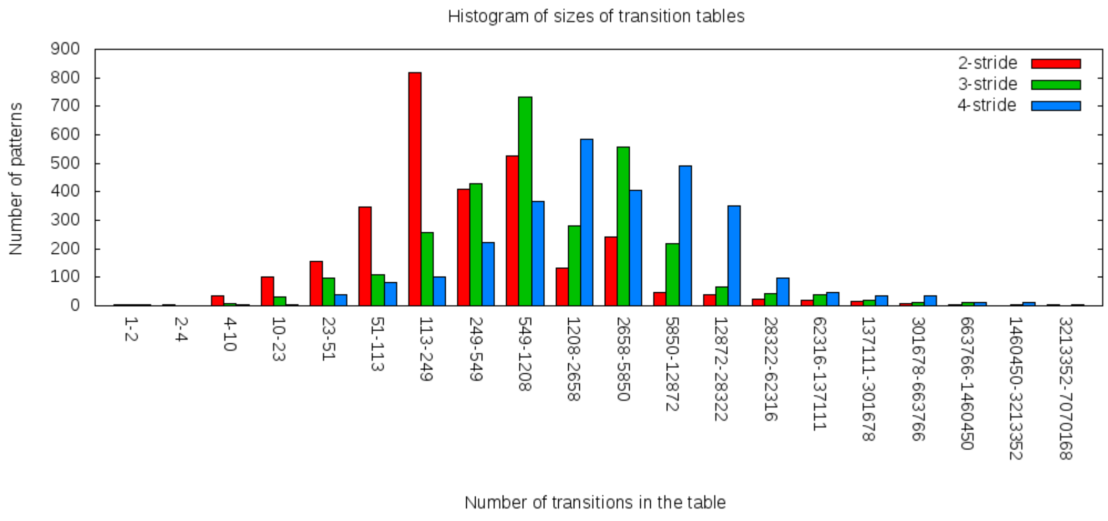


Figure C.6: Transition size of pattern automata for created for Snort regular expressions. Histogram shows how many pattern creates pattern automata with transition size presented at the x-axis. The transition size is showed for various number of character accepted by one transition of a pattern automaton. Many default states are allowed in an automaton.