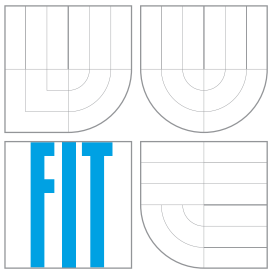
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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
AUTHOR

Ing. JAN KAŠTIL

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. ZDENĚK KOTÁSEK, CSc.

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 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
December 1, 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 [2] 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) [50] 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 [45] contains a receiver's and sender's IP address together with port numbers and TCP setting called flags. The IANA [3, 41] 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 [37] 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 [49, 35, 31, 19] or machine learning [34, 16, 44, 53] 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 [20], or port scanning [36].

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 [18]. 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 [48].

Vern Paxton [55] 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 [10, 11, 9, 56, 51] 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 3.

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 5. 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 5). The measurement performed in chapter 7 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 [32, 5, 33, 54] 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.

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 [42, 39, 52].

2.1 Definition of Regular Expression

The definition 1 from [42] defines a regular expression and regular languages in the way used in the information theory.

Definition 1. ***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^*.

Definition 2. ***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.

2.2 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.

Definition 3. ***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.*

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 [47].

The deterministic automaton ensures that there is at most one next state for any combination of input symbol and a given state.

Definition 4. ***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.

Theorem 2. ***Uniqueness of minimal DFA:*** *There is only one unique minimal deterministic finite automata.*

Theorem 3. ***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 [47].

Chapter 3

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.

3.1 Architectures Based on Nondeterministic Finite Automata

The first pattern matching architecture based on nondeterministic finite automata was introduced by Ullman in [56, 22]. 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 [51].

The NFA generated into FPGA is created as a union of NFA generated for every rule. Hutchings et al. [21] 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 [38].

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 [11] 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 [10], 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. [30] 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%.

Namjoshi and Narlikar [43] 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.

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.

3.2 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 Luchaup et al. [40] focused on the increasing of the throughput of the matching unit by accepting more input symbols in one step. 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.

The Kumar [32] 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 Delayed Input DFA (DDFA) introduces transitions without any label called delayed transitions.

Delayed 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 Delta Deterministic Finite Automaton (δDFA) [17] 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.

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 [7] or [17]. 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 4

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.

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.

4.1 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 $T : \Sigma^* \to \Sigma_{class}$ 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 4.1 demonstrates such situation.

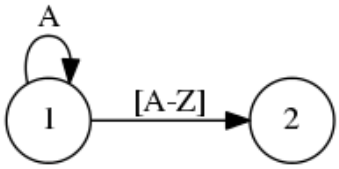


Figure 4.1: Overlapping symbols

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

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

Definition 6. ***Overlapping symbols:*** *Lets have alphabets Σ and Σ_{class} and alphabet transformation $T : \Sigma^* \to \Sigma_{class}$. 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.

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. The algorithm 1 generates the alphabet whose transformation can be done in deterministic fashion.

4.2 Pattern Automaton

All state of the art methods for generating multistriding automaton have serious limitation for practical use. Therefore we have designed pattern automaton, which allows the label

Algorithm 1 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 Σ 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

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 7. ***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 4. ***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 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. States that belong to the concatenated transitions are refered as Visited states. 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 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.

Theorem 5. ***Equivalence of pattern automaton and finite automaton:*** *Lets have an automaton $M(Q, \Sigma, \delta, s, f)$. The algorithm 2 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.*

Algorithms 2 and 3 are new algorithms presented in the thesis and are used to obtain fine trade off between memory requirements and matching throughput.

Algorithm 2 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 $M_p(Q_p, \Sigma_{classP}, \delta_p, s, F_p)$

Duplicate final states into automaton $M_D(Q_D, \Sigma_{classD}, \delta_D, s, F_D)$
$\Sigma_{classP} = \emptyset$
$\delta_p = \emptyset$
$F_p = F_D$
$Q_p = Q_D$
$D = \emptyset$
$Q_{process} = Q_D$
while $Q_{process} \neq \emptyset$ **do**
    $Q_{new} = \emptyset$
    for $q \in Q_{process}$ **do**
        if $\nexists q_o : (q_o, q) \in D$ **then**
            $q_o = q$
        else
            Find q_o, such $(q_o, q) \in D$
        end if
        Compute all n-character symbols $q_S = \{s | s \in \Sigma^*_{class} \wedge |s| = n \wedge (q_o, s) \in Dom(\delta^*_D)\}$
        for $sym \in q_S$ **do**
            if $V(q_o, sym) \cap F = \emptyset \vee \delta^*(q_o, sym) \in F_D$ **then**
                Add a new transition $\delta_p = \delta_p \cup (q, sym, \delta^*_D(q_o, sym))$
                Add symbol into alphabet $\Sigma_{classP} = \Sigma_{classP} \cup \{sym\}$
            else if $\exists q_{td} : (\delta^*(q_o, sym), q_{td}) \in D$ **then**
                Add new transition into duplicated state $\delta_p = \delta_p \cup (q, sym, q_{td})$
                Add new symbol into alphabet $\Sigma_{classP} = \Sigma_{classP} \cup \{sym\}$
            else
                Generate new state $q_n \notin Q_p$
                Add new state into set of states $Q_p = Q_p \cup \{q_n\}$
                Add new state into processing pipeline $Q_{new} = Q_{new} \cup \{q_n\}$
                The new state is final state $F_p = F_p \cup \{q_n\}$
                Store, which state is being duplicated $D = D \cup \{(\delta^*_D(q_o, sym), q_n)\}$
                Add a new transition $\delta_p = \delta_p \cup (q, sym, q_n)$
                Add a symbol into alphabet $\Sigma_{classP} = \Sigma_{classP} \cup \{sym\}$
            end if
        end for
    end for
    $Q_{process} = Q_{new}$
end while

Algorithm 3 Alphabet determinization for pattern symbols

Input: Nondeterministic pattern alphabet Σ_N
Output: Deterministic pattern alphabet Σ_D
Output: Mapping $M : \Sigma_N \to \Sigma_D$

function AddSymbol(Sym, Σ_D, $M : \Sigma_N \to \Sigma_D$, $M_{back} : \Sigma_D \to 2^{\Sigma_N}$)
    if $sym \notin M_{back}$ **then**
        $M_{back} = M_{back} \cup (sym \to \{sym\})$
    end if
    if $\nexists p \in \Sigma_D : p \cap sym$ **then**
        $\Sigma_D = \Sigma_D \cup \{sym\}$
        $M = M \cup (sym \to sym)$
    else
        Remove conflicting symbol from alphabet $\Sigma_D = \Sigma_D \setminus \{p\}$
        Remove conflicting symbol from mapping $\forall x \in \Sigma_N : M = M \setminus \{(x \to p) : \forall x\}$
        Remove intersection from symbol p: $p_{new} = p - sym \cap p$
        Remove intersection from new symbol: $sym_{new} = sym - sym \cap p$
        Discover set of original symbols p_o of p, such as $(p \to p_o) \in M_{back}$
        Discover set of original symbols of sym sym_o, such as $(sym \to sym_o) \in M_{back}$
        Add intersection to mapping $\forall p_i \in p_o : M = M \cup (p_i \to p \cap sym)$
        Add intersection to mapping $\forall sym_i \in sym_o : M = M \cup (sym_i \to p \cap sym)$
        Add intersection to mapping $M_{back} = M_{back} \cup (p \cap sym \to p_o \cup sym_o)$
        Add intersection into alphabet $\Sigma_D = \Sigma_D \cup \{p \cap sym\}$
        for $s \in p_{new}$ **do**
            Add a symbol from difference into M: $\forall p_i \in p_o : M = M \cup (p_i \to s)$
            Add a symbol from difference into M_{back}: $M_{back} = M_{back} \cup (s \to p_o)$
            $\Sigma_D = \Sigma_D \cup \{sym\}$
        end for
        Order sym_{new} by size of symbol
        for $s \in sym_{new}$ **do**
            Add a symbol from difference into M: $\forall sym_i \in sym_o : M = M \cup (sym_i \to s)$
            Add a symbol from difference into M_{back}: $M_{back} = M_{back} \cup (s \to p_o)$
            Recursively call AddSymbol(s,Σ_D,M,M_{back})
        end for
    end if
end function
Order Σ_N by the size of pattern symbols
$\Sigma_D = \emptyset$
$M = \emptyset$
$M_{back} = \emptyset$
for $s \in \Sigma_N$ **do**
    AddSymbol(s,Σ_D,M,M_{back})
end for

Chapter 5

Analysis of regular expressions used in Intrusion Detection Systems

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 [27] and in [26]. The presented experiments were performed for the L7 and Snort ruleset.

5.1 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 3 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 8. ***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.

5.1.1 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[4] dataset and L7 rules [1].

Figure 5.1 reports the saturation of automata generated from the Snort 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 1. The histogram presents automata with three different optimisations.

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 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 5.1.

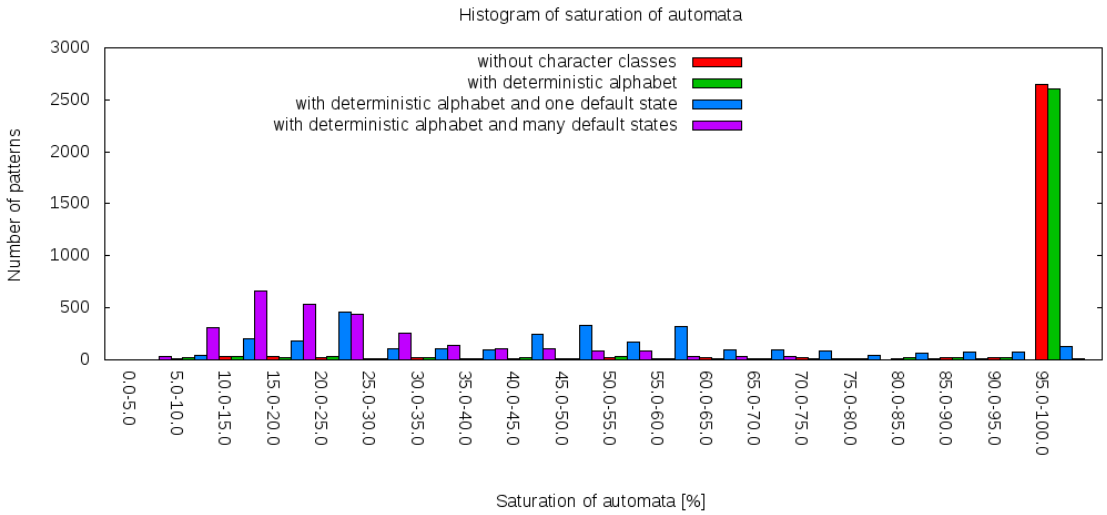


Figure 5.1: Saturation of transition table of DFA with deterministic alphabet created from Snort regular expressions. Histogram show how many patterns are with saturation presented at the x-axis. The x-axis is annotated in percentage. The level of saturation is shown for various optimisation methods.

deterministic alphabet, one default state The introduction of the default state reduces the size of automata and produces automata with low saturation. The combination of deterministic alphabet and one default state produces automata with larger saturation then automata without character classes and with default state.

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 5.1. The comparing these two optimisations according to the size of automata indicates that *many default state* optimisation reduces the size of the automaton.

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.

5.1.2 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.

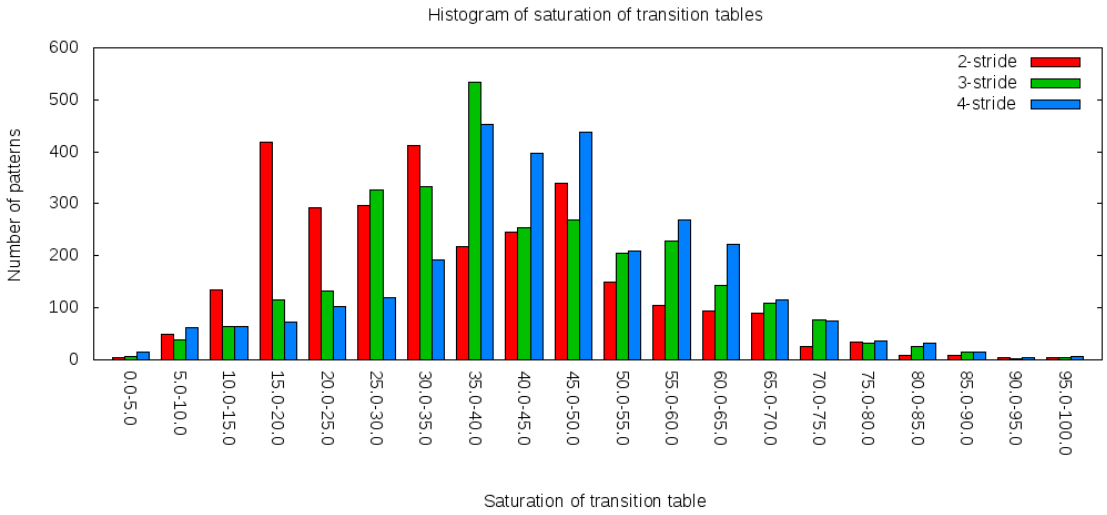


Figure 5.2: Saturation of transition table of pattern automata created from Snort regular expressions. Histogram shows how many pattern creates automata with saturation of transition table presented at the x-axis. The axis is annotated in percentages. The level of saturation is shown for various number of character accepted by one transition of pattern automaton.

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.

Experiments with pattern automaton was published by Kastil and collective [24, 28] or in czech language [29].

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 5.2 describes the distribution of saturations of transition table for pattern automata with *many default states* generated from Snort ruleset. 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, 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.

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.

Chapter 6

Perfect Hashing based Deterministic Automata

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 [8]. 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 6.1 shows more detailed version of the architecture.

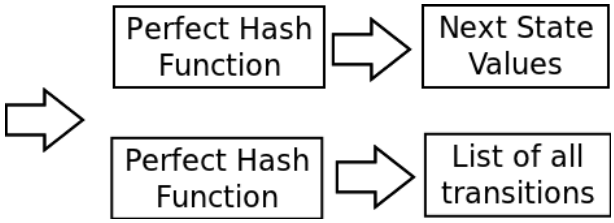


Figure 6.1: Perfect Hashing Automaton

It can be seen from figure 6.1, 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 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 6.2 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.

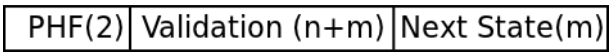


Figure 6.2: Structure of memory line

Numbers in the bracket in the figure 6.2 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 6.3. 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.

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.

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

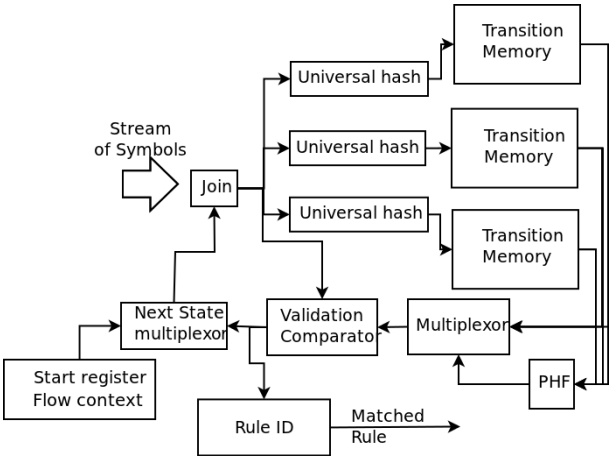


Figure 6.3: Perfect Hashing Automaton

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 [13] in time of 11 - 21 clock cycles [12]. 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 [14]. 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 [15] 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 6.3, 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. 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.

6.0.3 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 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 6.3. 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.

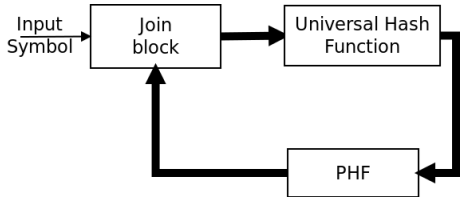


Figure 6.4: Pipelining in Perfect hashing

The figure 6.4 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 6.5

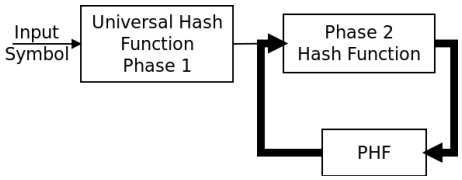


Figure 6.5: Pipelining in Perfect hashing

The bold lines in the figure 6.5 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.

6.1 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 [8] 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 [23].

6.1.1 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 9. *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. 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[1] 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.

6.1.2 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 6.1.1.

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 6.6 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

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

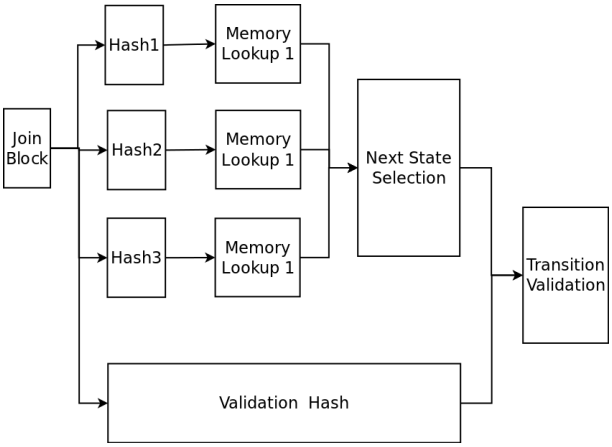


Figure 6.6: Time for computation of validation hash

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.

6.1.3 Reliability of probability Automata

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 6.1 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.

$$f_r(k) = (1 - P_{fail})^k \tag{6.1}$$

The mean number of symbols between errors of the probability automaton t_c can be computed from the f_r by the equation 6.2.

$$t_c = \sum_{\forall i \in \mathbb{N}_1} i \times f_r(i) \tag{6.2}$$

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 6.7 shows the dependency of mean number of symbols between errors and the amount of memory available for transition validation.

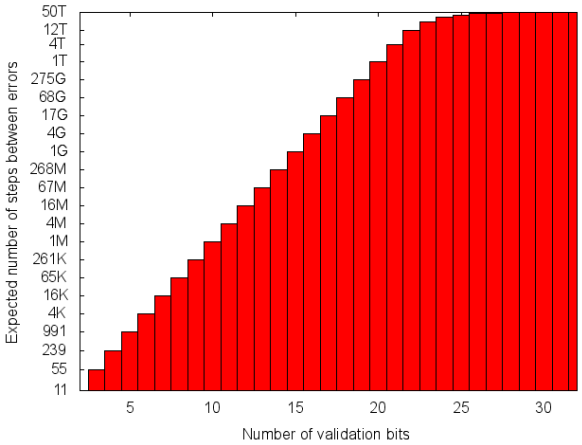


Figure 6.7: The expected mean number of symbols between errors of probability automata in number of processed symbols

The Y axes of figure 6.7 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.

The maximal size of IPv4 datagram is 2^{16} bytes [46], while IPv6 increases this limit up to $2^{32} - 1$ bytes [6]. 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.

Chapter 7

Experimental Results

The experimental results in chapter 5 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 6 described perfect hashing automaton and probability automaton pattern matching unit. This chapter reports the results of experiments performed with these two hardware architectures.

7.1 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.

7.1.1 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 7.1 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 7.1 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

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 7.1: Properties of hardware implementation of PHFDFA with 1 cycle latency

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 matching to 3 clock cycles. Table 7.2 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 7.2 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.

7.1.2 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.

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 7.2: Properties of hardware implementation of PHFDFA with 3 cycle latency

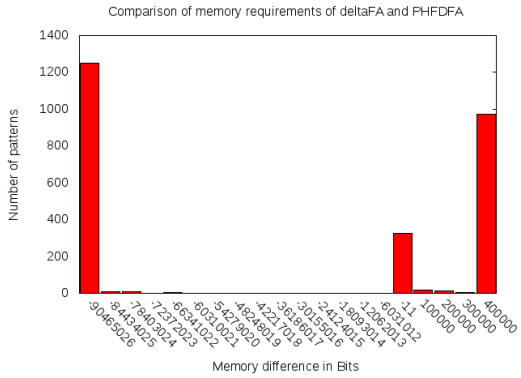
The δDFA accepts only one symbol per transition which limits its throughput. To increase throughput, several parallel instances of δDFA has to be run at once which increases the amount of memory required for pattern matching. The PHFDFA and Probability Automaton may accept arbitrary number of symbols 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 7.1. 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.

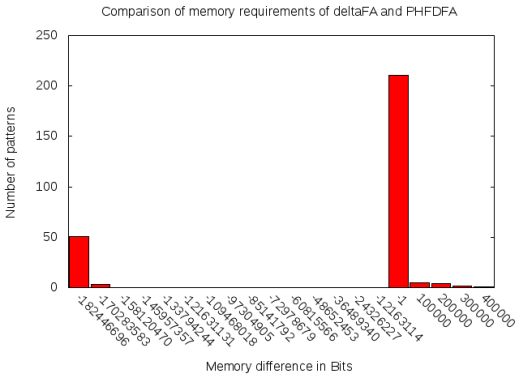
Figure 7.1a compares the PHFDFA with δDFA for Snort ruleset. The negative value on the x-axis means that the δDFA is more efficient while positive value indicates that these patterns should be implemented by PHFDFA. It can be seen, that patterns are divided into three groups. The first group contain 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 7.1b 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.

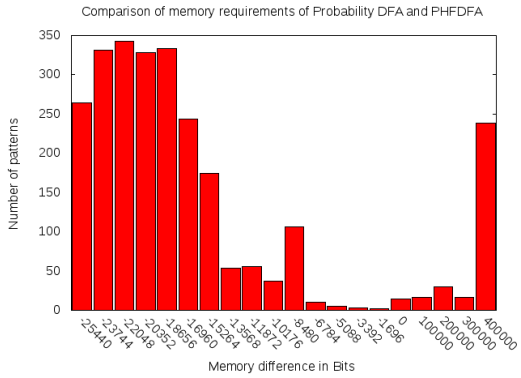
The comparison between PHFDFA and probability automaton for Snort ruleset is shown



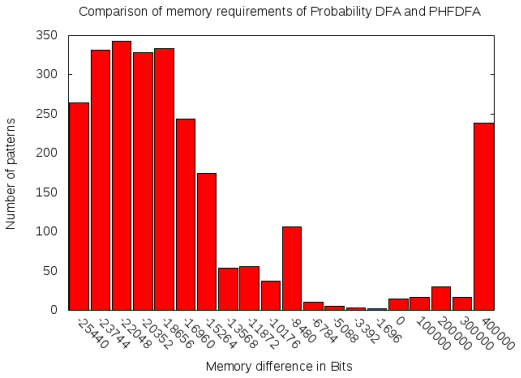
(a) The difference in number of bits required to store transition table for δDFA a PHFDFA. Histogram shows how many pattern in the Snort ruleset has at least memory savings presented on x-axis if implemented by PHFDFA instead of δDFA.



(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 7.1: Comparison of memory requirements of different architectures

in Figure 7.1c. 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 7.1a. 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.

Chapter 8

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 3 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 4 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 [24] 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 5 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 [26]. 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 [24, 29, 28] while the second architecture, called *Probability automaton*, allows small portion of errors in the matching process to reduce memory requirements of the system [23]. The analysis performed in chapter 6.1.3 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 [25]. The memory used for implementation of transition table is the limiting factor of the clock frequency of both designs. The measurement performed in chapter 7 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.

8.1 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] Internet world stats. <www.internetworldstats.com/stats.htm>.

[3] Service name and transport protocol port number registry. <www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>.

[4] Snort homepage. <www.snort.org>.

[5] 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.

[6] D Borman, S Deering, and R Hinden. Rfc 2675: Ipv6 jumbograms, 1999.

[7] 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.

[8] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

[9] 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.

[10] 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.

[11] 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.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*

[13] 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>.

[14] J-P. Deschamps and G. Sutter. Comparison of FPGA implementation of the mod M reduction. *Latin American applied research*, 37:93 – 97, 01 2007.

[15] PM Ebert, James E Mazo, and Michael G Taylor. Overflow oscillations in digital filters. *Bell System Technical Journal*, 48(9):2999–3020, 1969.

[16] 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.

[17] 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.

[18] Ziemba G. et al. Rfc 1858: Security considerations for ip fragment filtering. 1995.

[19] 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.

[20] 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.

[21] 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.

[22] AnnaR. Karlin, HowardW. Trickey, and JeffreyD. Ullman. Algorithms for the compilation of regular expressions into plas. *Algorithmica*, 2(1-4):283–314, 1987.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Jan Kaštil. Vyhledávání regulárních výrazů ve vysokorychlostním síťovém provozu. In *Počítačové architektury a diagnostika 2009*, pages 89–94. Tomas Bata University in Zlín, 2009.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] Cynthia Bailey Lee, Chris Roedel, and Elena Silenok. Detection and characterization of port scan attacks, 2003.

[37] 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.

[38] 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.

[39] Peter Linz. *An Introduction to Formal Language and Automata.* Jones and Bartlett Publishers, Inc., USA, 2006.

[40] 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.

[41] 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.

[42] Alexander Meduna. *Automata and languages: theory and applications.* Springer-Verlag, London, UK, 2000.

[43] K. Namjoshi and G. Narlikar. Robust and fast pattern matching for intrusion detection. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, March 2010.

[44] 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.

[45] University of Southern California. Rfc 793:transmission control protocol. 1981.

[46] Jon Postel et al. Rfc 791: Internet protocol. 1981.

[47] Elaine Rich. *Automata, computability and complexity: theory and applications.* Pearson Prentice Hall Upper Saddle River, 2008.

[48] K. Rowett and S. Sikdar. Intrusion detection system, September 29 2005. US Patent App. 11/125,956.

[49] 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.

[50] Abhijit Sarmah. Intrusion detection systems: Definition, need and challenges. October 2001.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] Jeffrey D. Ullman. Combining state machines and regular expressions for automatic synthesis of vlsi circuits. Technical report, Stanford, CA, USA, 1982.