FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Generating NFAs for Hardware Accelerated Pattern Matching through High-Level Synthesis

Pedro Miguel Ribeiro Alves

Master in Informatics and Computing Engineering

Supervisor: Prof. Nuno Miguel Cardanha Paulino

Co-Supervisor: Prof. João Manuel Paiva Cardoso

February 15, 2024

Generating NFAs for Hardware Accelerated Pattern Matching through High-Level Synthesis

Pedro Miguel Ribeiro Alves

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

President: Prof. João Carlos Viegas Martins Bispo
Referee: Prof. Mário Pereira Véstias
Referee: Prof. Nuno Miguel Cardanha Paulino

February 15, 2024

Abstract

Signature-based Intrusion Detection Systems (IDSs) rely on searching incoming packets for known patterns that may pose a threat to a system. Such patterns are often represented through a list of Regular Expressions (RegExes), as is the case of SNORT, one of the most well-known Network Intrusion Detection Systems (NIDSs).

As the amount of data shared amongst computer networks keeps getting bigger due to an increase in traffic and available bandwidth, signature-based IDSs face a greater demand as they are required to monitor an ever-growing number of packets. This means that, sometimes, software implementations of RegExes pattern-matching algorithms are not fast enough in order to monitor every packet in a timely fashion.

In these cases, one can resort to Hardware Accelerators, such as Field-Programmable Gate Arrays (FPGAs). Although these devices are usually designed through a Register-Transfer Level (RTL) specification, High-Level Synthesis (HLS) has made designing programmable logic more accessible by being capable of translating a functional specification of a system written in a high-abstraction language such as C or C++ to an RTL logic specification.

HLS designs of RegEx pattern-matching algorithms on Programmable Logic Devices are still a majorly unexplored field. With that in mind, a RegEx matching solution based on Non-deterministic Finite Automata (NFAs), developed with HLS, is proposed. This solution provides support for Perl Compatible Regular Expressions (PCRE) features which are often absent from such hardware-based RegEx matching proposals, namely back-references and, to a lesser extent, bounded quantifiers.

Support for both back-references and bounded quantifiers is offered along with multiple options regarding their implementation. Results for the RegEx matching solution's general performance are presented, as well as results for the individual efficiency of these two operators, accompanied by an analysis of their corresponding implementation alternatives.

A characterization of existing RegExes rulesets is also performed, with various features regarding their composition and their expressions' inner structure being extracted and analyzed. This serves both as its own standalone contribution, offering an insight into the current state of RegEx utilization across various domains, and as a way to benchmark the proposed matching solution, allowing for the quantification of the coverage of use cases it provides.

Keywords: Finite Automata, Regular Expression, Hardware Accelerators, FPGA, High-Level Synthesis

ACM Classification: • Hardware → Integrated circuits → Reconfigurable logic and FPGAs; • Theory of computation → Formal languages and automata theory → Regular languages;

Resumo

Os Intrusion Detection Systems (IDSs) baseados em assinaturas baseiam-se na procura de padrões conhecidos nos pacotes de entrada que possam representar uma ameaça para um sistema. Tais padrões são frequentemente representados através de uma lista de Regular Expressions (RegExes), como é o caso do SNORT, um dos mais conhecidos Network Intrusion Detection Systems (NIDSs).

Como a quantidade de dados partilhados entre redes de computadores continua a aumentar devido ao aumento do tráfego e da largura de banda disponível, os IDSs baseados em assinaturas enfrentam um maior desafio, uma vez que são obrigados a monitorizar um número cada vez maior de pacotes. Isto significa que, por vezes, as implementações de software dos algoritmos de correspondência de padrões RegExes não são suficientemente rápidas para monitorizar todos os pacotes em tempo útil.

Nestes casos, pode-se recorrer a aceleradores de hardware, como Field-Programmable Gate Arrays (FPGAs). Embora estes dispositivos sejam normalmente configurados através de uma especificação Register-Transfer Level (RTL), o High-Level Synthesis (HLS) tornou a configuração de lógica programável mais acessível ao ser capaz de traduzir uma especificação funcional de um sistema escrito numa linguagem de elevada abstração, como C ou C++, para uma especificação lógica RTL.

O uso de HLS para a implementação de algoritmos de correspondência de padrões RegEx em dispositivos de lógica programável são ainda um campo muito pouco explorado. Com isso em mente, é proposta uma solução de correspondência de RegExes baseada em Non-deterministic Finite Automata (NFAs), desenvolvida com HLS. Esta solução oferece suporte para operadores Perl Compatible Regular Expressions (PCRE) que estão frequentemente ausentes de tais propostas de correspondência RegEx baseadas em hardware, nomeadamente back-references e, em menor grau, quantificadores delimitados.

O suporte para back-references e quantificadores delimitados é oferecido juntamente com várias opções relativamente à sua implementação. São apresentados resultados para o desempenho geral da solução de correspondência de RegExes, bem como resultados para a eficiência individual destes dois operadores, acompanhados de uma análise das suas respetivas alternativas de implementação.

É também efectuada uma caraterização de conjuntos de regras RegExes existentes, sendo extraídas e analisadas várias características relativas à composição destes conjuntos e à estrutura interna das suas expressões. Isto serve tanto como uma contribuição individual, proporcionando uma perspetiva sobre o estado atual da utilização de RegEx em vários domínios, como também uma forma de avaliar a solução de correspondência proposta, permitindo a quantificação da cobertura dos casos de utilização que esta fornece.

"We believe that we invent symbols.
The truth is that they invent us;
we are their creatures, shaped by their hard defining edges."

Gene Wolfe, The Shadow of the Torturer

Contents

1 Introduction **1**
- 1.1 Context and Motivation . . . 1
- 1.2 Objectives . . . 1
- 1.3 Document Structure . . . 2

2 Background **3**
- 2.1 Regular Expressions . . . 3
  - 2.1.1 Finite Automata . . . 3
  - 2.1.2 Deterministic and Non-deterministic Finite Automata . . . 3
  - 2.1.3 Regular Expression Syntax . . . 4
  - 2.1.4 Back-references as Non-regular Syntax Extensions . . . 5
- 2.2 FPGAs and FPGA Programming . . . 5
  - 2.2.1 FPGAs . . . 5
  - 2.2.2 HLS FPGA Development . . . 6
- 2.3 High-Level Synthesis . . . 7
  - 2.3.1 Data Streaming . . . 8
  - 2.3.2 Pipelining . . . 8
  - 2.3.3 Loops . . . 8
  - 2.3.4 Unsupported Constructs . . . 8

3 State of the Art **10**
- 3.1 NFA Construction . . . 10
- 3.2 Character Decoding . . . 11
- 3.3 Affix Sharing . . . 11
- 3.4 Multi-Character Input Matching . . . 12
- 3.5 Overview . . . 13
- 3.6 Analysis . . . 16

4 Proposed Approach **17**
- 4.1 Collected Rulesets . . . 17
- 4.2 Ruleset Features . . . 18
- 4.3 Software Artifacts . . . 19
  - 4.3.1 Regular Expression (RegEx) Analyzer . . . 20
  - 4.3.2 RegEx Matcher . . . 21

5 RegEx Matcher Implementation **24**
- 5.1 Epsilon-NFA Generation . . . 24
  - 5.1.1 Supported RegEx Operators and Flags . . . 24

5.1.2 Epsilon-Non-deterministic Finite Automaton (NFA) Creation 24
5.2 Transformation into NFA 36
5.2.1 Epsilon Transition Removal 36
5.2.2 Capture Transition Removal 39
5.2.3 Counter Transition Removal 40
5.2.4 Anchor Transition Removal 41
5.2.5 Extended NFA Formal Definition 42
5.3 Code Generation 42
5.3.1 Template Engine 43
5.3.2 General Template Implementation 44
5.3.3 Capture Groups and Back-references 47
5.3.4 Bounded Quantifiers 49

6 Experimental Results **51**
6.1 Ruleset Features Analysis 51
6.1.1 Ruleset Size 51
6.1.2 Operator and flag utilization 52
6.1.3 Expression length 53
6.1.4 Capture group length 55
6.1.5 Bounded quantifier repetitions 56
6.2 RegEx Matcher 57
6.2.1 Supported RegExes 57
6.2.2 Experimental setup 58
6.2.3 General performance 58
6.2.4 Capture group and back-reference performance 60
6.2.5 Bounded quantifier performance 62

7 Conclusion and Future Work **65**

References **68**

A Ruleset Analysis Results **73**

List of Figures

2.1 Automata for $(\texttt{a}|\texttt{b})^*\texttt{b}$ . . . 4
2.2 Simple LUT implementation of a 2-bit XOR operation . . . 6
2.3 HLS steps (Source: [12]) . . . 7
2.4 Pipelining of a three-step operation . . . 8

3.1 NFA construction rules . . . 10
3.2 NFA construction for $(\texttt{a}|\texttt{b})^*\texttt{c}$ . . . 11
3.3 Sub-expression sharing . . . 12

4.1 Syntax tree for $x = 3 \times (7-2)$ . . . 20
4.2 Architectural Diagram for the Regular Expression (RegEx) Analyzer . . . 21
4.3 Equivalent NFAs representing $(ab)^+$ . . . 22
4.4 Architectural Diagram for the RegEx Matcher . . . 23

5.1 Syntax Tree for $[bc]a^+|c$ . . . 26
5.2 Automaton for a . . . 27
5.3 Automaton for . . . 27
5.4 Character Class Automata . . . 27
5.5 Automaton for $\backslash 1$ . . . 28
5.6 Anchor Automata . . . 28
5.7 Concatenation Construction Rule for $r_1 r_2$ . . . 28
5.8 Alternation Construction Rule for $r_1|r_2$ . . . 29
5.9 Quantifier Construction Rule for r_1* . . . 29
5.10 Quantifier Construction Rule for r_1+ . . . 30
5.11 Quantifier Construction Rule for $r_1?$ . . . 31
5.12 Automaton for $a\{4,6\}$ generated through expression repetition . . . 31
5.13 Automaton for $a\{4,6\}$ generated through the extended transition set . . . 32
5.14 Generated automaton for $(a|b)$ . . . 34
5.15 User interface showcasing the available automata generation flags . . . 34
5.16 Automata visualization enabled through the debug flag . . . 35
5.17 Non-deterministic Finite Automata (NFAs) for ab before/after the application of Algorithm 2 (the unreachable state 3 will then be removed by the application of Algorithm 3) . . . 37
5.18 NFA matching flowchart . . . 44
5.19 Simplified NFA for $(ab\{2,4\})\{3\}$ . . . 45
5.20 Simplified automaton for $.^*(a^+b)\backslash 1$ . . . 49
5.21 Simplified automaton for $.^*a\{3\}$ . . . 50

6.1 Total operator frequency . . . 52

6.2 Operator occurrence by expression 53
6.3 Flag occurrence 54
6.4 Maximum input string lengths 54
6.5 Maximum capture group lengths 55
6.6 Maximum referenced capture group lengths 56
6.7 Maximum repetitions for bounded quantifiers 57
6.8 FF and LUT usage for both bounded quantifier implementations 63
6.9 Achieved clock period and BRAM usage for both bounded quantifier implementations 63

List of Tables

2.1 Subset of PCRE operators 4

3.1 Summary and comparison of state-of-the-art Regular Expression (RegEx) acceleration approaches 14

5.1 Supported Operators 25
5.2 Supported Flags 25

6.1 Number of RegExes per ruleset 51
6.2 Supported expressions of each RegEx ruleset 58
6.3 Resulting Non-deterministic Finite Automata (NFAs)' descriptions 59
6.4 Synthesis results of the expressions' NFAs 60
6.5 Capture group/non-capture group resource usage comparison 61
6.6 Back-reference resource usage comparison 62

A.1 Total Perl Compatible Regular Expressions (PCRE) operator utilization 73
A.2 Number of expressions using each PCRE operator 73
A.3 Total PCRE flag utilization 73
A.4 Expressions' maximum input string lengths 74
A.5 Maximum capture group lengths 75
A.6 Maximum referenced capture group lengths 75
A.7 Maximum bounded quantifier repetitions 76

Listings

4.1 ANTLR Grammar for simple arithmetic 19
5.1 ANTLR rule defining capture groups 33
5.2 Recursive method of extracting a state's epsilon closure 37
5.3 FreeMarker workflow example 43
5.4 Topmost function template code 44
5.5 State structure 45
5.6 Transfer of counter variables 45
5.7 NFA state transition logic template section 46
5.8 State transition code for [ab]c 46
5.9 Template code for the repetition of the transition logic 47
5.10 Transition logic repetition code for [ab]c 47
5.11 Capture Buffer Structure 47
5.12 Buffer insertion function 48
5.13 Buffer comparison function 48
5.14 Counter structure 49
5.15 Counter equality function 49
6.1 Subset of Protomata Regular Expressions (RegExes) 55
6.2 Perl Compatible Regular Expressions (PCRE) with reference to constant group . 56
6.3 Chosen PCREs used for results extraction 58

Abbreviations

ASIC	Application-Specific Integrated Circuit
BRAM	Block Random-Access Memory
CFG	Control Flow Graph
DFA	Deterministic Finite Automaton
DFG	Data Flow Graph
FA	Finite Automaton
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
HLS	High-Level Synthesis
I/O	Input/Output
IDS	Intrusion Detection System
LUT	Lookup Table
NFA	Non-deterministic Finite Automaton
NIDS	Network Intrusion Detection System
PCRE	Perl Compatible Regular Expressions
PLD	Programmable Logic Device
RegEx	Regular Expression
RTL	Register-Transfer Level
SRAM	Static Random-Access Memory

Chapter 1

Introduction

1.1 Context and Motivation

The purpose of Regular Expressions (RegExes) is to search for specific occurrences of complex textual patterns in a given dataset. They have a wide variety of applications in different fields such as data mining [3, 15], natural language processing [21], genetic analysis [39] and deep packet inspection [46].

Deep packet inspection is especially important for Network Intrusion Detection Systems (NIDSs) since some of these security systems rely on the analysis of incoming packets in order to identify attacks on the network, as is the case of SNORT [7]. This analysis is done through a search for certain content patterns, stored as a set of rules, known to potentially be malicious [2].

The constant increases in network traffic [23] and available bandwidth pose a challenge to the NIDSs' task of analyzing data in real time. This means that the speed of RegEx matching algorithms is a critical factor in preventing them from becoming a bottleneck to the network or causing the NIDS to miss packets.

Hardware-based implementations of RegEx matching algorithms present themselves as a solution to this challenge, representing a faster alternative to software implementations of the same algorithms [40, 29]. Field-Programmable Gate Arrays (FPGAs) have been vastly explored as a platform to provide fast RegEx matching [40, 29, 50, 4, 25], with some of their innate characteristics, mainly their reconfigurability, which allows them to adapt to the constant updates of NIDSs' rulesets, making them a promising hardware accelerator for this endeavor.

1.2 Objectives

While most proposals of RegEx matching algorithms implementations on FPGAs resort to Register-Transfer Level (RTL) designs, this work aims to implement a Non-deterministic Finite Automaton (NFA)-based RegEx matching algorithm using High-Level Synthesis (HLS). The proposed solution features support for bounded quantifiers and back-references, which are Perl Compatible Regular Expressions (PCRE) features that are often omitted from such RegEx matching proposals.

Besides the matcher, a characterization of selected RegEx rulesets is also presented, serving as its own standalone contribution, as it provides an overview of the current state of RegEx utilization across various domains. The ruleset characterization focuses on extracting and analyzing select attributes from the RegExes contained within the various rulesets, namely the length of the expressions, PCRE operator usage, capture group length, and bounded quantifier repetitions. This analysis also proves to be useful when it comes to discussing the results of the proposed RegEx matcher and its different implementation options.

1.3 Document Structure

This dissertation is composed of seven chapters: an introduction of the document outlining its context, motivation, and objectives (Chapter 1); an overview of some key background topics (Chapter 2); a presentation of the current state of the art (Chapter 3); a description of the approach taken to fulfill the proposed objectives (Chapter 4); an in-depth explanation of the implementation of the RegEx matcher (Chapter 5); the presentation of the achieved results (Chapter 6); a final conclusion with the main takeaways and future work (Chapter 7).

Chapter 2

Background

2.1 Regular Expressions

In this section, the underlying concepts inherent to the implementation of Regular Expression (RegEx) pattern-matching algorithms will be explored, as well as how RegEx patterns are defined, particularly focusing on the back-reference operator.

2.1.1 Finite Automata

Kleene's theorem [34] implies that every language defined by a regular expression can be represented by a finite automaton and vice-versa.

A Finite Automaton (FA) is an algebraic structure defined for an input alphabet Σ, composed of a set of finite states Q, an initial state q_0, a set of accepting states F, and a transition function δ, which takes state q_i and input I_j and outputs the set of next states Q_{i+1} [37].

$$Q_{i+1} = \delta(q_i, I_j)$$

An input string I is accepted by the automaton if, after processing every symbol I_j in I, an accepting state is reached.

2.1.2 Deterministic and Non-deterministic Finite Automata

Deterministic Finite Automata (DFAs) and Non-deterministic Finite Automata (NFAs) differ in the size of the set Q_{i+1} obtained through the application of the transition function for a given state: while the size of Q_{i+1} is unlimited for NFAs, it has a maximum of one element for DFAs. This means that in a NFA, a given state can have multiple transitions with the same input symbol, while a DFA state has, at most, a single transition per input symbol.

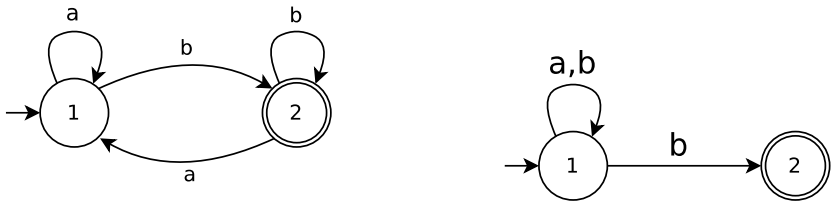
Every NFA can be converted into a DFA [35]. However, both these types of FAs have distinct characteristics.

NFAs typically have a lower number of states. For a n-state NFA, an equivalent DFA requires at most 2^n states. [28]

The traversal of a DFA is straightforward due to each state transitioning to only one state given a certain input. Because of their non-deterministic nature, the same cannot be said for NFAs. This leads to different approaches when it comes to parsing an input string in the case of NFAs.

One option is to follow possible paths through the NFA for the given input string until one reaches an accepting state. This constitutes a recursive depth-first search with backtracking, which for a n-state NFA might lead to an exploration of 2^n paths.

An alternative is to consider multiple states as active at the same time. In this approach, a set of active states is kept, and as the input string is parsed, the set of active states is updated according to the transition function. This results in various NFA paths being explored concurrently.



(a) DFA (b) NFA

Figure 2.1: Automata for $(\texttt{a}|\texttt{b})^*\texttt{b}$

2.1.3 Regular Expression Syntax

Originally, RegExes were defined as a sequence of symbols and three operators: union, concatenation, and iterate [5] (also known as the Kleene star operator). Modern RegEx syntax has been expanded in order to increase their expressiveness, with new features being added, such as character classes, wildcards, anchors, bounded quantifiers, and back-references.

While there isn't a single widely accepted standard for the definition of RegExes, the Perl Compatible Regular Expressions (PCRE) specification is very commonly used. Table 2.1 presents a subset of PCRE operators.

Table 2.1: Subset of PCRE operators

Operator	Description	Example Regex	Matching Strings
$*$	Zero or more occurrences	$\texttt{ab}^*$	a, ab, abb...
$+$	At least one occurrence	$\texttt{ab}^+$	ab, abb, abbb...
?	Zero or one occurrence	$\texttt{ab}^?$	a, ab
()	Group	$(\texttt{ab})^+$	ab, abab, ababab...
$\vert$	Or	$\texttt{a}\vert\texttt{b}$	a, b
.	Match Any character (Wildcard)	$\texttt{a}.^*$	a, ab, acd, a...
[...]	Match any character within the brackets (Character Class)	$\texttt{a}[\texttt{bc}]$	ab, ac
$\{\texttt{n},\texttt{m}\}$	Between n and m occurrences (Bounded Quantifier)	$\texttt{a}\{2,3\}$	aa, aaa
$\backslash\texttt{n}$	Match the *nth* group (Back-Reference)	$([\texttt{ab}])\texttt{c}\backslash\texttt{1}$	aca, bcb

2.1.4 Back-references as Non-regular Syntax Extensions

One of the most useful extensions introduced to the syntax of RegExes are back-references. This operator allows for previously matched input, in the form of a capture group, to be referenced within the expression. The pattern (”|′).*\1, for example, allows matching an expression's opening quotation marks to its closing ones. The capture group (”|′) stores if either single or double quotation marks were first used, and then they're matched by referencing this group via \1. In \n, *n* refers to the order of the group being referenced by its order of appearance in the expression.

Back-references represent a non-regular extension to the RegEx' syntax, as they fall outside the domain of regular languages due to their reliance on previous input, creating the need for some type of memory keeping. This means that these extended RegExes can no longer be defined by simple FAs, which makes their implementation non-trivial.

Back-references are particularly hard to implement on devices such as Field-Programmable Gate Arrays (FPGAs). While these devices are capable of having memory storage, their memory resources are fairly limited, and the overuse of these resources can lead to major performance losses. This represents a restriction on the length of the back-references that can be supported. Expressions that have back-references to groups of unknown lengths, such as (a$^+$)c\1, pose an even greater challenge, as it's impossible to know beforehand how many memory units have to be allocated (see Subsection 2.3.4) or even if there are enough available.

2.2 FPGAs and FPGA Programming

This section introduces the concept of FPGAs and the use of High-Level Synthesis (HLS) as their configuration method.

2.2.1 FPGAs

FPGAs belong to a family of integrated circuits known as Programmable Logic Devices (PLDs), which are devices whose hardware can be reconfigured in order to model other digital circuits [16]. This allows FPGAs to provide specialized solutions with lower costs and reduced turnaround time when compared to non-reconfigurable hardware accelerators such as Application-Specific Integrated Circuits (ASICs) [26]. Their ability to be reprogrammed also means that they are able to cope with the ever-occurring updates to the Network Intrusion Detection Systems (NIDSs)' rule sets, making them a promising candidate to be used within this context.

Simple FPGAs are primarily composed of three types of elements: logic elements, Input/Output (I/O) elements, and wiring elements [16]. Logic elements are usually implemented as lookup tables (LUTs), which are memory representations of the truth table of an arbitrary logical expression in Static Random-Access Memory (SRAM). A k-input LUT is composed of 2^k bit memory cells and a 2^k-input multiplexer. A k-input LUT can perform up to 2^{2^k} different logical functions, which means that a k-input LUT can be implemented with, at most, 2^{k-n} n-input LUTs [36]. Figure 2.2 shows a LUT implementation of an XOR operation.

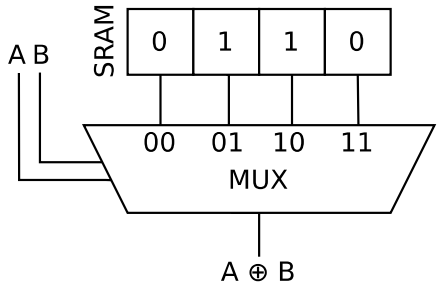


Figure 2.2: Simple LUT implementation of a 2-bit XOR operation

Wiring elements, in the form of both switch and connection blocks, serve as a connector between elements, while I/O elements act as an interface between the I/O pins and the FPGA's wiring elements.

FPGAs also have elements associated with their configuration or the storage of data, in the form of Block Random-Access Memory (BRAM).

2.2.2 HLS FPGA Development

The development of FPGA accelerators has been traditionally done through Hardware Description Languages (HDLs), such as Verilog and VHDL, which in turn are synthesized into a Register-Transfer Level (RTL) specification [1].

HLS, also known as behavioral synthesis, provides a higher-abstraction level alternative to HDLs that allows FPGA applications to be developed with general-purpose programming languages, for instance, C or C++, offloading the hardware description itself to the HLS tool [18].

Lahti et al. [22] present five major advantages of HLS:

1. The ease of the design effort due to a higher abstraction design environment. The focus of the design process is on the actual behavior of the system instead of its micro-architectural details, which in turn leads to a lower introduction of bugs.

2. The acceleration of the verification process. The behavior of the system is simpler and faster to test using software verification tools compared to RTL simulation tools.

3. A faster design space exploration. Small changes in the system's architecture can be explored through small tweaks in the HLS tools, requiring little source code modifications. Transformations such as pipelining and loop unrolling can easily be explored, while, in HDL, similar changes would require significant changes to the source code.

4. Increased code portability between platforms. For the same HLS code, different RTL descriptions can be synthesized according to the target platform.

5. Higher accessibility to software developers. HLS uses familiar general-purpose languages in contrast to more niche HDLs and abstracts most of the hardware-specific implementation details.

HLS tools go through various steps in order to convert a behavioral description of the system into an RTL description (see Figure 2.3), which is then itself converted into actual hardware through a logic synthesis tool. The first of these steps is the *compilation* of the behavioral description into a Data Flow Graph (DFG), representing the dependencies between data, and a Control Flow Graph (CFG), modeling the course of operations. An *allocation* step is performed in order to define the type and number of hardware resources needed. The *scheduling* process determines the order and timing of operations in terms of clock cycles, and the mapping of language constructs to hardware resources is known as *binding*. Finally, these last three independent steps are followed by the *generation* of a RTL description [12].

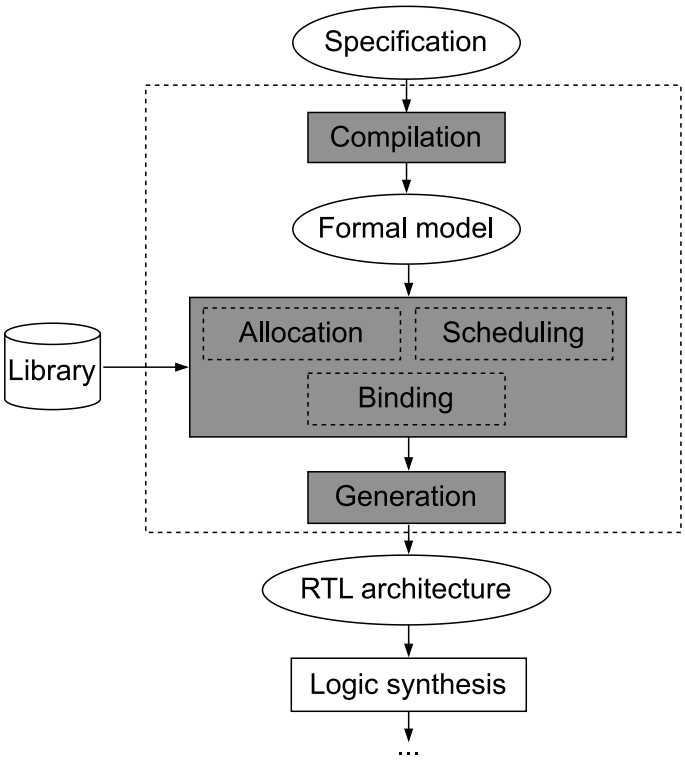


Figure 2.3: HLS steps (Source: [12])

2.3 High-Level Synthesis

Despite raising the abstraction level, HLS is still a tool for generating hardware descriptions of various systems, which implies a programming paradigm where this has to be kept in mind.

This section goes over various HLS programming concepts that are supported by Vitis HLS [1], a tool developed by Xilinx for the development of FPGA algorithms on their devices through C-like programming languages. Restrictions inherent to this programming paradigm are also presented.

[1] Vitis High-Level Synthesis User Guide https://docs.xilinx.com/r/en-US/ug1399-vitis-hls

2.3.1 Data Streaming

When developing hardware applications, exploiting sequential memory accesses is crucial in order to design highly performant solutions, as these are far more efficient than random memory accesses. Data streaming is the process of feeding data to a particular operation in a sequential and continuous fashion, taking full advantage of this notion.

2.3.2 Pipelining

Pipelining is a technique that attempts to achieve maximum resource utilization by splitting complex tasks into smaller independent tasks that can be performed concurrently. This allows for multiple instances of the main task in multiple completion stages to coexist concurrently.

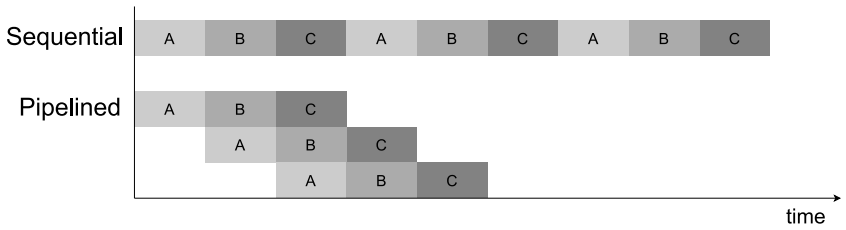


Figure 2.4: Pipelining of a three-step operation

2.3.3 Loops

Unrolling Loop unrolling allows some or all of a loop's iterations to occur in parallel instead of sequentially. An unrolled looped demands different hardware resources for its various iterations, as opposed to rolled loops which re-utilize a single piece of hardware. This results in a trade-off between performance and resource utilization.

Merging Automatic loop merging looks for loops that can be joined (loops with similar headers, for example) and merges their respective bodies in a single loop, reducing the total number of iterations.

Nested Loop Flattening Since it takes one clock cycle to move between inner and outer loops, transforming nested loops into an optimized single loop reduces the overall necessary clock cycles. This transformation is known as loop flattening and can only be performed when solely the innermost loop has statements in its body, and every inner loop has compile-time constant bounds.

2.3.4 Unsupported Constructs

In order for a behavioral description to be synthesizable, all the necessary resources for its hardware implementation have to be known at the time of compilation. This implies that higher-abstraction constructs that violate this notion are not supported, namely dynamic memory allocation and recursion.

The restriction on the dynamic allocation of memory presents a major issue for the implementation of back-references, as the length of some of these sub-expressions might be variable and, therefore, unable to be determined at the time of synthesis.

Chapter 3

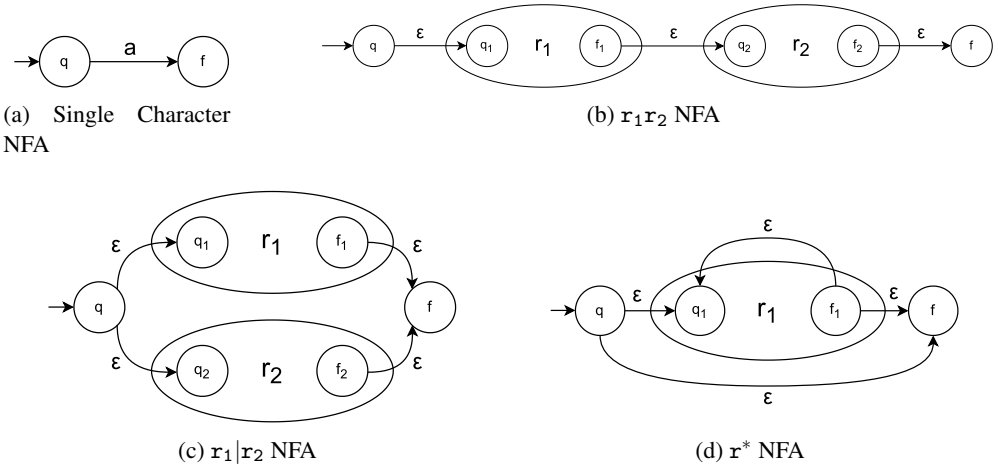
State of the Art

This chapter first highlights some valuable contributions regarding the implementation of Regular Expression (RegEx) pattern-matching engines on Field-Programmable Gate Arrays (FPGAs), and then provides a broader overview and discussion of recent related work.

3.1 NFA Construction

Sidhu and Prasanna [40] utilize a method for defining Non-deterministic Finite Automata (NFAs) from RegExes akin to the ones proposed earlier by Thompson [43], and both McNaughton and Yamada [27].

A set of rules for generating NFAs for an expression that matches a single character, and the expressions $\mathtt{r}_1\mathtt{r}_2$, $\mathtt{r}_1|\mathtt{r}_2$, and $\mathtt{r}^*$ are presented, as depicted in Figure 3.1. The states q_n and f_n represent the start and acceptance states of the automaton n, respectively, and a ε-transition represents a transition that can occur without consuming any input symbol.



(a) Single Character NFA

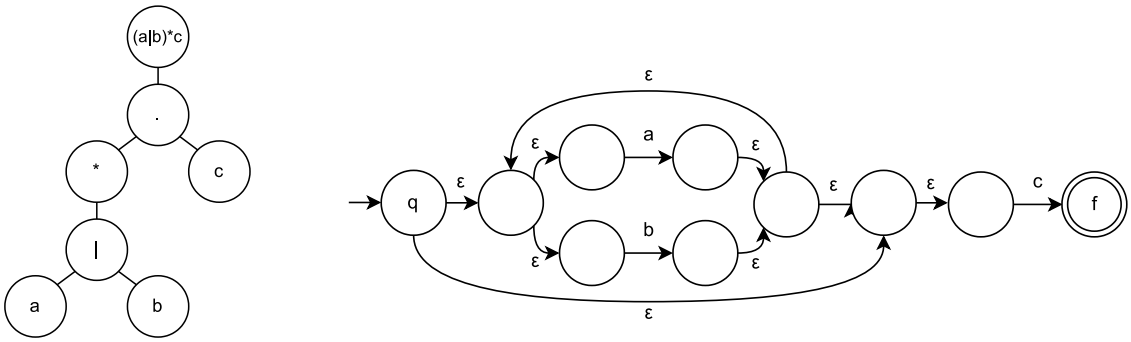
(b) $\mathtt{r}_1\mathtt{r}_2$ NFA

(c) $\mathtt{r}_1|\mathtt{r}_2$ NFA

(d) $\mathtt{r}^*$ NFA

Figure 3.1: NFA construction rules

Given that complex RegExes can be built by the concatenation of simpler ones, upon parsing an expression into its constituent sub-expressions and applying the aforementioned rules to these sub-expressions by traversing the parse tree in post-order, it is possible to obtain a NFA that models the original expression. As an example, Figure 3.2 shows an application of this method for $(\texttt{a}|\texttt{b})^*\texttt{c}$;



(a) Expression parse tree

(b) Expression NFA

Figure 3.2: NFA construction for $(\texttt{a}|\texttt{b})^*\texttt{c}$

3.2 Character Decoding

Clark and Schimmel [9] propose a method to greatly reduce the necessary resources for the implementation of a RegEx pattern-matching engine on a FPGA while also increasing its throughput.

It states that each character comparison unit does not need to perform a full 8-bit comparison. An 8-to-256 decoder can be used, and instead of broadcasting a character's full 8 bits to every character comparison unit, a unit would only receive a single bit corresponding to the output of the decoder for the given unit's target character.

This method reduces not only logic resources but also routing resources, as broadcasting 8 bits to each of the n comparison units would need $8n$ connections from the input character, while routing the single output bit to its respective unit would reduce the number of connections to n.

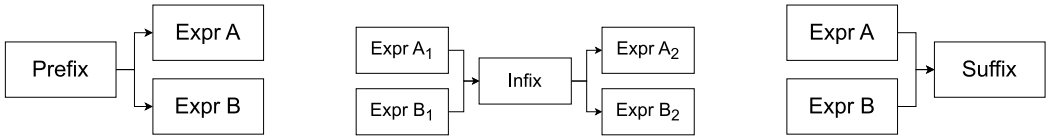
With this approach, and with the added optimization of re-utilizing expression prefixes, their design was capable of processing an 8-bit character each clock cycle, at 100 MHz, with a throughput of 800 Mb/s on a Xilinx Virtex-1000 FPGA, for the entire SNORT rule-set at the time.

3.3 Affix Sharing

Lin. et al. [24] present a way where unit blocks that match sub-expressions representing all types of affixes, not only prefixes but also infixes and suffixes, can be shared amongst expressions containing the same affixes.

Contrary to the re-utilization of sub-expression matching blocks for expressions with the same prefixes, a direct re-utilization of infixes and suffixes leads to an expression losing its identity after entering the re-utilized block. Cases such as the one presented in Figure 3.3b can lead to incorrect

matches such as $\texttt{Expr}\texttt{A}_1[\texttt{Infix}]\texttt{Expr}\texttt{B}_2$, where sub-expressions from expression *A* get mixed with sub-expressions from *B*. In the case of suffix sharing (see Figure 3.3c), the same loss of expression identity occurs, making it impossible to determine if the final matched expression was either *A* or *B*.



(a) Prefix Sharing (b) Infix Sharing (c) Suffix Sharing

Figure 3.3: Sub-expression sharing

This expression differentiation problem was solved by using a JK Flip-Flop to act as a switch, memorizing which expression triggered the entry in the affix matching module. This was then used in conjunction with a demultiplexer in order to guide the output of the affix module.

A problem with this sharing architecture was identified: the switch could be triggered by another expression while an infix sub-pattern was still in processing. Patterns `abcdefgh` and `dedefpq`, sharing the infix pattern `def`, are given as an example of this. For the input string 'abcdefgh', the switch would be triggered by the sub-string 'abc', and then, while still processing the sub-pattern `def`, the sub-string 'de' would trigger the switch a second time, leading to the input string incorrectly not being matched by the pattern `abcdefgh`.

This problem, named the critical section problem due to its similarity with software critical sections, was solved by identifying patterns where the sharing of infixes would result in this issue and avoiding sharing their infixes altogether.

For the SNORT rule-set and a Xilinx Virtex2 XC2V6000 as the target device, an approach supporting both affix-sharing and also character decoding as proposed in [9] was designed. It showed, on average, an area of 3273 slices, 2.5 characters per slice, a minimum period of 7.91 ns, and a throughput of 1065 Mb/s.

3.4 Multi-Character Input Matching

Yamagaki et al. [47] proposes a more efficient method for processing multiple characters per clock cycle compared to earlier propositions by Clark et al. [10] and Sutton [41]. An algorithm for transforming an NFA capable of processing n symbols per transition into an NFA with $2n$-symbol transitions is presented as a solution (see Algorithm 1). Repeating this algorithm k times would result in a 2^k-character NFA.

Targeting an Altera Stratix II (EP2S180) FPGA, for 2691 rules from the SNORT 2.4 ruleset, a 2-character NFA implementation achieved a throughput of 2.30 Gb/s, which represented an improvement of 184% when compared to a single-character NFA. This number increased to 3.63 Gb/s when using a 4-character NFA. An 8-character NFA resulted in a disproportional increase of the logic resources utilized as a result of the higher level of complexity in transition logic. Average

Algorithm 1 Multi-character NFA transformation [47]

Input: n-character NFA
Output: $2n$-character NFA

```
1:  add self-edge labeled ε to initial node
2:  for each final node f in NFA do
3:      add new final node m
4:      connect f and m with an edge labeled ε
5:  end for
6:  for each node n in NFA do
7:      for each node i having an edge to node k do
8:          for each node j having an edge from node n do
9:              if edge (n,j) is not line 1 self-edge then
10:                 add new edge (i,j)
11:                 concatenate labels of edges (i,n) and (n,j)
12:                 add concatenated label to the new (i,j) edge
13:             end if
14:         end for
15:     end for
16: end for
17: remove original NFA edges
18: remove edges inserted at lines 1-5
```

performance increases, with the numbers of supported RegExes ranging between 64 and 2691, were not proportional to the number of characters processed per transition, being approximately 170%, 250%, and 290% for 2-, 4-, and 8-character NFAs, respectively.

3.5 Overview

Table 3.1 aggregates various approaches regarding the matching of RegExes on FPGAs.

István et al. (2016) [17] Hardware-accelerated SQL queries specifically targeting the string-matching operators `LIKE` and `REGEXP_LIKE`, developed through *Intel's Altera Heterogeneous Architecture Research Platform*. This implementation is based on NFAs, which have their number of states reduced by identifying chains of consecutive characters, called tokens, and compressing each of these sequences into a single NFA state.

Wang et al. (2015) [45] FPGA implementation of a RegEx pattern-matcher for the ClamAV virus detection ruleset. Like the previous system, this approach compresses the underlying NFAs by operating through tokens, representing character segments instead of standalone characters. It is stated that this method can reduce the size of NFAs by over 95%. Furthermore, patterns that share tokens amongst them are identified and are processed through a specialized NFA known as a memory-based NFA [31].

		Implementation Details		Results	
Author (Year)	Rule Set (Domain)	Automata	Synthesis Method	Target Device	Throughput
Sidhu and Prasanna (2001) [40]	Snort (NIDS)	NFA-based	HDL (VHDL)	Virtex X CV100	Non-Disclosed
Clark and Schimmel (2003) [9]	Snort (NIDS)	NFA-based	HDL (JHDL)	Xilinx Virtex-1000	800 Mb/s
Lin et al. (2007) [24]	Snort and Trend Micro (NIDS)	NFA-based	HDL	Xilinx Virtex2 XC2V6000	Average of 1065 Mb/s (Snort)
Yamagaki et al. (2008) [47]	Snort (NIDS)	NFA-based	HDL	Altera Stratix II (EP2S180)	3.63 Gb/s
Wang et al. (2015) [45]	ClamAV (Malware Detection)	NFA-based	HDL	Xilinx Virtex-6	94 MB/s
Guo and Jiang (2015) [14]	Snort (NIDS)	NFA-based	Non-Disclosed	Altera Stratix II (EP2S180)	512 Gb/s
Yang et al. (2016) [49]	Bro, Snort and L7-filter (NIDS)	DFA-based	HDL	Xilinx Virtex-7	29.59 Gb/s
Or et al. (2016) [30]	ClamAV (Malware Detection)	NFA-based	HDL	Xilinx Virtex-6	144 MB/s
Gogte et al. (2016) [13]	Snort (NIDS) and RegExLib (General Purpose)	DFA-based	HDL	Altera Arria V	400 MB/s
István et al. (2016) [17]	Non-Disclosed	NFA-based	HLS (OpenCL)	Altera Stratix V 5SGXEA	25.6 GB/s
Jin et al. (2018) [20]	Snort and Bro (NIDS)	NFA-based	HDL (Verilog)	Non-Disclosed	Non-Disclosed
Yang et al. (2018) [48]	Bro, Snort and L7-filter (NIDS)	DFA-based	Non-Disclosed	Xilinx Virtex-7	140 Gb/s
Comodi et al. (2018) [11]	Non-Disclosed	Not Automata-based	HDL	Xilinx Virtex-7	66.56 Gb/s
Ceška et al. (2019) [8]	Snort (NIDS)	NFA-based	HDL (VHDL)	Xilinx Virtex UltraScale+ VU9P	Beyond 100 Gb/s
Jiang et al. (2021) [19]	Non-Disclosed	Not Automata-based	HDL	Cyclone IV GX	3.1 Gb/s
Sert and Bazlamacci (2021) [38]	Snort (NIDS)	NFA-based	HDL	Non-Disclosed	51.5 Gb/s
Parravicini et al. (2021) [32]	PROSITE and Brill	NFA-based	HDL (SystemVerilog)	Xilinx Ultra96v2	Non-Disclosed
Callanan et al. (2021) [6]	Snort (NIDS)	NFA-based	HLS (OpenCL)	Arria 10 PAC	17.9 Gb/s

Table 3.1: Summary and comparison of state-of-the-art RegEx acceleration approaches

PiDFA [49] *PiDFA* stands for parallel-input DFA and is an alternative to Deterministic Finite Automaton (DFA) stride doubling algorithms, employed to match several input characters simultaneously. This is done primarily due to a parallelizable method called DFA Transition Merging, which reads multiple input characters and creates temporary merging transition tables according to these characters, choosing the one to be applied in accordance with the current DFA state. A merging transition table represents the transition sequences the input characters could trigger across the DFA, according to its transition table.

HARE [13] *HARE*, or Hardware Accelerator for Regular Expressions, extends *HAWK*, a hardware accelerator for exact fixed-length string-matching, so that it can support some RegEx functionalities, such as Kleene operators $(+, *)$, alternation operators $(|, ?)$, and character classes $([\mathtt{a}-\mathtt{z}])$. It is capable of matching multiple characters per clock cycle through the use of various bit-split automata [42].

Ceška et al. (2019) [8] With deep packet inspection in mind, a multi-stage RegEx-matching architecture utilizing approximate NFAs is proposed. Instead of resorting to a NFA that exactly recognizes a language defined by a RegEx, multiple NFAs modeling approximations of that same language are used. The precision of the automata is low at the top-most stages and higher at the deeper stages, allowing for input that surely doesn't match a RegEx to be discarded at the earliest stages. In order to derive the approximated automata, states are labeled with their significance (the probability that they're reached when processing a packet), which is determined using training network traffic.

Sert and Bazlamacci (2021) [38] This approach achieves NFAs capable of two character transitions through a three-stepped process. Firstly, a RegEx is divided into its sub-expressions using the $($, $)$ and $|$ operators as delimiters; then each of these sub-expressions is further decomposed into pairs of characters; finally, each of these pairs is categorized as representing one of the six possible cases: $\mathtt{ab}$, $\mathtt{a}$, $\mathtt{a}^*\mathtt{b}$, $\mathtt{a}^*$, $\mathtt{ab}^*$, or $\mathtt{a}^*\mathtt{b}^*$. Each of these groups has its respective automaton translation, allowing for the complete NFA to be built by piecing the groups' automata together.

Callanan et al. (2021) [6] A High-Level Synthesis (HLS) design of a hardware accelerator for RegEx-matching developed with *Intel's OpenCL SDK*. It is based on multi-stride NFAs, processing up to 8 characters at a time, which are built through the algorithm presented in Section 3.4. These automata are modeled as an OpenCL implementation of a state machine, obtained through a source code template engine, which is then fed to the HLS tool in order to be synthesized into its hardware representation.

3.6 Analysis

It is clear that most of the work being done on the field of RegEx matching on FPGAs belongs within the context of Network Intrusion Detection System (NIDS), which goes in accordance with these system's needs for highly performant RegEx matching solutions, capable of high throughput packet analysis.

With advances in both FPGA devices and hardware accelerated RegEx matching in the last years, implementations went from processing capabilities in the megabit range to ones capable of processing hundreds of gigabits.

However, it is apparent that most research is also being done within the context of low-level hardware descriptions, with few systems being developed with HLS. This makes the development of RegEx matching engines through HLS a largely unexplored field, with little being known about its overall performance, advantages, and potential limitations.

The implementation of back-references on hardware solutions is also often neglected, with this operator being mostly left out of the set of operators supported by hardware implementations of RegEx matching systems, developed with HLS or otherwise.

Chapter 4

Proposed Approach

This chapter goes over the rulesets selected for analysis, the respective analysis measures, and the developed software artifacts, focusing on how these influence and integrate with one another.

4.1 Collected Rulesets

Rulesets belonging to different pattern-matching domains were collected, making up a total dataset of several thousands of Regular Expressions (RegExes), providing a broad range of different expressions to analyze.

Below follows a description of every RegEx pattern-matching ruleset collected:

- **ClamAV's** main database, containing multiple malware signatures, meant for file analysis in the search for potential malware.
- Both **Snort 2.9** and **Snort 3 Community** rules which contain patterns for known threats regarding incoming network packets, placing these rulesets in the domain of Network Intrusion Detection Systems (NIDSs).
- An openly available set of **Yara** rules [1], containing patterns used to detect malware, vulnerabilities, and other potential software exploits.
- The protein motifs utilized by **Protomata**, belonging to the **Prosite** [33] protein database, used to look for specific motifs in protein sequences.
- A set of both **Suricata 4.0** and **Suricata 5.0** (also a NIDS) rules, openly provided [2] by the Proofpoint cybersecurity company.

Some of the extracted rulesets don't provide a Perl Compatible Regular Expressions (PCRE) version of their contained patterns and consequently can't directly be compiled into a list of PCREs, such as the case of the Yara, ClamAV, and Protomata rulesets. For these cases, the

[1] https://github.com/Yara-Rules/rules
[2] https://rules.emergingthreats.net/

software tools and scripts provided by the AutomataZoo benchmark suite [44] were employed, allowing the translation of each ruleset's corresponding rule format into a PCRE format.

4.2 Ruleset Features

In order to perform an analysis of the different RegEx rulesets, different characterization features have to be extracted. These characteristics then go on to provide a summary of the current state of the utilization of RegExes across various application domains and guide the developed features of the presented pattern-matcher.

As the majority of the published hardware-accelerated RegEx matchers don't provide support for either back-references or bounded quantifiers, a greater emphasis was placed on extracting characteristics regarding precisely these two PCRE features, so that they could support the development of different options when it comes to implementing them.

In terms of general ruleset characteristics, the following measurements are considered:

- The amount of times that a PCRE operator appears in the ruleset;
- The number of different expressions in which a PCRE operator appears;
- The amount of times that a PCRE flag appears in the ruleset;
- The maximum input lengths supported by a ruleset's various expressions;

Knowing the prevalence of various PCRE features provides meaningful insight when determining whether a specific feature should be supported or not. With this information, it is also possible to ascertain how critical the lack of support for back-references and bounded quantifiers is for the real-world usage of the aforementioned majority of accelerators that don't support them.

Even though the maximum input length capable of being matched by an expression isn't directly related to how demanding said expression's pattern is to match, there generally is a positive relation between this metric and the number of states belonging to the expression's corresponding automaton representation. Generally, the bigger an expression is, the larger its automaton counterpart gets, making it more cumbersome on the Field-Programmable Gate Array (FPGA) resources and making its matching slower.

The length of capture groups is a fundamental statistic in the matter of defining how large the FPGA resources allocated as memory should be. Concerning capture groups and, by extension, back-references, various statistics with respect to their length are considered:

- The lengths of a ruleset's capture groups;
- The lengths of a ruleset's capture groups that end up being referenced by a back-reference;
- The lengths of referenced capture groups that appear in the ruleset and have a known fixed size;

The reason for analyzing referenced capture groups separately is that capture groups are often used solely due to their grouping capabilities, whereas when only pattern matching is being performed without any post-processing, as is the case, non-capture groups (toggled by inserting ? : at the start of the group) should be used instead.

A second specification is done to isolate referenced capture groups whose size is constant and can be determined by analyzing the expression. This specification is performed to check for further unnecessary usage of capture groups and back-references because if the sub-expression being referenced is constant, it can just be repeated literally in the expression. For example, in $(abc)^{+}d\backslash 1$, the need for capturing the string literal "abc" leads to the needless allocation of memory on the FPGA when the expression could just be rewritten as $(?:abc)^{+}dabc$.

Lastly, information about the various numbers of repetitions present in each ruleset's bounded quantifiers is collected.

4.3 Software Artifacts

Two major software artifacts were developed: a RegEx analyzer, capable of extracting statistics from multiple sets of RegEx rules, and a hardware-accelerated RegEx matcher synthesized through High-Level Synthesis (HLS).

Both artifacts use the same method to perform the parsing of the RegExes, which is a PCRE grammar used in conjunction with the ANTLR parser generator tool [3] in order to create a PCRE parser. Listing 4.1 shows an example of a grammar capable of processing basic arithmetic expressions.

```
grammar arithmetic;

root : equation* EOF ;
equation : expression comparison expression ;
expression
    : expression POW expression
    | expression (TIMES | DIV) expression
    | expression (PLUS | MINUS) expression
    | LPAREN expression RPAREN
    | atom
    ;
atom : number | variable ;
number : DOUBLE ;
variable : VARIABLE ;
comparison : EQUAL | GREATER | LESSER ;

VARIABLE : [a-zA-Z]+ [0-9]* ;
DOUBLE : [0-9]+ ('.' [0-9]+)? ;
LPAREN : '(' ;
```

[3] ANTLR Parser Generator https://www.antlr.org/

```
20 RPAREN : ')' ;
21 PLUS : '+' ;
22 MINUS : '-' ;
23 TIMES : '*' ;
24 DIV : '/' ;
25 GREATER : '>' ;
26 LESSER : '<' ;
27 EQUAL : '=' ;
28 POW : '^' ;
```

Listing 4.1: ANTLR Grammar for simple arithmetic

4.3.1 RegEx Analyzer

Extracting information from a RegEx is performed through a two-step process. The first step is parsing the expression, resulting in a tree structure called a syntax tree. This tree represents a decomposition of the RegEx into its different components, with every node representing a different RegEx syntactic unit, like a character literal or a quantifier, for example. The second step corresponds to making a traversal along every tree node and extracting all of the information deemed meaningful along the way.

Completing the simpler arithmetic example shown earlier, Figure 4.1 shows the generated syntax tree for the equation $x = 3 \times (7 - 2)$, where it is apparent that every piece of syntactic and semantic information belonging to the original arithmetic expression is retained and easily extractable.

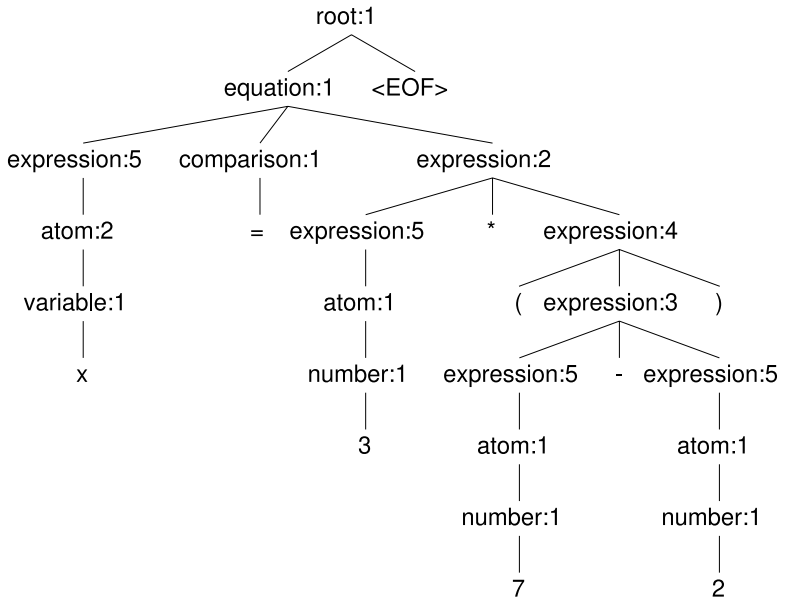


Figure 4.1: Syntax tree for $x = 3 \times (7 - 2)$

The process of extracting metrics from a RegEx is expanded to operate instead on the multiple RegExes that make up the chosen rulesets, with metrics being aggregated together as each expression is processed. For each extracted metric, a single file in Comma-Separated Values (CSV) format is generated, where the results for that metric are contained and sorted by each ruleset.

To summarize, an architectural diagram of this software artifact is presented in Figure 4.2.

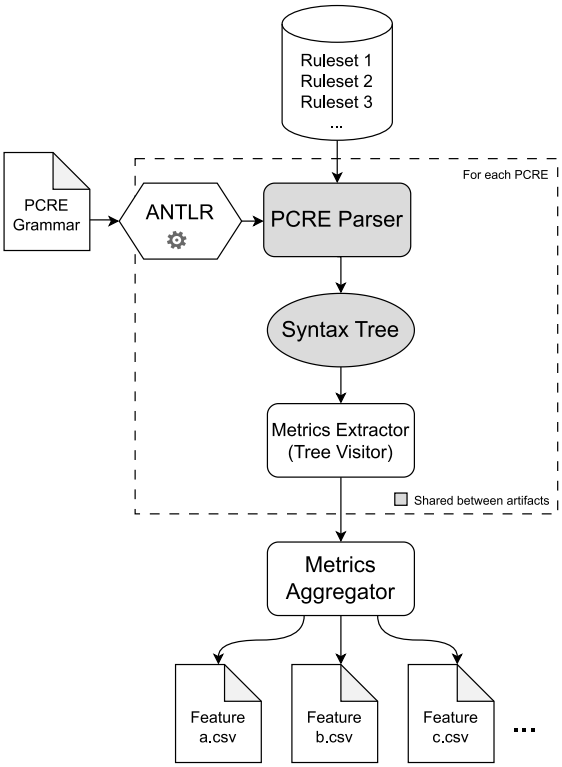


Figure 4.2: Architectural Diagram for the RegEx Analyzer

4.3.2 RegEx Matcher

In order to deploy the hardware-accelerated RegEx matcher on a FPGA, there needs to be a way to convert a set of RegExes into an equivalent high-level code description that can be synthesized by a HLS tool is necessary.

Two fundamental parts go into implementing such a mechanism: the generation of the automata that correspond to the different RegExes belonging to the rulesets, followed by the translation of these automata into a code description that can be synthesized into its hardware counterpart through HLS.

The process of generating the expression's automata equivalent is further split in two. Firstly, each expression is parsed and then converted into an Epsilon-Non-deterministic Finite Automaton (NFA) by traversing the syntax tree generated by the parsing process.

An Epsilon-NFA is a type of NFA that contains transitions known as epsilon transitions, which are transitions belonging to a Finite Automaton (FA) that can be activated without consuming any

input from the input string. This type of intermediate NFA is used in order to ease the process of converting PCREs into their FA equivalent, as going straight from PCREs to NFAs would prove to be more challenging.

The second step of converting PCREs into their corresponding NFAs is performing the removal of the epsilon transitions utilized earlier, converting the previously obtained Epsilon-NFA into an equivalent standard NFA.

As an example of both types of NFAs, Figure 4.3 shows an Epsilon-NFA and its equivalent NFA, both capable of accepting input composed by repetitions of the string "ab", analogous to the RegEx $(ab)^+$.

(a) Epsilon-NFA (b) NFA

Figure 4.3: Equivalent NFAs representing $(ab)^+$

After a PCRE is converted into its NFA counterpart, an automaton is implemented as an equivalent C++ code description, akin to a state-machine, which can be synthesized by Xilinx' Vitis HLS tool into a hardware description that can then run on a FPGA.

The process of translating an automaton into a code description is performed through a source code templating engine. The engine makes use of a written template containing the overall structure of a generic code description with various placeholders, which are then substituted to create a specialized description for each automaton.

Finally, after achieving a valid source code description that models the parallel execution of all of the PCREs' automata, that description is passed on to Vitis HLS, which is tasked with the translation of the matcher's software description into a hardware one, targeting a specific FPGA. The hardware description can then be run on a FPGA, taking character strings as input, outputting a binary 'yes' or 'no' output for each one of them, corresponding to whether the input string matches a RegEx contained in the matcher's description or not.

Figure 4.4 represents an architectural diagram for the proposed RegEx matcher, outlining the entire process necessary in order to deploy a hardware-accelerated RegEx matcher on a FPGA.

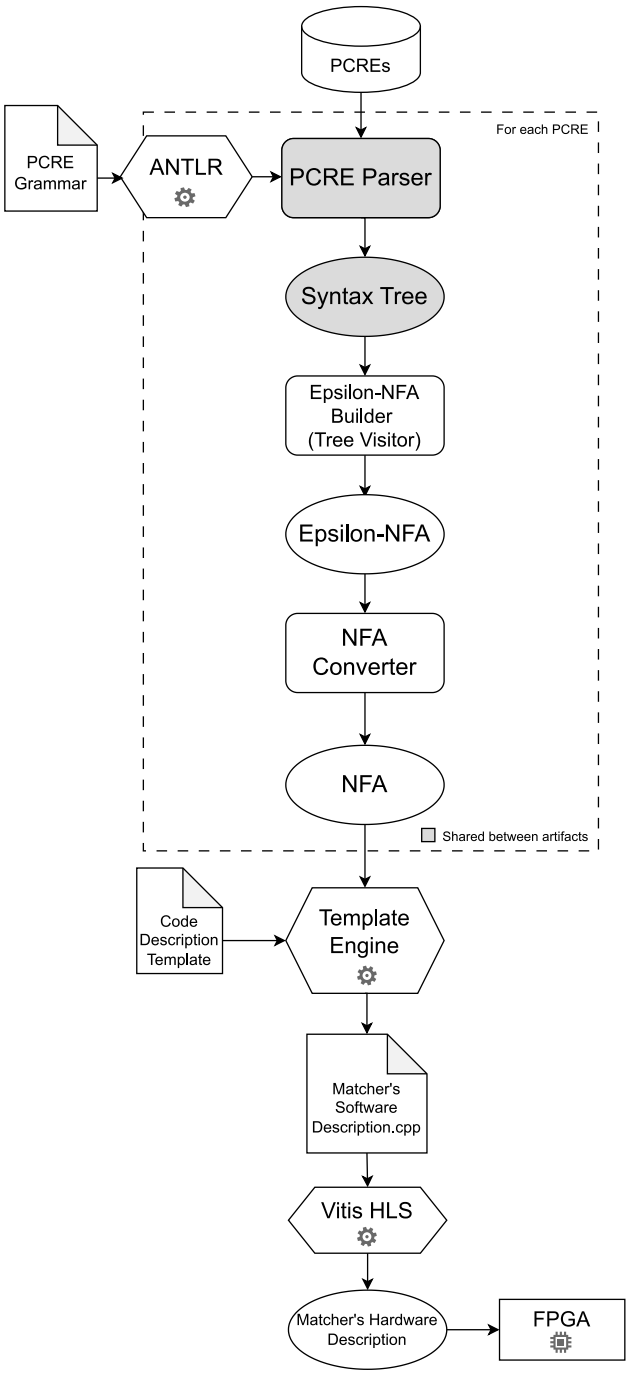


Figure 4.4: Architectural Diagram for the RegEx Matcher

Chapter 5

Regular Expression (RegEx) Matcher Implementation

This chapter describes the implementation steps required to obtain a software description of a RegEx matcher that is convertible into its corresponding hardware description through High-Level Synthesis (HLS). The result of this process is a RegEx matcher implementation ready for being deployed on a Field-Programmable Gate Array (FPGA).

5.1 Epsilon-NFA Generation

In this section, it is explained how an extended Epsilon-Non-deterministic Finite Automaton (NFA) conversion of a RegEx is obtained, with this representing an initial automaton translation that is later expanded upon. It is outlined which Perl Compatible Regular Expressions (PCRE) operators and flags are supported, along with how they are represented within the automata.

5.1.1 Supported RegEx Operators and Flags

The proposed accelerator supports all of the PCRE operators that don't require either positive or negative lookahead, as supporting these operators would prove to be a significant undertaking compared to the relatively small number of expressions that actually employ them. Regarding the various expressions' flags, only standard PCRE flags are supported, and not any ruleset-specific ones.

Tables 5.1 and 5.2 respectively provide a comprehensive view of various RegEx operators and flags and their corresponding status regarding whether the proposed solution supports them or not.

5.1.2 Epsilon-NFA Creation

This section covers the first phase of converting a RegEx into its corresponding NFA representation. This step is responsible for the parsing of the pattern and for generating its analogous Epsilon-NFA representation.

Name	Supported
Alternation	Yes
Concatenation	Yes
Character Classes	Yes
Regular Quantifiers	Yes
Greedy Quantifier Type (Default Type)	Yes
Lazy Quantifier Type	No
Possessive Quantifier Type	No
Bounded Quantifiers	Yes
Start/End Anchors	Yes
Start/End Anchors	Yes
Capture/Non-Capture Groups	Yes
Back-references	Yes
Lookarounds	No

Table 5.1: Supported Operators

Name	Supported
i	Yes
m	Yes
s	Yes
x	Yes
A	Yes
Ruleset Specific (R, U, I, P, ...)	No

Table 5.2: Supported Flags

RegEx Parser

When it comes to building the automata representation of a RegEx pattern, the first step is performing the parsing of the expression. To achieve this, an openly available PCRE grammar [1] for the ANTLR parser generator tool was utilized. With this tool, it is possible to generate a syntax tree, which is a tree-like decomposition of the various units that make up an expression. As an example, such a tree can be seen in Figure 5.1.

After the syntax tree is built, a process akin to the McNaughton–Yamada–Thompson algorithm [43, 27] can be applied to build the Epsilon-NFA, where the various unitary constituents that make up an expression can then merged back together according to certain rules depending on the operators acting on them.

[1] Bart Kiers' PCRE grammar https://github.com/antlr/grammars-v4/blob/master/pcre/PCRE.g4

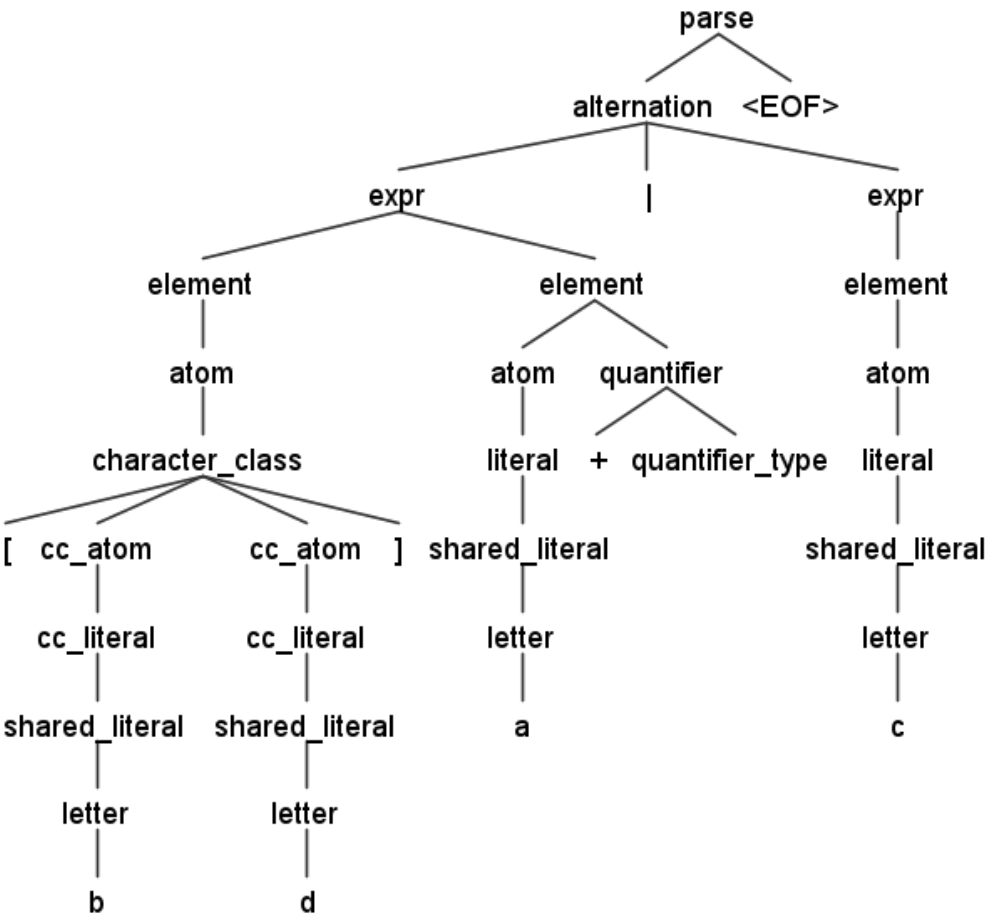


Figure 5.1: Syntax Tree for $[bc]a^{+}|c$

The Epsilon-NFA is constructed by performing a post-order traversal of every tree node and incrementally adding each sub-automata, representing a distinct tree branch, to a stack. Upon exiting a fork in the tree, these sub-automata can be merged through the application of a merging rule corresponding to the operator that generated the divergence.

Terminal Nodes

In this context, terminal nodes are nodes that can directly be translated into an automaton with a single transition, being the most basic constituent unit of a RegEx. Terminal nodes can be either character literals, wildcards, character classes, or back-references.

Character Literals Represented by a single-transition automaton with a character transition that matches a single specific character.

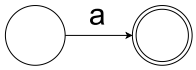


Figure 5.2: Automaton for a

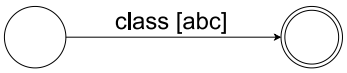
Wildcards Represented by a single-transition automaton with a wildcard transition that matches any single character, excluding the new-line character unless the *s* flag is active.

Figure 5.3: Automaton for .

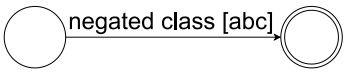
Character Classes Character classes allow a single character to be matched from a set of various characters. This set can be represented by declaring each character explicitly or by using character ranges. As an example, both $[abc]$ and $[a-c]$ would match either the character a, b, or c.

Classes can be negated by inserting a '^' after the class's opening bracket. This has the behavior of matching any character that isn't a member of the defined set of characters.

This feature is represented by a single-transition automaton with a class transition that matches any single character contained in its character set. In the case of a negated character class, the transition matches any single character that isn't included in the given set of characters.



(a) Automaton for [abc]



(b) Automaton for [^abc]

Figure 5.4: Character Class Automata

Back-references Back-references are a feature that allows previously matched input to be repeated later on in the expression. The input to be repeated has to be stored through the use of a capture group, which can then be referenced, either by name (if the group being referenced was named) or by its order of appearance in the expression (with the first capture group having an index of 1). For example, in the expression $(a|b)\backslash 1$, the capture group $(a|b)$ stores whether the first matched character was an 'a' or a 'b' and then, depending on the stored character, the back-reference $\backslash 1$ would look to match that input again.

This feature is represented by a single-transition automaton with a back-reference transition that matches any previous input that was stored by the capture group it references.

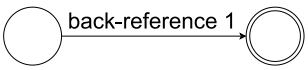


Figure 5.5: Automaton for $\backslash 1$

Anchors Anchors serve to enforce the location of where a match occurs in relation to its location within the input string.

A start anchor is expressed by the symbol '^'. It either matches the start of the input string or the start of a new line in the case that the *m* flag (known as multi-line flag) is active.

The symbol '$' represents an end anchor, and these differ from a start anchor by matching the end of the input string or line instead of the start.

Both anchors are represented by a single-transition automaton with a start/end anchor transition that matches its respective start or end location.

(a) Automaton for ^ (b) Automaton for $

Figure 5.6: Anchor Automata

Concatenation and Alternation

The concatenation operator is a linking operator used to chain smaller RegEx units together, allowing for more complex patterns to be built. It is an implicit operator that is present whenever two RegEx units are written consecutively. In the expression $r_1 r_2$, the implicit concatenation operator present between RegExes r_1 and r_2 gives the expression the meaning of 'match r_1 *then* match r_2'.

The construction rule for the concatenation operator is shown in Figure 5.7.

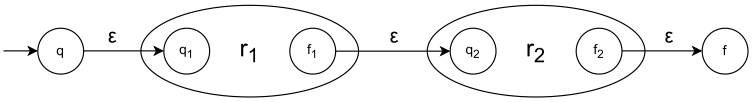


Figure 5.7: Concatenation Construction Rule for $r_1 r_2$

Like concatenation, the alternation operator also allows for RegExes to be linked with each other. This operator is expressed through the '|' symbol so that in the expression $r_1|r_2$, the alternation operator present between RegExes r_1 and r_2 gives the expression the meaning of 'match r_1 *or* match r_2'.

The construction rule for the alternation operator is shown in Figure 5.8.

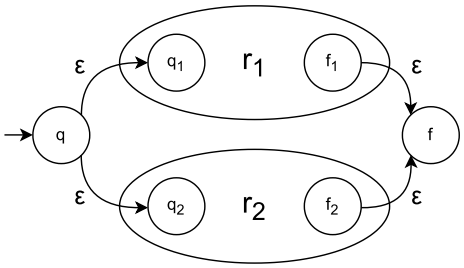


Figure 5.8: Alternation Construction Rule for $r_1|r_2$

Regular Quantifiers

Quantifiers allow the repetition of RegEx units. Regular quantifiers, as opposed to bounded quantifiers, which will be discussed later, can easily be implemented with simple Finite Automata (FAs), requiring no extensions or more complex building behaviors.

'Zero or more' Quantifier The 'zero or more' quantifier is expressed with the '$*$' symbol, also known as the Kleene star. In the expression r_1*, the quantifier gives the expression the meaning of 'match r_1 *zero or more times*'

The construction rule for this quantifier is shown in Figure 5.9.

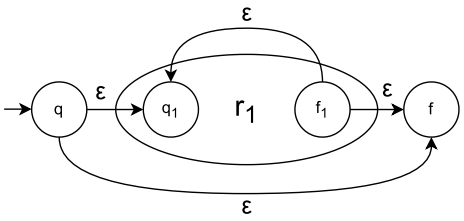


Figure 5.9: Quantifier Construction Rule for r_1*

'One or more' Quantifier The 'one or more' quantifier is expressed with the '$+$' symbol. In the expression r_1+, the quantifier gives the expression the meaning of 'match r_1 *one or more times*'

The construction rule for this quantifier is shown in Figure 5.10.

'Zero or one' Quantifier The 'zero or one' quantifier is expressed with the '?' symbol. In the expression $r_1?$, the quantifier gives the expression the meaning of 'match r_1 *zero or one times*'

The construction rule for this quantifier is shown in Figure 5.11.

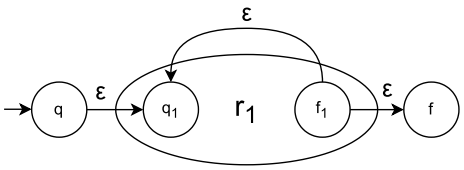


Figure 5.10: Quantifier Construction Rule for r_1+

Bounded Quantifiers

Bounded quantifiers, like regular quantifiers, allow for RegEx elements to be repeated. These provide more expressive power than their regular counterparts due to their capability of having different numbers of repetitions expressed as positive integers.

A bounded quantifier's number of repetitions can be defined as an exact number or as an interval of allowed repetitions. While an interval's lower bound must be defined explicitly, its upper limit can be omitted, taking on the implicit endpoint of $+\infty$. This means that bounded quantifiers can be directly used to express three different repetition meanings: match *exactly x* times, match *at least x* times, and match *between x* and *y* times.

Regarding the translation of this PCRE feature to FAs, there are two possible approaches: a naive approach, where the pattern meant to be repeated is explicitly repeated an appropriate number of times in the FA, together with regular quantifiers when optional or unknown repetitions need to be fulfilled; and an approach where regular FAs are expanded to possibly include counter comparisons and increments in their transition functions, allowing a transition to occur a determined amount of times.

Explicit Repetition By explicitly repeating the sub-FA representing the RegEx element being referenced by the bounded quantifier, an automaton's number of states and respective transitions grow exponentially big. This makes this solution, however simple, very inefficient for quantifiers with a large number of repetitions.

Below are the different types of bounded quantifiers implemented by employing this method.

- *'Match exactly x times'* quantifier: In order to match a RegEx element a determined amount of times, its FA equivalent is concatenated together the required amount of times, leading to transformations such as from $a\{4\}$ into *aaaa*.

- *'Match at least x times'* quantifier: This situation can be interpreted as matching a RegEx an exact number of times, with this number corresponding to the quantifier's lower bound, and then matching it again zero or more times. This results in expressions such as $a\{4,\}$ being translated into $aaaaa*$

- *'Match between x and y times'* quantifier: For cases where the number of repetitions is expressed as an interval, repetition numbers up to and including the interval's lower limit can

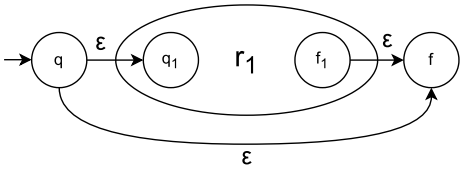


Figure 5.11: Quantifier Construction Rule for r_1?

again be translated as replications of the pattern concatenated together. Each individual repetition constrained between the interval's lower and upper limits can be interpreted as being optional. For example, the pattern $a\{4,6\}$ can be interpreted as 'match a exactly 4 times, then match it either 0,1 or 2 more times'. To achieve this optional behavior, the ? (zero or one) quantifier is used so that RegExes like $a\{4,6\}$ ultimately become $aaaaa?a?$, resulting in the automaton shown in Figure 5.12 (which suppresses epsilon transitions resulting from the various concatenations for improved clarity).

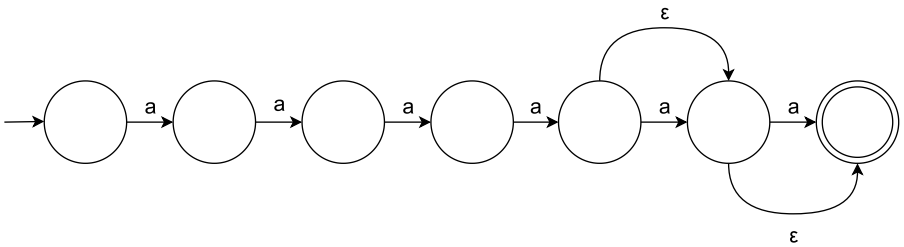


Figure 5.12: Automaton for $a\{4,6\}$ generated through expression repetition

Extended FA with Counters An alternative to creating copies of the sub-FA on which a bounded quantifier actuates is extending an automaton's transitions to set, check, and increment counter variables. This represents a trade-off between automata with a lower amount of states and transitions at the cost of each transition involving counters becoming more complex.

For this approach, a new set of epsilon transitions is introduced, each linked to a counter variable tasked with keeping count of the number of times that an expression referenced by a bounded quantifier has been repeated. These new transitions are the following:

- A *'Set'* transition, responsible for initializing the counter variable attached to it with a default starting value of 1;
- A *'Lesser'* comparison transition, activated if the value of the counter variable attached to it is lesser than the transition's own comparison value. An activation of this transition increments the value of its attached counter by one.

  This transition is used to represent the repetitions that don't yet satisfy the number of repetitions required by a given bounded quantifier;

- *'Equal'*, *'At Least'* and a *'Between'* comparison transitions. These are similar to the *'Lesser'* comparison transition in the sense that they also perform a comparison between a counter's value and their own comparison values while incrementing the counter. Each represents the different acceptance comparisons corresponding to the three types of bounded quantifiers mentioned earlier.

  The *'Between'* transition has two attached comparison values: one for the lower limit of the repetition range and one for the upper limit of the repetition range.

Independently of the type of bounded quantifier being implemented, a copy of the sub-automaton on which the quantifier actuates is created and placed before the original sub-automaton, with a *'Set'* transition linking them together. This is performed in order to capture when a first match of the pattern referenced by the quantifier has occurred, initializing the quantifier's counter with its starting integer value of 1, meaning that it's ready to count further repetitions.

The next step adds the comparison transitions corresponding to the type of comparison dictated by the bounded quantifier type that is being implemented. This results in an acceptance comparison transition outgoing to an acceptance state being added, and, for quantifiers that allow multiple numbers of pattern repetitions instead of an exact number of repetitions, an acceptance transition that loops to the beginning of the quantifier pattern's original automaton.

To illustrate the automata generated by this approach, Figure 5.13 demonstrates the automaton for the RegEx $a\{4,6\}$ generated through the application of this method. It is noted that in this type of bounded quantifier, where a counter's value is compared against an interval, the corresponding looping transition could just check if the counter's current value is lesser than the interval's upper limit (6 in the example).

The use of the *'Set'* transitions to reset a counter's value means that automata generated from expressions with bounded quantifiers are as easy to encapsulate within loops as automata generated from other types of expressions. This, in addition to the creation of independent counter variables and transitions assigned to each bounded quantifier, results in functionally correct automata capable of supporting RegExes with nested bounded quantifiers.

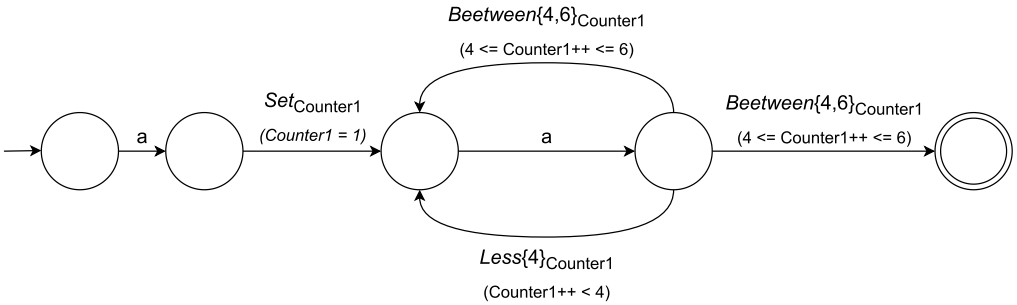


Figure 5.13: Automaton for $a\{4,6\}$ generated through the extended transition set

Capture Groups

RegEx capture groups, like their non-capturing counterpart, serve to group sub-expressions within an expression so that different RegEx operators can then be applied to the group as a whole, akin to how parenthesis are used in mathematical expressions.

Unlike non-capture groups, capture groups also have the dedicated purpose of storing which input matches the patterns that they represent. This feature works in tandem with back-references, which rely on referencing previously matched input so that an exact copy of that same input can be matched again.

In order to translate this PCRE feature into its automata equivalent, there are two requirements that need to be fulfilled. The first one is indicating both where a capture begins and where it ends, and the second one is identifying the capture itself.

By default, capture groups are identified by an integer representing the group's order of appearance in the expression (starting at index 1). However, capture groups can optionally be named, and then that name can be used to reference them. Listing 5.1 displays the ANTLR rule utilized to define all the possible ways of expressing capture groups, as supported by the PCRE specification.

```
1 capture
2  : '(' '?' '<' name '>' alternation ')' //named
3  | '(' '?''\'' name '\'' alternation ')' //named
4  | '(' '?' 'P' '<' name '>' alternation ')' //named
5  | '(' alternation ')' //unnamed
6  ;
```

Listing 5.1: ANTLR rule defining capture groups

In the proposed implementation, every capture group is internally identified by a positive integer corresponding to its appearance order in the overall expression, whether the group was named or not. To obtain this representation, a mapping between every named capture group and its order of appearance is kept.

To accomplish both of the two aforementioned requirements, two new types of epsilon transitions, each possessing a capture group integer identifier, were added to the set of transitions supported by the epsilon-NFA:

- A '*Capture Start*' transition, denoting where input regarding its attached capture group identifier should start being captured;
- A '*Capture End*' transition, signifying that the matched input no longer should be captured when concerning the transition's attached capture group identifier.

One of each of these new transition types is used to delimit a capture group's boundaries in the generated automata, with a '*Capture Start*' placed before the capture group's first automaton state and a '*Capture End*' appended to the end of the capture group's automaton.

As a result of this solution, Figure 5.14 shows the generated automaton for the RegEx $(a|b)$.

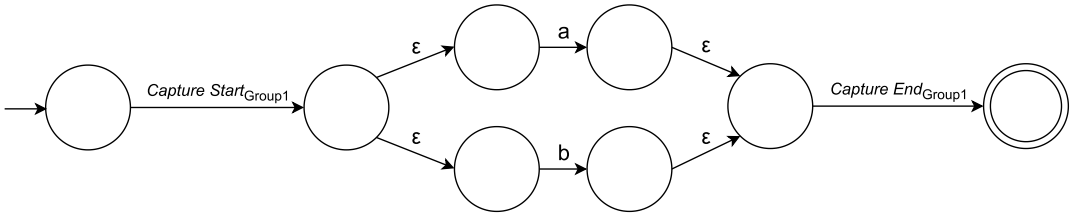


Figure 5.14: Generated automaton for $(a|b)$

Generation Flags

In order to assess how different implementation choices and optimizations regarding the translation of PCREs into automata end up altering the final solution's performance, a set of automata generation flags is made available.

Figure 5.15 depicts the user interface of the software developed to translate PCREs into their corresponding HLS-compliant software descriptions, where the available automata generation flags can be seen.

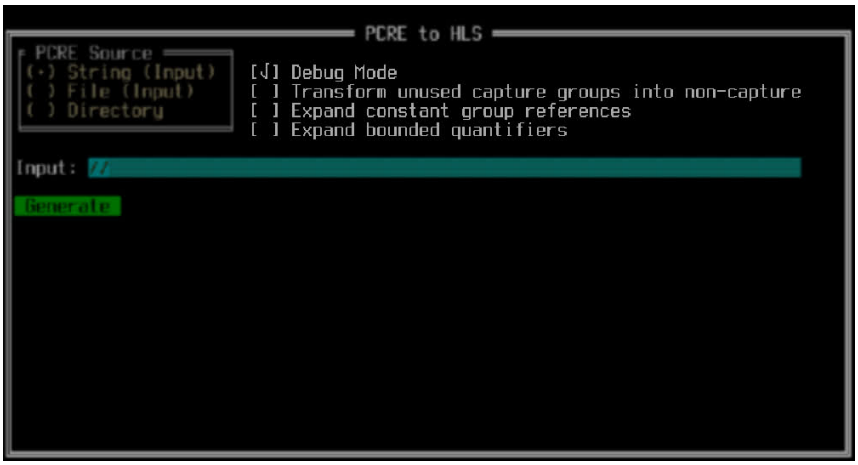


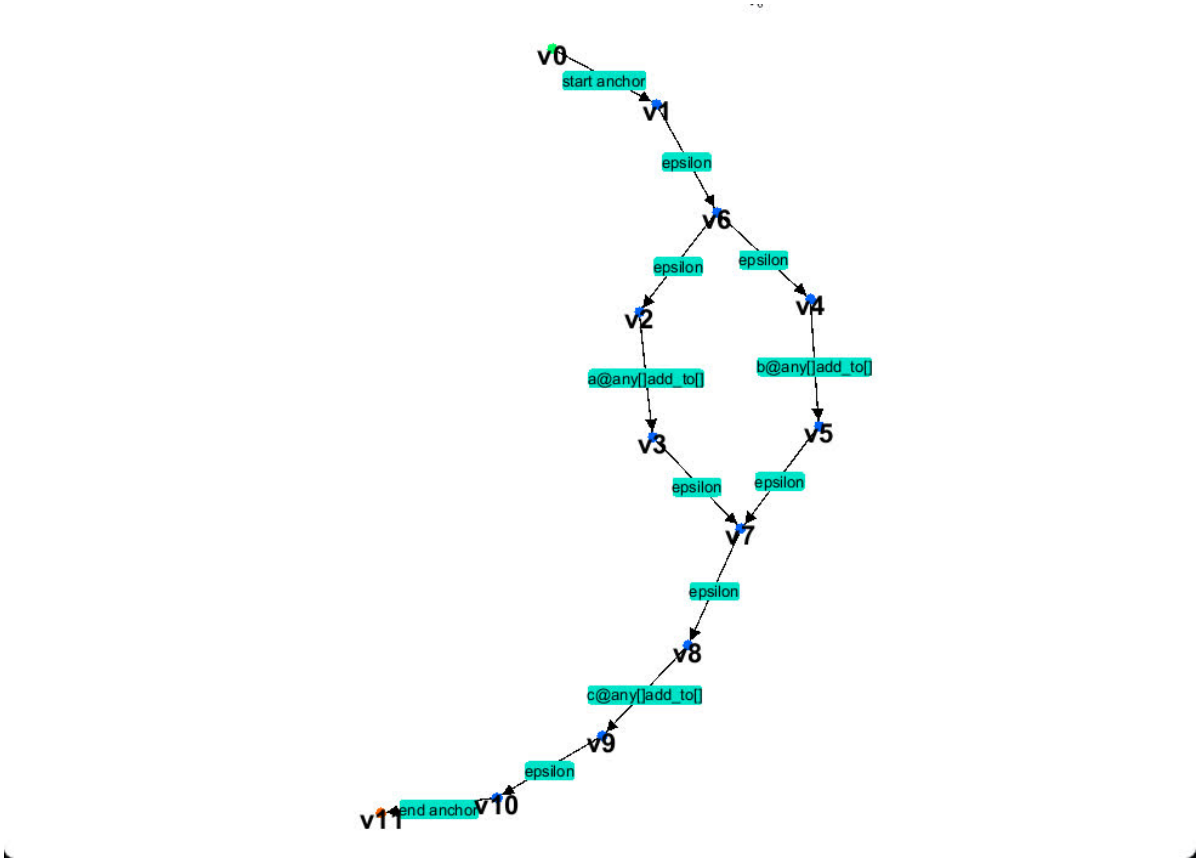
Figure 5.15: User interface showcasing the available automata generation flags

'Debug Mode' This flag is unique in the sense that it doesn't influence either the automata being generated or their respective software description. Its purpose is to allow the visualization of the automata being generated, both in their intermediate epsilon-NFA form and in their final NFA form, and to display the syntax trees extracted from each expression that is parsed.

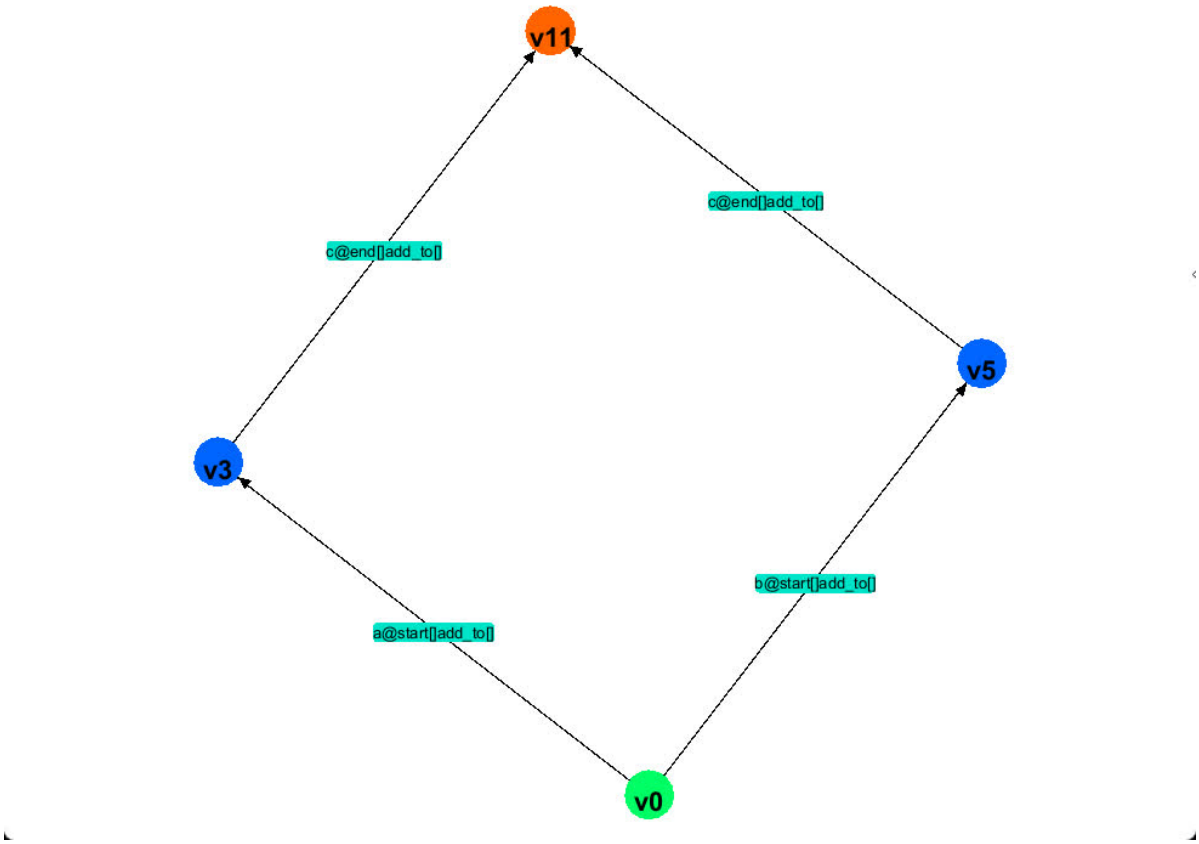
The syntax tree demonstrated earlier in Figure 5.1 was generated through the use of this flag, achieved through the visualization features made available by the same ANTLR library that was used to perform the parsing of the PCREs.

Figure 5.16 shows an example of the automata visualization enabled through the use of this flag for the RegEx $\hat{}(?:a|b)c\$$, obtained through the graph visualization capabilities offered by the GraphStream[2] graph library.

[2]GraphStream library https://graphstream-project.org/



(a) Epsilon-NFA for ^(? : $a|b$)c\$



(b) NFA for ^(? : $a|b$)c\$

Figure 5.16: Automata visualization enabled through the debug flag

'Transform unused capture groups into non-capture' In order to store the input matched by capture groups so that it can be referenced later, buffers have to be allocated. This means that if a capture group is used with no intention of being referenced and solely due to their grouping ability, buffers are being allocated unnecessarily, leading to a waste of resources.

To prevent this unnecessary allocation of storage resources, a flag was created, which, when enabled, looks to convert groups that are declared as being capture groups but end up not being referenced by any back-reference into non-capture groups.

An example of the effects of toggling this flag is the conversion from the automaton generated by the RegEx $(a|b)c$ into the automaton equivalent of $(?:a|b)c$, where the capture group $(a|b)$ is transformed into the non-capture group $(?:a|b)$, resulting in no unnecessary storage being created to cache whether the group matched with an 'a' or a 'b'.

'Expand constant group references' Similar to the previous flag, this flag looks to prevent unnecessary allocation of storage resources. In the cases where a back-reference refers to a capture group that has a constant string as its matching input, like in the capture group present in $(ab)c\backslash 1$, the back-reference can be switched by that same constant string contained within the group, and the capturing group can be made non-capturing, resulting in the aforementioned expression generating the automaton equivalent of $(?:ab)cab$.

'Expand bounded quantifiers' When it comes to the implementation of bounded quantifiers, both approaches described in Section 5.1.2 are implemented, with the second one, where the generated NFAs are extended to support transitions with counters, being the one utilized by default. This flag switches to the first described implementation of bounded quantifiers, where the sub-automaton corresponding to an expression referenced by a bounded quantifier is explicitly repeated in the overall automaton. When this flag is toggled off, automata of the sort shown in Figure 5.13 are generated, and when it's toggled on, automata like the one shown in Figure 5.12 are generated instead.

5.2 Transformation into NFA

Multiple intermediate transitions are utilized in the first step of the NFA generation process. While these transitions ease the creation of the initial NFAs, no input is matched by triggering them, with the information they convey being transferable to transitions that do, making them removable.

The various intermediate transition removal steps are described within this section, delineating how the firstly generated Epsilon-NFAs can be transformed into their extended NFA equivalent.

5.2.1 Epsilon Transition Removal

During the initial phase of translation between a RegEx into an automaton, multiple simple epsilon transitions were introduced in order to implement the various regular operators that constitute the RegEx syntax, such as concatenation, alternation, and the different regular quantifiers.

Since epsilon-NFAs are no more expressive than simple NFAs, being that these transitions have no inherent expressive value, epsilon transitions can be removed while producing no alterations to the overall language being represented.

To remove epsilon transitions, one must be able to calculate an automaton state's epsilon closure, which represents the set of all states that can be reached from a given automaton state by exclusively following epsilon transitions. Listing 5.2 shows the recursive implementation utilized to extract a state's (referred to as '`vertex`') epsilon closure.

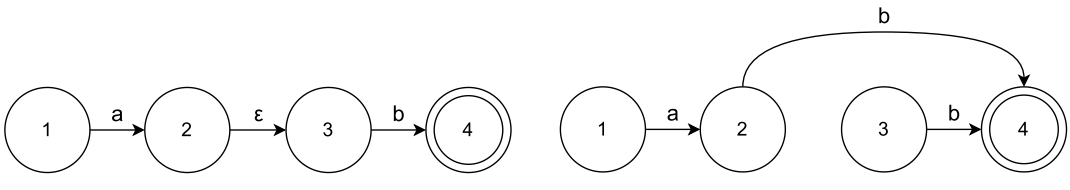
```
1 private static void getEpsilonClosure(Graph<String, DefaultEdge> graph, String
      vertex, Set<String> closure) {
2     closure.add(vertex);
3     for (DefaultEdge edge : graph.outgoingEdgesOf(vertex)) {
4         if (edge.getClass() == EpsilonEdge.class) {
5             String out_vertex = graph.getEdgeTarget(edge);
6             getEpsilonClosure(graph, out_vertex, closure);
7         }
8     }
9 }
```

Listing 5.2: Recursive method of extracting a state's epsilon closure

After being able to extract a state's epsilon closure, performing the removal of epsilon transitions becomes a matter of, for every state, obtaining its epsilon closure and then transferring back every non-epsilon transition outgoing from the states belonging to said closure to the original closure originating state. If a state is a member of the epsilon closure of an acceptance state, it is converted into an acceptance state.

Algorithm 2 depicts the epsilon transition removal process described, employed during the transition from epsilon-NFAs into NFAs.

As is the case for state number 3 in Figure 5.17, this approach results in states that previously only possessed incoming epsilon transitions being left in the resulting automaton as unreachable states. A further step is then necessary to perform the removal of the resulting unreachable states caused due to the earlier application of the epsilon removal step.



(a) Before epsilon removal (b) After epsilon removal

Figure 5.17: NFAs for *ab* before/after the application of Algorithm 2 (the unreachable state 3 will then be removed by the application of Algorithm 3)

Algorithm 2 Epsilon transition removal from NFA

Input: *NFA* with epsilon transitions
Output: *NFA* with no epsilon transitions

```
1:  for each state s ∈ NFA do
2:      closure ← NFA.getClosure(s)
3:      if closure ∩ NFA.getAcceptanceStates() ≠ ∅ then
4:          NFA.addAcceptanceState(s)
5:      end if
6:      for each state r ∈ closure do
7:          for each transition t ∈ NFA.outgoingTransitionsFrom(r) do
8:              if t is epsilon transition then
9:                  continue
10:             end if
11:             source ← s
12:             target ← NFA.getTarget(t)
13:             NFA.addTransition(source, target, t)
14:         end for
15:     end for
16: end for

17: for each transition e ∈ NFA do
18:     if e is epsilon transition then
19:         NFA.removeTransition(e)
20:     end if
21: end for
```

Unreachable states are identified by either not having any incoming transitions or by exclusively possessing self-transitions, coming from and to themselves. Algorithm 3 applies this notion to identify and remove an automaton's unreachable states.

Algorithm 3 Unreachable states removal from FA

Input: *FA* with unreachable states
Output: *FA* with no unreachable states

```
1:  unreachable ← {}
2:  for each state s ∈ FA do
3:      if s = FA.getStartState() then
4:          continue
5:      end if
6:      incoming ← FA.incomingTransitionsTo(s)
7:      if incoming = ∅ then
8:          unreachable.add(s)
9:      else
10:         only_self_edges ← True
11:         for each transition t ∈ incoming do
12:             if FA.getSource(t) ≠ FA.getTarget(t) then
13:                 only_self_edges ← False
14:                 break
15:             end if
16:         end for
17:         if only_self_edges = True then
18:             unreachable.add(s)
19:         end if
20:     end if
21: end for

22: for each state u ∈ unreachable do
23:     FA.removeState(u)
24: end for
```

5.2.2 Capture Transition Removal

The meaning implied by the *'Capture Start'* and *'Capture End'* transitions, used to convey capture groups within the generated NFAs, is that every input symbol that is matched between a pair of these two transitions must be stored. These pairs of transitions can be omitted, with their meaning being expressed directly by the transitions positioned between them.

In order to omit capture transitions, the remaining transitions are associated with capture information constituted by a capture group identifier and a boolean flag denoting if a transition represents the start of a given capture. The boolean flag is necessary to know when captured data should be reset in order to start a new capture. A transition can have various amounts of capture information depending on how many captures it is a member of. This addition of information to each transition represents a further extension to the type of NFAs being used.

To perform the removal of capture transitions, every pair of states associated with a capture group's start and end transitions is extracted from the NFA. Every possible path through the NFA starting and ending at a given capture pair's states is then calculated and traversed, with the information regarding its capture group identifiers set accordingly. The first transition taken along each path represents the start of a capture group, so its boolean 'start of capture' flag is set as being true. A capture transition is substituted by a standard epsilon transition which is later removed by the process described in Section 5.2.1.

Both steps taken to perform the removal of capture transitions are shown in Algorithm 4.

Algorithm 4 Capture transitions removal from NFA

Input: *NFA* with capture transitions
Output: *NFA* with no capture transitions

```
1:  Map<groupID, (start_state, end_state)> capture_data
2:  for each transition t ∈ NFA do
3:      if t is capture transition then
4:          if t is 'Capture Start' transition then
5:              capture_data[t.groupID].first ← NFA.getTarget(t)
6:          else
7:              capture_data[t.groupID].second ← NFA.getSource(t)
8:          end if
9:          NFA.addTransition(NFA.getSource(t), NFA.getTarget(t), newEpsilonTransition())
10:         NFA.removeTransition(t)
11:     end if
12: end for

13: for each (groupID, capture_pair) ∈ capture_data do
14:     paths ← NFA.getPathsBetween(capture_pair.first, capture_pair.second)
15:     for each path p ∈ paths do
16:         for each transition e ∈ p do
17:             is_start ← p.indexOf(e) = 0
18:             e.addCaptureInfo(groupID, is_start)
19:         end for
20:     end for
21: end for
```

5.2.3 Counter Transition Removal

Like the case of capture transitions, specialized counter transitions can be omitted by performing a further addition to the NFAs' transition set definition, with their meaning being conveyed instead by attaching additional information to a NFA's regular transitions.

The removal of counter transitions is done by traversing a NFA looking for these types of transitions. Whenever a state with an outgoing counter transition is found, every transition incoming to that said state is linked with the counter transition's information and then transferred to the counter transition's target state. This process is shown by Algorithm 5.

Algorithm 5 Counter transitions removal from NFA

Input: *NFA* with counter transitions
Output: *NFA* with no counter transitions

```
1:  DepthFirstIterator<State> (NFA.getStartState()) iterator
2:  while state s ← iterator.next() do
3:      for each transition t ∈ NFA.outgoingTransitionsFrom(s) do
4:          if t is not counter transition then
5:              continue
6:          end if
7:          for each transition e ∈ NFA.incomingTransitionsTo(s) do
8:              source ← NFA.getSource(e)
9:              target ← NFA.getTarget(t)
10:             transferred_transition ← e.copy()
11:             transferred_transition.addCounterInfoFrom(t)
12:             NFA.addTransition(source,target,transferred_transition)
13:         end for
14:         NFA.removeTransition(t)
15:     end for
16: end while
```

This transformation may lead to the creation of unreachable states in the NFA, meaning that it should be followed up by a pass of the unreachable state removal algorithm shown in Algorithm 3.

5.2.4 Anchor Transition Removal

The removal of anchor transitions is performed in a way similar to the removal of normal epsilon transitions, which was explained in 5.2.1.

For the start of line/string anchors, the process is entirely the same: for every state, its closure under start anchors is obtained, representing every state that is reachable by following only start anchor transitions, then each transition coming out of a state belonging to said closure is copied back to the closure's source state. The copied transitions are extended to possess information signalizing that they should only be taken if the input being processed is at the beginning of a line/string.

For the end of line/string anchors, this approach is modified to not transfer back transitions outgoing from a state's closure set but to instead transfer forward transitions incoming to a state to its closure set. The transferred transitions are extended to possess information signaling that they should only be taken if the input being processed is at the end of a line/string.

Note that the closure of each state is being utilized due to expressions with strings of consecutive anchors, such as ^^^*ab*$$$, being valid according to PCRE syntax.

5.2.5 Extended NFA Formal Definition

The multiple transition extensions added along the NFA generation process result in NFAs which can be formally described by the following 9-tuple$(\Sigma, Q, q_0, F, C, G, B, I, \delta)$:

- $\boldsymbol{\Sigma}$ represents the input alphabet, which is a finite set of input symbols over which the transitions are defined
- $\boldsymbol{Q}$ is the finite set of states that constitute the NFA
- $\boldsymbol{q_0} \in Q$ is the NFA's initial state
- $\boldsymbol{F} \subseteq Q$ is the finite set which represents the NFA's acceptance states
- $\boldsymbol{C}$ is the finite set of counter variables, each associated with a different bounded quantifier PCRE operator
- $\boldsymbol{G}$ is the finite set of input storage variables, each associated with a different RegEx capture group
- $\boldsymbol{B} \subseteq G$ is the finite set of input storage variables that are referenced by back-reference operators
- $\boldsymbol{I}$ is the finite set of positive integers representing the index of each input symbol within the input string
- $\boldsymbol{\delta} : Q \times \Sigma \times I \times C \times B \rightarrow Q_{i+1}$ is the NFA's transition function

Buffer variables contained in C are utilized solely for performing the storage of input declared by their corresponding capture groups. Buffer variables which are also contained within set B are the ones whose stored input is actually verified in order to match back-reference transitions, hence why only these are taken into account when defining the NFA's transition function.

Set I is necessary because, in order to follow the matching location restrictions imposed by PCRE anchors, not only an input symbol has to be taken into account but also its location within the overall input string.

5.3 Code Generation

In this section, the process of creating a C++ description for the RegEx matcher out of the previously generated NFAs is detailed. A template engine is used for this process, making the developed template the section's primary focus.

5.3.1 Template Engine

To convert the generated NFAs into their equivalent matching algorithms, the Apache FreeMarker template engine [3] Java library is used.

A template, written in the FreeMarker Template Language, allows code to be dynamically created in order to represent a software description of each generated automaton, through the usage of various template placeholder variables, conditional expressions, or loops.

The interaction between the generated NFAs and the template is done seamlessly since the data that is fed to the template comes in the form of custom JavaBeans whose attributes can be directly accessed via dot notation within the template. With this notion, a list of different JavaBean representations of the various NFAs is used to send data to the template.

Listing 5.3 depicts a source code generation implementation, using an example template, where the entire FreeMarker workflow is showcased, from how data is declared to how it is sent to the template and then accessed by it.

```
1  // Automaton.java
2  public class Automaton {
3      private String expression;
4      private Set<State> states;
5      ...
6  }

8  // State.java
9  public class State {
10     private int id;
11     ...
12 }

14 // CodeGenerator.java
15 Set<Automaton> automata = generateAutomata(expressions);
16 Map<String, Object> root = Map.of("automata", automata);
17 Template template = FTLconfig.getTemplate("template.ftlh");
18 template.process(root, new OutputStreamWriter(new FileOutputStream(cpp_file)));

20 // exampleTemplate.ftlh
21 int main() {
22     <#list automata as automaton>
23         std::cout << "The generated automaton for ${automaton.expression} has the
               following ${states?size} states:<#list automaton.states as state> state
               .id</#list>" << std::endl;
24     </#list>
25     return 0;
26 }
```

Listing 5.3: FreeMarker workflow example

[3] Apache FreeMarker template engine https://freemarker.apache.org/

5.3.2 General Template Implementation

The generated C++ code description of the RegEx matcher is composed of two main types of methods: various functions dedicated to representing the matching algorithm of each RegEx's generated NFA, and a single topmost function which runs all of the aforementioned NFA functions, passing them their input strings and registering if any of them performed a match.

Listing 5.4 shows the FreeMarker template code responsible for creating the matcher's topmost function. The '`automaton`' functions correspond to each PCRE's generated NFA matching algorithm.

```
1 bool runAutomata(char input[], int input_size) {
2     <#list automata as automaton>const bool a${automaton?counter} = automaton${
          automaton?counter}(input, input_size);</#list>
3     return <#list automata as automaton>a${automaton?counter}<#if !automaton?
          is_last> || </#if></#list>;
4 }
```

Listing 5.4: Topmost function template code

Each NFA's matching algorithm is implemented as a software state machine which allows multiple states to be active at any given time. Aside from representing an automaton's behavior, this translation is direct in the sense that it mimics its overall structure and state-transition logic.

NFA matching is done in an iterative way with two sets of states being kept, one representing the current iteration's state activation status and the other representing which states will be considered active in the next iteration. Matching stops when either no transitions can be triggered, or when an acceptance state is reached, which represents a match. A flowchart representing this process is shown in Figure 5.18.

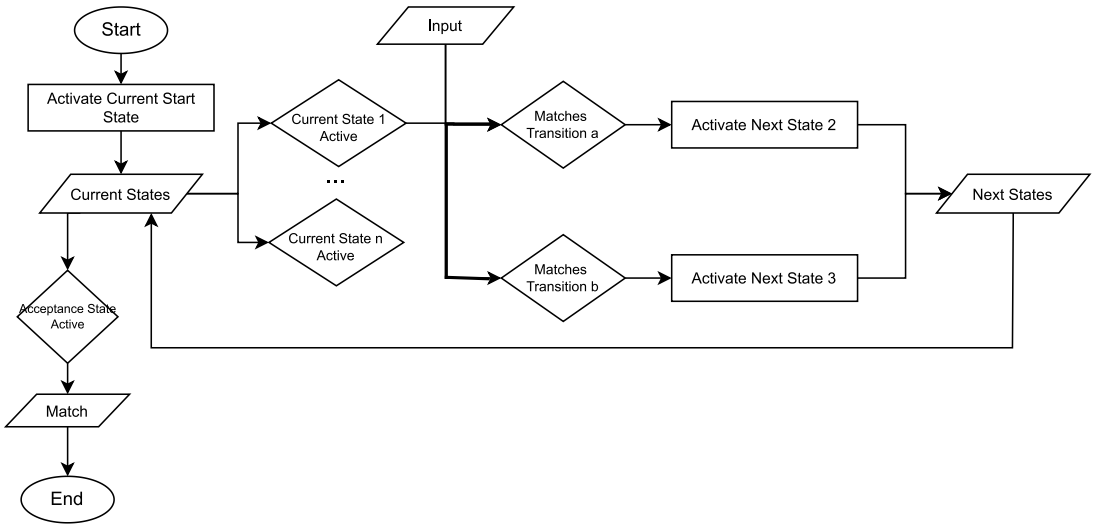


Figure 5.18: NFA matching flowchart

Given the non-deterministic behavior of NFAs, the existence of multiple possible branching paths within an automaton may end up creating divergences regarding the values stored within its internal variables. An example of this is when nested bounded quantifiers are utilized, like in the expression $(ab\{2,4\})\{3\}$ whose simplified automaton can be seen in Figure 5.19. In this example, when two 'b' symbols have already been read, and a third one is being processed, both transitions outgoing from state 2 have to be considered, since it's not known whether the next symbol will be an 'a', leading to a new iteration of the outer quantifier, or a 'b', representing a new iteration of the inner quantifier. This requires a way to keep multiple variable values simultaneously.

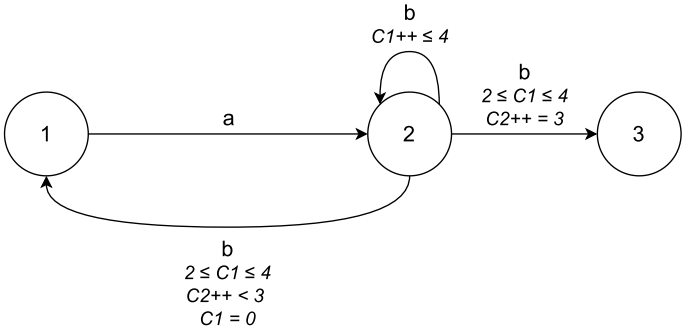


Figure 5.19: Simplified NFA for $(ab\{2,4\})\{3\}$

To solve this, each state keeps information regarding the status of its own internal variables, as seen in Listing 5.5. For the example given earlier of the input string 'abbb' for the expression $(ab\{2,4\})\{3\}$, this allows state 2 to hold the value 3 and 0 for C1 and C2 respectively, and state 1 the value 0 for C1 and 1 for C2.

```
1 template <size_t C, size_t B>
2 struct State {
3     int input_index = -1;
4     std::array<Counter, C> counters = {};
5     std::array<Buffer, B> buffers = {};
6 };
```

Listing 5.5: State structure

When NFAs makes use of counters of capture buffers, the status of these variables is transferred from one state to another whenever a transition linking the two of them is triggered. This is performed through methods similar to the one seen in Listing 5.6.

```
1 template <size_t C, size_t B>
2 void transferCounters(State<C,B> &from, State<C,B> &to)
3 {
4     to.counters = from.counters;
5 }
```

Listing 5.6: Transfer of counter variables

To implement each NFAs corresponding state transition logic, a template section similar to the one shown in Listing 5.7 is used. The methods `getTransitionComparison` and `getTransitionBody` seen in the template section are custom-developed FreeMarker functions responsible for generating the code corresponding to a transition check and a transition activation, respectively. For simple token-only transitions, without any bounded quantifiers, capture groups, or back-references, these functions generate code corresponding to basic token checks and to the activation and status transfer of the set of next active states.

```
<#list automaton.states as state>
const bool is_current${state.id}_active = stateIsActive(current[${state.id}]);
</#list>
<#list automaton.states as state>
<#if (state.outgoing_transitions?size > 0)>
if (is_current${state.id}_active)
{
    const int input_token = input[current[${state.id}].input_index];
    int input_increment;
    <#list state.outgoing_transitions as transition>
    ${getTransitionComparison(transition, automaton)}
    {
        transition_occurred = true;
        ${getTransitionBody(transition, automaton)}
    }
    </#list>
}
</#if>
</#list>
```

Listing 5.7: NFA state transition logic template section

Listing 5.8 shows an example of the code that is generated to represent the state transition logic of the NFA corresponding to the RegEx $[ab]c$.

```
//...
const bool is_current0_active = stateIsActive(current[0]);
const bool is_current1_active = stateIsActive(current[1]);
const bool is_current2_active = stateIsActive(current[2]);
if (is_current0_active)
{
    const int input_token = input[current[0].input_index];
    int input_increment;
    if ((input_increment = (input_token == 97 || input_token == 98)))
    {
        transition_occurred = true;
        next[1].input_index = current[0].input_index + input_increment;
    }
}
if (is_current1_active)
{
    const int input_token = input[current[1].input_index];
    int input_increment;
```

```
19      if ((input_increment = input_token == 99))
20      {
21          transition_occurred = true;
22          next[2].input_index = current[1].input_index + input_increment;
23      }
24  }
25  //...
```

Listing 5.8: State transition code for [ab]c

The state transition logic is repeated until either no further transitions can be triggered or an acceptance state is reached, with the set of next states being copied to represent the set of the now current states, as in the template section shown in Listing 5.9.

```
1  do {
2  // ... TRANSITION LOGIC ...
3  in_end_state = <#list automaton.end_states as end_state>current[${end_state.id}<#if !
       end_state?is_last> || </#if></#list>;
4  memcpy(current, next, ${automaton.states?size} * sizeof(State<${template_arguments}>));
5  std::fill(next, next + ${automaton.states?size}, State<${template_arguments}>());
6  } while (transition_occurred && !in_end_state);
```

Listing 5.9: Template code for the repetition of the transition logic

Listing 5.10 completes the example shown in Listing 5.8 by showcasing the code responsible for the repetition of the transition logic for the expression $[ab]c$.

```
1  do {
2  // ... TRANSITION LOGIC ...
3  in_end_state = is_current2_active;
4  memcpy(current, next, 3 * sizeof(State<0, 0>));
5  std::fill(next, next + 3, State<0, 0>());
6  } while (transition_occurred && !in_end_state);
```

Listing 5.10: Transition logic repetition code for [ab]c

5.3.3 Capture Groups and Back-references

When an automaton transition corresponds to a symbol that is nested inside a capture group, that symbol has to be stored in a buffer so that it can later be referenced. To implement this behavior, a buffer structure, which can be seen in Listing 5.11, is implemented.

```
1  struct Buffer {
2      std::array<int, 32> data = {};
3      size_t size = 0;
4      unsigned int allowed_offsets = 0;
5  };
```

Listing 5.11: Capture Buffer Structure

A transition that needs to store its input makes a call to the method `bufferEnqueue`, shown in Listing 5.12, inside its corresponding transition activation code within a NFA's overall transition logic. An adaptation of this method also allows for the insertion of buffers within buffers, for expressions like $(a^+b)(\backslash 1|c)\backslash 2$, when back-references are used inside capture groups.

```
1  void bufferEnqueue(Buffer &buffer, int element, bool clear)
2  {
3      if (clear)
4          bufferClear(buffer);
5      buffer.data[buffer.size++] = element;
6  }
```

Listing 5.12: Buffer insertion function

Back-reference transitions make use of the method `bufferCheck`, shown in Listing 5.13, in order to perform their transition activation check within its corresponding NFA's transition logic.

```
1  bool bufferCheck(const Buffer &buffer, const char *input, int input_size, int
       input_index, int *input_increment)
2  {
3      //...
4      for (unsigned int i = 0; i <= buffer.allowed_offsets; i++)
5      {
6          bool valid = true;
7          for(int j = 0; j + i < buffer.size && valid; j++)
8              valid = input[input_index + j] == buffer.data[j + i];
9
10         if (valid)
11         {
12             *input_increment = buffer.size - i;
13             return true;
14         }
15     }
16
17     return false;
18 }
```

Listing 5.13: Buffer comparison function

The `allowed_offsets` structure variable seen in Listings 5.13 and 5.11 is to account for the non-determinism present in expressions such as $.^*(a^+b)\backslash 1$, whose corresponding simplified automaton can be seen in Figure 5.20. For this RegEx's automaton, all the input strings 'aaabab', 'aaabaab', and 'aaabaaab' have to match, but every starting 'a' symbol simultaneously causes both a wildcard self-transition within state 1 and a buffer reset transition from state 1 to state 2, leading to buffer B1 being constantly, making it store a single 'a' character after processing multiple starting 'a' symbols. The variable `allowed_offsets` solves this by being incremented whenever

taking a buffer reset transition to a state that was already active, instead of totally resetting a buffer's state. For the aforementioned cases, this results in buffer B1 storing the string 'aaa' with the `allowed_offsets` value of 2. This value means that when performing checks on B1, matches ignoring either its first or both first and second stored characters should also be considered valid, as seen in `bufferCheck` shown in Listing 5.13.

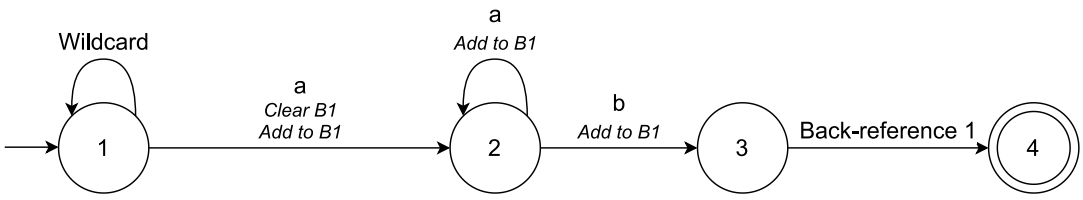


Figure 5.20: Simplified automaton for $.^*(a^+b)\backslash 1$

5.3.4 Bounded Quantifiers

To store a bounded quantifier's respective repetitions, a counter structure is implemented. This structure can be seen on Listing 5.14.

```
struct Counter {
    unsigned int value = 0;
    unsigned int value_offset = 0;
};
```

Listing 5.14: Counter structure

Transitions whose triggering involves comparisons with values stored within counters do so by calling different comparison functions corresponding to each of the different comparison operators required to implement this feature, as listed in Section 5.1.2. Listing 5.15 shows the function used to check if a counter's stored value is equal to a given target value.

```
bool counterCheckEqual(const Counter &counter, const unsigned int target_value)
{
    bool valid = false;
    for (unsigned int i = 0; i <= counter.value_offset && !valid; i++)
        valid = (counter.value - i + 1) == target_value;

    return valid;
}
```

Listing 5.15: Counter equality function

Again, like in the case for the code implementation of back-references and capture groups discussed in Section 5.3.3, an offset variable, `value_offset`, is used to account for non-deterministic

behavior. In the simplified automaton for $.^*a\{3\}$ shown in Figure 5.21, it is clear that both transitions outgoing from state 1 will keep getting triggered when processing a large string of 'a' symbols, leading to the value stored within the counter variable C1 continuously being reset, never reaching its intended target value. To account for this behavior, `value_offset` gets incremented when a counter reset transition to an already active state is triggered, instead of totally resetting the counter's value. This variable's value is used to offset a counter's current value when performing comparison operations like the one shown in Listing 5.15.

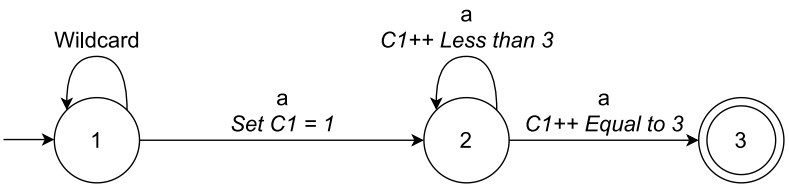


Figure 5.21: Simplified automaton for $.^*a\{3\}$

Chapter 6

Experimental Results

This chapter presents the results extracted from both the analysis performed on various Regular Expression (RegEx) rulesets and the implemented RegEx matcher. A description of the experimental environment used to extract results about the RegEx matcher is given, along with individual segments in which the performance of bounded quantifiers and both capture groups and back-references are discussed separately.

6.1 Ruleset Features Analysis

This section presents the extraction results of various RegEx characteristics, listed in Section 4.2, from a sample of selected RegEx rulesets. The sample of chosen RegEx rulesets is composed of ClamAV's main database, Snort 2.9 and Snort 3 Community, Protomata's protein motifs, and openly available sets of Yara, Suricata 4 and 5 rules.

6.1.1 Ruleset Size

In total, 107629 expressions belonging to various RegEx rulesets are analyzed. Table 6.1 demonstrates how these RegExes are distributed amongst the various selected rulesets.

Ruleset	Number of RegExes
ClamAV	33397
Snort2.9	17491
Snort3 Community	1080
Protomata	1309
Yara	28875
Suricata4	15137
Suricata5	10340

Table 6.1: Number of RegExes per ruleset

6.1.2 Operator and flag utilization

As can be seen in the chart depicted in Figure 6.1, the data representing each Perl Compatible Regular Expressions (PCRE) operator's frequency is marked by the prevalence of the concatenation operator across the entirety of the collected rulesets, with this operator appearing in close to the totality of expressions.

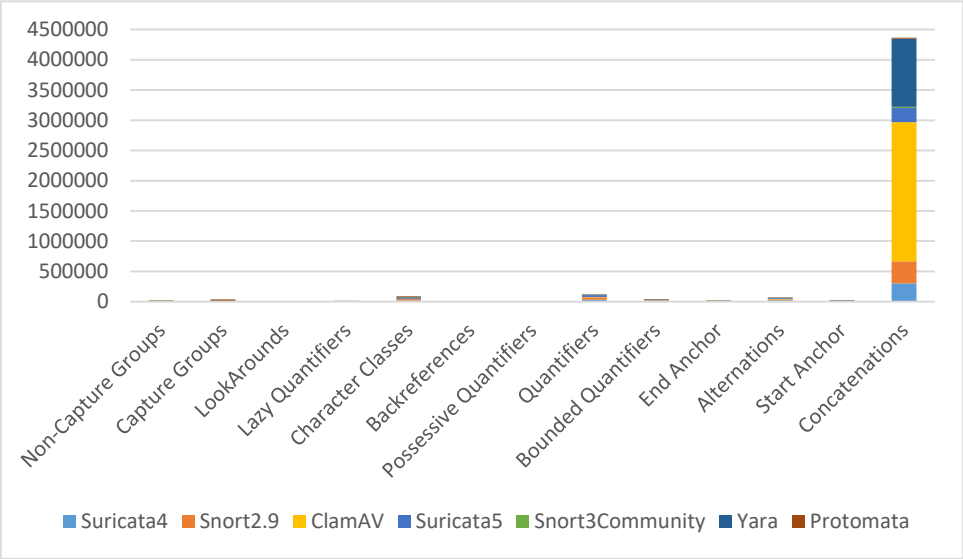


Figure 6.1: Total operator frequency

Considering data regarding operator usage by expression while omitting the concatenation operator, which is shown in the chart represented in Figure 6.2, it is clear that the two next most reoccurring operators are quantifiers, appearing in around 36.33% of all expressions, and character classes, which appear in about 26.69% of the accounted expressions.

Possessive quantifiers and lookarounds, two operators that are not supported by the developed matcher, represent the PCRE operators with the lowest presence across the entirety of the considered rulesets, with their utilization never going upwards of 5% for any ruleset. The other unsupported PCRE feature, lazy quantifiers, has prevalent use in the Protomata ruleset, being present in about 73% of expressions, with every quantifier used in this ruleset being tagged as lazy.

Bounded quantifier use across all the expressions is very significant. Depending on the ruleset, this operator can be used in up to 73.72% of a ruleset's expressions, as is the case of the Protomata ruleset. However, its use can also fall below the 1% mark, as in the case of the ClamAV and Yara rulesets.

In general, back-references are not a very common operator, with some rulesets, such as Protomata, ClamAV, and the chosen Yara ruleset, not having a single occurrence of this operator. It has significant use in the Snort 3 Community ruleset, however, with 22.13% of the expressions contained in that ruleset making use of back-references. It is noted that even though Protomata doesn't use back-references, every single one of its 1309 expressions uses capture groups.

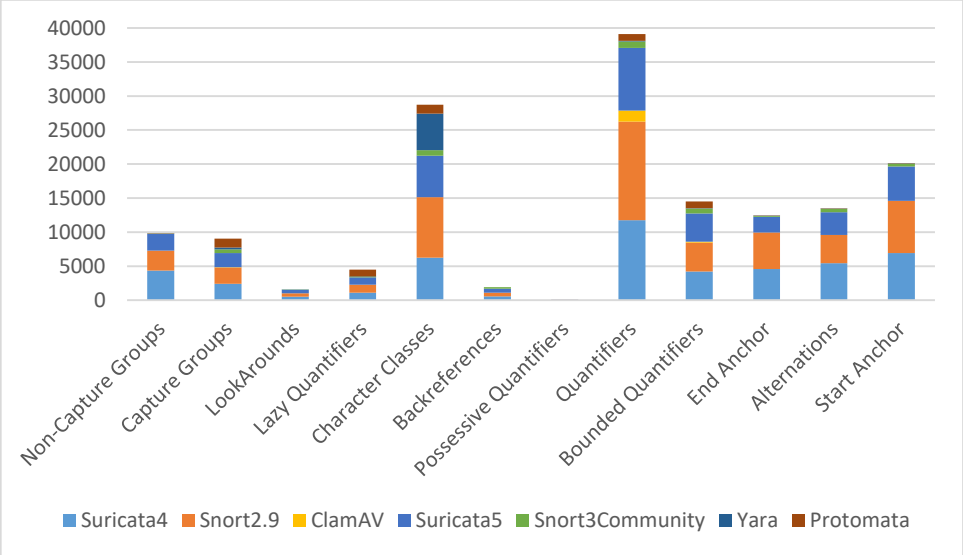


Figure 6.2: Operator occurrence by expression

Regarding PCRE flags, the '*i*' flag, which signalizes that character matches should be performed in a case-insensitive fashion, is the most prevalent one, with close to a quarter of all of the accounted expressions (25.76%) making use of it. The '*s*' and '*m*' flags fall behind the '*i*' flag, with each having about 5% presence in the overall rules.

The flags '*A*' and '*x*' report virtually no utilization. This is due to the ability of the start anchor operator (^ operator) to perform the same function as the '*A*' flag, and the lack of need for comments within the rulesets' RegExes themselves, which makes the '*x*' flag unnecessary.

The rulesets belonging to the domain of Network Intrusion Detection Systems (NIDSs), these being the Snort and Suricata rulesets, make the greatest use of PCRE flags, with most of these flags being ruleset-specific flags used to indicate where to look for matches within a given network packet. The remaining rulesets make no use of PCRE flags, apart from the chosen Yara ruleset, where the '*i*' flag appears a relatively small number of times, with an appearance rate of 8.11%.

The chart presented in Figure 6.3 provides a visualization of the overall occurrence of the various PCRE flags across the chosen rulesets.

6.1.3 Expression length

From all the expressions whose maximum input string length could be statically determined by analyzing the expression itself, the results shown in Figure 6.4 were collected. Most of the data refers to the ClamAV and Yara rulesets, which, due to their relatively smaller use of quantifiers and prevalence of RegExes that actually represent constant strings, have the most expressions with constant input lengths.

Of all the considered expressions, the vast majority of expressions (70.85%) have a maximum input string length equal to or under 50 symbols, with about half of the overall expressions having a maximum input string length that falls below 30 symbols.

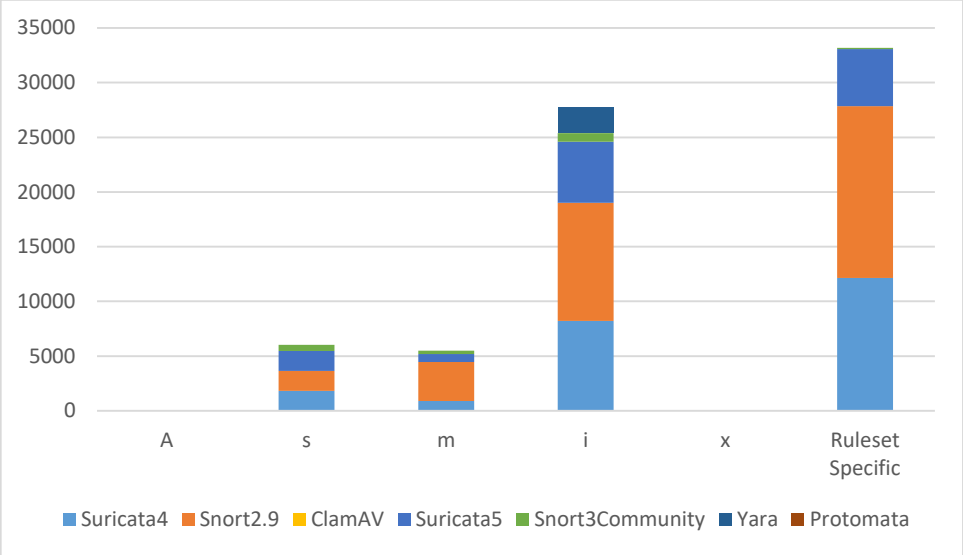


Figure 6.3: Flag occurrence

ClamAV, and Yara, to a lesser extent, have a large number of expressions whose matching input string length is above the overall length average, with a big number of expressions matching input strings with over 100 symbols.

Out of all of the 1309 Protomata RegEx, 1200 of them have a maximum input length equal to or under 15 symbols, with all 1309 expressions having a determinable maximum input length that never surpasses 30 input symbols. This is due to Protomata's RegEx following limited repetition patterns, not using any quantifier with an unknown upper bound, such as the '*match zero or more times*' ($*$ operator) and '*match one or more times*' ($+$ operator) regular quantifiers, or the '*match at least x times'* quantifier, with the form $\{x,\}$.

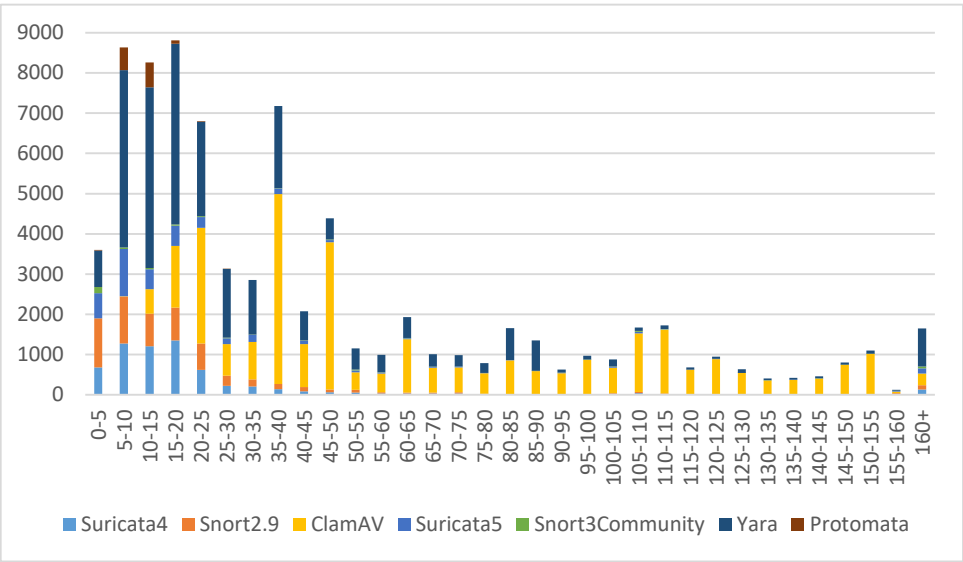


Figure 6.4: Maximum input string lengths

6.1.4 Capture group length

Capture groups whose maximum capture length can be determined by analyzing the group itself have their maximum length compiled in the chart depicted in Figure 6.5. Data is primarily composed of capture groups from the Protomata ruleset, whose RegExes always envelop their used character classes in capture groups, as seen in the subset of Protomata PCREs shown in Listing 6.1. This is likely to enable some form of post-processing to be performed after an input string matches an expression, making it possible to know which character from a character class was matched.

```
/(([RX])([GX])([DBX]))/
/(([NBX])([^P])([STX])([^P]))/
/(([LIVMFEZX])([FYX])([PX])([WX])([MX])([KRQZTAX]))/
/(([LX])([MX])([AX])([EZQZX])([GX])([LX])([YX])([NBX]))/
```

Listing 6.1: Subset of Protomata RegExes

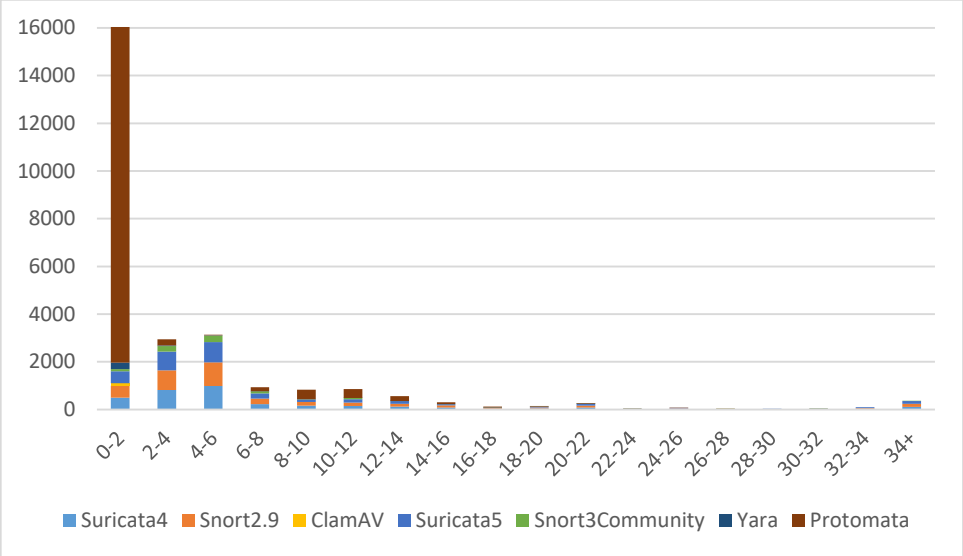


Figure 6.5: Maximum capture group lengths

When considering only capture groups that are referenced by back-references, whose data is on Figure 6.6, only the rulesets that utilize back-references are represented, with ClamAV, Protomata, and the Yara ruleset being left out. Snort's version 3 community ruleset is also mostly absent due to its expressions' back-references relying upon capture groups whose maximum size can't possibly be determined through expression analysis alone.

From the data for the remaining rulesets, it is noted that about half (48.39%) of the referenced capture groups have a maximum length that is under 4 input symbols, with more than two-thirds (71.63%) requiring less than 12 symbols to store the captured input. The largest capture group recorded required 255 symbols, with this being an extreme outlier.

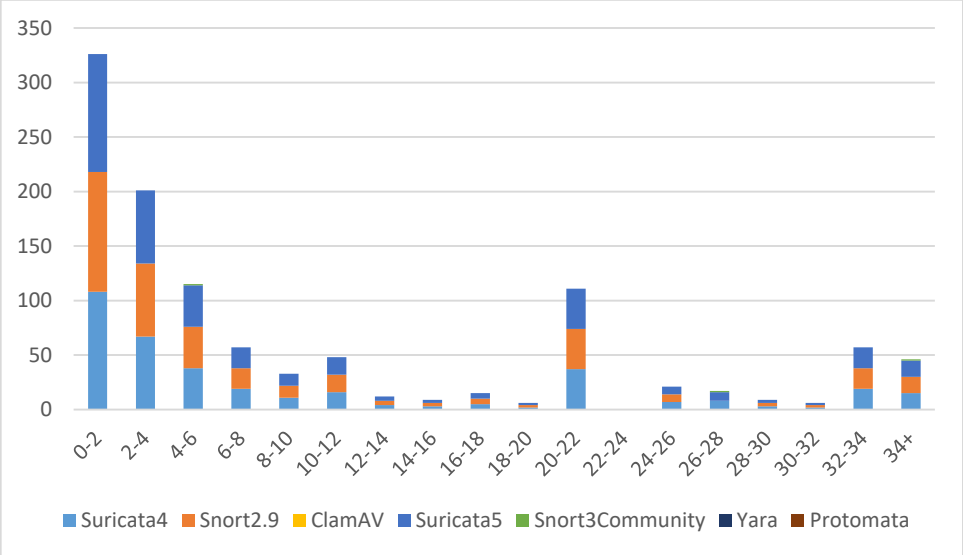


Figure 6.6: Maximum referenced capture group lengths

Back-references that call upon constant capture groups appear in only one expression amongst the totality of all rulesets, with it being shown in Listing 6.2. In this expression, the named capture group $(? < pushvar > \backslash.push\backslash x28\backslash x22)$ represents the string '.push("' and is referenced by its name various times throughout the expression as $(?P = pushvar)$. The constant string within this capture group has a length of 7 symbols.

```
/^(?:\x22|[\x20-\x7e]{0,70}\x22)\x29\x3b\s*[^\.]+(?<pushvar>\.push\x28\x22)(?:\x22
    |[\x20-\x7e]{0,70}\x22)\x29\x3b\s*[^\.]+(?P=pushvar)(?:[\x20-\x7e]{0,70}\x22)\
    x29\x3b\s*[^\.]+(?P=pushvar)(?:[\x20-\x7e]{0,70}\x22)\x29\x3b\s*[^\.]+(?P=
    pushvar)/
```

Listing 6.2: PCRE with reference to constant group

6.1.5 Bounded quantifier repetitions

For all of the different varieties of bounded quantifiers, data collected for their number of repetitions is compiled in the chart represented in Figure 6.7. For quantifiers with both a lower and an upper limit of repetitions, only the upper limit is considered.

Besides a spike in the upper numbers of repetitions caused due to some expressions in the Snort and Suricata rulesets utilizing quantifiers with around a thousand repetitions, most expressions use quantifiers with a far lower amount of maximum repetitions, being that half of all the considered bounded quantifiers employ at most 6 repetitions.

Protomata and Snort's community version 3 represent opposites in this regard, as 70.22% of Protomata's bounded quantifiers demand less than 4 repetitions, while 71.50% of bounded quantifiers that make up this version of Snort require upwards of 950 repetitions. The relatively

small number of bounded quantifiers present in ClamAV and the Yara rulesets also tend to use fairly small numbers of repetitions.

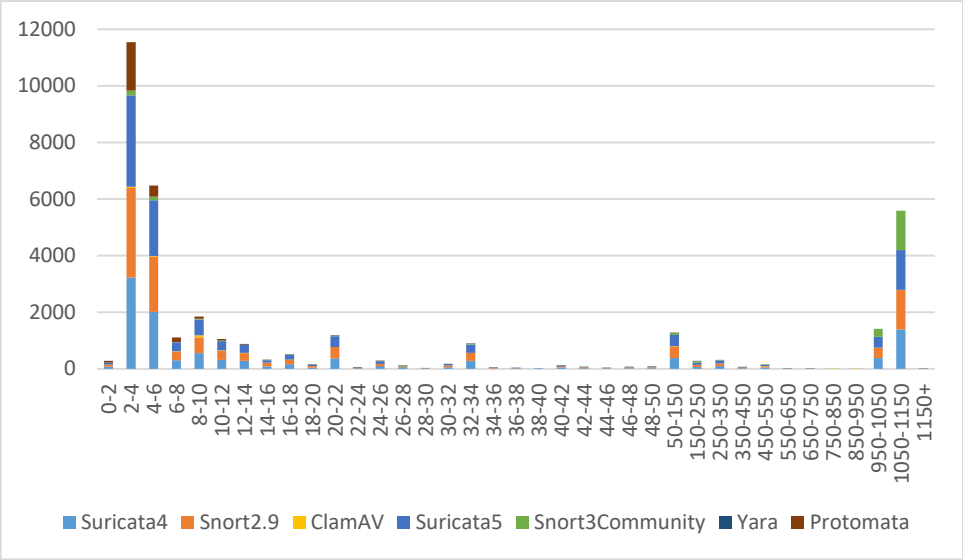


Figure 6.7: Maximum repetitions for bounded quantifiers

6.2 RegEx Matcher

This section starts by detailing the portions of each ruleset that are supported by the RegEx matcher. Then, a description of the environment utilized to test and extract performance results from the matching solution, followed by said performance results, is provided.

Performance results are split between general performance results, where expressions with neither bounded quantifiers nor back-references are tested, and individual sections where these two operators, along with their implementation alternatives, are tested individually.

6.2.1 Supported RegExes

The set of PCRE operators supported by the developed matcher, presented earlier in Table 5.1, leaves out both lazy and possessive quantifier modifiers, as well as lookarounds. The information extracted regarding the utilization of each PCRE operator, presented in Section 6.1.2, makes it possible to determine the impact of not including these operators in the matcher's feature set.

Table 6.2 presents the portion of each selected RegEx ruleset that is supported by the matcher by excluding expressions that utilize any of the aforementioned unsupported PCRE operators.

In total, out of all the 107629 expressions belonging to the selected rulesets, 101540 are supported by the matcher, with this number being equivalent to 94.343% of all the chosen RegExes. Individually, each ruleset has support for the vast majority of its expressions, with the exception of the Protomata ruleset due to the frequent use of lazy quantifiers within its RegExes.

Ruleset	Total Expressions	Supported Expressions
ClamAV	33397	33397 (100%)
Snort2.9	17491	15856 (90.652%)
Snort3 Community	1080	890 (82.407%)
Protomata	1309	344 (26.280%)
Yara	28875	28874 (99.997%)
Suricata4	15137	13474 (89.014%)
Suricata5	10340	8705 (84.188%)

Table 6.2: Supported expressions of each RegEx ruleset

6.2.2 Experimental setup

To test and extract results from the developed matcher, expressions are chosen randomly from the set of all RegExes that make up the rulesets picked for analysis (introduced in Section 4.1), with these being: Snort's version 2.9 and Snort community's version 3, ClamAV, Protomata, and public Suricata 4, Suricata 5 and Yara rulesets.

Synthesis of the hardware-accelerated matcher design is performed through Vitis HLS version 2023.1.1. It targets a Xilinx Alveo U250 data accelerator card with a Gen3 x16 PCIe interface and four DIMMs of 16 GB, 2400 MT/s, DDR4 memory for a total of 64 GB of off-chip memory. This accelerator card uses the UltraScale+ XCU250 Field-Programmable Gate Array (FPGA), which is composed of 216K Configurable Logic Blocks (CLBs), 1728K lookup tables (LUTs), 3456K Flip-flops (FFs), 12288 Digital Signal Processing (DSP) slices, and 54 MB of on-chip memory achieved through both 36 Kb Block Random-Access Memory (BRAM) modules and 1280 UltraRAM memory blocks of 288 Kb each.

All of the synthesis results are achieved by performing synthesis using Vitis Kernel Flow with a target clock period of 5 ns, equivalent to a clock frequency of 200 MHz.

6.2.3 General performance

To extract performance metrics from the developed accelerated RegEx matcher, 6 expressions utilizing different PCRE features are chosen randomly from the set of RegExes that make up the chosen rulesets, with each expression belonging to a different ruleset. These RegEx make use of all the supported PCRE features (stated in Section 5.1.1) besides bounded quantifiers and capture groups/back-references, as these are discussed individually in the sections that follow.

Listing 6.3 shows the PCREs chosen to extract overall performance results from the RegEx matcher, along with the ruleset to which they belong. Note that no expression belonging to the Protomata ruleset is chosen due to every single RegEx it contains utilizing capture groups.

```
1 /\x81\x7c\x79\x60\xea\x77\xe8\x83\x7c\x79/ (ClamAV)
2 /<title>\s*(?:log\s*in|sign\s*in)/i (Suricata 4)
3 /^\/(?:kas|ser|mac)[0-9]+\.png$/i (Suricata 5)
4 /\x3C\x3C[^>]*\x2FJavaScript/smi (Snort 2.9)
```

```
5 /awstats.pl?[^\r\n]*logfile=\x7C/i (Snort 3 Community)
6 /\$mysql_key\s*=\s*@?base64_decode/ (Yara)
```

Listing 6.3: Chosen PCREs used for results extraction

Regarding the makeup of the Non-deterministic Finite Automata (NFAs) generated for each of the chosen PCREs, Table 6.3 compiles the descriptions of each expression's corresponding NFAs concerning their total number of states, transitions, and how many states are considered acceptance states. These three characteristics are taken into account due to the way they directly translate into conditional statements when generating a NFA's corresponding software description.

	States	Transitions	Acceptance States
ClamAV RegEx	11	10	1
Suricata4 RegEx	22	28	2
Suricata5 RegEx	16	18	1
Snort2.9 RegEx	15	16	1
Snort3 RegEx	21	24	1
Yara RegEx	28	33	1

Table 6.3: Resulting NFAs' descriptions

The number of states that a RegEx's corresponding NFA has is directly related to how many matchable symbols are contained within the RegEx. For the analyzed PCREs, all their corresponding NFAs have a number of states which is equal to the expression's number of matchable symbols plus one. Matchable symbols are the ones that are actually compared with symbols coming from an input string. Given the Snort 2.9 RegEx `\x3C\x3C[^>]*\x2FJavaScript` as an example, this RegEx has three matchable symbols written in hexadecimal, one belonging to the character class `[^>]` and the remaining 10 constituting the string 'JavaScript', totaling 14 matchable symbols within the expression, making its corresponding NFA have a total of 15 states.

In general, the generated NFAs have a number of transitions which is fairly similar to the number of states, with these diverging further when more regular quantifiers (`*`, `+`, `?` operators) and alternations (`|` operator) are utilized, with the number of added transitions due to alternations depending on how many alternation options are chained together.

NFAs with multiple acceptance states are created when an expression has branching options that never get merged with the rest of the expression. Examples of these are the Suricata 4 expression `<title>\s*(?:log\s*in|sign\s*in)` and the Suricata 5 expression `^\/(?:kas|ser|mac)[0-9]+\.png$`. While the diverging options `(?:kas|ser|mac)` present at the start of the second expression all get merged with the main expression's body upon reaching the character class `[0-9]`, this never happens with the branching group `(?:log\s*in|sign\s*in)` due to its location being at the end of the expression. The number of these sorts of unmerged branches within a RegEx directly coincides with its corresponding NFA number of acceptance states.

When it comes to the synthesis results of the generated NFAs' code descriptions, Table 6.4 presents the achieved clock period and FPGA resource utilization when synthesizing a hardware implementation of each of the aforementioned PCRE's corresponding NFAs.

The implementation's top function `runAutomata`, responsible for calling each individual NFA's `automaton` function, shares its clock period with the NFA function with the highest clock period. This means that the implementation's overall clock period is 4.458 ns, equivalent to a clock frequency of about 224 MHz, coinciding with the slowest clock speed obtained by the synthesizing Suricata 5 PCRE's automaton function.

ClamAV's and Yara's chosen PCREs comparatively lower clock periods in relation to the other expressions' NFAs' synthesis results are likely due to both these expressions not making use of the '`i`' case-insensitive flag. This flag makes character comparisons more cumbersome by enforcing both a lower-case comparison and an upper-case comparison to be performed, effectively doubling the overall number of comparisons made.

	Clock Period (ns)	FF	LUT	BRAM	URAM
ClamAV RegEx	2.876	1006	1571	0	0
Suricata4 RegEx	4.003	4790	4569	0	0
Suricata5 RegEx	4.458	3194	3508	0	0
Snort2.9 RegEx	4.003	3048	3090	0	0
Snort3 RegEx	4.036	4437	4261	0	0
Yara RegEx	2.985	5829	5431	0	0

Table 6.4: Synthesis results of the expressions' NFAs

The average matching time achieved by running the RegEx matcher on a Xilinx Alveo U250 ten consecutive times, for a 256-byte input string, is 0.46471 milliseconds. In comparison, using grep [1], on an Intel 10750H running at 2.6GHz, takes 133.25456 milliseconds, being around 287 times slower.

6.2.4 Capture group and back-reference performance

Capture groups and back-references are RegEx features that are inherently linked together. Capture groups are tasked with storing a matching portion of a given input string so that that same exact input portion can be matched again through the use of a back-reference. However, capture groups also open up the capability to analyze and process segments of a matched input string, and are often used as the default grouping operator when RegExes are written.

Cases, where capture groups are used without a corresponding back-reference, represent occasions where capture groups can be transformed into non-capture groups. From a RegEx matching standpoint, this transformation bears no impact on the input strings that are matched by a given

[1] GNU Grep https://www.gnu.org/software/grep/

expression, with its semantics being kept intact, and provides a way to minimize the matcher's resources, since no unnecessary storage buffers are allocated.

To measure the resource costs of implementing capture groups, RegExes from the Protomata ruleset featuring different numbers of unreferenced capture groups are selected and synthesized with and without the aforementioned capture group transformation. Protomata is selected for the extraction of these sorts of RegExes due to the prevalence of unreferenced capture groups in every single one of its expressions, as stated in Section 6.1.4.

Table 6.5 presents the resource utilization of five Protomata RegExes with increasing numbers of capture groups, providing a comparison between utilizing capture groups as written in the original expressions, and substituting them with non-capture groups. All the expressions are tested with the buffers corresponding to capture groups set to store at most 8 input symbols. This buffer width is enough to support the majority of capture groups with known lengths from all the analyzed rulesets, as shown in Figure 6.6.

Capture Transformation	Capture Groups	Clock Period (ns)	FF	LUT	BRAM
/(([RX])([GX])([DBX]))/					
No	4	3.13	31961	11113	0
Yes	0	2.72 (x0.869)	1483 (x0.046)	1320 (x0.106)	0
/(([NBX])([^P])([STX])([^P]))/					
No	5	2.962	4674	17495	1
Yes	0	2.985 (x1.008)	993 (x0.212)	1320 (x0.075)	0 (x0)
/(([GX])([AX])([KX])([RX])([HX]))/					
No	6	4.319	3388	16706	1
Yes	0	3.107 (x0.719)	1191 (x0.352)	1488 (x0.089)	0 (x0)
/(([KX])([KRX])([CX])([GX])([HX])([LMQZRX]))/					
No	7	4.319	3936	19653	1
Yes	0	2.72 (x0.630)	1390 (x0.353)	1663 (x0.085)	0 (x0)
/(([SX])([KX])([RX])([KX])([YX])([RX])([KX]))/					
No	8	4.122	4480	20414	1
Yes	0	3.107 (x0.754)	1590 (x0.355)	1746 (x0.086)	0 (x0)

Table 6.5: Capture group/non-capture group resource usage comparison

It is clear that the substitution of capture groups for non-capturing ones provides decreases in resource utilization of all types, while also providing meaningful decreases in clock period. When employing non-capture groups, BRAMs are not utilized at all, and, when this is also the case when using capture groups, it is at the cost of immense FF and LUT utilization. Similar across all the selected RegExes, the substitution of capture groups majorly decreases the usage of LUTs.

Capture Group	Back-references	Clock Period (ns)	FF	LUT	BRAM
Yes	Yes	3.232	4647	14882	2
Yes	No	3.291	4700	16821	1
No	No	2.91	3807	12875	0

Table 6.6: Back-reference resource usage comparison

Regarding the testing of back-references, the RegEx highlighted earlier in Listing 6.2 is selected for results extraction. The fact that this expression contains a back-reference that references a fixed string makes it a prime candidate to test back-references, since it can be substituted by the constant string it references without altering the overall meaning of the expression. This allows what is essentially the same expression to be tested with and without back-references.

Table 6.6 shows the synthesis results of the same rule with both back-references and capture groups, with capture groups only (with the back-reference being substituted by the fixed string it references), and with neither back-references nor capture groups.

6.2.5 Bounded quantifier performance

The two different approaches to implementing bounded quantifiers, previously presented in Section 5.1.2, are compared by picking an expression that makes use of this operator and synthesizing variations of that same expression with differing numbers of repetitions.

The chosen RegEx is `/AUTH\s[^\n]{100}/smi`, selected due to its overall compact size and due to the fact that the quantifier is only being applied to a character class with a single negated symbol. As this type of character class requires only a single non-equality comparison, its complexity is of the simplest order, making this expression a prime candidate to benefit from the bounded quantifier's implementation where its operand is explicitly repeated within the expression's corresponding automaton.

In order to focus singularly on bounded quantifiers, the expression's flags are removed, turning the test RegEx into `/AUTH\s[^\n]{x}/`, with 'x' taking on multiple repetition values.

Figure 6.8 depicts how the usage of FFs and LUTs scales according to the number of repetitions declared by the bounded quantifier, for both the implementation which uses transitions with counters and the implementation that explicitly repeats the quantifier's operand. For this last implementation alternative, the utilization of FFs and LUT scales somewhat linearly with the increase of repetitions, while for the first one, which employs counters, resource utilization remains constant no matter the value of the quantifier's repetitions.

The recorded FF and LUT resources utilization behavior resembles what happens at the automaton level, with states and transitions scaling linearly in relation to a quantifier's repetitions for the implementation that does not use counters, while these remain the same for the implementation that does.

In Figure 6.9, a chart is shown where BRAMs usage and the achieved design's clock period in relation to the number of required repetitions, for both bounded quantifier implementations, are

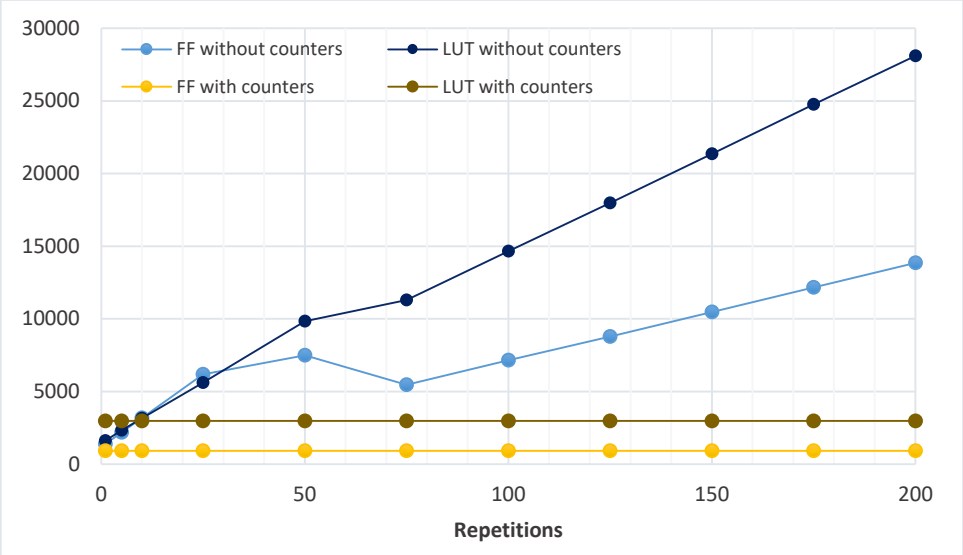


Figure 6.8: FF and LUT usage for both bounded quantifier implementations

presented. Again, like the case for FFs and LUTs, no changes in clock period or BRAM usage are registered for the implementation with counters since the matcher's design is essentially the same independently of the target repetition value.

In this expression, for the implementation without counters, 2 BRAMs start being utilized in between 100 and 125 repetitions and onwards. This coincides with a 21% increase in clock period, from 2.811 ns to 3.407 ns, surpassing the alternative implementation's constant clock period of 3.339 ns. Higher BRAM counts are achieved when targeting even higher numbers of repetitions but with no meaningful impact on the clock period.

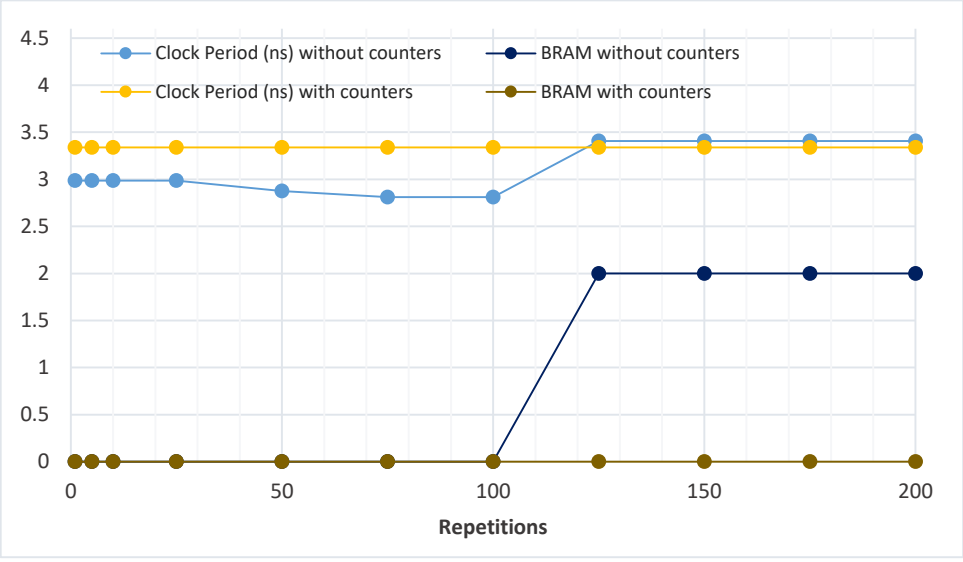


Figure 6.9: Achieved clock period and BRAM usage for both bounded quantifier implementations

Overall, the implementation of bounded quantifiers that uses counters to keep track of the

number of repetitions is much more resource-efficient, keeping resource requirements constant independently of a quantifier's target repetitions, with FF usage always being lower than in the alternative implementation.

The implementation where a bounded quantifier's operand is explicitly repeated shows an advantage in lower LUT utilization for repetitions under 10 (for expressions of the nature of the one tested) and in lower clock periods. This last advantage is only maintained until BRAMs start being utilized, at which point this implementation's obtained clock periods become slightly higher than the alternative, making explicit repetition an objectively worse approach in these scenarios.

Chapter 7

Conclusion and Future Work

The increasingly higher volumes of data over recent times and across multiple fields have generated a need for fast Regular Expression (RegEx) pattern-matching solutions. This is particularly true for applications where pattern-matching has to be performed close to real-time, such as the case of Network Intrusion Detection Systems (NIDSs).

Field-Programmable Gate Arrays (FPGAs) have proven to be a suitable platform for the implementation of proficient RegEx matching implementations, capable of providing more performant solutions than functionally equivalent software ones. Their capability of being re-configurable is also a major advantage, as they are capable of keeping up with changes in the RegEx rulesets, unlike other hardware accelerator platforms, which have to be substituted with redesigned hardware.

Through the survey of the state of the art, it is apparent that while the implementation of hardware-accelerated RegEx matching on FPGAs is not an entirely new topic, the majority of the work in this field focuses on lower-level Register-Transfer Level (RTL) implementations of such matching solutions and not on higher-abstraction High-Level Synthesis (HLS) alternatives. Additionally, the majority of surveyed solutions either lack any mention regarding the implementation of back-references or state that they are not supported. To a lesser extent, this is also true for bounded quantifiers, with some solutions resorting to naive repetition approaches in order to implement them.

The analysis of various RegEx rulesets shows that, overall, the lack of support for back-references causes small decreases in the portion of a ruleset's expressions that are supported by a matching solution that is missing this feature. While the selected Yara, Protomata, and ClamAV rulesets don't use back-references at all, Snort's 3 community version features back-references in 22.13% of its expressions, which is a significant portion and makes it the most affected ruleset by the lack of support for this Perl Compatible Regular Expressions (PCRE) operator.

Even though back-references are not that frequent, capture groups are fairly prevalent, meaning that most of the input data being recorded by these capture groups is not actually being used during the matching process.

In comparison to back-references, bounded quantifiers are shown to be a more frequent operator, being present in various degrees in all of the selected RegEx rulesets, with the selected

Snort and Suricata rulesets having a large proportion of these quantifiers that require upwards of a thousand of repetitions.

The proposed RegEx matching solution, designed with HLS and supporting both back-references and bounded quantifiers, features implementation options to save resources on unused capture group data and both a naive implementation of bounded quantifiers, where expressions are explicitly repeated, and a more scalable one, using counter variables to keep track of repetitions. This solution supports 94.343% of all of the 107629 analyzed expressions, with the majority of every ruleset's expressions being supported with the exception of Protomata, due to the lack of support for lazy quantifiers.

The removal of capture groups that aren't referenced by a back-reference proves to be a significant optimization option, both lowering the overall resource usage and showing improvements in clock period. With this option toggled, no Block Random-Access Memory (BRAM) modules are allocated, and major decreases in Flip-flop (FF) utilization and especially lookup table (LUT) utilization are achieved, with decreases of upwards of 64% and 90%, respectively, being registered.

Regarding both bounded quantifier implementations, the naive implementation with explicit repetition only proves to be beneficial for quantifiers referencing small sub-expressions with rather low repetition numbers. Comparisons between both implementations, using a single-character quantifier (the best-case scenario for the naive implementation), show that resource utilization between both implementations breaks even at around 10 repetitions, scaling somewhat linearly for the naive implementation and remaining constant for the one using counters. In regards to the achieved clock period, the break-even point is reached further, at around 100 repetitions, while also remaining constant for the implementation employing counters. Overall, the implementation of bounded quantifiers that uses counters to track the number of repetitions is far more flexible, being capable of representing both the repetition of large expressions as well as large numbers of repetitions more efficiently, due to its constant scaling properties.

The storage buffers and counters, utilized to respectively implement back-references and bounded quantifiers, are a major challenge when it comes to implementing a Non-deterministic Finite Automaton (NFA)-based RegEx matcher. The non-deterministic nature of these automata makes it so that these structures don't share the same state across every automaton state, as would be the case in a Deterministic Finite Automaton (DFA), leading to the duplication of resources and a more complex matching logic for transitions utilizing these structures.

Furthermore, in cases when both a regular transition and a back-reference transition can be taken, this leads to cases where the outgoing states are reading input characters from different places within the input string. This means that the input string isn't necessarily processed sequentially, making the streaming of this data unfeasible and prohibiting the development of a solution that follows the producer-consumer paradigm of FPGA design.

Future work should begin by addressing this issue, by changing the way back-reference transitions are matched within the RegEx matcher's code description. As a possible solution, instead of checking if the entire buffer matches a whole portion of the input string, this check could be performed character by character, in step with how the input string should be processed, with each

buffer instance storing a reference to which of its containing characters is currently being compared. With that addressed, additional measures can be taken to further improve the matcher's performance, such as the implementation of input character decoders or multi-character NFAs.

References

[1] E.M. Aboulhamid and J. Lapalme. Recent trends in hardware/software description languages. In *Proceedings of the 12th IEEE International Conference on Fuzzy Systems (Cat. No.03CH37442)*, pages 185–188, December 2003.

[2] Tamer Abuhmed, Aziz Mohaisen, and Daehun Nyang. A Survey on Deep Packet Inspection for Intrusion Detection Systems. *ArXiv*, February 2008.

[3] Normi Sham Awang Abu Bakar. Using regular expressions for mining data in large software repositories. In *The 5th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 1–6, November 2014.

[4] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, CoNEXT '07, pages 1–12, New York, NY, USA, December 2007. Association for Computing Machinery.

[5] Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[6] Devon Callanan, Luke Kljucaric, and Alan George. Accelerating Regular-Expression Matching on FPGAs with High-Level Synthesis. In *International Workshop on OpenCL*, IWOCL'21, pages 1–8, New York, NY, USA, April 2021. Association for Computing Machinery.

[7] Brian Caswell and Jay Beale. *Snort 2.1 Intrusion Detection, Second Edition*. Elsevier, June 2004. Google-Books-ID: rzd3PLH3vDsC.

[8] Milan Ceška, Vojtech Havlena, Lukáš Holík, Jan Korenek, Ondrej Lengál, Denis Matoušek, Jirí Matoušek, Jakub Semric, and Tomáš Vojnar. Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 109–117, April 2019. ISSN: 2576-2621.

[9] Christopher R. Clark and David E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In Peter Y. K. Cheung and George A. Constantinides, editors, *Field Programmable Logic and Application*, Lecture Notes in Computer Science, pages 956–959, Berlin, Heidelberg, 2003. Springer.

[10] C.R. Clark and D.E. Schimmel. Scalable pattern matching for high speed networks. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–257, April 2004.

[11] Alessandro Comodi, Davide Conficconi, Alberto Scolari, and Marco D. Santambrogio. TiReX: Tiled Regular eXpression Matching Architecture. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 131–137, May 2018.

[12] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, July 2009. Conference Name: IEEE Design & Test of Computers.

[13] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. HARE: Hardware accelerator for regular expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, October 2016.

[14] Huifang Guo and Kunpeng Jiang. High Throughput Regular Expression Matching Algorithm. In *2015 International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 368–372, December 2015. ISSN: 2472-7555.

[15] Leticia I. Gómez and Alejandro A. Vaisman. RE-SPaM: Using Regular Expressions for Sequential Pattern Mining in Trajectory Databases. In *2008 IEEE International Conference on Data Mining Workshops*, pages 395–398, December 2008. ISSN: 2375-9259.

[16] Masahiro Iida. What Is an FPGA? In Hideharu Amano, editor, *Principles and Structures of FPGAs*, pages 23–45. Springer, Singapore, 2018.

[17] Zsolt István, David Sidler, and Gustavo Alonso. Runtime Parameterizable Regular Expression Operators for Databases. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 204–211, May 2016.

[18] Tomonori Izumi and Yukio Mitsuyama. Design Flow and Design Tools. In Hideharu Amano, editor, *Principles and Structures of FPGAs*, pages 87–115. Springer, Singapore, 2018.

[19] Nan Jiang, Ping Lin, Yulong He, Zhuozhi Tan, and Jin Hu. Design of A New Type of Regular Expression Matching Engine Based on FPGA. In *2021 IEEE 15th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pages 159–163, October 2021. ISSN: 2163-5056.

[20] Xin Jin, Jun Lin, and Zhongfeng Wang. A Novel Compiler for Regular Expression Matching Engine Construction. In *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 251–256, October 2018.

[21] L. Karttunen, J.-P. Chanod, G. Grefenstette, and A. Schille. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328, December 1996. Publisher: Cambridge University Press.

[22] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D. Hämäläinen. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, May 2019. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[23] Jie Li, Andreas Aurelius, Viktor Nordell, Manxing Du, Åke Arvidsson, and Maria Kihl. A five year perspective of traffic pattern evolution in a residential broadband access network. In *2012 Future Network & Mobile Summit (FutureNetw)*, pages 1–9, July 2012.

[24] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of Pattern Matching Circuits for Regular Expression on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(12):1303–1310, December 2007. Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[25] Le Hoang Long, Tran Trung Hieu, Vu Tan Tai, Nguyen Hoa Hung, Tran Ngoc Thinh, and Dinh Duc Anh Vu. Enhanced FPGA-based architecture for regular expression matching in NIDS. In *ECTI-CON2010: The 2010 ECTI International Confernce on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pages 666–670, May 2010.

[26] Pierre Marchal. Field-programmable gate arrays. *Communications of the ACM*, 42(4):57–59, April 1999.

[27] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, March 1960. Conference Name: IRE Transactions on Electronic Computers.

[28] A. R. Meyer and M. J. Fischer. Economy of description by automata, grammars, and formal systems. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 188–191, East Lansing, MI, USA, October 1971. IEEE.

[29] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pages 127–136, New York, NY, USA, December 2007. Association for Computing Machinery.

[30] Nga Lam Or, Xing Wang, and Derek Pao. MEMORY-Based Hardware Architectures to Detect ClamAV Virus Signatures with Restricted Regular Expression Features. *IEEE Transactions on Computers*, 65(4):1225–1238, April 2016. Conference Name: IEEE Transactions on Computers.

[31] Derek Pao, Nga Lam Or, and Ray C. C. Cheung. A memory-based NFA regular expression match engine for signature-based intrusion detection. *Computer Communications*, 36(10):1255–1267, June 2013.

[32] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D. Santambrogio. CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching. *ACM Transactions on Embedded Computing Systems*, 20(5s):51:1–51:24, September 2021.

[33] Indranil Roy, Ankit Srivastava, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. High Performance Pattern Matching Using the Automata Processor. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1123–1132, May 2016. ISSN: 1530-2075.

[34] Jacques Sakarovitch. Kleene's theorem revisited. In Alica Kelemenová and Jozef Kelemen, editors, *Trends, Techniques, and Problems in Theoretical Computer Science*, Lecture Notes in Computer Science, pages 39–50, Berlin, Heidelberg, 1987. Springer.

[35] Kai Salomaa and Sheng Yu. NFA to DFA transformation for finite languages. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*, Lecture Notes in Computer Science, pages 149–158, Berlin, Heidelberg, 1997. Springer.

[36] Tsutomu Sasao and A. Mishchenko. LUTMIN : FPGA Logic Synthesis with MUX-Based and Cascade Realizations. 2009.

[37] Vipula Sateesh, Connor Mckeon, Jared Winograd, and Andre DeHon. Pipelined Parallel Finite Automata Evaluation. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 108–116, December 2019.

[38] Kamil Sert and Cüneyt F. Bazlamacci. NFA Based Regular Expression Matching on FPGA. In *2021 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pages 1–5, November 2021.

[39] L. Sharmila, U. Sakthi, A. Geethanjali, and Suresh Sagadevan. Regular Expression Based Pattern Matching for Gene Expression Data to Identify the Abnormality Gnome. In *2017 Second International Conference on Recent Trends and Challenges in Computational Models (ICRTCCM)*, pages 301–305, February 2017.

[40] R. Sidhu and V.K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 227–238, March 2001.

[41] P. Sutton. Partial character decoding for improved regular expression matching in FPGAs. In *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*, pages 25–32, December 2004.

[42] Lin Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 112–122, June 2005. ISSN: 1063-6897.

[43] Ken Thompson. Programming Techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[44] Jack Wadden, Tommy Tracy, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Jeffrey Udall, Matthew Wallace, Mircea Stan, and Kevin Skadron. AutomataZoo: A Modern Automata Processing Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 13–24, September 2018.

[45] Xing Wang, Nga Lam Or, Ziyan Lu, and Derek Pao. Hardware Accelerator to Detect Multi-Segment Virus Patterns. *The Computer Journal*, 58(10):2443–2460, October 2015.

[46] Chengcheng Xu, Shuhui Chen, Jinshu Su, S. M. Yiu, and Lucas C. K. Hui. A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms. *IEEE Communications Surveys & Tutorials*, 18(4):2991–3029, 2016. Conference Name: IEEE Communications Surveys & Tutorials.

[47] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. High-speed regular expression matching engine using multi-character NFA. In *2008 International Conference on Field Programmable Logic and Applications*, pages 131–136, September 2008. ISSN: 1946-1488.

[48] Jiajia Yang, Lei Jiang, Xu Bai, Huailiang Peng, and Qiong Dai. A High-Performance Round-Robin Regular Expression Matching Architecture Based on FPGA. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7, June 2018. ISSN: 1530-1346.

[49] Jiajia Yang, Lei Jiang, Qiu Tang, Qiong Dai, and Jianlong Tan. PiDFA: A practical multi-stride regular expression matching engine based On FPGA. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–7, May 2016. ISSN: 1938-1883.

[50] Yi-Hua Yang and Viktor Prasanna. High-Performance and Compact Architecture for Regular Expression Matching on FPGA. *IEEE Transactions on Computers*, 61(7):1013–1025, July 2012. Conference Name: IEEE Transactions on Computers.

Appendix A

Ruleset Analysis Results

Tables A.1 to A.7 present the results belonging to each Regular Expression (RegEx) feature extracted from the selected rulesets, providing a complete view of the results presented as charts in Section 6.1.

	Non-Capture Groups	Capture Groups	LookArounds	Lazy Quantifiers	Character Classes	Back-references	Possessive Quantifiers	Quantifiers	Bounded Quantifiers	End Anchor	Alternations	Start Anchor	Concatenations
Suricata4	9188	5516	911	2894	18190	1326	10	34224	10534	4622	20806	6928	300309
Snort2.9	7555	5519	864	2946	23694	1328	9	39776	10416	5385	19130	7675	368405
ClamAV	0	103	0	0	0	0	0	2964	243	0	1088	0	2297430
Suricata5	7464	5091	911	2849	18022	1326	10	31316	10497	2357	18735	5052	233114
Snort3Community	12	2331	52	496	4484	247	0	6630	2354	127	2697	406	23673
Yara	0	357	0	3	10803	0	0	272	87	1	160	3	1126752
Protomata	2	18594	0	2505	13071	0	0	2505	2505	9	2	7	15990

Table A.1: Total Perl Compatible Regular Expressions (PCRE) operator utilization

	Non-Capture Groups	Capture Groups	LookArounds	Lazy Quantifiers	Character Classes	Back-references	Possessive Quantifiers	Quantifiers	Bounded Quantifiers	End Anchor	Alternations	Start Anchor	Concatenations
Suricata4	4345	2399	531	1122	6229	558	10	11768	4220	4584	5440	6927	15068
Snort2.9	2941	2402	480	1146	8912	553	9	14463	4230	5353	4124	7672	17424
ClamAV	0	29	0	0	0	0	0	1656	136	0	29	0	33397
Suricata5	2498	2101	531	1094	6085	558	10	9178	4180	2320	3350	5051	10271
Snort3Community	9	515	42	148	827	239	0	958	693	126	499	397	1049
Yara	0	308	0	1	5362	0	0	110	63	1	45	3	28870
Protomata	2	1309	0	965	1309	0	0	965	965	9	2	7	1309

Table A.2: Number of expressions using each PCRE operator

	A	s	m	i	x	Ruleset Specific
Suricata4	1	1817	887	8222	0	12134
Snort2.9	1	1838	3571	10797	0	15716
ClamAV	0	0	0	0	0	0
Suricata5	1	1810	731	5588	0	5212
Snort3Community	2	558	301	774	0	114
Yara	0	0	0	2342	0	0
Protomata	0	0	0	0	0	0

Table A.3: Total PCRE flag utilization

Max. Input Length	Suricata4	Snort2.9	ClamAV	Suricata5	Snort3Community	Yara	Protomata
0-5	682	1219	0	623	156	907	14
5-10	1276	1168	3	1174	41	4411	564
10-15	1203	810	609	502	27	4490	623
15-20	1349	816	1537	498	32	4495	87
20-25	621	652	2881	262	21	2353	18
25-30	223	254	781	150	13	1707	3
30-35	204	177	929	177	8	1362	0
35-40	141	124	4722	139	6	2045	0
40-45	85	110	1068	82	3	727	0
45-50	59	68	3665	57	10	527	0
50-55	52	69	436	50	17	526	0
55-60	29	30	465	30	7	429	0
60-65	28	30	1323	27	6	519	0
65-70	28	29	616	27	2	305	0
70-75	31	33	614	32	2	274	0
75-80	11	13	510	11	0	242	0
80-85	5	6	847	5	0	794	0
85-90	9	10	570	9	1	749	0
90-95	10	10	515	10	4	80	0
95-100	5	5	858	5	2	95	0
100-105	22	17	633	24	5	177	0
105-110	36	34	1458	37	20	83	0
110-115	15	15	1586	12	4	93	0
115-120	1	2	620	1	0	55	0
120-125	4	4	878	4	0	57	0
125-130	2	1	538	2	1	90	0
130-135	3	2	354	3	2	45	0
135-140	7	6	353	7	0	47	0
140-145	2	2	403	2	1	48	0
145-150	0	9	737	0	1	54	0
150-155	1	1	1019	1	1	73	0
155-160	18	18	42	18	2	23	0
160+	127	111	288	130	50	943	0

Table A.4: Expressions' maximum input string lengths

Capture Group Length	Suricata4	Snort2.9	ClamAV	Suricata5	Snort3Community	Yara	Protomata
0-2	502	504	103	490	90	272	17012
2-4	820	825	0	782	231	28	262
4-6	992	991	0	846	270	18	27
6-8	230	231	0	213	92	3	163
8-10	157	155	0	110	9	2	398
10-12	149	149	0	136	48	1	380
12-14	119	118	0	117	5	1	205
14-16	89	89	0	44	10	0	75
16-18	30	29	0	29	2	0	36
18-20	42	42	0	41	5	0	15
20-22	81	81	0	79	3	1	15
22-24	8	8	0	8	1	0	1
24-26	22	24	0	22	0	0	3
26-28	11	3	0	11	1	0	2
28-30	13	13	0	13	0	0	0
30-32	14	14	0	14	1	0	0
32-34	31	31	0	31	0	0	0
34+	117	118	0	116	10	0	0

Table A.5: Maximum capture group lengths

Capture Group Length	Suricata4	Snort2.9	ClamAV	Suricata5	Snort3Community	Yara	Protomata
0-2	108	110	0	108	0	0	0
2-4	67	67	0	67	0	0	0
4-6	38	38	0	38	1	0	0
6-8	19	19	0	19	0	0	0
8-10	11	11	0	11	0	0	0
10-12	16	16	0	16	0	0	0
12-14	4	4	0	4	0	0	0
14-16	3	3	0	3	0	0	0
16-18	5	5	0	5	0	0	0
18-20	2	2	0	2	0	0	0
20-22	37	37	0	37	0	0	0
22-24	0	0	0	0	0	0	0
24-26	7	7	0	7	0	0	0
26-28	8	0	0	8	1	0	0
28-30	3	3	0	3	0	0	0
30-32	2	2	0	2	0	0	0
32-34	19	19	0	19	0	0	0
34+	15	15	0	15	1	0	0

Table A.6: Maximum referenced capture group lengths

Repetitions	Suricata4	Snort2.9	ClamAV	Suricata5	Snort3Community	Yara	Protomata
0-2	76	74	3	69	1	4	57
2-4	3227	3169	36	3234	160	18	1702
4-6	2009	1960	22	1967	118	15	385
6-8	309	302	7	311	11	5	167
8-10	559	552	74	559	23	8	78
10-12	321	316	20	323	31	3	42
12-14	283	278	5	287	8	3	21
14-16	90	120	5	89	9	0	20
16-18	169	157	1	170	16	0	9
18-20	50	50	1	50	0	0	7
20-22	376	396	6	375	20	2	4
22-24	19	19	0	19	0	0	1
24-26	94	81	2	94	13	0	1
26-28	32	27	0	31	31	0	1
28-30	13	11	0	13	0	0	0
30-32	57	59	2	57	1	0	0
32-34	285	284	2	286	21	10	1
34-36	16	16	0	16	1	0	2
36-38	13	14	1	13	0	0	0
38-40	2	0	0	2	0	0	0
40-42	40	38	3	40	1	0	1
42-44	26	26	0	26	3	0	0
44-46	12	12	1	12	0	0	2
46-48	23	22	0	23	0	0	1
48-50	27	25	0	28	8	2	0
50-150	386	404	21	384	72	8	3
150-250	70	71	5	69	59	5	0
250-350	94	83	11	94	24	4	0
350-450	23	20	2	23	13	0	0
450-550	47	46	2	47	27	0	0
550-650	2	2	2	2	0	0	0
650-750	3	3	0	3	0	0	0
750-850	0	0	2	0	0	0	0
850-950	0	0	1	0	0	0	0
950-1050	378	377	0	378	285	0	0
1050-1150	1398	1398	0	1398	1398	0	0
1150+	5	4	6	5	0	0	0

Table A.7: Maximum bounded quantifier repetitions