FPGA design for Intrusion Detection/Prevention applications

MASTER’S THESIS

Michael Dyrmose

JULY 1, 2008
DTU FOTONIK, DEPARTMENT OF PHOTONICS ENGINEERING
DTU, TECHNICAL UNIVERSITY OF DENMARK
Abstract

This project presents an approach to increase the performance of intrusion detection and prevention by using dedicated hardware. The most demanding challenge of intrusion detection is payload inspection, i.e. examining the actual data content of network packets. The open-source intrusion detection system Snort facilitates the use of Perl Compatible Regular Expressions (PCRE) to describe patterns for payload data matching. This project provides a way of implementing a large number of PCRE-based Snort rules in parallel on a Field Programmable Gate Array thus providing a substantial increase in performance.
Acknowledgements

Writing a master’s thesis can be a long and draining process, where new problems suddenly can arise at any time. I would like to thank my supervisor Michael S. Berger for always taking time to help me understand and address the problems at hand.

I would also like to thank friends and family who has taken time reading and correcting the report as well as providing invaluable feedback.

Michael Dyrmose, July 2008
# Contents

1 Introduction 1

2 Background 3
  2.1 Snort ................................. 3
    2.1.1 Snort architecture .................. 4
    2.1.2 Snort rules ......................... 4
  2.2 FPGAs ................................. 5
  2.3 Regular expressions .................... 6
    2.3.1 Alternation .......................... 6
    2.3.2 Quantifiers .......................... 6
    2.3.3 Grouping ............................. 7
  2.4 Perl Compatible Regular Expressions ..... 7
    2.4.1 Basic operators and quantifiers ...... 7
    2.4.2 Escape sequences ..................... 8
    2.4.3 Specific character classes ........... 8
  2.5 Finite automata ........................ 9
    2.5.1 Deterministic finite automata ........ 9
  2.6 Non-deterministic finite automatas ...... 10
    2.6.1 Converting regular expressions to NFA .. 12

3 Related work 17

4 Implementation 19
  4.1 Logic structures ........................ 20
    4.1.1 Comparators .......................... 21
    4.1.2 Concatenation entity ................. 21
    4.1.3 OR-entity ............................ 22
    4.1.4 AND-entity ............................ 22
    4.1.5 Quantifiers ........................... 22
  4.2 Top-level structure ..................... 23
## 5 The ranged quantifier

5.1 Logical structure ............................................. 30
5.2 The kill entities ............................................. 31
  5.2.1 Constructing the kill entity ......................... 33
  5.2.2 Example of kill entity construction ............. 35

## 6 Software

6.1 Software description ......................................... 41
6.2 Automatic VHDL source-code generation .................. 42
  6.2.1 PCRE extraction ........................................ 42
  6.2.2 Insertion of concatenation operators .......... 42
  6.2.3 Infix to postfix conversion ....................... 44
  6.2.4 Constructing the syntax tree ..................... 47
  6.2.5 Checking for ranged quantifiers .............. 49
  6.2.6 Generating VHDL source-code from syntax tree .. 50
  6.2.7 Generating VHDL top-level entity ............. 50

## 7 Evaluation

7.1 Functionality ................................................ 54
  7.1.1 Snort rules used in the software evaluation .... 54
  7.1.2 Evaluating designs in Quartus II .............. 58
7.2 Potential ..................................................... 62
  7.2.1 Source-code size ..................................... 62
  7.2.2 FPGA resource consumption .................... 64

## 8 Discussion

## 9 Future work

## 10 Conclusion

## A Executing snort2vhdl

## B VHDL source-code

  B.1 Positive character comparator ......................... 81
  B.2 Concatenation entity .................................... 82
  B.3 OR-entity .................................................. 83
  B.4 AND-entity ................................................. 84
  B.5 \*-quantifier ............................................. 85
  B.6 +-quantifier ............................................... 85
  B.7 ?-quantifier ............................................... 86
  B.8 Ranged quantifier ......................................... 86
B.9 Top-level entity ........................................ 88
List of Figures

2.1 A DFA accepting all bit sequences containing at least two 1’s . 10
2.2 An NFA accepting all bit sequences ending in either 00 or 0110 11
2.3 NFA matching a single input symbol . . . . . . . . . . . . . . 12
2.4 Concatenation rule connecting two sub-NFAs . . . . . . . . . . 13
2.5 Construction rule alternating between two sub-NFAs (OR op-
erator) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.6 Construction rule for an arbitrary number of sub-NFAs (*-
quantifier) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.7 Construction rule for a positive number of sub-NFAs (+-quantifier) 14
2.8 Construction rule for matching zero or one occurrences of a
sub-NFA (?-quantifier) . . . . . . . . . . . . . . . . . . . . . . 15
2.9 NFA representing ((a*b)(c|d)) constructed using the rules . . . 15
4.1 Logical structures corresponding to NFA construction rules . . 25
4.2 Logical structure showing the design of the top-level entity
and connections to the sub-entities . . . . . . . . . . . . . . . . 26
4.3 Logical structure showing the design of the top-level entity
and connections to the sub-entities . . . . . . . . . . . . . . . . 27
5.1 Counter incrementing in a ranged quantifier based on input
signal from a sub-entity . . . . . . . . . . . . . . . . . . . . . . 30
5.2 Incorrect counter incrementing in a ranged quantifier based on
input signal from a sub-entity . . . . . . . . . . . . . . . . . . . 30
5.3 Correct counter incrementing in a ranged quantifier by using
the kill signal . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.4 Logical structure of the ranged quantifier . . . . . . . . . . . . 32
5.5 Tree representation of the regular expression ((a*b)(c|d)){3, 5} 36
5.6 Tree showing the first step of the kill entity generation process 37
5.7 Tree representation of the expression ¬(a*b) . . . . . . . . . . 37
5.8 Tree representation of the expression (a*b)¬(c|d) . . . . . . . . 38
5.9 The complete kill entity design to the expression ((a*b)(c|d)){3, 5} 39
6.1 Each iteration in the postfix to infix conversion process of the expression \((abc|def) \gg g^*\) .................................................. 46
6.2 Tree representing a positive character class .......................... 49
6.3 Tree representing the regular expression \(a^* \gg (b|c)\) ............ 50

7.1 Rules 0-4 selected from the Snort ruleset ............................ 55
7.2 Rules 5-9 selected from the Snort ruleset ............................ 56
7.3 Content of payload file used in the evaluation of the generated design ................................................................. 57
7.4 Theoretical single thread scenario showing three regular expressions parsing the same input .............................. 58
7.5 Theoretical parallel scenario showing three regular expressions parsing the same input ........................................... 59
7.6 The resulting simulation waveform after synthesis in Quartus II 60
7.7 Graph showing the number of generated entities and comparators at different design sizes ........................................... 64
7.8 Standard FPGA logic block implementation .......................... 65
7.9 Graph showing the resource consumption on the Cyclone II chip at different design sizes .................................... 66

8.1 Graph showing the resource consumption on the Stratix II chip 69
8.2 Graph showing the estimated resource consumption of the Stratix II chip ......................................................... 71
Introduction

The Internet is growing at a rate that was unimaginable just a few years ago. An ever-increasing number of people are going on-line using network connectivity on a rising number of different platforms. Today cell phones, set-top boxes, digital cameras and even mp3-players are all being connected to the Internet, providing new exciting possibilities for end users. However, these new possibilities come at a price. Bandwidth demands are placing a serious load on the physical network connections used. The amount of data being transported between nodes on the network has seen a tremendous increase in recent years.

Increased network utilization has an impact on how Internet security should be dealt with. Monitoring network connections with bandwidths measured in terms of Gb/s is a very resource-demanding process. At the same time the increase in people on-line also gives rise to a growing number of threats and attacks taking place over the Internet. Due to this it is important to monitor a given network connection in order to identify and prevent potential attacks or to gather evidence in case of a successful attack.

This project presents an approach to monitoring high-speed network traffic for intrusion detection. The bottle-neck of intrusion detection is payload inspection, i.e. examining the actual data content of network packets. Monitoring very high-speed connections can result in dropped packets if the Network Intrusion Detection System (NIDS) running the monitoring service is
unable to keep up with the traffic flow. The goal of the research conducted is to show how intrusion detection can be improved by implementing payload inspection on a dedicated hardware device.

The focus of this project is on how to implement payload inspection rules from the open-source intrusion detection package Snort [18]. Because of the constant increase in malicious on-line activities, the size of the Snort rule-set is growing at a similar rate, putting more and more load on the system running the intrusion detection service. This growing computational demand for each packet has made it necessary to look at ways of offloading some of the demanding processing applications to dedicated hardware devices. Implementing parallel pattern search engines in dedicated hardware is a way to help improve the performance of an NIDS. Recent years’ development in the area of Field Programmable Gate Arrays (FPGAs) has made it an excellent choice for supporting intrusion detection, due to its flexible nature that allows easy implementation of proposed hardware designs. Implementing the payload inspection rules in a parallel fashion allows simultaneous execution of all rules on the same input, which can lead to an increase in performance, compared to the single thread approach obtained by using a standard CPU.

The primary goal of the project is to illustrate how to create parallel hardware designs that can be used to program an FPGA, based on the Snort rule-set. Another objective is to develop software that facilitates automatic generation of the hardware designs, based on single rules or the full rule-set from the Snort installation. The functionality of the implemented designs is proven by testing with generated network data as input.
2 Background

This report describes the research and work that has been conducted throughout the project. The report uses different theoretical models and approaches such as regular expressions and finite automata. In this section an introduction to these areas will be given, providing a fundamental knowledge that will help the reader understand the terms used in the project. First we take a look at Snort, the intrusion detection software used in the project. This is followed by an introduction to the FPGA - a programmable hardware device used to implement Snort rules. Regular expressions and the theory of converting expressions to hardware using non-deterministic finite automatas are introduced as the final section.

2.1 Snort

Snort is a very popular open-source NIDS package being developed by the company Sourcefire. It was initially written by Martin Roesch, who also founded Sourcefire and is still working there. Snort was introduced in late 1998 and started out as a mere network traffic packet sniffer. The software has continuously developed over time and is now a full-blown intrusion detection and prevention package with a large user community supporting its further development.
2.1 Snort

Snort is based on libpcap\(^1\) and is described as a lightweight security package. This refers to the fact, that it can easily be deployed on a given network, with very little disruption to the functionality and performance of the network. Its basic features has relatively low resource requirements and it is cross-platform as it supports just about any OS including Linux, Windows, *BSD, MacOS, Solaris and many more. This makes it possible to use the same Snort configuration on all servers of the network.

Snort can be used to perform a number of different surveillance tasks including protocol analysis and content pattern matching. A commonly used feature is the capability to detect/block a large number of different types of network attacks, e.g. buffer overflows and web application attacks. Intrusion detection in Snort is performed by classifying the network data on the TCP/IP stack of the system. Malicious traffic is identified by predefined rules that use a number of different methods for discovering intrusion patterns.

2.1.1 Snort architecture

The internal design of the Snort system is based on performance, simplicity and flexibility. The system can be divided into three subsystems; a packet decoder, a detection engine and a logging/alerting system. By parsing the rules before the packet sniffing is initialized, the pr. packet amount of processing required is kept to a minimum. The basic functionality of the packet decoder is analyzing raw network traffic, and ordering the network packets for the detection engine. The Snort detection engine is the place where the actual intrusion detection is taking place. The detection rules are applied to the input data and the installed plug-ins are executed. The logging/alerting system is taking care of providing feedback from the detection engine. An alert is raised if a rule has been triggered, and the event is logged if Snort is configured to do so [7]. The actual use and configuration of Snort lies outside the scope of this project, as focus will be on how to implement certain Snort rules in hardware.

2.1.2 Snort rules

The different attack signatures, i.e. the descriptions of known attack patterns, are categorized in a series of rule-files that together make up the complete Snort rule-set. The rules are written in a flexible language that describes which traffic to let through and which to block. Snort rules can

\(^1\)API (Application Programming Interface) for capturing network traffic
be described in different ways; one of them is by payload inspection, i.e., the
rule includes a description of what to look for in the actual data content of
network packets. Snort provides different ways of describing which patterns
to search for in the payload. Simple string search is one of the methods,
where the network traffic is searched for a defined string. In addition to
simple string based matching, Snort provides the opportunity of using Perl
Compatible Regular Expressions (PCREs) in the rules, thereby providing
a flexible notation of writing string recognition patterns. This project will
study how to improve the performance of payload inspection using these reg-
ular expressions. A thorough introduction to regular expressions and PCRE
is given later in this section. By using PCREs for content matching it is
possible to define complex patterns matching strings that otherwise would
be impossible to describe.

2.2 FPGAs

A field-programmable gate array (FPGA) is a hardware device that can be
configured to behave like a specific hardware design. It is widely used in
the design phase of hardware production for simulation purposes before the
hardware is manufactured in large quantities. FPGAs consist of a number
of logical cells that can be programmed by the user. The logic cells of the
FPGA can be programmed to perform different tasks, ranging from simple
NOT, OR, AND, etc. to complex combinational functions.

The logic cells in the FPGA are connected by a matrix of wires and pro-
grammable switches. The design is constructed by specifying the logic of the
cells and then closing the switches in the matrix, thus combining the basic
blocks in a way that creates the desired hardware circuit [2]. Different types
of FPGA boards exist with different types of cell-logic architecture. Some
FPGA-types also include memory elements in the collection of logic cells, e.g.
flip-flops. Physically the cells can be implemented as look-up tables or as a
combination of multiplexers and gates. This report will not go into detail
about the inner hardware structure of FPGAs.

Programming an FPGA is done by creating a design resembling the chip to be
implemented. The design of the chip can be described in two different ways.
One way is by drawing a logic circuit diagram that graphically describes how
the design should be implemented. Another approach is to specify the design
by writing source code in a hardware description language. When the design
is complete, it is run through a compiler and transferred to the FPGA-board.
The FPGA used in this project to implement regular expressions is the Altera DE1 board [16]. It is based on a Cyclone II-chip and the design is described in VHDL source code.

2.3 Regular expressions

As stated earlier, Snort provides the possibility of using Perl Compatible Regular Expressions [11] in payload inspection rules. This section provides an introduction to regular expressions in general which leads to a discussion of how finite automata are used to represent regular expressions.

A regular expression can be seen as a pattern that describes a set of multiple strings. The pattern is built by using both symbols (letters, numbers etc.) and operators. By combining symbols and operators it is possible to describe a complex set of multiple strings in a compact notation. The operators used in regular expressions include the alternation operator, the quantifier-family and the grouping operators.

2.3.1 Alternation

The alternation operator | provides a way of alternating between two parts of the expression. This makes it possible for a regular expression to match input strings containing different sub-expressions. The simple regular expression \( b|c \) uses the alternation operator between the two sub-expressions \( b \) and \( c \). This example of a very basic regular expression matches the strings 'b' and 'c'.

2.3.2 Quantifiers

The quantifier-family (\(?\), \(+\), \(*\)) in regular expressions consists of three different operators. The \(?\)-operator indicates that the sub-expression is accepted zero or one time. The \(+\)-operator defines the sub-expression to be present one or more times, and finally the \(*\)-operator indicates that the sub-expression should exist zero or more times. The previous example of a simple regular expression can be extended to include quantifiers:

\[ b^+|c^? \]

This expression matches the strings: \{'b', 'b...b', ∅, 'c'}\}. Note that the expression matches the empty set (\( ∅ \)) due to the sub-expression \( c^? \) that accepts zero or one occurrence of the symbol 'c'.
2.3.3 Grouping

Regular expressions use parenthesis to override operator precedence. By extending the previous example we get

\[ x(b^+|c^2)y, \]

that matches \{'xby', 'xb...by', 'xcy', 'xy'\}. Note the concatenation of the expression and the leading and trailing symbols \(x\) and \(y\).

By using operators on larger sub-expressions it is possible to create quite complex string matching patterns, such as \(x((a|b)^+ b^*c^*)yz^*\). It would be impossible to describe all the strings matched by this pattern without the use of regular expressions due to the presence of the quantifiers.

2.4 Perl Compatible Regular Expressions

The regular expression search patterns used in the payload inspection rules of Snort are described using Perl syntax. Perl Compatible Regular Expressions provide a number of extended functionalities compared to standard regular expressions, such as escape sequences and character classes. This section explains which operators and meta-characters from the PCRE-library that has been implemented in the project.

2.4.1 Basic operators and quantifiers

The basic operators (alternation, quantification and grouping) are all fully supported in the model proposed in this project. The implementation also supports a new member of the quantifier-family; the ranged quantifier. Ranged quantifiers are described by four different notations:

- \{x\} The expression is matched exactly \(x\) times.
- \{x,\} The expression is matched at least \(x\) times.
- \{,x\} The expression is matched at most \(x\) times.
- \{x,y\} The expression is matched at least \(x\) and at most \(y\) times.

The expression \((abc)\{2,4\}\) is matched by the input strings \{'abcabc', 'ab-abcabc', 'abcabcabcabc'\}. 7
2.4 Perl Compatible Regular Expressions

2.4.2 Escape sequences

Escape sequences are identified by the backslash (\) followed by one or more identifiers. Escape characters are used to define either specific special input symbols or predefined character classes. The escape sequence \w identifies a predefined word class that matches the characters \[a - z\], \[A - Z\], \[0 - 9\] and \_ (the underscore). By identifying the class using upper-class characters, the class is negative, e.g. \W matches any input symbol not in the predefined word class. Special unprintable characters such as the line-feed and tab can also be identified using escape sequences.

Predefined classes identified by escape sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\w</td>
<td>Any 'word' character (alphanumeric plus _)</td>
</tr>
<tr>
<td>\W</td>
<td>Any non-'word' character</td>
</tr>
<tr>
<td>\s</td>
<td>Any whitespace character</td>
</tr>
<tr>
<td>\S</td>
<td>Any non-whitespace character</td>
</tr>
<tr>
<td>\d</td>
<td>Any digit character (0-9)</td>
</tr>
<tr>
<td>\D</td>
<td>Any non-digit character</td>
</tr>
</tbody>
</table>

Special unprintable characters identified by escape sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\t</td>
<td>Horizontal tabulator (hex value: 0x09)</td>
</tr>
<tr>
<td>\a</td>
<td>Alarm/Bell (hex value: 0x07)</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return (hex value: 0x0D)</td>
</tr>
<tr>
<td>\n</td>
<td>New line/Line feed (hex value: 0x0A)</td>
</tr>
<tr>
<td>\f</td>
<td>New page/Form feed (hex value: 0x0C)</td>
</tr>
<tr>
<td>\e</td>
<td>Escape key (hex value 0x1B)</td>
</tr>
</tbody>
</table>

Escape sequences using the x identifier is commonly used in the Snort rules. \x defines an escape sequence that matches a specific character based on its hexadecimal value, e.g. \x5C matches the backslash and \x38 matches the ampersand (&).

Finally escape sequences are used to match special characters, which would otherwise be parsed in the expression, e.g. the brackets. By writing a backslash immediately before a bracket, it is matched as a normal input symbol, e.g. \( and \{.

2.4.3 Specific character classes

The predefined character classes that can be described using escape sequences can sometimes prove insufficient. Specific character classes are defined by one
or more characters surrounded by square brackets, e.g. \[abc\] that denotes a classes matching either the input symbol 'a', 'b' or 'c'. The character class only matches one input symbol, unless of course it is used in combination with a quantifier.

In addition to positive classes, negative character classes can also be described. When the first symbol in the class is a \(^\sim\) (the caret) the character class is defined as negative. Negative classes match anything but the expression specified, e.g. \[^{\sim}def\] matches any input symbol except 'd', 'e' and 'f'.

Escape sequences can be included in character classes for additional functionality, e.g. \[d\w] that matches any digit or any word character. This could also have been matched by using the expression \((d\w)\) which is how the character class could be described, using the alternation operator.

The character interval is another feature of character classes and is described by two characters separated by a hyphen, e.g. \[a−z\]. This matches any character that falls within the interval, in this case from lowercase a through lowercase z.

When a character interval uses a combination of upper- and lower-case characters, both cases are matched by the expression, e.g. \[b−X\] that matches characters b through x in both cases. Digit intervals can be specified in a similar fashion, e.g. \[4−8\]. By combining intervals in a class, it is possible to define a matching pattern much more compact than by using the alternation operator, e.g. \[b−h\{3, \ldots, 7\}\] = \((b|c|d|e|f|g|h)(3|4|5|6|7))\).

## 2.5 Finite automata

Finite automata are used in this report to describe how regular expressions can be seen as state machines that can be implemented in hardware. This section presents finite automata, by briefly introducing deterministic finite automatas and then exploring how these are extended into non-deterministic finite automatas that can be used to describe regular expressions. [5]

### 2.5.1 Deterministic finite automata

A deterministic finite automaton (DFA) is a directed graph consisting of nodes and transitions. One node is defined as the starting node and one or more nodes are defined as accepting or final nodes. The transitions between
two nodes are indicated graphically by an arrow labeled by the symbol that invokes the transition. A DFA runs on a string of input symbols, beginning at the starting node. Each input symbol will transfer the current state to another node or itself based on the outgoing transitions from the node that represents the current state. A DFA does not allow two outgoing transitions from the same node to have the same input symbol. Because of this only a single node can represent the current state at any given time. Figure 2.1 shows a DFA accepting all bit sequences containing at least two 1s.

![Figure 2.1 — A DFA accepting all bit sequences containing at least two 1's](image)

### 2.6 Non-deterministic finite automatas

A non-deterministic finite automaton (NFA) is a DFA with extended possibilities. The NFA allows multiple nodes to represent the current state at the same time. This is due to the fact that an NFA can have nodes with the same input symbol on different outgoing transitions. In the following a node is considered active if it represents the current state of the automata. Transitions occurring on no input known as the $\varepsilon$-transition are also introduced, as an extra feature of NFAs.

Figure 2.2 shows an NFA that accepts all bit sequences ending in either '00' or '0110'. Note how the starting node $q_0$ has two transitions that both use the input symbol '0'. When receiving a '0' with the current state being in node $q_0$, the NFA will activate both $q_0$ and $q_1$. When $q_1$ is active $q_3$ is also automatically activated due to $q_1$'s outgoing $\varepsilon$-transition. If the following input symbol is '1' the NFA would activate $q_2$. At this point $q_0$ and $q_2$ would be active, as the previously active $q_3$ node would lose its state, due to the lack of outgoing transitions labeled '1'. Another '1' as input would let $q_2$ activate $q_3$ and finally a '0' would activate the accepting node $q_4$. If the input bit sequence had ended in '010' the NFA would have ended in node $q_2$ as it does not have an outgoing transition labeled '0'. The first '0' would activate $q_1$ and thus $q_3$ due to the $\varepsilon$-transition, but as $q_3$ does not have an outgoing transition labeled '1' it would never end in the accepting node $q_4$. 

10
Table 2.1 — Transition function of NFA shown in Figure 2.2

<table>
<thead>
<tr>
<th>State</th>
<th>ε</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>∅</td>
<td>{q₀, q₁}</td>
<td>q₀</td>
</tr>
<tr>
<td>q₁</td>
<td>q₃</td>
<td>∅</td>
<td>q₂</td>
</tr>
<tr>
<td>q₂</td>
<td>∅</td>
<td>∅</td>
<td>q₃</td>
</tr>
<tr>
<td>q₃</td>
<td>∅</td>
<td>q₄</td>
<td>∅</td>
</tr>
<tr>
<td>*q₄</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

Figure 2.2 — An NFA accepting all bit sequences ending in either 00 or 0110

Formal definition

The formal definition of a non-deterministic finite automaton is described by the notation

\[ A = (Q, \Sigma, \delta, q₀, F) \]

where \( Q \) is the set of states, \( \Sigma \) is the set of input symbols, \( \delta \) is the transition function, \( q₀ \) defines the start state and \( F \) is the set of accepting states. The example shown in Figure 2.2 can be formally described by:

\( (q₀, q₁, q₂, q₃, q₄, \{0, 1\}, \delta, q₀, q₄) \)

The transition function \( \delta \) is defined by the transition table shown in Table 2.1.
2.6.1 Converting regular expressions to NFA

NFAs are introduced in this report as they play an important role in understanding how to implement regular expressions in hardware. A regular expression can be converted to an NFA that accepts the same input strings that are matched by the regular expression. [12] The constructed NFA can then be used to process the input string and perform the matching. In a later section it is described in detail how the NFA is converted to a hardware design that can be programmed into an FPGA device. In this section the focus will be on how a regular expression is converted to an NFA.

An NFA matching a regular expression can be constructed by using a few simple construction rules. The basic principle is to regard each part of the regular expression as a regular expression itself, and then apply the rules to create larger expressions. Figure 2.3 depicts an NFA matching a single input symbol. This is the most basic block in NFAs, and any NFA consists of at least one.

The following section introduces the NFA construction rules used to build NFAs from regular expressions. In the following the term sub-NFA refers to the smaller NFAs used when building larger automaton structures.

![Figure 2.3 — NFA matching a single input symbol](image-url)
Chapter 2: Background

2.6 Non-deterministic finite automatas

Concatenation

Combining a number of regular expressions in a given order is done by using the rule shown as Figure 2.4 subsequently. This rule is called concatenation and allows the construction of linked expressions.

![Figure 2.4 — Concatenation rule connecting two sub-NFAs](image)

Alternation

The OR-operator is used in regular expressions to alternate between different sub-expressions. This is constructed as an NFA by using the rule shown in Figure 2.5, which provides a way of alternating between two sub-NFAs. The resulting NFA will end in the accepting state if either one of the sub-NFAs are matching.

![Figure 2.5 — Construction rule alternating between two sub-NFAs (OR operator)](image)
Quantification

The quantifier-family of regular expressions is constructed as NFAs in three different ways.

The first quantifier we look at is the \(*\)-quantifier. It is constructed by adding an \(\varepsilon\)-transition from the final node of a sub-NFA to its starting node. An additional \(\varepsilon\)-transition is used to bypass the sub-NFA by linking the starting and accepting nodes of the surrounding NFA. This results in an NFA that accepts an arbitrary number of sub-NFA matches. By bypassing the sub-NFA with the \(\varepsilon\)-transition the NFA will also accept zero matches. The NFA construction rule of the \(*\)-quantifier is shown in Figure 2.6.

The \(+\)-quantifier used in regular expressions is related to the \(*\)-quantifier. The \(+\)-quantifier accepts any positive number of matches. The NFA construction rule is similar to that of the \(*\)-quantifier, except for the removal of the \(\varepsilon\)-transition bypassing the sub-NFA as shown in Figure 2.6. The construction rule for the \(+\)-quantifier is shown in Figure 2.7.

The \(?\)-quantifier is the final quantifier used in regular expressions. It accepts either zero or one matches by the sub-expression. The NFA construction rule for this quantifier is constructed as an alternation (Figure 2.5) between the sub-NFA and the empty string. This construction rule can be seen in Figure 2.8.
Combining the rules introduced in this section makes it possible to construct
NFAs matching complex regular expressions. In the following example the
regular expression \((a^*b)(c|d)\) is constructed as an NFA.

**Example of constructing an NFA from a regular expression**
The first step is to create an NFA matching the symbol 'a' using the rule
specified in Figure 2.3. A \(*\)-quantifier is added to that NFA using the rule
shown in Figure 2.6, and this is connected to an NFA matching the symbol
'b', using the rule from Figure 2.4. Two NFAs matching the symbols 'c' and
'd' are constructed based on Figure 2.3 and combined by using the construc-
tion rule shown in Figure 2.5, representing the OR-operator. Finally the two
NFAs are connected by the approach shown in Figure 2.4. The NFA rep-
resenting the complete expression is shown in Figure 2.9 with each segment
marked.

This chapter has provided the background knowledge of Snort, FPGAs,
regular expressions and how these can be converted into NFAs. The mate-
rial covered in this section will be used in the following chapters. With the
conversion of the regular expression to NFA, the expression can be imple-
mented in hardware. This is described in chapter 4, in fact any NFA can be
implemented in hardware.
Much research has already been made in the area of implementing pattern matching processes in hardware. Sidhu and Prasanna [12] presents a method for finding matches to a regular expression using FPGAs, this is done by creating a state machine design of an NFA corresponding to a regular expression. Parts of the approach proposed in this project are based on the work done by Sidhu and Prasanna.

The research performed by Sidhu and Prasanna in 2001 is commonly referred to in other papers written on the subject.


Sutton [14] extends the work by Sidhu and Prasanna to handle processing of bytes in parallel. The Snort Intrusion Filter for TCP (SIFT) introduced by Attig and Lockwood [1] operates as a preprocessor to the NIDS, filtering out acceptable network data ensuring that processing power is not wasted on unauthorized traffic. Cho and Mangione-Smith [3] presents a similar co-processor architecture used to monitor and identify a number of intrusion signatures.

This project proposes a method of automatically generating VHDL source-code from the rules of a standard Snort installation. Some of the research introduced above is referenced in various parts of the report.
4

Implementation

Being able to implement non-deterministic finite automata as an FPGA design is fundamental to implementing regular expressions. In this section an implementation approach is suggested by building logic circuits corresponding to the NFA construction rules shown in figures 2.3 - 2.8. The logic circuits of the FPGA design are described in VHDL source-code. The VHDL source code is designed by specifying a number of basic logical entities each representing one of the NFA construction rules. When each entity is introduced, a reference to graphical representation as well as the actual VHDL source-code is given. The design of some of the structures is based on work done by Sidhu and Prasanna. [12]

Deterministic finite automatas in hardware

As mentioned earlier, DFAs are a group within the area of finite automata, with less functionality than NFAs. The best way of understanding how NFAs can be constructed in hardware, is to first look at how DFAs would be implemented. DFAs can be designed in hardware by using one of two different techniques; using a memory block to store the different states of the DFA or using One-Hot encoding. The memory block approach is not a good option as it is only possible to look at one node in every clock cycle. [1] This would make it hard to extend the functionality to implement the multi-state features of the NFA. One-hot encoding associates a flip-flop with each node, and the current state is identified by the flip-flop storing a 1 at the given time. Each flip-flop is connected to a number of logic circuits that transfers the
1-bit to a new flip-flop in each clock cycle. This is a better implementation approach when seeking to extend DFAs to NFAs.

**Extending DFAs to NFAs**

By using one-hot encoding, it is possible to extend a DFA to an NFA. One of the features of the NFA is the possibility of having the same input symbol on multiple outgoing transitions, e.g. the input symbol 'a' can lead from an active node to a number of different nodes. This NFA functionality makes it possible for the NFA to have a number of nodes active at a given time. This is implemented in hardware by connecting the output of a given flip-flop to a number of other flip-flops instead of just one. By using this approach it is possible to maintain several states concurrently by having the '1' bit from the active flip-flop being transferred to a number of flip-flops. The other feature of the NFA, compared to the DFA, is the \( \varepsilon \)-transition, i.e. a state transition without any input. In hardware this is implemented by connecting the input of a flip-flop to the input of the destination flip-flop, thus bypassing it.

### 4.1 Logic structures

In section 2.6.1 it was shown how NFAs are constructed from regular expressions by using a number of simple basic rules, corresponding to the operators used. The step of constructing a hardware design from an NFA is done in a similar manner, by using a number of basic VHDL-entities corresponding to simple NFAs, and linking them together. In this section the basic entities used in the VHDL source-code are introduced.

In the following sections the term entity refers to a VHDL-entity and the term sub-entity refers to an entity that is connected as a child to a parent-entity at a higher level. A sub-entity can be as simple as matching a single input symbol or the root entity of a large number of interconnected entities.

The general structure of an implemented NFA is quite simple. A top-level entity controls the input to the NFA and the output coming from it. The input signal is used to control when entities should be active. This input signal can be seen as the '1'-bit that identifies which state is the current state. The signal is routed down through the sub-entities that together make up the design of the implemented NFA, ending in the very first entity, representing the starting node. The input signal from the top-level entity is at constant high, as the starting node is always active. The input signal is then routed on through the entities, ending as the output from the NFA design to the
top-level entity, indicating that the pattern has been matched.

### 4.1.1 Comparators

The comparator entity corresponds to an NFA matching a single input symbol as shown in Figure 2.3. This is the basic entity in the hardware design. The input signal to the comparator entity is connected to a flip-flop. The comparator entity is then constructed by combining the output from the flip-flop and the output from a character-comparator as the input to an AND-gate. By doing this it is only when the input signal is high that the comparator entity can produce a high output. The structure of the comparator can be seen in Figure 4.1(a). Source-code that describes a comparator entity is included in this report in appendix B.1. When generating the VHDL source-code for a number of Snort rules, no two comparator entities check for the same input. Comparators are reused, in order to save space in the final generated source-code.

The comparator entity can be designed to compare the input to specific input symbols or to intervals used in character classes. The special escape sequences are also implemented by using the comparator entity, where predefined classes such as \( \text{\textbackslash d} \) are treated as an interval, in this case \([0-9]\). Two different kinds of comparator entities exist; positive and negative. Positive comparator entities provide a high output if the input character-comparator matches the predefined matching data. Negative comparators do the opposite; if the input symbol does not match the predefined pattern the output is high.

### 4.1.2 Concatenation entity

The concatenation entity is used to control the order in which sub-entities are interconnected, and is a hardware implementation of the NFA construction rule shown in Figure 2.4. It is implemented by connecting the output from one entity to the input of the next. This allows the flip-flops representing the different states to forward the ‘1’-bit to the next state, without the need of any logic gates. In the VHDL source-code this is done as a port mapping between the output signal and the input signal. The output signal from the second entity is then used as the output of the concatenation entity. The design of the concatenation entity can be seen as Figure 4.1(b). VHDL source-code implementing the concatenation entity is included in appendix B.2. The word entity is a variation of the concatenation entity, providing a way of interconnecting any number of comparator entities. The concate-
nation entity and the word entity are basically the same. Each comparator is connected from the output pin of one comparator to the input pin of the next comparator and so on. The output from the final comparator is then mapped to the output of the word entity.

### 4.1.3 OR-entity

The OR-entity is an implementation of the alternation operator used in regular expressions and the NFA construction rule shown in Figure 2.5. The input to the OR-entity is connected to the input of two separate sub-entities. The output signals from the sub-entities are then connected to a logical OR-gate that produces a high output, if any of the sub-entities are matched. A logical structure describing the design of the OR-entity can be seen in Figure 4.1(c), and the source-code is included in appendix B.3.

### 4.1.4 AND-entity

The AND-entity is a somewhat special case. It is not an operator used in regular expressions, as a given input symbol cannot have two different values, and it is not possible to define as an NFA. The design of the AND-entity is similar to that of the OR-entity. The input to the entity is connected to the input of two separate sub-entities. The output signals from the two sub-entities are connected to an AND-gate, providing a high output if both sub-entities are providing a high output. Figure 4.1(d) shows a logical structure representing the AND-entity, and the source code is included in appendix B.4.

### 4.1.5 Quantifiers

Each member of the quantifier-family (\(*\, +\, \? \,\) and \{x,y\}) each have their own logical structure. The quantifiers are connected to a sub-entity that contains the sub-expression being matched, and produces an output based on the quantification operator. The structure of each quantifier is described in the following sections.

**The \(*\)-quantifier**

The \(*\)-quantifier provides a high output on an arbitrary number of matches of the sub-entity it is connected to. The implementation corresponds to the NFA rule shown previously in Figure 2.6. The logical structure consists of two OR-gates that both are connected to the sub-entity. The input signal to
Chapter 4: Implementation 4.2 Top-level structure

the quantifier entity and the output signal from the sub-entity are combined in both OR-gates. The output from the second OR-gate is then connected to the output of the quantifier entity. Figure 4.1(e) shows the logical structure of the *-quantifier. The VHDL source-code for the *-quantifier is included in appendix B.5.

The + -quantifier

The output from the + -quantifier is high on a positive number of matches, of the sub-entity it is connected to. It is related to the *-quantifier, as it can be regarded as a sub-entity connected to a *-quantifier that is connected to an identical sub-entity. The NFA rule is shown in 2.7 and its hardware implementation is shown in Figure 4.1(f). The VHDL source-code for the + -quantifier is included in appendix B.6.

The ? -quantifier

The ? -quantifier returns a 1-bit on exactly zero or one matches of the sub-entity it is connected to. It is an implementation of the NFA-rule shown in Figure 2.8, and is implemented by using one logical OR-gate. The input to the entity is connected to both the input signal of the sub-entity and to the OR-gate. The output from the sub-entity is also connected to the OR-gate, and the output from the OR-gate is then connected as the output of the quantifier entity. The logical structure of the entity is shown in Figure 4.1(g). The VHDL source-code for the ? -quantifier is included in appendix B.7.

The ranged quantifier structure

The output signal from the ranged quantifier is high if the sub-entity it is connected to, is matched a certain number of times that falls into a predefined range. The logical structure of the ranged quantifier is much more complex than that of the rest of the quantifier family. As it differs significantly from the entities it is described thoroughly in the next chapter.

4.2 Top-level structure

This entity is the top-level control component that every implemented NFA is connected to. It contains a register storing the payload data that is passed on to each NFA at one character in each clock cycle. The top-level entity provides each NFA with the input signals; i, clk, reset, halt and inputChar.
4.2 Top-level structure

The \textit{i} signal is the input signal being routed down through the NFA ending up in the comparator checking for the very first input symbol in the expression. The \textit{clk} signal is the signal produced by the onboard clock, and is used by the entity to pass on the input signal each time the clock goes high. The \textit{reset} signal is used to initialize each of the entities to its default state. The \textit{halt} signal is used only by the ranged quantifiers to know when the last input symbol has been sent. The \textit{clk}, \textit{reset} and \textit{halt} signals have been omitted from the descriptions of the logical structures as well as the figures, to simplify the understanding of the design. The \textit{inputChar} signal is a bit-vector containing the binary representation of the input symbol, and is only read by the comparators.

The \textit{sendChar} process of the top-level entity controls the signals directly related to the input data, i.e. \textit{i}, \textit{inputChar} and \textit{halt}. The top-level logical structure is shown in Figure 4.2, and the VHDL source-code is included in appendix B.9.

Figure 4.3 shows a hardware implementation of the regular expression \((ab)^*cd\), using the entities introduced in this chapter. Dashed lines indicate each of the comparators, as well as which sub-entities each operator refers to. The input and output signal in the figure is connected to the top-level entity shown in Figure 4.2, which has been omitted from this figure. The large arrows in the bottom of the figure indicate the input symbols sent to the comparators.
Figure 4.1 — Logical structures corresponding to NFA construction rules
Figure 4.2 — Logical structure showing the design of the top-level entity and connections to the sub-entities.
Figure 4.3 — Logical structure showing the design of the top-level entity and connections to the sub-entities
4.2 Top-level structure
The ranged quantifier is different from the other members of the quantifier family, as the number of accepted matches is defined by a range. As described in section 2.4.1, the quantifier can be defined as open or exact. The output from the entity is high, as long as the current number of counted matches falls within the predefined range.

However, counting the number of matches, poses a potential problem that is not applicable to the basic quantifiers; the counter has to be reset if the input data does not match the sub-expression being counted. This is explained in the following example.

**Example of input data to the ranged quantifier**

The simple regular expression \((ab)^4\) involves the use of a ranged quantifier. It defines that the substring ‘ab’ has to be matched exactly four times.

The regular expression matches the string ‘abababab’. Figure 5.1 shows the input signal (coming from the sub-entity) being counted on each input symbol. In each clock cycle, where the incoming signal is high, the counter is incremented.

The problem with the ranged quantifier rises on input strings such as ‘ababaabab’, where the extra a causes the input data not to match the ex-
Chapter 5: The ranged quantifier

5.1 Logical structure

The ranged quantifier entity consist of connections to two separate sub-entities as well as an OR-gate, two AND-gates and a counter keeping track of the number of matches found so far. The first sub-entity is an implementation of the sub-expression found so far. This sub-entity provides a high output whenever it is matched. The second sub-entity is a negative representation of the same expression. This secondary sub-entity is described by...
Chapter 5: The ranged quantifier  

5.2 The kill entities

A number of issues have to be taken into consideration, when defining the reset-functionality of the ranged quantifier. Simply inverting the output signal from the primary sub-entity is unfortunately not a liable solution for the kill signal. Resetting the counter when the primary sub-entity’s output signal is low is only correct if the pattern matched was just a single input symbol. Another approach is to reset the counter when all the comparators used in the pattern has low output at the same time. This would lead to correctly resetting the counter when an input symbol not used in the expression is parsed. However, this approach does not take into consideration the scenario where the first character of the pattern appears at the wrong time.

Using the previous example, the input string 'ababaabab', would never lead to all comparators’ output being low at the same time. The input signal to the comparator for the character 'a' is always high, because it is the first comparator of the word entity. This leads to the introduction of the kill signal.

Figure 5.3 — Correct counter incrementing in a ranged quantifier by using the kill signal

the term kill entity, and the output signal coming from it as kill signal in the remainder of the report.

The counter is implemented by a register storing a value that is incremented in each clock cycle, where the input signal coming from the matching sub-entity is high. The output signal from the ranged quantifier entity itself is high whenever the current number of counted matches falls within the pre-defined range.

The logical structure of the ranged quantifier is shown in Figure 5.1. The VHDL source-code for the ranged quantifier is included in appendix B.8.
5.2 The kill entities

**Definition:** The kill signal is high only when the input provided has made the expression unmatchable.

The definition of the kill signal states that the signal is high when the implemented regular expression has received input data that does not match the sub-pattern being counted. This provides a reset of the counter giving the quantifier a chance to be matched on a later series of input symbols.

Each time the counter in the ranged quantifier is incremented, the input signal going to the kill entities is set to high. It is not necessary to start the kill entity before the expression has been matched at least once. The construction of the entities that together makes up the kill entity is explained in detail below.
5.2.1 Constructing the kill entity

As stated above the kill entity is the negative representation of the expression that is being counted by the ranged quantifier. Because of this we have to define a negative representation of the entities that implements the sub-expression. A number of logic rules are used to do this and these are introduced in the following. An example of how the rules are used to generate the kill entities is shown in section 5.2.2.

This section introduces three basic rules that shows how to negate the basic entities; concatenation, OR and AND. In the following the symbols a and b represent two separate parts of a regular expression.

Negative representation of the concatenation entity

A concatenation of two entities is defined as the first element a occurring before the second element b. This leads to rule 5.1 that states that the negative version of a concatenation has occurred if either the first element is not a or if the first element a is followed by an element that is not b. The rule is implemented by using an OR-entity to alternate between the sub-entities.

\[ \neg(ab) = \neg a \lor \neg b \]  

(5.1)

Combining more than one element such as in word entities leads to a recursive building of the negative expression:

\[ \neg(abc) = \neg a \lor \neg b \lor \neg c \]

(5.2)

Negative representation of the OR-entity

An alternation between two entities is implemented by the OR-entity. The alternation is defined as either a or b has occurred. The negative representation of this is that neither a nor b has occurred. This leads to rule 5.3, where a negated OR-entity is represented by an AND-entity, as both operands have to be negative in order for the OR-entity to fail.

\[ \neg(a|b) = \neg a \land \neg b \]

(5.3)

Negative representation of the AND-entity

The negative version of the AND-entity is the opposite of the OR-entity. Rule 5.4 states that the negative version of a and b both occurring, is that either a or b does not occur.

\[ \neg(a \land b) = \neg a \lor \neg b \]

(5.4)
5.2 The kill entities

In formal logic, the negation rules 5.3 and 5.4 used on the OR-entity and the AND-entity are known as De Morgan’s laws. [10]

Negative representation of quantifiers

These simple rules are the foundation when constructing the kill entity. Aside from the basic operators we have to define negative equivalents for the quantifiers (¿, ¿ and ¿) in order to implement functionality from the PCRE-library that is supported. Defining a negative signal for a quantified expression requires a bit more work. If we look at an expression such as ¿ it is not possible to define a negative signal that follows the definition of the kill signal, i.e. the kill signal is high when it is impossible for the expression to match. As the ¿-quantifier matches an arbitrary number of a’s it is both matched and not matched at any time, regardless of the specified range defined in the ranged quantifier. Because of this, it is not possible to design a kill signal for an expression that ends in a quantifier itself. Given the definition of the kill signal, it would require knowledge of the sub-expression that follows the ranged quantifier.

Given the statement above, it can be concluded that specifying kill signals for ab¿, ab¿ ab¿ is not possible, while specifying kill signals for ¿b, ¿b and ¿b is. Likewise it does not make sense to talk about kill signals for expressions such as a¿|b, as this would lead to having to define a kill signal for ¿. Regular expressions using nested ranged quantifiers e.g. (ab{4, }cde){6, 18} are not supported in this implementation.

Specifying kill signals for expressions such as ¿b is possible and will be explained in the following three rules as each quantifier has its own negative equivalent.

As the expression ¿b is matched by an arbitrary number of ’a’s followed by a single ’b’, the negative expression can be defined as an arbitrary number of a’s followed by an element that is neither ’a’ nor ’b’. The final element has to be checked for ’a’, due to the quantifier.

\[ -(a^*b) = a^*(-a \land -b) \]  \hspace{1cm} (5.5)

The expression ¿b is matched by any positive number of ’a’s followed by a single ’b’. The negative expression is matched if the first element is not an a, or if the first element is a positive number of ’a’s followed by an element that is neither an ’a’ nor ’b’. Once again, the final element has to be checked
for 'a', due to the quantifier.

\[ \neg(a^+b) = \neg a|a^+\neg a \land b \]  \hspace{2cm} (5.6)

The expression \(a^2b\) is matched by either zero or one occurrences of 'a', followed by a single 'b'. The negative expression is matched if the first element is neither 'a' nor 'b', or if the first element is an 'a' and the second is not a 'b'. Because of the quantifier, the first element has to be checked for both 'a' and 'b'.

\[ \neg(a^2b) = (\neg a \land \neg b)|(a\neg b) \]  \hspace{2cm} (5.7)

This concludes the six rules used to construct the kill entity for the ranged quantifier. To generate a kill entity, the matched expression is parsed from right to left, using the rules recursively on the sub-expressions. In the following section an example of constructing a kill entity is given. By using a tree-like structure to graphically represent the entities, it is easier to understand how the kill entity is constructed by being able to refer to specific nodes in the tree. The tree-structure approach is also used explicitly in the software implementation of the source-code generation in chapter 6.

5.2.2 Example of kill entity construction

The regular expression \(((a^*b)(c|d))\{3,5\}\) includes a ranged quantifier and its tree representation is shown in Figure 5.5. The sub-entity of the ranged quantifier tree-node (ID: 9) is a concatenation node (ID: 8). Neither of the concatenation node’s sub-nodes are quantifiers. This means that rule 5.1 can be used.

**Step 1**

Applying rule 5.1 to the expression results in the expression \(\neg(a^*b)|(a^*b)\neg(c|d))\). To build this expression an OR-node has to be created. The first sub-node of the OR-node i.e. the left hand side of the operator corresponding to \(\neg(a^*b)\), is then created by applying the kill-generation routine recursively on that expression. Figure 5.6 shows how the tree is divided into two parts, while the first step is being performed.

**Step 2**

To generate the negative expression shown on the left hand side of the tree in Figure 5.6, rule 5.5 \(\neg(a^*b) = a^*|\neg a \land \neg b\) is needed. This is constructed by adding a concatenation-node (ID: 11) with \(a^*\) as its first sub-node and \((\neg a \land \neg b)\) as it’s second. The \(a^*\) is constructed by having a comparator (ID: 13) for the input symbol 'a' inserted as a subnode to a node representing
5.2 The kill entities

Chapter 5: The ranged quantifier

Figure 5.5 — Tree representation of the regular expression \((a^*b)(c|d)\{3,5\}\)

the \(^*-\)quantifier (ID: 12). This is inserted as the first sub-node to the concatenation node. Negative comparators are created for characters 'a' and 'b' (ID: 15 and 16), and inserted into an AND-node (ID: 14), which is then also inserted into the concatenation node. The concatenation node (ID: 11), now appears as the tree shown in Figure 5.7. It is then inserted as the left hand sub-node of the OR-node (ID: 10) from step 1 shown in Figure 5.6.

Step 3
The right hand side of the OR-node from step 1 is the expression \((a^*b)(c|d)\)\(^{\neg}\), that consists of a two separate expressions put together by a concatenation node (ID: 17). The first part \((a^*b)\) is not negative and can be copied from the original tree (ID: 1, 2, 3 and 6). The other part \((c|d)\)\(^{\neg}\) can be constructed by using rule 5.3. This is represented by creating an AND-node (ID: 18) and inserting two negative comparators for the characters 'c' and 'd' (ID: 19 and 20). The small tree is then inserted into the concatenation node (ID: 17). The tree resulting from step 3 is shown in Figure 5.8.
Chapter 5: The ranged quantifier

5.2 The kill entities

Figure 5.6 — Tree showing the first step of the kill entity generation process

Figure 5.7 — Tree representation of the expression \( \neg (a^* b) \)
Figure 5.8 — Tree representation of the expression \((a\cdot b) \neg (c|d)\)
Step 4

The final step is putting together the trees shown in Figure 5.7 and Figure 5.8, by adding them as sub-nodes to the OR-node (ID: 10) created in step 1. We now have constructed the complete kill entity to the ranged quantifier. The corresponding negative representation of the original expression is defined as \((a^* (\neg a \land \neg b)) \mid ((a^*b) (\neg c \land \neg d))\). Figure 5.9 shows the complete kill entity that is used to determine when the counter of the ranged quantifier should be reset due to a mis-match of the expression.

This concludes the explanation of the ranged quantifier. In the next chapter the software that has been developed during the project is introduced.

Figure 5.9 — The complete kill entity design to the expression \(((a^*b)(c|d))\{3, 5\}\)
5.2 The kill entities

Chapter 5: The ranged quantifier
The project also includes the development of software that automatically generates VHDL source-code for FPGA programming, based on the rule-set of a Snort installation. The design of the program and the issues that had to be addressed are discussed in this chapter.

6.1 Software description

The software called snort2vhdl has been developed in Java, using the Netbeans IDE. The Java source-code is not included in this report, it is however fully available along with a jar-file on the enclosed CD-ROM as well as online at http://www.mitspeciale.dk

Aside from the Snort rule-set the program uses two external files in the generation process. The payload-file contains the input data to be parsed by the implemented PCREs. The program uses the data in the file when creating the VHDL source-code for the top-level entity, controlling the parallel PCREs. In the filter-file the user can specify a list of filters that is used to avoid certain PCREs. If a PCRE contains the filter as a sub-string, the PCRE is not implemented.
6.2 Automatic VHDL source-code generation

The automatic source-code generation can be divided into seven individual steps. The steps are listed below, and each step is described in detail in this section.

1. PCREs are extracted from the Snort rule-set
2. Concatenation operators are automatically inserted into each PCRE
3. PCREs are converted from infix notation to postfix notation
4. A syntax tree is recursively built from the PCRE
5. The syntax tree is examined for ranged quantifiers and kill entities are built if necessary
6. VHDL source-code is generated for each PCRE
7. The top-level entity controlling the parallel PCREs is created

The first step extracts the PCREs from the Snort rule-set. Steps two through five are used to process the individual PCREs in a loop. The final two steps are used to generate the actual VHDL source-code.

6.2.1 PCRE extraction

The complete snort rule-set is divided into a number of files all ending with the .rules extension. Each file represent a specific category of rules e.g. ftp.rules, ddos.rules and virus.rules, just to name a few. These files are all located in the \rules directory of the Snort installation. The software scans each file in this directory line-by-line for rules that uses PCREs for payload inspection. When an expression is found a filtering process begins. The filters are used to identify PCRE-functionalities not supported in this implementation. The extracted PCREs that has not been filtered out, is stored in an ArrayList for further processing.

6.2.2 Insertion of concatenation operators

After extracting the PCREs, each expression is processed one at a time. The first step on the road to hardware implementation is to automatically insert concatenation operators in the string of the PCRE. These operators are used to identify where the NFA-rule shown in Figure 2.4 should be used and eventually where the VHDL concatenation entity shown in Figure 4.1(b) should
be constructed in a later process. The concatenation operator is identified by the symbol $\gg$.

The concatenation function loops through the input string containing the PCRE one character at a time, using a stack to build the resulting string. The automatic concatenation function is designed as a two-step process, in order to simplify the function. The first step is adding concatenation identifiers, which is followed by a step that removes redundant identifiers.

The concatenation operator $\gg$ is added to the string on four occasions:

- Before a start parenthesis, e.g. $\gg($
- After an end parenthesis, e.g. $)\gg$
- Before an expression using a quantifier, and after the quantifier itself, e.g. $\gg a^* \gg$
- Before and after a character class, e.g. $\gg[a,b] \gg$

The next step is a trimming process that removes unnecessary concatenation entities, by using the following rules.

- Never after a start parenthesis, e.g. $)$
- Never before an end parenthesis, e.g. $\gg)$
- Never before a quantifier, e.g. $\gg\{x,y\}$ or $\gg?$
- Never before or after an alternation operator $|$, e.g. $\gg| \gg$ or $|\gg$

An example of the automatic concatenation process is given here:

Parsing the regular expression

$abc\ast de|f$

provides the output string

$ab \gg c^* \gg d \gg e|f$
6.2.3 Infix to postfix conversion

When each PCRE has been run through the automatic concatenation parser, the next step is to convert the notation of the expression from infix to postfix. Infix notation is the notation normally used when writing any type of expression. It means that the operator is written between the operands it is referring to, e.g. the regular expression \(a|b\), that matches strings 'a' or 'b'. In postfix notation the operator is written after the operands. The same example would then be written as \(ab\) in postfix notation.

Converting to postfix notation is introduced to simplify constructing the entities. By converting to postfix notation it is possible to eliminate the use of parenthesis, e.g. \(a \gg (b|c)^*\) as the operator | refers to the previous two elements and the \(^\ast\)-quantifier refers to the previous element \((b|c)\).

The operators are either unary or binary. Unary operators only refer to one element, this applies to operators such as the quantifiers (\(?\), \(*\), \(\dagger\) and \(\{x,y\}\)). Binary operators refers to two elements, and this applies to operators such as \(\&\) and the concatenation (\(\gg\)).

The function performs another task along with the postfix conversion. During the same process word identifiers are inserted into the expression. Word identifiers are represented by the symbol \(\text{ââ}'\) (the currency symbol), and surrounds a sequence of characters that is matched in the arranged order. Using the word identifiers, the actual result from the conversion process given the previous example is

\[a \gg (b|c)^* = \text{ââ}'a\text{ââ}'\text{ââ}'b\text{ââ}'\text{ââ}'c\text{ââ}'\text{ââ}'\]

Word identifiers are introduced for saving space in the generated VHDL source-code, so that each two characters do not require the presence of a concatenation entity. A word entity can have any number of comparators connected to it and not just two as the concatenation entity. Word entities connected to just one character comparator are however removed later in the process to save additional space.

The conversion process runs from the beginning of the string, one character at a time. When a character is found, the process continues until an operator is found. This defines the beginning and the end of a word. The characters in the word are written to the output string, surrounded by word identifiers. However, when an operator is found things are a bit more complex, that requires the use of an operator stack. If the operator found is a
Chapter 6: Software  

6.2 Automatic VHDL source-code generation

start parenthesis or a concatenation operator it is pushed to the stack. If it
is an end parenthesis, the stack is popped and written to the postfix string
until the top of the stack contains a start parenthesis, which is thrown away.
If the operator is a unary operator such as the quantifiers or a character class
the operator is written directly to the output postfix string. OR-operators
refer to the entire previous expression, so when an OR-operator is found the
elements are moved from the stack to the output string until either another
OR-operator or a parenthesis is found. The final operator is left on the stack.
When all characters in the input string have been parsed, the remaining op-
erators on the stack are pop’ed and added to the string. The conversion is
now completed and the postfix string is returned from the function.

Example showing infix to postfix conversion
In the following example the expression \((abc|def) \gg g^*\) is provided as the
input. The postfix string is built in a number of steps shown in Figure
6.1. The iteration number refers to the loop that runs through the input
string, one character at a time. The example shows how the infix expression
\((abc|def) \gg g^*\) is converted to the postfix expression \(\Box abc\Box\Box def\Box\Box g\Box g^* \gg\),
where the \(\Box\) identifies a word sequence.
### Iteration 1
Input: '('  
Stack = {()}  
Postfix = ''  
Operator is pushed to stack.

### Iterations 2-4
Input: 'a, b, c'  
Stack = {()}  
Postfix = 'abc'  
Word identifiers are wrapped around string and written to the output.

### Iteration 5
Input: '|'  
Stack = {(),()}  
Postfix = 'abc'  
Or-operator is pushed to stack.

### Iterations 6-8
Input: 'd, e, f'  
Stack = {(),()}  
Postfix = 'abcdef'  
Word identifiers are wrapped around string and written to the output string.

### Iteration 9
Input: ')'  
Stack = {()}  
Postfix = 'abc'  
Operators are moved from the stack and written to output until top of stack contains (, which is thrown away.

### Iteration 10
Input: 'g'  
Stack = {()}  
Postfix = 'g'  
Operator is pushed to the stack.

### Iteration 11
Input: '*'  
Stack = {()}  
Postfix = 'g'  
Unary operators are written directly to the output.

### Iteration 12
Input: ''  
Stack = {}  
Postfix = 'g'  
The resulting output of the function: 'abc|def|g|'

---

**Figure 6.1** — Each iteration in the postfix to infix conversion process of the expression: \((abc|def) \gg g\)
6.2.4 Constructing the syntax tree

With the regular expression on the postfix notation form, a syntax tree representing it can be created. The tree-structure approach was introduced when describing the generation of the kill entity for the ranged quantifier, and the approach is also used in the actual software implementation. By describing the regular expression as a tree it is easier to generate the VHDL source-code, as the VHDL entities are connected in a tree-like fashion.

The tree structure is implemented in Java as the class regexNode, which contains variables storing type, ID and data as well as an ArrayList of its sub-nodes. The type variable identifies which kind of entity the node represents. The full list of different types is shown in Table 6.1. Trees are constructed by creating objects of the regexNode class and linking the objects to the object representing the parent node, by inserting the object in the sub-node ArrayList of the parent object. In the remainder of the report, the term node refers to an object of the regexNode class. The procedure for constructing a syntax tree is somewhat similar to that of the postfix conversion process. The postfix notation expression is read character by character and uses a stack to control the order of the nodes in the tree.
Word identifiers

If a word identifier is found, a node of the type word is created along with a comparator for each symbol in the word. The comparator nodes are inserted into the ArrayList of the word-node, creating the link between the nodes. The word node is then pushed to the tree stack, which is used to control the build process. If the word consists of just one character the comparator-node is pushed to the stack instead of the word node. This is done to save space in the final VHDL source-code.

Quantifiers

Quantifier operators refer to the previous element, so when a quantifier is found, the item at the top of the stack is popped. By using the stack it is always possible to refer to the entire previous element, which can be a simple comparator or a complex expression. A quantifier node is created and the node from the stack is inserted as a sub-node in the ArrayList of the quantifier node. The quantifier node, now containing the previous expression as a sub-node, is pushed to the stack so it can be referenced as the previous element for other operators.

Character classes

If a character class is found the situation is more complex. As introduced in section 2.4.3 character classes defines a group of characters and can be either positive or negative. An example of a positive character is \([a - z0 - 9]\). This class is treated as an alternation between each element in the class \((a - z|0 - 9|_.)\), meaning that the character class matches any character in one of the two intervals or the underscore character. This is implemented by creating a comparator node for each element (interval or symbol). When running through the character class each comparator node is pushed to a different stack; the class stack. With all comparators on the class stack it is time to build the actual class node. This is done in a loop that runs while the class stack is not empty. If the class is a positive character class the node combining the comparators is of the type OR. If it is a negative character class such as \([a - f0 - 9]\), it is built with negative comparators using a node of the type AND to combine the comparators. Figure 6.2 shows a tree-structure representing the positive character class \((a - z|0 - 9|_.)\). The top-level OR-entity containing the tree is finally pushed to the tree stack.
Chapter 6: Software  6.2 Automatic VHDL source-code generation

Figure 6.2 — Tree representing a positive character class

Concatenation/OR

Both the concatenation- and the OR-entities contains two sub-entities. Due to the use of the tree stack the two nodes to be inserted as sub-nodes are present at the top of the stack when the concatenation identifier or the OR-operator is read from the postfix input string. A node representing the entity is created and the two nodes are removed from the stack and inserted as sub-nodes. The combining node is then pushed to the top of the stack.

By parsing the input expression one character at a time and using the stack, the final result will be just one node on the stack. This node will be the top node of the complete tree representing the regular expression. By inserting nodes as sub-nodes to the parent node, the tree is linked together, and it is possible to traverse the tree during the generation of the VHDL source-code.

Example of tree-generation
Given the infix expression $a^* (b|c)$, the auto-concatenated expression is written as $a^* \gg (b|c)$. By converting the expression to postfix notation, the result is $a^*bc \gg$. The tree built from this expression is shown in Figure 6.3.

6.2.5 Checking for ranged quantifiers

The next step of the source-code generation is building the kill entities to the ranged quantifiers. The tree-structure is traversed checking each node for being of the ranged quantifier type. If this is the case the kill entities are constructed by calling a recursive function on the sub-node of the ranged quantifier node. The function builds the kill entity, and inserts it as sub-node
6.2 Automatic VHDL source-code generation

Figure 6.3 — Tree representing the regular expression $a^{*} \supseteq (b|c)$

to the ranged quantifier. A thorough example of how the kill-generation is implemented, was shown in section 5.2.2, and is omitted here. The resulting tree, i.e. the top-node is added to an ArrayList storing all the processed PCREs.

6.2.6 Generating VHDL source-code from syntax tree

The kill entity generation function was the final step of the loop parsing the PCREs individually. The next step is generating VHDL source-code from the processed PCREs. This is done by calling the function convertTree2VHDL on each of the elements in the ArrayList. This function loops through the items in the ArrayList, processing them individually. A StringBuffer storing the VHDL source-code for each PCRE is created by traversing the tree, calling the node2vhdl function on each individual node. The different entities are designed as VHDL source-code as introduced in chapter 4. The node2vhdl function adds the source-code of a given entity type to the StringBuffer, while correctly port-mapping the signals to the correct sub-entities. The VHDL source-code of each PCRE is written to the output file before processing the next PCRE.

6.2.7 Generating VHDL top-level entity

The final task of snort2vhdl is constructing the top-level VHDL-entity that provides the input data to the implemented PCREs. This is done by adding a pre-defined VHDL-entity, containing the functions used. Signals are dynamically connected to each of the implemented PCREs to provide an output telling the status of each. The content of the payload file is stored in the
top-level entity, as it contains the data to be parsed as the input symbols. The top-level entity is written to the output file, and statistics of the implementation is provided to the user.

This concludes the description of the automatic generation of VHDL source-code by snort2vhdl. Appendix A explains how snort2vhdl is executed.
This section presents an evaluation of the designs generated by snort2vhdl. As stated earlier a number of different functionalities from the PCRE-library are not supported. In this evaluation, a filter file has been created to skip the PCREs using unimplemented functionality.

The payload data used to evaluate the design approach proposed in the project is user-generated. Although this is not as realistic as a real-time scenario, it will still be able to prove the design generated contains PCREs that correctly matches their respective patterns.

The generated designs will be evaluated with focus on two aspects, functionality and potential.

**Functionality**

The functionality of the generated designs is vital to the project. It is proven by implementing a number of PCREs, and determining if they correctly match input data described by the expression. The functionality test, also evaluates if the implemented PCREs are arranged in a parallel setup thus providing a better performance than a single thread setup.
7.1 Functionality

Potential

All FPGA boards consist of a finite number of logic elements available for programming. This part of the evaluation will look at how many PCREs it is possible to implement on the Altera board provided.

7.1 Functionality

Validating the generated designs is a very important part of the project. Executing snort2vhdl on the full rule-set of Snort results in the generation of a design consisting of over 5000 PCREs arranged in parallel. Proving that each PCRE from the rule-set is implemented correctly would be an enormous task. Instead 10 different PCREs have been selected from the total rule-set and will be tested in this section. The PCREs have been chosen based on the operators used in the expression. Together the 10 selected PCREs make up a small rule-set that uses all the operators from the PCRE library that is supported. This provides a test case that is able to validate the full functionality of the PCRE implementation model proposed.

7.1.1 Snort rules used in the software evaluation

In the following the Snort rules that have been selected for the evaluation are presented, along with the matching payload data that is used to test the functionality of the PCRE.
Chapter 7: Evaluation

7.1 Functionality

Extracted Snort rule #0:

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule set</th>
<th>PCRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SPYWARE-PUT</td>
<td>pcre: &quot;Referer\x3A[^\r\n]*http\x3A\x2F\x2Fmysearch\dropspam\com\index.php?tpid=&quot;</td>
</tr>
</tbody>
</table>

Description: This rule is used to identify network traffic relating to the dropspam software that is identified as spyware by Snort. The rule inspects the Referer data field of the HTTP-packets.

Operators used: The PCRE defined in this rule uses the negative character class (\[^\r\n\]) and the *-quantifier in combination. The expression also uses different special hex-characters, e.g. \x3A (:) and \2F (.).


Extracted Snort rule #1:

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule set</th>
<th>PCRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WEB-CLIENT</td>
<td>pcre: &quot;/&lt;OBJECT \s+[^&gt;]<em>classid\s</em>=\s*[\x22\x27]\s*\clsid\s*:\x7B?\s*47C6C527-6204-4F91-849D-66E234DEE015?/&quot;</td>
</tr>
</tbody>
</table>

Description: This rule is used to identify ActiveX object access from the 3rd party browser-toolbar Search Assistant.

Operators used: The PCRE defined in this rule uses the both the negative character class (\[^>\]) and the positive character class ([\x22\x27]), both in combination with a quantifier (* and ?). The expression also uses different special characters such as \s (whitespace) and hex-characters.

Matching payload: <OBJECT classid = ' clsid:47C6C527-6204-4F91-849D-66E234DEE015

Extracted Snort rule #2:

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule set</th>
<th>PCRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SPYWARE-PUT</td>
<td>pcre: &quot;tid\x3D{([0-9A-z]+-){4}[0-9A-z]+}&quot;;</td>
</tr>
</tbody>
</table>

Description: This rule is used to identify traffic from the the 3rd party browser-toolbar isearch. It is a hijacking application used to direct users to the isearch.com domain.

Operators used: The PCRE defined in this rule uses positive character classes containing character intervals (A-z and 0-9) both in combination with the +-quantifier. The ranged quantifier (4) is also used in the expression.

Matching payload: tid={0-AA-1-BB-C}

Extracted Snort rule #3:

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule set</th>
<th>PCRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>POLICY</td>
<td>pcre: &quot;User-Agent\x3A[^\r\n]+Google\x3A/&quot;</td>
</tr>
</tbody>
</table>

Description: This rule is used to identify traffic originating from the Google Desktop search program.

Operators used: The PCRE uses the the negative character class (\[^\r\n\]) in combination with the +-quantifier.

Matching payload: User-Agent: Google Desktop

Extracted Snort rule #4:

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule set</th>
<th>PCRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>EXPLOIT</td>
<td>pcre: &quot;Entry[/file/[0-9].[/]&quot;, ^\x0a\annotate\x0a&quot;;</td>
</tr>
</tbody>
</table>

Description: This rule identifies a CVS rsh annotate revision overflow attempt.

Operators used: The PCRE defined in this rule uses positive character classes containing character intervals (0-9) and the . (period). The character class is tied to an open end ranged quantifier ([\71]). The special character \x0a corresponds to the newline character, which is indicated in the matching payload as \<lf>.

Matching payload: Entry /file/123456789123456789123456578912345657892000123456789123456578920001234 657980123456//xy<lf> annotate<lf>

Figure 7.1 — Rules 0-4 selected from the Snort ruleset
### 7.1 Functionality

#### Chapter 7: Evaluation

**Extracted Snort rule #5:**

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule-set</th>
<th>PCRE</th>
<th>Description</th>
<th>Operators used</th>
<th>Matching payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>WEB-MISC</td>
<td>pcre: &quot;/Content-Length\s*:</td>
<td>This rule identifies attempts to specify a negative Content-Length of an HTTP-packet.</td>
<td>The PCRE defined in this rule uses the OR-operator (</td>
<td>), which in this case allows alternation between \s* and \s*\r\n\s+. All three simple quantifiers are used in this expression.</td>
</tr>
</tbody>
</table>

**Extracted Snort rule #6:**

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule-set</th>
<th>PCRE</th>
<th>Description</th>
<th>Operators used</th>
<th>Matching payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>DOS</td>
<td>pcre: &quot;/Cache-Control\s*:</td>
<td>This rule identifies denial of service attempts on the Apache server using mod_cache.</td>
<td>The PCRE defined in this rule uses the OR-operator (</td>
<td>) multiple times. The *- and +-quantifier are also used.</td>
</tr>
</tbody>
</table>

**Extracted Snort rule #7:**

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule-set</th>
<th>PCRE</th>
<th>Description</th>
<th>Operators used</th>
<th>Matching payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SPECIFIC-</td>
<td>pcre: &quot;/Authorization\s*:</td>
<td>Microsoft SPNEGO ASN.1 library heap corruption overflow attempt</td>
<td>The PCRE defined in this rule uses the OR-operator (</td>
<td>), to allow alternation between YE4G.{40}LgMc and YIIQ.{40}QUFB. Special hex-characters and the \s (whitespace) indicator are also used.</td>
</tr>
</tbody>
</table>

**Extracted Snort rule #8:**

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule-set</th>
<th>PCRE</th>
<th>Description</th>
<th>Operators used</th>
<th>Matching payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>VOIP-SIP</td>
<td>pcre: &quot;/Remote-Party-ID\s*:</td>
<td>This rule is used to identify Cisco 7940/7960 INVITE Remote-Party-ID denial of service attempt.</td>
<td>This rule uses the ranged quantifier with both high and low specified ([1,3]). The \d operator used matches any digit.</td>
<td>Remote-Party-ID: csip:root@217.88.Ñ</td>
</tr>
</tbody>
</table>

**Extracted Snort rule #9:**

<table>
<thead>
<tr>
<th>#</th>
<th>Snort rule-set</th>
<th>PCRE</th>
<th>Description</th>
<th>Operators used</th>
<th>Matching payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>MYSQL</td>
<td>pcre: &quot;/DATE_FORMAT\s*:</td>
<td>This rule is used to identify a Date_Format denial of service attempt.</td>
<td>The rule makes extensive use of the special hex-character operator, e.g. \x22 (&quot;), and the *- and + quantifier. The OR-operator is used to alternate between different formats of specifying the input.</td>
<td>DATE_FORMAT( &quot;akd\is\et&quot;</td>
</tr>
</tbody>
</table>

Figure 7.2 — Rules 5-9 selected from the Snort ruleset
Chapter 7: Evaluation

7.1 Functionality

Figure 7.3 — Content of payload file used in the evaluation of the generated design

The payload data content specified in each of the rules above is combined in the file payload.txt that is read by snort2vhdl. The full contents of the payload file are shown in Figure 7.3. In the payload file, the regex statements before each matching payload are included to be able to distinguish between the data strings.

Executing snort2vhdl on the 10 selected expressions provides the following output from the program:

VHDL source-code successfully generated!
Total PCREs: 10, unsupported: 0

Statistics:
 Comparator: 81 (11 negative)
 Operator entities:
 | OR-entities: 12
 | AND-entities: 7
 Concatenation/Word-entities: 96
 Ranged quantifiers: 6
 |-quantifiers: 18
 +-quantifiers: 12
 ?-quantifiers: 3

Total number of operator entities: 154
File: `vhdl_out.vhd` size: 266023

The output states that the generated VHDL source-code based on the ten introduced PCREs consists of 154 entities and 81 different comparators. The 11 negative comparators are used in the negative character classes such as `[^\n\r]` used in PCRE #3 and `[^@]` used in PCRE #8, as well as the ranged quantifier used in PCRE #2, #4, #7 and #8. The generated VHDL-source code is included on the attached CD-rom as well as on-line.

### 7.1.2 Evaluating designs in Quartus II

The functionality of the generated designs are evaluated using the software package Quartus II (ver. 7.1) developed by Altera Corp. [16]. The Quartus II tool is used for analysis and synthesis of VHDL designs, and can also be used for the final programming of the FPGA board.

In a typical Snort set-up running on a system with a single thread CPU only one PCRE can parse the input symbol during a CPU clock cycle. In order to test a number of PCREs on the same input, the input data has to be repeated for each PCRE. Figure 7.4 describes this scenario, where the grey areas portray situations where the regular expression is not being parsed. The x indicates the time when the input data string to the regular expression is repeated. The lines show feedback from regular expressions, where a high line indicates a match.

A parallel implementation provides all of the implemented regular expressions with the same input symbol in the same CPU clock cycle. The output signals in a parallel scenario are shown in Figure 7.5, where the time-line is the same as in the single thread scenario. By comparing the two figures, it is evident that the parallel implementation would have the potential of great increase in performance, as the input only has to be read once in total,

![Figure 7.4 — Theoretical single thread scenario showing three regular expressions parsing the same input](image-url)
Chapter 7: Evaluation

7.1 Functionality

Figure 7.5 — Theoretical parallel scenario showing three regular expressions parsing the same input.

instead of once for each regular expression.

The goal of the functional evaluation is to prove two arguments:

1. The implemented regular expressions correctly match the user-generated payload data. This is shown by the output signal from each of the implemented PCREs attaining a high output value at least once during the simulation.

2. The implemented PCREs are being matched in parallel. This is shown by the output signals from the implemented PCREs all attaining a high output, before the input string is looped, as shown in Figure 7.5.

The generated VHDL source-code is evaluated in Quartus II by compiling and synthesizing it. An input waveform is needed for the simulation, to define the input signals top the top-level of the design, i.e. the CPU clock and the reset signal. The result of the simulation is a waveform showing how each output signal has behaved during the execution. The output signals in the resulting waveform, describes how the output signals from each implemented PCRE behave. As far as the input signals goes, the clock signal can be set to an arbitrary clock cycle length as this is a simulation. The reset signal is set to high during the first clock cycle to initialize the logical circuits present in the design.

Running the Quartus II simulation on the design based on the PCREs from the introduced Snort rules, using the predefined payload as input characters results in the waveform shown in Figure 7.6. The waveform shows the two input signals to the top-level controller followed by the output signals from the PCREs. Each PCRE has its own output signal titled regexOutput[x] where x refers to the number of the chosen PCREs stated earlier. When the
7.1 Functionality

Figure 7.6 — The resulting simulation waveform after synthesis in Quartus II

The resulting simulation waveform after synthesis in Quartus II

The signal is high it indicates that the pattern has been matched by the provided input data.

The first signal in the waveform is the CPU clock. In Figure 7.6 the zoom level has been set so the entire timeline of the evaluation is visible. Because of the zoom level, the frequent changes in the clock signal are indistinguishable and appear as a solid black line. The next line represents the reset signal, which is only high in the very first clock cycle, in order to reset and initialize all the circuits of the design.

The regexOutput[0] signal represents the output signal from the first of the implemented PCREs. The signal goes high after a certain amount of time which shows that the input has been matched by PCRE #0:

```plaintext
pcre: */\Referer\/[a-zA-Z]+://\.[\ze]*http://\.[\ze]*\.[\ze]*\.[\ze]*\.[\ze]*\.[\ze]*\.\.\ze*\.[\ze]*\.[\ze]*\.[\ze]*\.[\ze]*\.[\ze]*\.[\ze]*\.[\ze]*\.[\ze]*\.
```
This proves that the PCRE matches some of the content from the payload file. As the file contains the matching payload defined in the presentation of the chosen PCREs, this was expected. The regexOutput[0] signal only goes high once during the simulation, showing that it does not match anything else from the payload file.

The following nine signals in Figure 7.6 represent the output from the other implemented expressions. The figure shows that all of the output signals go high once during the simulation run. This proves that each of the regular expressions is matched by the input data provided. The output signal regexOutput[5] is high for a longer period of time. This is because that particular regular expression ends in a +-quantifier referring to the escape sequence \d. The input data matching the regular expression ends in '134566', matching the entire expression with each digit.

The last signal in the waveform is called restartInput. This signal is usually not generated in the design as the input data is only parsed once. It is included in this evaluation to indicate when the last symbol of the input has been found, to be able to compare the simulation output waveform to the expected parallel scenario shown in Figure 7.5. The input restart signal goes high just after the last of the implemented PCREs have provided a high output indicating a match. This shows that the parallel setup is implemented correctly. It would otherwise have been necessary to repeat the payload data content for each of the PCREs resulting in an output waveform at ten times the length. By studying the output from Quartus II it can be concluded that the PCREs have been implemented in a way that matches the right input in a parallel setup.
7.2 Potential

The size of the generated FPGA designs plays an important role. An FPGA is constrained by its finite number of available logical blocks, i.e. registers and look-up tables. The available number of these logical blocks defines the maximum size of the designs it is possible to implement. In this section it is examined how many PCREs can be implemented on the Altera DE1 board provided in the project.

7.2.1 Source-code size

The VHDL source-code generated by snort2vhdl contains a large number of entities that together constitutes the generated FPGA design of the implemented expressions. The size of the VHDL source-code varies greatly on the number of implemented PCREs. To illustrate this, twenty different design sizes have been generated with a PCRE count ranging from 50 to 1000, with a 50 PCRE increase from each level to the next. The PCREs are randomly selected with no duplicates allowed.

Each PCRE level has been generated ten times, to provide a reasonable average describing each level. The average number of entities and comparators required to generate each design size is shown in Table 7.1, along with a graphical representation of the data in Figure 7.7. Figure 7.7 shows that the total number of entities closely follows a linear relation to the number of implemented PCREs. The linearity can also be determined by looking at the table data. The last two columns in Table 7.1 shows the number of entities and comparators pr. PCRE. The number of generated VHDL entities pr. PCRE has an average of 9.7 with little deviation. A rough estimate of the size of a generated design can then be given by multiplying the number of implemented PCREs by 10.

The number of generated character comparators follows a different trend. As snort2vhdl re-uses previously constructed character comparators in later PCREs, the number of new comparators created for additional expressions is decreasing as the PCRE count grows. As shown in Table 7.1 the average number of comparators generated for an FPGA-design based on 50 PCREs is 84. The number of comparators in a design with 1000 PCREs that implements 20 times more regular expressions only uses the double number of character comparators. The reuse of comparators does not change the consumption of FPGA resources, but it does have a positive influence on the compile time. Resource consumption is discussed more thoroughly in the

<table>
<thead>
<tr>
<th>PCRE Count</th>
<th>Average Entities</th>
<th>Average Comparators</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>9.7</td>
<td>84</td>
</tr>
<tr>
<td>100</td>
<td>19.4</td>
<td>168</td>
</tr>
<tr>
<td>150</td>
<td>29.1</td>
<td>252</td>
</tr>
<tr>
<td>200</td>
<td>38.8</td>
<td>336</td>
</tr>
<tr>
<td>250</td>
<td>48.5</td>
<td>420</td>
</tr>
<tr>
<td>300</td>
<td>58.2</td>
<td>504</td>
</tr>
<tr>
<td>350</td>
<td>67.9</td>
<td>588</td>
</tr>
<tr>
<td>400</td>
<td>77.6</td>
<td>672</td>
</tr>
<tr>
<td>450</td>
<td>87.3</td>
<td>756</td>
</tr>
<tr>
<td>500</td>
<td>97.0</td>
<td>840</td>
</tr>
<tr>
<td>550</td>
<td>106.7</td>
<td>924</td>
</tr>
<tr>
<td>600</td>
<td>116.4</td>
<td>1008</td>
</tr>
<tr>
<td>650</td>
<td>126.1</td>
<td>1092</td>
</tr>
<tr>
<td>700</td>
<td>135.8</td>
<td>1176</td>
</tr>
<tr>
<td>750</td>
<td>145.5</td>
<td>1260</td>
</tr>
<tr>
<td>800</td>
<td>155.2</td>
<td>1344</td>
</tr>
<tr>
<td>850</td>
<td>164.9</td>
<td>1428</td>
</tr>
<tr>
<td>900</td>
<td>174.6</td>
<td>1512</td>
</tr>
<tr>
<td>950</td>
<td>184.3</td>
<td>1596</td>
</tr>
<tr>
<td>1000</td>
<td>194.0</td>
<td>1680</td>
</tr>
<tr>
<td>PCREs</td>
<td>Entities</td>
<td>Comparators</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>50</td>
<td>425.3</td>
<td>84</td>
</tr>
<tr>
<td>100</td>
<td>951.7</td>
<td>100.1</td>
</tr>
<tr>
<td>150</td>
<td>1373.4</td>
<td>105.1</td>
</tr>
<tr>
<td>200</td>
<td>1970.8</td>
<td>113.2</td>
</tr>
<tr>
<td>250</td>
<td>2492.4</td>
<td>123.9</td>
</tr>
<tr>
<td>300</td>
<td>2880.8</td>
<td>123.5</td>
</tr>
<tr>
<td>350</td>
<td>3409</td>
<td>126.7</td>
</tr>
<tr>
<td>400</td>
<td>3855.4</td>
<td>134.1</td>
</tr>
<tr>
<td>450</td>
<td>4375.5</td>
<td>138.1</td>
</tr>
<tr>
<td>500</td>
<td>4864.2</td>
<td>139.6</td>
</tr>
<tr>
<td>550</td>
<td>5409.8</td>
<td>146.5</td>
</tr>
<tr>
<td>600</td>
<td>5769</td>
<td>146.8</td>
</tr>
<tr>
<td>650</td>
<td>6540.4</td>
<td>150.2</td>
</tr>
<tr>
<td>700</td>
<td>6741.6</td>
<td>158.1</td>
</tr>
<tr>
<td>750</td>
<td>7486.4</td>
<td>155</td>
</tr>
<tr>
<td>800</td>
<td>7906.6</td>
<td>155.4</td>
</tr>
<tr>
<td>850</td>
<td>8228.1</td>
<td>164.7</td>
</tr>
<tr>
<td>900</td>
<td>8737.1</td>
<td>163.8</td>
</tr>
<tr>
<td>950</td>
<td>9350.1</td>
<td>165.9</td>
</tr>
<tr>
<td>1000</td>
<td>9740.7</td>
<td>166</td>
</tr>
<tr>
<td>Avg.</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 7.1** — The number of generated entities and comparators in VHDL source-code at different design sizes
next section.

![Figure 7.7 — Graph showing the number of generated entities and comparators at different design sizes](image)

### 7.2.2 FPGA resource consumption

During the compilation of the VHDL source-code, different optimization techniques are used by the compiler to reduce the size of the design. This makes the number of entities in the source-code differ from the number of logical blocks required in the FPGA for the final programming. As all FPGA boards have a limited number of resources in terms of logical elements, e.g. registers and lookup-tables, this is an important factor.

A logic block in a standard FPGA consists of a look-up table connected to a register and an output, as shown in Figure 7.8. The output from the block is either from the register or directly from the lookup-table. The actual logical hardware composition of the FPGA chip will not be discussed further in this report, and is only introduced to understand why the number of look-up tables and registers is interesting. The compilation of some of the very large designs has unfortunately proven difficult in Quartus II, as limitations in the software prevent the compilation of very complex designs. Because of this it has been necessary to perform the compilation in a separate program.
For analyzing the logical requirements, twelve different design sizes have been compiled. The size of the designs ranges from 50 to 600 PCREs, where each design size has been compiled between five and ten times with different random expressions, to provide a reasonable average. The results from the compilation are shown in Table 7.2. The compilation results shown in Table 7.2 are represented graphically in Figure 7.9. The Altera DE1 board provided for the project is based on the Cyclone II chip (model: EP2C20F484C7N). This specific chip contains 18752 programmable LUTs and 18752 programmable registers. This maximum level is drawn as the dashed line in the figure.

Table 7.2 — Resource consumption on the Cyclone II chip at different design sizes

<table>
<thead>
<tr>
<th>PCREs</th>
<th>Look-up tables</th>
<th>Registers</th>
<th>LUT/PCRE</th>
<th>REG/PCRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>2164</td>
<td>758</td>
<td>52.28</td>
<td>15.17</td>
</tr>
<tr>
<td>100</td>
<td>3952</td>
<td>1373</td>
<td>39.52</td>
<td>13.73</td>
</tr>
<tr>
<td>150</td>
<td>5631</td>
<td>2128</td>
<td>37.54</td>
<td>14.19</td>
</tr>
<tr>
<td>200</td>
<td>7247</td>
<td>2827</td>
<td>36.24</td>
<td>14.14</td>
</tr>
<tr>
<td>250</td>
<td>8569</td>
<td>3430</td>
<td>34.28</td>
<td>13.72</td>
</tr>
<tr>
<td>300</td>
<td>10484</td>
<td>4257</td>
<td>34.95</td>
<td>14.19</td>
</tr>
<tr>
<td>350</td>
<td>11214</td>
<td>4559</td>
<td>32.04</td>
<td>13.02</td>
</tr>
<tr>
<td>400</td>
<td>13254</td>
<td>5427</td>
<td>33.14</td>
<td>13.57</td>
</tr>
<tr>
<td>450</td>
<td>14149</td>
<td>5764</td>
<td>31.44</td>
<td>12.81</td>
</tr>
<tr>
<td>500</td>
<td>15870</td>
<td>6563</td>
<td>31.74</td>
<td>13.13</td>
</tr>
<tr>
<td>550</td>
<td>18435</td>
<td>8026</td>
<td>33.52</td>
<td>14.59</td>
</tr>
<tr>
<td>600</td>
<td>20070</td>
<td>8700</td>
<td>33.45</td>
<td>14.50</td>
</tr>
</tbody>
</table>
As can be seen in the graph the maximum number of PCREs that can be implemented in the provided board is just below 600. The total number of PCRE-based Snort rules supported is over 5000, so the provided board is not even close to being large enough to implement the full design.

Figure 7.9 — Graph showing the resource consumption on the Cyclone II chip at different design sizes
The results of the experiments performed in the evaluation section proved that the regular expressions correctly matched the generated input. The evaluation also showed that the Altera DE1 board only had available resources for implementing about 1/10th of the supported Snort rule-set.

The Altera DE1 board is based on the Cyclone II-chip, which is a low-end FPGA-chip, as it is designed with focus on simplicity and low-cost. A high-end FPGA-board, such as ones based on the Stratix II-chip, contains a much higher number of available logical elements. This makes it possible to implement much larger designs. The Stratix II chip contains more advanced logic, so a direct comparison of the number of available of internal logical elements cannot be done. In order to provide a valid comparison the same generated design levels have been compiled for the Stratix II. Each level size has been compiled between five and ten times, to provide reasonable average data for the comparison. The data from the Stratix II compilation is shown in Table 8.1, and represented graphically in Figure 8.1.

The Stratix II chip contains 143520 programmable look-up tables and 150386 programmable registers. The compilation results shows that while 600 PCREs was too much for the Cyclone II-based board to handle, this design size only takes up about 5.2 % of the look-up tables in the Stratix II-chip and 3.9 % of the available registers. The calculations are shown below.
Chapter 8: Discussion

<table>
<thead>
<tr>
<th>PCRE</th>
<th>Look-up tables</th>
<th>Registers</th>
<th>LUT/PCRE</th>
<th>REG/PCRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>981</td>
<td>587</td>
<td>19.62</td>
<td>11.74</td>
</tr>
<tr>
<td>100</td>
<td>1543</td>
<td>1050</td>
<td>15.43</td>
<td>10.50</td>
</tr>
<tr>
<td>150</td>
<td>2180</td>
<td>1602</td>
<td>14.53</td>
<td>10.68</td>
</tr>
<tr>
<td>200</td>
<td>2587</td>
<td>1899</td>
<td>12.94</td>
<td>9.50</td>
</tr>
<tr>
<td>250</td>
<td>3127</td>
<td>2382</td>
<td>12.51</td>
<td>9.53</td>
</tr>
<tr>
<td>300</td>
<td>3956</td>
<td>3137</td>
<td>13.19</td>
<td>10.46</td>
</tr>
<tr>
<td>350</td>
<td>4328</td>
<td>3410</td>
<td>12.37</td>
<td>9.74</td>
</tr>
<tr>
<td>400</td>
<td>5064</td>
<td>4019</td>
<td>12.66</td>
<td>10.05</td>
</tr>
<tr>
<td>450</td>
<td>5503</td>
<td>4353</td>
<td>12.23</td>
<td>9.67</td>
</tr>
<tr>
<td>500</td>
<td>5972</td>
<td>4730</td>
<td>11.94</td>
<td>9.46</td>
</tr>
<tr>
<td>550</td>
<td>6548</td>
<td>5201</td>
<td>11.91</td>
<td>9.46</td>
</tr>
<tr>
<td>600</td>
<td>7464</td>
<td>5910</td>
<td>12.44</td>
<td>9.85</td>
</tr>
<tr>
<td>Avg.</td>
<td>N/A</td>
<td>N/A</td>
<td>13.57</td>
<td>10.05</td>
</tr>
</tbody>
</table>

Table 8.1 — Resource consumption on the Stratix II chip at different design sizes

\[
\frac{7464 \text{ LUTs}}{143320 \text{ LUTs}} = 5.2\% \quad \frac{5910 \text{ Registers}}{150386 \text{ Registers}} = 3.9\%
\]

The compilation data shows, that the usage of look-up tables and registers follows a linear growth. By using this assumption it is possible to estimate the maximum number of PCREs that can be implemented on the Stratix II-chip. By extrapolating the average value of LUT/PCRE and REG/PCRE, i.e. 13.57 and 10.05 the data in Table 8.2 is calculated. The estimated data is shown graphically in Figure 8.2. The available number of look-up tables and registers are shown in the figure as dashed lines. Given the linear assumption, Figure 8.2 shows that the number of PCREs that can be implemented is between ten and eleven thousand. The line indicating look-up table is the first to cross a maximum-level line. The estimated exact number is calculated below, where \(x\) is the number of PCREs.

\[
\frac{143520 \text{ LUTs}}{13.57 \text{ LUTs/PCRE}} \times x \\
\downarrow
\]

\[x = 10670.6 \text{ PCREs}\]

Given the data used for determining the average, the total number of PCREs that can be implemented is 10670. This result is an estimate that can vary to some extent. It does show, however, that a Stratix II-based board is fully capable of storing all the Snort rules supported.

A direct comparison between the performance of a standard Snort setup and
Figure 8.1 — Graph showing the resource consumption on the Stratix II chip

the performance of proposed hardware design is difficult. A simple comparison of the clock rate between an FPGA board and a single thread workstation is not possible as the hardware architecture is not comparable.

If we assume that the Stratix II-based board will synthesize the design at 100 MHz, the throughput, i.e. the amount of data that is processed, is easily calculated. The proposed implementation parses one input character (1 byte) in each clock cycle. Hz is defined as the number of clock cycles pr. second, which in this case translates to the number of input symbols pr. second. A clock frequency of 100 MHz thus provides a throughput of 100MB/s. Network speeds are often specified as bits/sec., so by multiplying with 8 bits pr. byte we can calculate the theoretical performance.

\[
\text{Throughput} = 100 \text{ MHz} \times 8 \text{ bits} \\
\downarrow \\
\text{Throughput} = 800 \text{ Mbit/s}
\]

The throughput of the hardware device is independent of the number of implemented PCREs, which would not be the case on a single thread computer running the intrusion detection service.

69
Chapter 8: Discussion

It has not been possible to do performance measurements of an actual Snort installation in this project, so in order to make a comparison we look at related work. Research conducted by Mitra, et. al. presents a performance evaluation between an FPGA implementation and a standard single thread Snort set-up [9]. The computer used is a SGI XEON 5160 workstation, running at 3.0 GHz with 16GB of memory. Different designs of upto 200 PCREs (limited by their FPGA) were evaluated on a 2 GB tcpdump file of network data.

The performance of the workstation showed a linear decrease in throughput at increasing rule counts. At 200 implemented PCREs the workstation maintained a throughput of 36Mbit/s.

Theoretically it was possible to implement 10000 PCREs on the Stratix II board. By following the linear assumption of the workstation performance, it would be able to maintain a throughput of

\[ 36 \text{ Mbit/s} \times \frac{200}{10000} \text{ PCREs} = 0.72 \text{ Mbit/s} \]

at a rule count of 10000. Taking the theoretical throughput of 800 Mbit/s for the Stratix II board into consideration, the relative increase in performance of substituting the workstation with a dedicated device would be

<table>
<thead>
<tr>
<th>PCRE</th>
<th>Look-up tables</th>
<th>Registers</th>
<th>LUT/PCRE</th>
<th>REG/PCRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>13574</td>
<td>10052</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>2000</td>
<td>27149</td>
<td>20104</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>3000</td>
<td>40723</td>
<td>30155</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>4000</td>
<td>54298</td>
<td>40207</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>5000</td>
<td>67872</td>
<td>50259</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>6000</td>
<td>81446</td>
<td>60311</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>7000</td>
<td>95021</td>
<td>70362</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>8000</td>
<td>108595</td>
<td>80414</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>9000</td>
<td>122169</td>
<td>90466</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>10000</td>
<td>135744</td>
<td>100518</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>11000</td>
<td>149318</td>
<td>110569</td>
<td>13.57</td>
<td>10.05</td>
</tr>
<tr>
<td>12000</td>
<td>162893</td>
<td>120621</td>
<td>13.57</td>
<td>10.05</td>
</tr>
</tbody>
</table>

Table 8.2 — Estimated resource consumption of the Stratix II chip at large design sizes
Chapter 8: Discussion

Figure 8.2 — Graph showing the estimated resource consumption of the Stratix II chip

\[
\frac{800 \text{ Mbit/s}}{0.72 \text{ Mbit/s}} = 1111.
\]

Given the assumptions above it can be concluded that the performance increase when looking at an intrusion detection system of 10000 Snort PCRE rules is a factor 1111. Many uncertainties exists in this conclusion, and perhaps 100 MHz is a too modest assumption, it is however obvious that the introduction of an FPGA in an intrusion detection system would provide a substantial performance increase.
The project presents an approach to creating VHDL source-code specifying the design of a large number of PCREs. Due to timing constraints a number of features originally planned for the project have had to be scrapped. These implementation shortcomings and more are discussed in this chapter, as this project has much potential for more research in different areas.

Real network traffic

Feeding the FPGA with a tcpdump of actual network traffic or even better, implementing the pattern matching design on an FPGA device with a network connection was planned. Unfortunately time has not allowed much research in this area. It would be a very interesting area to look more thoroughly into. Due to this a performance test has not been carried out.

Limitations in PCRE implementation

The PCRE language provides extra functionality that is not implemented in this project. Features such as back-references have not been implemented due to lack of time. Back-references are operators, e.g. \1 that can reuse parts of the pattern itself. \1 would be a reference to the first sub-pattern in parenthesis. The back-reference can be seen as a type of addressable memory storing the available references. This feature can be implemented on the FPGA by using the RAM block on the chip, according to Mitra, et. al. [9].
The operator `^` (the caret) is used in a regular expression to match the beginning of the line. The `$` (the dollar sign) operator is used to match the end of the line. These operators have not been implemented as it is (at this point) not possible to parse actual network payload data.

In the ranged quantifier section it was stated that ranged quantifiers referring to expressions ending in another quantifier are not supported, e.g. `(ab^*){4}`. This is because of the definition of the kill entity. In some specific cases, however it is possible to convert the expression to an implementable counterpart. In this example the expression can be converted to the expression `a(b^a){3}b^*`.

**General limitations**

An implementation of the design on a live FPGA with a network interface would require a better way of determining which regular expression has been matched. At this point each implemented expression has its own output signal, which is not optimal as it requires a pin for each expression. More research in this area would result in a better implementation. One approach could be to group the regular expression engines in clusters and create a recursive function that derives which engine has been matched at a given time.

Another improvement that could be looked into is the use of prefix-sharing. By combining regular expressions with the same prefix, it would be possible to save space when implementing similar expression. Lee et. al. [8] and Tripp [15] has conducted research in this area.

Another area of research could be studying a combined PC/FPGA environment where only the most CPU-intensive PCREs of the Snort installation are moved to the FPGA. Using this approach it would be possible to obtain an increase in performance despite using a board based on a small FPGA-chip.

There are of course many more possibilities of extending and enhancing the work done in this project. The suggestions given in this section is just to give an idea of the potential.
The research conducted in this project shows that it is possible to improve intrusion detection by moving CPU-intense applications to dedicated hardware. In this project FPGA designs were generated based on rules from a standard Snort installation. The implemented rules are based on payload inspection using Perl Compatible Regular Expressions.

The main advantage of employing PCREs on dedicated hardware is the possibility of arranging multiple expressions in parallel. In a parallel setup all the implemented rules simultaneously receives the same input symbol in each clock cycle. This provides parallel execution of several PCREs as opposed to a standard single thread setup, where the input data has to be repeated for each expression.

In section 7.1.2, an evaluation proved the functionality of the generated designs. By studying the output signals from the implemented regular expressions, it could be concluded that the expressions were correctly matching the provided input. The parallelism was also proven by studying the output signals as the implemented regular expressions all provided matching outputs without having to loop the input data.

In the discussion a performance increase was estimated by looking at results from related work. Although the estimation is based on a number of assumptions, it showed that introducing a dedicated hardware device, such
as an FPGA, in an intrusion detection system has potential for a dramatic increase in performance.
Executing snort2vhdl

snort2vhdl is executed from the command-line using a number of different optional arguments.

By default the Snort rule-set directory is specified as C:\SNORT\RULES. This setting can be overridden by using the -d argument. The user is then able to specify the location of the Snort rules directory.

The default file name for the file containing the payload data is called payload.txt. By using the -p argument is possible to specify a different name/path for the input file that should be used.

The default name for the filter file is filters.txt, this can be overridden by using the -f argument.

By using the -v argument it is possible to specify the name of the generated file containing the VHDL source-code, the default filename is vhdl_out.vhd.

The -r argument provides a way to extract just a specific number of PCREs. This is useful if the target board is too small to fit the entire rule-set. Table A.1 shows the valid optional command-line arguments of snort2vhdl.

The resulting output from the execution of the program provides information about the generated source-code, such as the number of PCREs implemented and entities created. Executing the program without any arguments
Chapter A: Executing snort2vhdl

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d&lt;path&gt;</td>
<td>The path of the Snort rule-set directory (default setting: C:\SNORT\RULES)</td>
</tr>
<tr>
<td>-f&lt;filename.ext&gt;</td>
<td>Name of the file containing PCRE exclusion filters (default setting: filters.txt)</td>
</tr>
<tr>
<td>-p&lt;filename.ext&gt;</td>
<td>Name of the payload file containing input data for the program (default setting: payload.txt)</td>
</tr>
<tr>
<td>-v&lt;filename.ext&gt;</td>
<td>Name of the generated output file (default setting: vhdl_out.txt)</td>
</tr>
<tr>
<td>-r&lt;number&gt;</td>
<td>Number of randomly selected PCREs to be implemented (default setting: no random selection)</td>
</tr>
<tr>
<td>-h</td>
<td>Help screen showing the valid arguments.</td>
</tr>
</tbody>
</table>

Table A.1 — Optional command-line arguments of snort2vhdl

using the rule-set from Snort ver. 2.7.0, gives the following output:

VHDL source-code successfully generated!
Total PCREs: 5455, unsupported: 73

Statistics:
Comparator entities: 246 (86 negative)

Operator entities:
OR-entities: 16194
AND-entities: 972
Concatenation/Word-entities: 27025
Ranged quantifiers: 3618
*quantifiers: 7037
+quantifiers: 1827
?quantifiers: 1446

Total number of operator entities: 58119

File: ‘vhdl_out.vhd’ size: 63683325

The output states that the generated design consists of 5455 PCREs, built by creating 58119 VHDL-entities and 245 comparators. The size of the generated source-code is quite large, 63 Mb. The comparator count is not limited to single character comparators. Character interval comparators, e.g. \[a – z\] are also included; explaining the validity of the high number. The
number of unsupported PCREs specifies how many PCREs were skipped due to lack of implemented functionality.
B.1 Positive character comparator

--- Special Character Comparator 'R' charIndex=0

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity comparator0 is
  port(clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector(7 downto 0); o: out std_logic);
end comparator0;

architecture behv of comparator0 is
  signal resultComp: std_logic := '0';
signal iDelay: std_logic;
begin
  process(clk, reset)
  begin
    if (reset='1') then
      iDelay <= '0';
    elsif (clk'event and clk='1') then
      iDelay <= i AFTER 1 ns;
    end if;
  end process;
  process(inputChar)
  begin
    if inputChar=std_logic_vector(to_unsigned(character 'pos ('R'),8)) then
      
81
B.2 Concatenation entity

---

Entity E1: Generated WORD/CONC-entity

library ieee;
use ieee.std_logic_1164.all;

entity E1 is
  port( clk, i, halt, reset: in std_logic;
        inputChar: in std_logic_vector(7 downto 0);
        o: out std_logic);
end E1;

architecture behv of E1 is

  component comparator0
    port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
  end component;

  component comparator1
    port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
  end component;

  component comparator2
    port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
  end component;

  component comparator3
    port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
  end component;

  component comparator4
    port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
  end component;

  component comparator5
    port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
  end component;

  component comparator6
    port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
  end component;

  component comparator7

end behv;
Chapter B: VHDL source-code

B.3 OR-entity

B.3 OR-entity

library ieee;
use ieee.std_logic_1164.all;

entity E13 is
  port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector(7 downto 0);
       o: out std_logic);
end E13;

architecture behv of E13 is
  component comparator71
    port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
  end component;
  component comparator70
    port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
  end component;
  signal i1, i2: std_logic;
begin
  ENTcomparator70: comparator70 port map (clk=>clk, i=>i, halt
  =>halt, reset=>reset, inputChar=>inputChar, o=>i1);
  ENTcomparator71: comparator71 port map (clk=>clk, i=>i, halt
  =>halt, reset=>reset, inputChar=>inputChar, o=>i7);
end behv;
B.4 AND-entity

1 — Entity E41: Generated AND-entity

2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity E41 is
5   port ( clk, i, halt, reset: in std_logic;
6          inputChar: in std_logic_vector(7 downto 0);
7          o: out std_logic);
8 end E41;
9 architecture behv of E41 is
10   component negComparator4
11     port (clk, i, halt, reset: in std_logic; inputChar: in
12          std_logic_vector; o: out std_logic);
13 end component;
14   component negComparator3
15     port (clk, i, halt, reset: in std_logic; inputChar: in
16          std_logic_vector; o: out std_logic);
17 end component;
18   signal i1, i2: std_logic;
19   signal i1found, i2found: std_logic;
20   signal andOut: std_logic;
21 begin
22   process (clk, reset)
23   begin
24     if (reset='1') then
25       i1found <= '0'; i2found <= '0';
26     elsif (clk='1' and clk'event) then
27       if (i='1' or (i1found='1' and i2found='1')) then
28         i1found <= '0' AFTER 1 ns; i2found <= '0' AFTER 1 ns
29       elsif (i='0') then
30         if (i1='1') then
31           i1found <= '1' AFTER 1 ns;
32         end if;
33         if (i2='1') then
34           i2found <= '1' AFTER 1 ns;
35         end if;
36       end if;
37     end if;
38   end process;
39   process (i1, i2, i1found, i2found)
40   begin
41     if ((i1found='1' and i2found='1') or (i1='1' and i2found

Chapter B: VHDL source-code

B.5 *-quantifier

1 — Entity E70: Generated STAR-entity
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity E70 is
5 port (clk, i, halt, reset: in std_logic;
6       inputChar: in std_logic_vector(7 downto 0);
7       o: out std_logic);
8 end E70;
9 architecture behv of E70 is
10 component comparator172
11 port (clk, i, halt, reset: in std_logic; inputChar: in
12      std_logic_vector; o: out std_logic);
13 end component;
14 signal i1, outputR1: std_logic;
15 begin
16 outputR1 <= i or i1;
17 o <= i or i1;
18 end behv;

B.5 *-quantifier

1 — Entity E70: Generated STAR-entity
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity E70 is
5 port (clk, i, halt, reset: in std_logic;
6       inputChar: in std_logic_vector(7 downto 0);
7       o: out std_logic);
8 end E70;

B.6 +-quantifier

1 — Entity E80: Generated PLUS-entity
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity E80 is
5 port (clk, i, halt, reset: in std_logic;
6       inputChar: in std_logic_vector(7 downto 0);
7       o: out std_logic);
8 end E80;
9 architecture behv of E80 is
10 component comparator202
11 port(clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
12 end component;
13 signal i1, o1: std_logic;
14 begin
15 o1 <= i OR i1;
16 o <= i1;
17 ENTcomparator202_first: comparator202 port map (clk=>clk, i =>i1, halt=>halt, reset=>reset, inputChar=>inputChar, o=>i1);
18 end behv;

B.7 ?-quantifier

--- Entity E79: Generated IS-entity
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity E79 is
5 port( clk, i, halt, reset: in std_logic;
6 inputChar: in std_logic_vector(7 downto 0);
7 o: out std_logic);
8 end E79;
9 architecture behv of E79 is
10 component comparator200
11 port(clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector; o: out std_logic);
12 end component;
13 signal i1, outputR1: std_logic;
14 begin
15 o <= i or i1;
16 ENTcomparator200: comparator200 port map (clk=>clk, i=>i, halt=>halt, reset=>reset, inputChar=>inputChar, o=>i1);
17 end behv;

B.8 Ranged quantifier

--- Entity E68: Generated RANGE-entity
2 library ieee;
3 use ieee.std_logic_1164.all;
4 entity E68 is
5 port( clk, i, halt, reset: in std_logic;
6 inputChar: in std_logic_vector(7 downto 0);
7 o: out std_logic);
8 end E68;
9 architecture behv of E68 is
10 component E66 is
Chapter B: VHDL source-code

B.8 Ranged quantifier

```
port( clk, i, halt, reset: in std_logic; inputChar: in std_logic_vector(7 downto 0);
    o: out std_logic);
end component;

component E67 is
  port( clk, i, halt, reset: in std_logic;
    inputChar: in std_logic_vector(7 downto 0);
    o: out std_logic);
end component;

signal i1, o1, startKill: std_logic;
signal foundCount: integer := 0;
signal iKill: std_logic;
begin
  setCounter: process(clk, reset)
  begin
    if (reset='1') then
      foundCount <= 0;
    elsif (clk'event and clk='1') then
      if (i1='0') then
        if (iKill='1') then
          foundCount <= 0 AFTER 1 ns;
        end if;
      else
        if (iKill='0') then
          foundCount <= foundCount + 1 AFTER 1 ns;
        else
          foundCount <= 1 AFTER 1 ns;
        end if;
      end if;
    end if:
  end process;

  setOutput: process(i1, foundCount, iKill)
  begin
    if (i1='1' and foundCount>=70) then
      o <= '1';
    else
      o <= '0';
    end if;
  end process;

  ol <= (i or i1) and not halt;
  startKill <= i and i1;
in_E66: E66 port map (clk=>clk, i=>o1, halt=>halt, reset=>reset, inputChar=>inputChar, o=>i1);
kill_E67: E67 port map (clk=>clk, i=>startKill, halt=>halt, reset=>reset, inputChar=>inputChar, o=>iKill);
end behv;
```
B.9 Top-level entity

1 — Generated Top-level RegEx control entity

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity regExp is
  port( clk, reset : in std_logic;
        regexOutput : out std_logic_vector(2 downto 0));
end regExp;

architecture behv of regExp is

-- convert hex character to 4 bit signed

function hex2bits(char : character) return std_logic_vector is
  variable bitResult: std_logic_vector(3 downto 0);

begin
  case char is
    when ' ' => bitResult:="0000";
    when '0' => bitResult:="0000";
    when '1' => bitResult:="0001";
    when '2' => bitResult:="0010";
    when '3' => bitResult:="0011";
    when '4' => bitResult:="0100";
    when '5' => bitResult:="0101";
    when '6' => bitResult:="0110";
    when '7' => bitResult:="0111";
    when '8' => bitResult:="1000";
    when '9' => bitResult:="1001";
    when 'a' => bitResult:="1010";
    when 'b' => bitResult:="1011";
    when 'c' => bitResult:="1100";
    when 'd' => bitResult:="1101";
    when 'e' => bitResult:="1110";
    when 'f' => bitResult:="1111";
    when others => ASSERT (false) REPORT "Error! Invalid character!" SEVERITY failure;
  end case;

  return bitResult;
end hex2bits;

component E6

  port( clk, i, halt, reset : in std_logic;
        inputChar: in std_logic_vector(7 downto 0);
        o: out std_logic);
end component;

component E36

  port( clk, i, halt, reset : in std_logic;
        inputChar: in std_logic_vector(7 downto 0);
        o: out std_logic);
end component;

component E53

end component;
```
47    port( clk, i, halt, reset: in std_logic;
        inputChar: in std_logic_vector(7 downto 0);
        o: out std_logic);
50    end component;
51    signal inputChar: std_logic_vector(7 downto 0) := "00000000"
49
52    signal charIndex: integer range 1 to 100;
53    signal i, halt: std_logic := '0';
54    constant payloadData: string := "726567657820303a0a526566657265723a2087";
55
56    begin
57    process(clk, reset)
58    begin
59        if (reset = '1') then
60            i <= '1';
61            charIndex <= 1;
62            inputChar <= "00000000";
63        elsif (clk'event and clk = '1') then
64            if (charIndex <= payloadData'length) then
65                inputChar <= hex2bits(payloadData(charIndex)) &
66                                hex2bits(payloadData(charIndex+1)) AFTER 1 ns
67            charIndex <= charIndex + 2 AFTER 1 ns;
68        elsif (charIndex > payloadData'length) then
69            inputChar <= "00000000" AFTER 1 ns;
70            i <= '0';
71            halt <= '1';
72        end if;
73    end if;
74    end process;
75    regex0: E6 port map (clk=>clk, reset=>reset, i=>i, halt=>
76        halt, inputChar=>inputChar, o=>regexOutput(0));
77    regex1: E36 port map (clk=>clk, reset=>reset, i=>i, halt=>
78        halt, inputChar=>inputChar, o=>regexOutput(1));
79    regex2: E53 port map (clk=>clk, reset=>reset, i=>i, halt=>
80        halt, inputChar=>inputChar, o=>regexOutput(2));
81
82
```
Bibliography


