Design of Efficient FPGA Circuits for Matching Complex Patterns in Network Intrusion Detection Systems

A Thesis
Presented to
The Academic Faculty

By

Christopher R. Clark

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science in
Electrical and Computer Engineering

Georgia Institute of Technology
December 2003

Design of Efficient FPGA Circuits
for Matching Complex Patterns in
Network Intrusion Detection Systems

Approved by:

Dr. David E. Schimmel, Advisor
School of ECE

Dr. Sudhakar Yalamanchili
School of ECE

Dr. Wenke Lee
College of Computing

December 8, 2003

TABLE OF CONTENTS

TABLE OF CONTENTS iii

LIST OF TABLES v

LIST OF FIGURES vi

SUMMARY vii

1 BACKGROUND AND RELATED WORK 1

1.1 Pattern Matching in Network Intrusion Detection 1

1.2 Pattern Matching in Software 3

1.2.1 Single Pattern Algorithms 3

1.2.2 Multiple Pattern Algorithms 5

1.2.3 Intrusion Detection Algorithms 7

1.3 Pattern Matching with Reconfigurable Hardware 12

1.3.1 Introduction 12

1.3.2 Brute-Force Approach 14

1.3.3 Finite Automata Approaches 15

2 INTRODUCTION 21

2.1 Motivation 21

2.2 The HardIDS Project 22

3 IMPLEMENTATION 24

3.1 Efficient Circuit Design 24

3.2 High-Throughput Circuit Design 28

3.3 Support for Complex Patterns 30

3.3.1 Case Insensitivity 30

3.3.2 Bounded-length Wildcards 31

3.4 Approximate Pattern Matching 34

3.5 Protocol Analysis 36

3.6 Automated Circuit Generation 38

3.7 System Integration 38

4 EVALUATION 40

4.1 Comparison of Circuit Design Approaches 40

4.1.1 Capacity Scalability 41

4.1.2 Throughput Scalability 44

4.2 Comparison with Previous Work 45

4.3 Complexity of Approximate Matching 47

REFERENCES 48

LIST OF TABLES

Table 1: Speed and area comparison of different design approaches (Virtex-1000) 43

Table 2: Area and throughput for different input widths (Virtex2-6000) 44

Table 3: Comparison with Previous Work 46

LIST OF FIGURES

Figure 1: Snort rule example 2

Figure 2: Aho-Corasick pattern matching machine example 6

Figure 3: AC_BM heuristics 10

Figure 4: Corresponding NFA and circuit components 19

Figure 5: Corresponding NFA and circuit for ((a|b)*)(cd) 20

Figure 6: HardIDS data flow 23

Figure 7: Match function for the character "`a`" (`0110 0001`) 25

Figure 8: Comparison of NFA design approaches 26

Figure 9: Data path of the FPGA pattern matching processor 27

Figure 10: Four-way parallel decoder NFA circuit for pattern "`snort`" 29

Figure 11: Use of position rule options 31

Figure 12: Wildcard with lower-bounded length 33

Figure 13: Wildcard with upper-bounded length 33

Figure 14: NFA for "`abcd`" with $k \leq 2$ 35

Figure 15: Circuit for "`abcd`" with $k \leq 2$ 36

Figure 16: Protocol analysis 37

Figure 17: Capacity comparison of different design approaches (Virtex-1000) 43

Figure 18: Complexity of approximate matching circuits (Virtex2-6000) 47

SUMMARY

The objective of this research is to design and develop a reconfigurable string matching co-processor using field-programmable gate array (FPGA) technology that is capable of matching thousands of complex patterns at gigabit network rates for network intrusion detection systems (NIDS). The motivation for this work is to eliminate the most significant bottleneck in current NIDS software, which is the pattern matching process. The tasks involved with this research include designing efficient, high-performance hardware circuits for pattern matching and integrating the pattern matching co-processor with other NIDS components running on a network processor. The products of this work include a system to translate standard intrusion detection patterns to FPGA pattern matching circuits that support all the functionality required by modern NIDS. The system generates circuits efficient enough to enable the entire ruleset of a popular NIDS containing over 1,500 patterns and 17,000 characters to fit into a single low-end FPGA chip and process data at an input rate of over 800 Mb/s. The capacity and throughput both scale linearly, so larger and faster FPGA devices can be used to further increase performance. The FPGA co-processor allows the task of pattern matching to be completely offloaded from a NIDS, significantly improving the overall performance of the system.

1 BACKGROUND AND RELATED WORK

1.1 Pattern Matching in Network Intrusion Detection

Network intrusion detection is the process of analyzing data sent over a computer network in order to detect the presence of any malicious or suspicious content in the data. The analysis may include the identification of *unusual* patterns in the data (anomaly-based detection) or the recognition of *specific* patterns in the data (signature-based detection). A signature-based network intrusion detection system (NIDS), also known as a rule-based NIDS, uses a set of signatures, or rules, which describe properties of network packets that indicate suspicious activity or known attacks. The NIDS compares each incoming packet to each of the rules, and generates an alert when a packet exhibits all of the properties specified by a rule. Some of the common properties analyzed for each packet include protocol, source address and port, destination address and port, data size, and data content.

The NIDS of primary interest in this research, Snort, is an example of a signature-based NIDS. Snort is one of the most widely-deployed NIDS in use today. Some of the reasons that it has become so popular are that it is free, open-source, highly configurable, and it has a large, well-maintained rule database, an active group of developers, and a responsive support community. Another attraction of Snort is its rule specification language, which is powerful but much easier to understand than that of other NIDS languages. An example of a Snort rule is shown in Figure 1. This rule will trigger an alert for an attempted buffer overflow attack on an FTP server. The table below the rule details the properties that a packet must posses in order to match the rule. The part of the rule before the parentheses is called the rule header, and the part of the rule inside the pa-

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP EXPLOIT
stat overflow"; flags:A+; dsize:>1000; content:"stat "; nocase;)
```

Protocol:	TCP
Source Address:	From External Network
Source Port:	Any
Destination Address:	To Internal Network
Destination Port:	21 (FTP)
TCP Flags:	ACK
Data size:	Greater than 1000 bytes
Payload data contains: (case-insensitive match)	"stat "

Figure 1: Snort rule example

rentheses contains the rule options. Each of the rule options is a Boolean predicate that returns true or false. The rule options form a logical conjunction, which means that all the predicates must evaluate to true in order for a rule match to occur.

Typically, a rule header is quite general and will be matched by a large amount of normal network traffic. The rule options are more specific and help to filter out harmless traffic. Often, the most discriminating filtering is provided by the content rule option, which searches the data payload for the presence of specific strings. However, simply checking for the existence of one or more patterns in a packet's payload is not sufficient for accurate detection of attacks. Sometimes, an alert may be generated for a packet that contains the patterns specified in a rule even though the packet is not part of an attack. For example, a web page describing how a particular attack works might be wrongly considered an active attack on a web server. Such invalid alerts are called false positives and can be avoided through the use of more detailed specifications of the attack patterns. In order to reduce false positives, a NIDS and its rule language should allow a rule to spec-

ify the position, ordering, and spacing of patterns as part of an attack's signature. Some of the necessary capabilities of a NIDS pattern matcher are listed below:

- Find a pattern that occurs after an offset from the beginning of the packet data.
- Find a pattern that occurs (or does not occur) within a certain number of bytes from the beginning of the packet data or from a specified offset.
- Find a pattern that occurs (or does not occur) after another pattern.
- Find a pattern that occurs (or does not occur) after another pattern within a certain number of bytes or after a certain number of bytes.

The Snort rule language provides mechanisms for specifying all of the above pattern relationships, and the Snort detection engine supports their semantics.

1.2 Pattern Matching in Software

The task of string pattern matching occurs in many applications and has been studied for decades. Over the years, new algorithms have been developed to improve the performance of pattern matchers in certain situations. In the following sections, some commonly used algorithms are briefly discussed and references to more detailed explanations are provided.

1.2.1 Single Pattern Algorithms

In single pattern matching algorithms, the input text is scanned looking for an occurrence of the target pattern. If the application needs to check for the presence of multiple target patterns, the input text must be scanned once for each pattern. The simplest of these algorithms, known as the naïve, or brute-force, technique, does a character-by-character match of the pattern at each position in the input text. The process starts with

the left-most characters of the text and the pattern aligned. Characters are compared from left to right until a mismatch is found. After a mismatch, the pattern is shifted right one position and the match process restarts with the comparison of the first pattern character with the second text character. If m is the length of the pattern and n is the length of the input text, then the worst-case number of character comparisons in the brute-force algorithm is $O(mn)$.

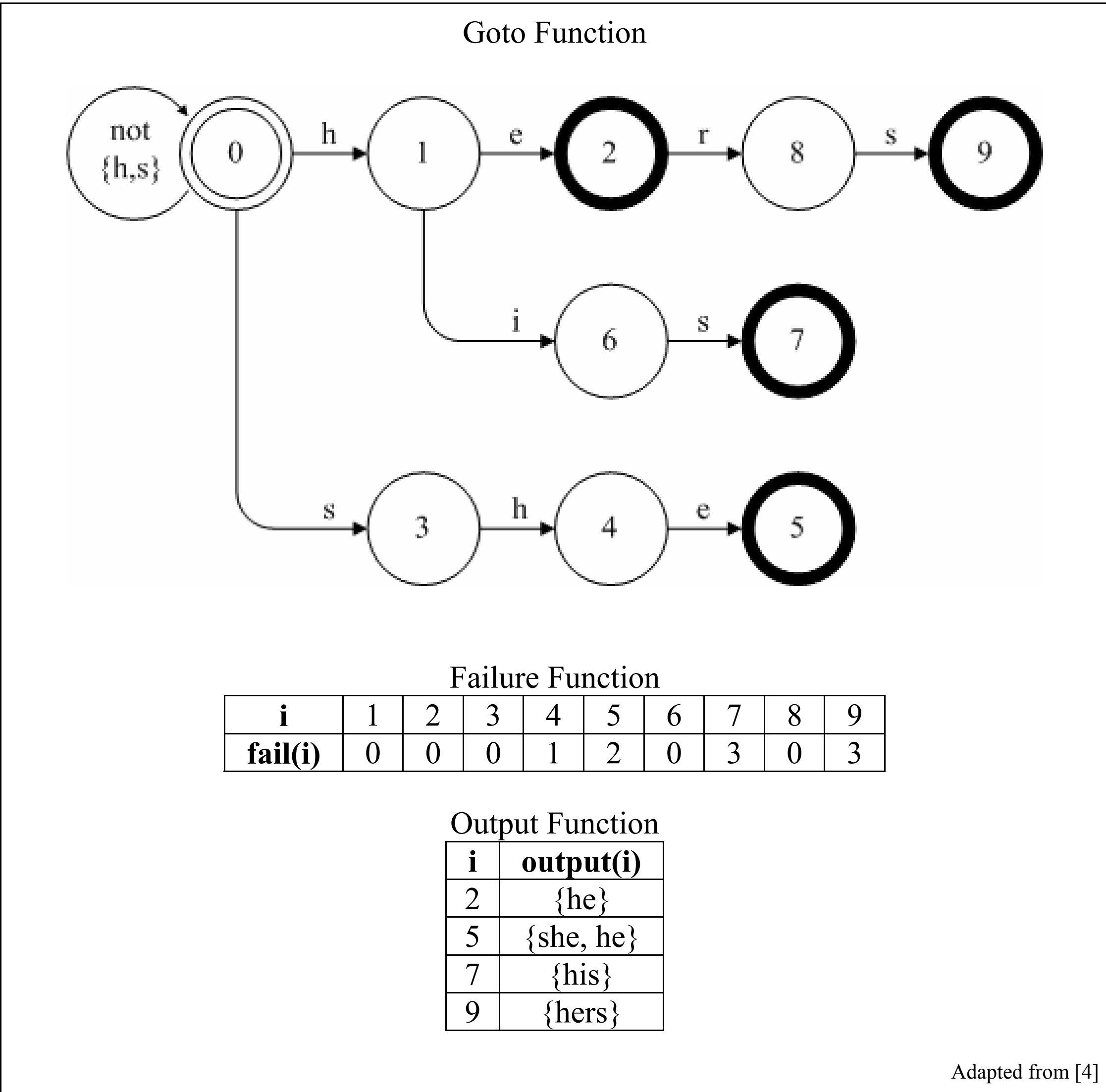
Knuth, Morris, and Platt (KMP) discovered that knowledge of the properties of the target pattern could be used to reduce the required number of comparisons [1]. The KMP algorithm is similar to the brute-force algorithm except for the way mismatches are handled. In the brute-force algorithm, the pattern is always shifted right one position after a character mismatch, regardless of the number of character matches that occurred before the mismatch. This can lead to wasted comparisons at positions that cannot possibly result in a complete pattern match. The KMP algorithm eliminates these unnecessary comparisons by using information about repeated substrings in the pattern. The pattern is preprocessed to generate a look-up table that indicates how many positions the pattern may be shifted to the right based on the position in the pattern where a mismatch occurs. The preprocessing takes $O(m)$ time, the skip table uses $O(m)$ memory, and the worst-case number of character comparisons is $O(n)$.

Boyer and Moore (BM) observed that even more character comparisons could be skipped if the comparison started with the rightmost character of the pattern and proceeded from right to left [2]. Like the KMP algorithm, the BM algorithm preprocesses the pattern to generate a table of mismatch skip values based on pattern position. In addition, the BM algorithm generates another table of skip values based on the value of the

pattern character involved in the mismatch. This table requires an entry for each character in the alphabet used. The value stored in the table for each character is either the rightmost position of that character in the pattern, or the length of the pattern if the character does not occur in the pattern. When a mismatch occurs, the appropriate values from each table are retrieved and the maximum of the two values indicates the number of character positions that the pattern may be shifted to the right before the next comparison. This algorithm requires $O(m)$ memory for the first table and $O(q)$ memory for the second table, where q is the size of the alphabet. The worst case number of comparisons is $O(n)$, but the expected number of comparisons is less than n and decreases as the pattern length increases. A study [3] of single pattern matching algorithms has shown that the BM algorithm is the quickest in all cases except for when binary alphabets or very short patterns are involved.

1.2.2 Multiple Pattern Algorithms

A multiple pattern matching algorithm searches a text string for the occurrence of any pattern in a set of patterns using only a single pass through the text. The most well-known technique of this type is the Aho-Corasick (AC) algorithm [4]. The AC algorithm preprocesses the set of patterns to construct a pattern matching machine based on a deterministic finite automaton (DFA). The matching procedure works by reading successive characters from the input string, making state transitions based on each character, and producing output when a complete pattern is matched. There are three functions involved in the process: a goto function, a failure function, and an output function. Figure 2 shows an example of these functions for the set of patterns {he, she, his, hers}.



Failure Function

i	1	2	3	4	5	6	7	8	9
fail(i)	0	0	0	1	2	0	3	0	3

Output Function

i	output(i)
2	{he}
5	{she, he}
7	{his}
9	{hers}

Figure 2: Aho-Corasick pattern matching machine example

The goto function determines if a state transition can be made based on the current state and the value of the input character. If the input matches one of the characters on an arc leaving the current state, then the state pointed to by the arc becomes the next state. If the input does not match any of the characters on the arc leaving the current state, then the failure function determines the next state. The failure function causes a transition to another intermediate state if the current partial match represents a prefix of another pattern in the set. Otherwise, the failure function causes a transition to the initial state (0). The output function is consulted after each state transition to determine if the current state represents a pattern match. The process continues until the end of the input text is reached.

1.2.3 Intrusion Detection Algorithms

The most computationally intensive task of a rule-based NIDS is searching each incoming packet's payload for the patterns specified by each of the rules [5]. Therefore, the algorithm used for pattern matching has a significant effect on the overall performance of the system. This section discusses research that has attempted to improve pattern matching algorithms for NIDS. Due to the availability of the source code and the rule database, these studies have focused on Snort. However, the algorithms and results presented here are applicable to other NIDS and to other network security applications.

The baseline pattern matching algorithm used for comparison purposes is the implementation in Snort version 1.9 and earlier. This design divides the rules in the database into groups based on their rule headers and stores them in a two-dimensional linked list. The distinct rule headers are stored as rule-tree nodes (RTNs) in a linked list. Attached to

each RTN is a linked list of option tree nodes (OTNs). An OTN contains all of the rule options for a single rule. The detection process for an incoming packet works as follows:

1. The packet's header fields are compared to the fields of each RTN in the list. If no matching node is found, the analysis for the packet is complete.
2. If a matching RTN is found, the associated list of OTNs is searched. At each OTN, the rule options are evaluated sequentially. If an option check fails, the search continues at the next OTN. If all of the option checks in a given OTN succeed, then an alert is generated and no more rules are checked against this packet.

In order to reduce the amount of pattern matching required, the 'content' rule options are always checked last for each OTN. The Boyer-Moore algorithm (described in section 1.2.1) is used for content pattern matching.

Since Snort groups related rules together under the same RTN, these rules often contain common substrings in their content options. However, because Snort applies the BM algorithm to each OTN's content options individually, it cannot take advantage of results from previous pattern matching attempts. A frequent occurrence in the Snort rule database is that many rules in an OTN list will have a common prefix in their content options. In these cases, a single comparison operation could potentially eliminate many rules from further consideration, significantly reducing the amount of pattern matching required for a packet that does not contain the shared prefix.

The common prefix insight above served as the motivation for some researchers to develop a new pattern matching algorithm for Snort [6]. They describe the new algorithm as "a Boyer-Moore-like algorithm applied to a set of keywords held in an Aho-

Corasick-like keyword tree that overlays common prefixes of the keywords." They call the algorithm AC_BM, but the only similarity to the AC algorithm is the use of a pattern tree. During initialization, one tree is generated for each RTN. The matching process uses techniques similar to BM to skip character comparisons, but instead of sliding a single pattern across the text, AC_BM slides the whole pattern tree across the text. Initially, the shortest pattern in the tree is aligned with the right end of the text, and the tree moves left towards the beginning of the text. Unlike BM, the character comparisons are performed from left to right with respect to the patterns. Two skip tables are generated using modified versions of the BM heuristics called the bad character shift and the good prefix shift, which are demonstrated in Figure 3. When a mismatch occurs, the bad character shift suggests shifting to the next occurrence in the pattern tree of the mismatching text character. There are two possible cases for the good prefix shift:

1. If a substring of the partial match occurs as a complete prefix of another pattern, the shift is to the character following the prefix in that pattern.
2. If a prefix of the partial match occurs as a suffix of another pattern, the tree is shifted so that the suffix is aligned with the matching characters of the text. This case is illustrated in Figure 3.

To ensure that no matches are missed, the algorithm cannot skip over more characters than the length of the shortest pattern. Therefore, the shift value used is the maximum of the values from the two heuristics if it is less than the length of the shortest pattern; otherwise, the length of the shortest pattern is used.

Patterns: time, tired, tiring, tinted, tinsel

Before Shift:

```
                                 r e d
                                / \
                               /   i n g
                               |
tree ->                      t i m e
                               |
                               \   t e d
                                \ /
                                 n s e l
                             *
text ->  t i m e i s o n m y s i d e
```

After Shift:

```
                           r e d
                          / \
                         /   i n g
                         |
tree ->                t i m e
                         |
                         \   t e d
                          \ /
                           n s e l
                       *
text ->  t i m e i s o n m y s i d e
```

Patterns: time, tired, tiring, tomato, tornado

Before Shift:

```
                         r e d
                        / \
                       /   i n g
                       |
tree ->              t i m e
                     |
                     \   r n a d o
                      \ /
                       o m a t o
                     *
text ->  a u t o m a t o n e
```

After Shift:

```
                 r e d
                / \
               /   i n g
               |
tree ->      t i m e
             |
             \   r n a d o
              \ /
               o m a t o
             *
text ->  a u t o m a t o n e
```

From [6]

Bad Character Shift: The initial comparison fails on the text character ‘s’, so the tree is shifted to the next occurrence of ‘s’ in the tree.

Good Prefix Shift: There is a prefix match of ‘to’ that occurs as a suffix of ‘tomato’, so the tree is shifted to align suffix with the matching text.

Figure 3: AC_BM heuristics

Integrating the AC_BM algorithm into Snort requires some changes to the Snort detection engine. The OTN list is sorted so that all of the rules without content options are before the rules with content options. This means that the non-content rules will be checked before the content rules. If none of the non-content rules match the current packet, then the AC_BM algorithm is invoked. If no patterns are found, then the analysis is done for this packet. Otherwise, the pattern matching will stop when the first match is found and the rest of the rule options in the rule corresponding to the matched pattern will be checked. The implementation of AC_BM in [6] does not support case-sensitive matching, but states that this could be accomplished by adding a second tree to each RTN for the case-sensitive patterns. Also, only one content string from each rule is included in the pattern tree. If a rule has a hit from the tree matching, any additional content strings are checked using the standard BM algorithm.

Due to the first-match design and rule matching order of Snort, there are some issues with this implementation of AC_BM. Since standard Snort scans packets from left to right while AC_BM scans from right to left, the first match found by each algorithm might be different. Combining strings from multiple rules into a tree causes the relative order of the rules to be lost, which can result in a less-precise match because AC_BM will choose the shortest matching pattern first. Also, placing the more general non-content rules first could cause some attacks to be missed. Many of these shortcomings could be eliminated with a more careful implementation. The authors state that their implementation was just a proof-of-concept and that they were mostly interested in improving the pattern matching performance.

The experiments show that the AC_BM implementation scales better than BM as the number of rules is increased. When tested with the entire Snort 1.6.3 rule database, AC_BM was about 18 percent faster than standard BM. In a test with only content rules, the execution time for AC_BM increased about 30 percent as the number of rules was increased from 200 to 786, while the time for standard BM increased over 200 percent. However, AC_BM uses approximately three times as much memory as standard BM due to the storage requirements of the skip tables.

At about the same time, another group independently studied multiple pattern search algorithms for Snort [5]. They did not modify the structure of the Snort rule engine and treated the group of patterns associated with each RTN as a separate pattern set. Their experiments showed that the most efficient algorithm varies with the size of the set. Thus, they advocate using a hybrid approach. Highest performance was achieved by using the Boyer-Moore (with Horspool optimization) algorithm for sets of size one, a setwise Boyer-Moore-Horspool algorithm that they developed for sets of size two through 100, and the standard Aho-Corasick algorithm for sets of size greater than 100. This hybrid approach yielded a 52 percent reduction in pattern matching time over the standard Snort algorithm (single pattern Boyer-Moore).

1.3 Pattern Matching with Reconfigurable Hardware

1.3.1 Introduction

The task of searching a collection of data for a set of patterns using a single pattern matching algorithm involves carrying out a large number of independent computations, where each computation consists of searching the input data for one pattern. The work of this task can be divided among multiple processing elements by assigning different com-

putations to each processing element. Since there are no dependences between the computations, they can all be executed simultaneously, resulting in an overall speedup that is proportional to the number of processing elements. When a processing element can be constructed out of a relatively small amount of reconfigurable logic, as is the case for a text pattern matcher, a single FPGA chip can be programmed to contain many thousands of processing elements. Since all processing elements can work on the input data at the same time, the FPGA performance can be several orders of magnitude better than that of a general-purpose processor. Unlike a software implementation, the FPGA's number of computation cycles remains constant as the number of patterns is increased (assuming there are enough available logic resources to implement a processing element for each pattern).

Due to the large number of applications using string pattern matching and the potential speedup offered by reconfigurable logic, there have been many pattern matching systems developed based on FPGAs. Several of these systems have been designed for interactive applications in which the target patterns are not known until run-time. The focus of these research projects was the development of techniques for quickly reconfiguring the logic based on the input patterns. The total execution time required for one application instance can be expressed as the sum of its three components [7]:

1. T_M: the time required to develop specialized circuitry for the given problem instance and map it into an FPGA bitstream. This is usually done on the host computer and is often the largest component.
2. T_R: the time required to reconfigure the FPGA. This usually involves downloading the bitstream to the FPGA over some type of interconnect.

3. T_E: The time required for the FPGA to execute the desired task on the input and produce output.

Reducing T_M was addressed in [8] by preprogramming the FPGAs with skeleton circuitry, and then using partial reconfiguration at runtime to insert word detectors into the circuit. Although this significantly reduced T_M, its effectiveness was limited by a lengthy T_R. A different technique, called self-reconfiguration, was taken by [7] in order to reduce both T_M and T_R. In this approach, all three components were performed on the FPGA device. First, a circuit capable of reading a problem instance and generating appropriate circuitry was developed and preprogrammed into the FGPA. At runtime, a problem instance was placed in memory accessible to the FPGA. Then, the FPGA circuit generator read the problem instance and reconfigured other parts of the device to implement the specialized circuits. They were able to achieve a T_M+T_R time on the order of 10^6 faster than CAD tools and 10^3 faster than software-directed partial reconfiguration.

A separate body of research exists for applications in which the patterns are known in advance rather than being provided at run-time. In an application such as network intrusion detection, where the patterns change infrequently (on the order of days), T_M and T_R are less important. Instead, the goals are to minimize T_E and maximize pattern density, even it requires a lengthy T_M to generate the circuitry.

1.3.2 Brute-Force Approach

The most straight-forward method to building pattern matching circuits is known as the brute-force approach. The brute-force algorithm produces circuits that perform a full comparison of every target pattern against the input in each clock cycle. In other words, no match state is saved across cycles. The input text from a packet payload is broadcast

to all of the pattern matchers and shifted past the target patterns at a rate of one character per clock cycle. A pattern matcher for a length m string contains an m-character shift register for buffering input characters, m character comparators, and a match output that is true when all of the comparators signal a match between the contents of the input shift register and the target pattern. Once the shift register is full, the pattern matcher performs m parallel character comparisons per clock cycle.

The algorithm can be generalized to process i characters per clock cycle by instantiating i copies of each pattern and shifting the content of the buffer i characters in each cycle. To properly detect all possible positions of the pattern in the input stream, each copy of the pattern must start at a different offset relative to the beginning of the input buffer. The first copy starts at offset 0 and the ith copy starts at offset i-1. Cho, et. al. [9] implemented a brute-force design that processed 4 characters (32 bits) per clock cycle. Sourdis, et. al. [10] increased the throughput of Cho's design significantly through aggressive pipelining, which also resulted in increased latency and lower character density.

1.3.3 Finite Automata Approaches

Since some of the hardware pattern matching techniques rely on finite automata, their theory is briefly discussed here in the context of pattern matching. A finite automation (FA) representation of a character string is a directed graph in which the nodes represent match states and each edge is labeled with a character that will cause the associated transition when matched by the input. There is one initial state labeled q and one or more final, or accepting, states labeled f_i. An FA processes an input string and either accepts or rejects it. The string is only accepted if its characters match the labels on any path from the initial state to a final state. In a nondeterministic finite automation (NFA) there is a

null character denoted by ε that matches any input. An NFA can be converted to an equivalent deterministic finite automation (DFA) by eliminating all edges labeled ε and ensuring that all nodes have no more than one outgoing edge labeled with each character.

On a serial processor, there is a tradeoff between the low memory usage of an NFA and the high processing speed of a DFA. Constructing an NFA from a regular expression of length n takes $O(n)$ time and the NFA requires $O(n)$ memory. The NFA processes each input character in $O(n)$ time. If the NFA is then converted to a DFA, the total construction time is $O(2^n)$ and the DFA requires $O(2^n)$ memory, but it is able to process each character in $O(1)$ time. However, there are optimization techniques that significantly reduce the construction time and memory required for the DFA in many cases. Therefore, in practice, software algorithms based on DFAs are more efficient than those based on NFAs.

Sidhu and Prasanna showed that the memory-speed tradeoff can be avoided with FPGAs by presenting a pattern matching approach that has both low construction overhead and optimal performance [11]. This is achieved by directly implementing the NFA in logic. The NFA is constructed in $O(n)$ time, uses $O(n^2)$ FPGA area, and can process one input character on every clock cycle. The key observation is that an FPGA can easily implement the types of NFA transitions that the conversion to a DFA eliminates by storing the state using a one-hot-like encoding. A state with multiple outgoing edges with the same label translates into connecting the output from a state flip-flop to the input of multiple state flip-flops. This enables multiple transitions to occur in a single clock cycle and allows multiple active states. The edges labeled ε in an NFA are implemented by connecting the input of the source flip-flop to the input of the destination flip-flop, which ef-

fectively eliminates the source flip-flop since its output is not connected. This simple transition logic makes an NFA-based matcher more efficient in hardware than a DFA-based matcher. The complex transition logic used by a DFA-based matcher to ensure that there is only one active state requires more circuit area and a longer cycle time. As pointed out by Moscola, Lockwood, et. al. [12], a DFA-based matcher has an advantage over an NFA-based matcher for applications that require the match state to be saved and reloaded since a single active state can be encoded more efficiently than multiple active states. Having multiple active states is undesirable in software because each state must be evaluated serially, increasing processing time. However, multiple active states are not a problem in an FPGA because all states can be evaluated in parallel without an increase in processing time.

An NFA and a corresponding circuit can be constructed for an arbitrarily large regular expression by recursively breaking the expression into sub-expressions and applying some simple rules. Figure 4 shows the corresponding NFA and circuit for each of the standard regular expression building blocks:

(a) This NFA will accept the character ‘c’. The circuit output will be one, if and only if the character comparator output is one and the flip-flop output is one.

(b) This NFA and circuit will match a string containing either of the regular expressions r_1 or r_2. N_i represents the NFA or circuit to implement r_i.

(c) This NFA and circuit will match a string containing r_1 followed by r_2.

(d) This NFA and circuit will match a string containing zero or more occurrences of r_1.

Figure 5 shows how these components can be combined to build an NFA and circuit for the expression ((a|b)*)(cd), which will match a string containing any sequence of zero or more 'a' or 'b' characters followed by the characters 'cd'.

One study [13] demonstrated the feasibility of using an NFA for NIDS pattern matching. They combined patterns from different rules in a subset of the Snort rule database into a single regular expression and then used the approach described above to generate an NFA pattern matching circuit. An interesting contribution of this work was the discovery that patterns with common prefixes cause duplicate circuitry to be generated, and that eliminating the duplicates and sharing a single copy of the circuit allows more patterns to be stored in the FPGA. One drawback of this implementation is that is was not possible to determine which Snort rule was associated with a match since the patterns were combined into a single regular expression. Also, the performance of the design decreased rapidly as the number of stored pattern characters was increased.

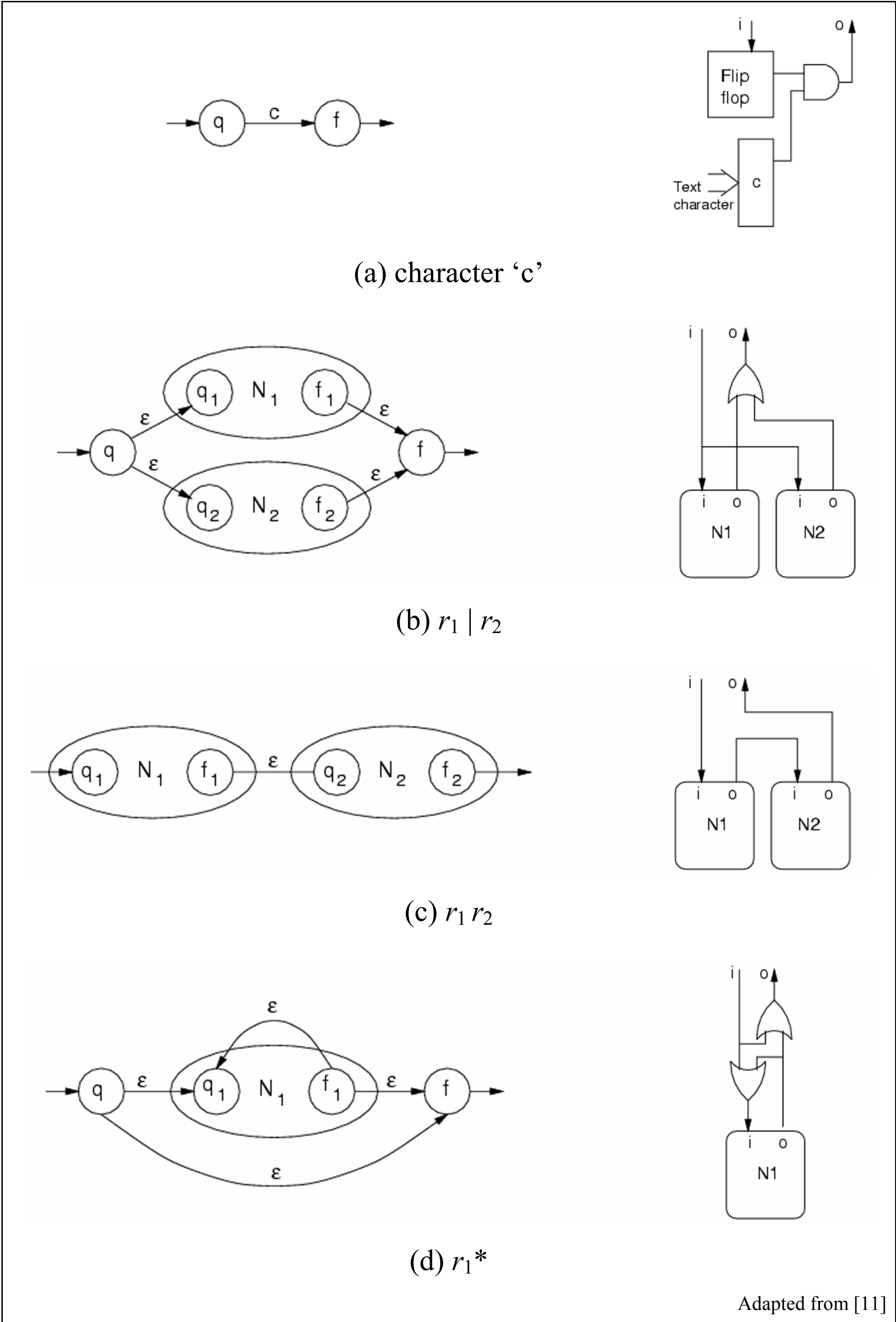


Figure 4: Corresponding NFA and circuit components

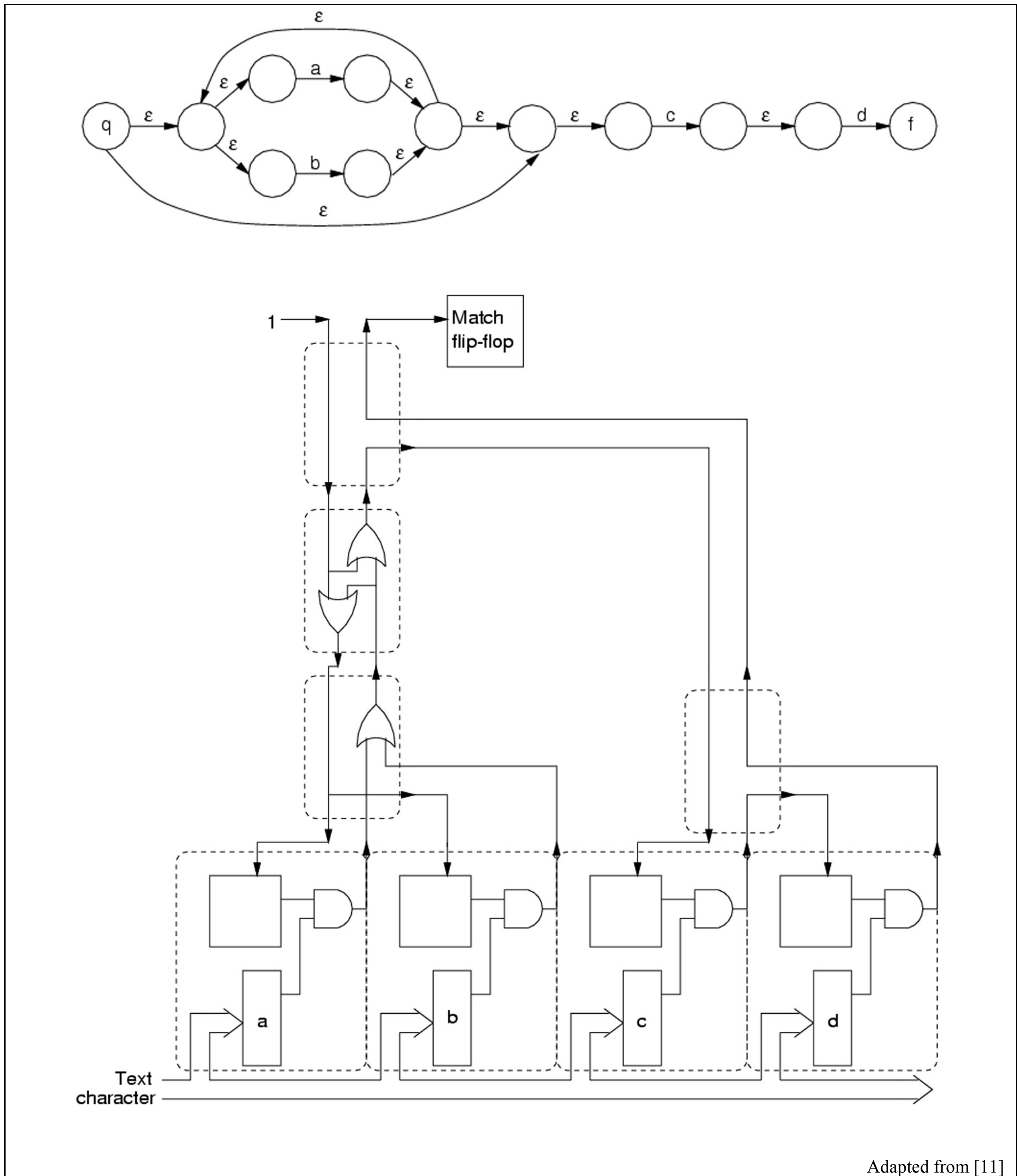


Figure 5: Corresponding NFA and circuit for ((a|b)*)(cd)

2 INTRODUCTION

2.1 Motivation

Despite improvements in software pattern matching algorithms such as those discussed in Section 1.2, content pattern matching is still the most computationally-intensive task of a signature-based IDS. Therefore, in most systems, the overall packet-processing rate of an IDS is directly determined by the time spent in the pattern matching routine for each packet. Often, many rules must be disabled to allow the IDS to keep up with the network line speed, which reduces its effectiveness since potential attacks might be missed. With network speeds continually increasing and new attack signatures being created daily, IDS pattern matching performance is becoming even more critical.

The proposed solution to the pattern matching performance problem presented in this research is to offload the pattern matching task to an FPGA co-processor. An FPGA is well-suited for highly-regular pattern matching operations and can easily outperform a high-end general-purpose CPU running a software pattern matching algorithm, while allowing the CPU to execute other operations in parallel with the pattern matching.

Depending on the circuit design implemented, a single modern FPGA has enough logic to implement hundreds to thousands of individual pattern matching circuits all operating simultaneously. The amount of parallelism available in an FPGA makes the design of an optimal pattern matching algorithm for an FPGA inherently different than that of an optimal algorithm for a sequential CPU. In an FPGA design, it is desirable to minimize the amount of control logic because this enables a higher-throughput data path. An algorithm that simply shifts the pattern by one is often faster than an algorithm that

uses a complex function to determine the shift amount, even though the latter requires less comparison operations.

There are some existing FPGA pattern matching designs that support simple regular expressions and some for domain-specific applications (e.g., searching DNA databases), but there are no designs that support all the requirements of pattern matching for network intrusion detection as described in Section 1.1. This research focuses on the design of efficient, high-speed pattern matching circuits capable of supporting the complex patterns found in network intrusion detection applications.

2.2 The HardIDS Project

This research is part of a project at Georgia Tech known as the Hardware IDS, or HardIDS, project. The goal of this project is to design and develop a complete network intrusion detection system implemented using embedded hardware components consisting of network processors and FPGAs. A prototype of the HardIDS system is being developed using commercial off-the-shelf components consisting of an Intel IXP1200 network processor and Xilinx Virtex and Virtex2 FPGAs. The IXP1200 contains a StrongARM CPU core, which runs an embedded Linux operating system and a modified version of the open-source IDS software, Snort [14]. The IXP1200 also contains six packet processing engines, or microengines, which perform many low-level operations on packets before they are passed to Snort. The FPGA operates on packets in parallel with the IXP1200 and sends information to Snort for analysis. This research includes integrating the FPGA-based pattern matcher with this system.

The flow of data through the various components of the intrusion detection system is shown in Figure 6. The flow starts with Ethernet packets entering the microengines from

the network ports. If a packet is part of a fragmented IP datagram, it is buffered until the full IP datagram can be reassembled. The next stage is packet header matching, in which various header fields are checked to determine whether further analysis needs to be performed. At this point, packets for some protocols are sent to a TCP stream tracking and reassembly module, and others are sent to the FPGA pattern-matcher. The stream-processing module performs protocol-specific processing on the data, and determines when to send the data to the pattern-matcher. The FPGA searches incoming packets and chunks of streams for all of its stored patterns and sends the rule match results to the NIDS analysis engine running on the StrongARM processor in the NP.

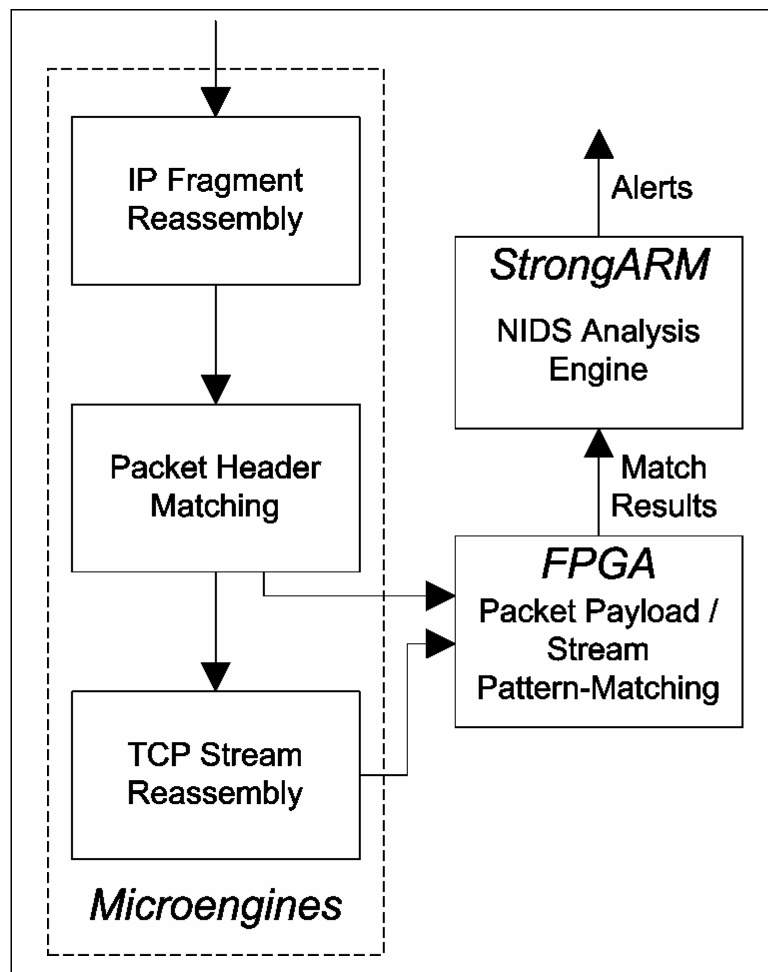


Figure 6: HardIDS data flow

3 IMPLEMENTATION

This section presents the design and implementation of a fully-featured FPGA pattern matching processor for network intrusion detection systems. The key capabilities of the pattern matcher include high pattern character capacity, high throughput, and support for complex pattern specifications.

3.1 Efficient Circuit Design

The primary pattern matching circuit design used in this research builds on the NFA design introduced by Sidhu and Prasanna in [11] and described in Section 1.3.3. A circuit that implements an NFA for pattern matching consists of a pipeline of character match units that processes one input character in each clock cycle. A character match unit consists of a match function and one bit of memory storage. The match function evaluates to true if the current input character matches a preprogrammed character code. The storage element is a flip-flop that stores the value of the previous unit's output (the current match state). The output of a character match unit (the next match state) is determined by the logical AND of the match function output and the flip-flop's output. On a typical FPGA that uses look-up tables (LUTs) to implement logic functions, one way to implement the match function is by performing an 8-bit comparison using two 4-input LUTs and an AND function. Figure 7 illustrates this technique for the character "`a`". Each LUT takes four bits of the 8-bit value of the current character as input and produces a high output if the input matches the programmed character's value. The two LUT output values are used as inputs to an AND gate whose output becomes the output of the match function. Using this approach, one character match unit requires two logic ele-

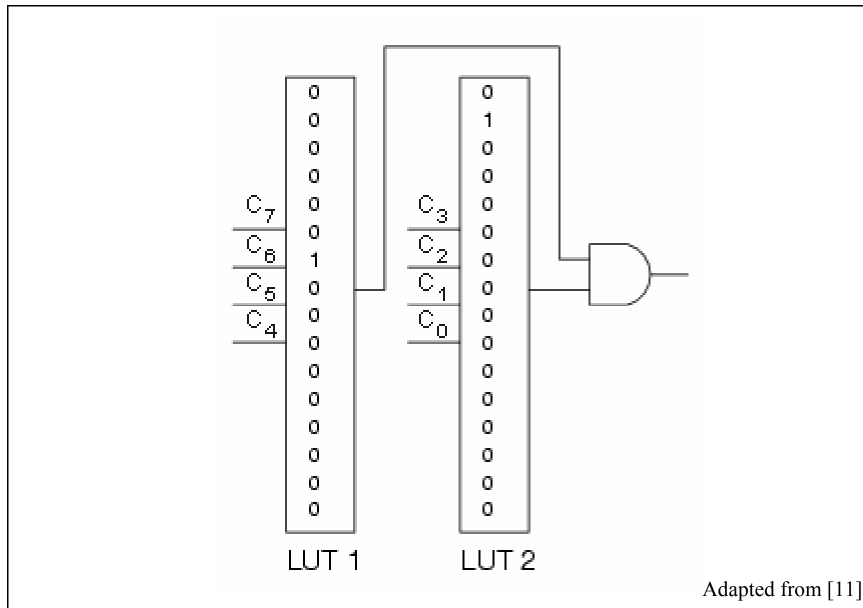


Figure 7: Match function for the character “a” (`0110 0001`)

ments, where a logic element is defined as one 4-input LUT and a flip-flop. The mapping of this type of character match unit to two Xilinx Virtex logic elements is shown in [13]. This thesis demonstrates a novel optimization that allows a character match unit to fit into a single logic element.

The key observation leading to the reduction in the area of a character match unit is that a full 8-bit comparison does not need to be performed by each unit. In fact, with a pattern set containing several thousand characters it is likely that there will be hundreds of identical 8-bit comparisons performed. These redundant comparisons waste valuable logic and routing resources and can be eliminated. To make a match decision, each character match unit only needs a single bit of information—whether or not the input character matches the unit’s programmed target character. Rather than performing distributed comparisons in each unit to obtain this bit of information, it is more efficient to perform all possible comparisons once in a centralized location and send the results to each unit.

For 8-bit characters, this can be achieved by using a shared 8-to-256 decoder and connecting the appropriate one-bit output of the decoder to each unit. By sharing the character comparison results in this way, it is possible to fit a character match unit into one logic element. Compared to the highest-capacity design found in previous works [13], the shared decoder approach doubles the maximum pattern capacity of a given reconfigurable logic device. Figure 8 shows the differences between the two designs for the pattern ‘`snort`’.

In addition to using less *logic* resources, the shared decoder approach uses *routing* resources more efficiently. If C is the number of character match units, the distributed comparators approach requires a total of $8*C$ connections from the input character to the character match units, while the shared decoder approach only requires C connections. Since C increases as the number of patterns is increased and since the character match

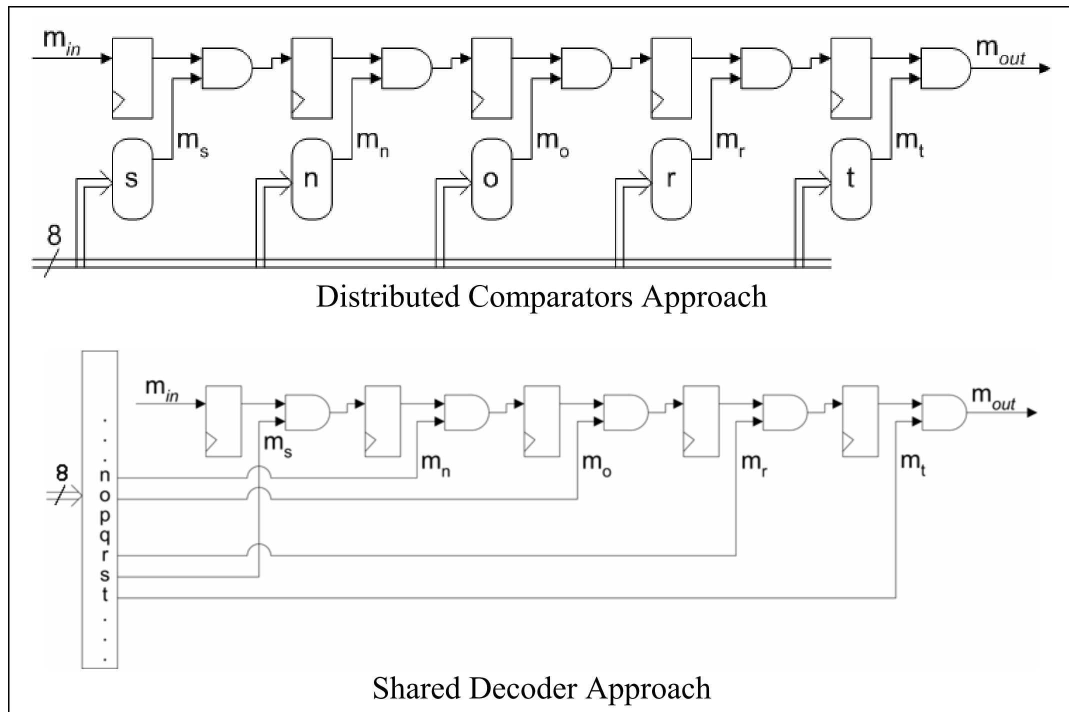


Figure 8: Comparison of NFA design approaches

distribution is in the critical path, the number of routing connections affects the capacity/cycle-time tradeoff. The operating frequency of an FPGA design is determined by how well the place-and-route software can connect logic blocks using a limited number of wires. In densely-packed designs with a large number of connections, congestion forces longer routes to be used for some connections. Thus, the shared decoder approach with eight-times fewer connections scales much better, achieving a higher operating frequency for a given capacity.

A block diagram showing the data path of the complete shared decoder pattern matching module is shown in Figure 9. The design is pipelined to process one character per clock cycle. An input buffer reads incoming 32-bit data words and serializes the bytes to output 8-bit characters. Next, the current character is decoded and match signals are distributed to the individual pattern matching units. Each pattern matching unit has an output that signals when a match is detected. For rules with multiple patterns, all of the corresponding pattern match outputs are passed through an AND gate to generate a rule match output. The rule match signals for all *N* rules are stored in a match vector. After the last character of a packet is processed, the output encoder packs the match results into 32-bit words and sends them to the NIDS analysis engine.

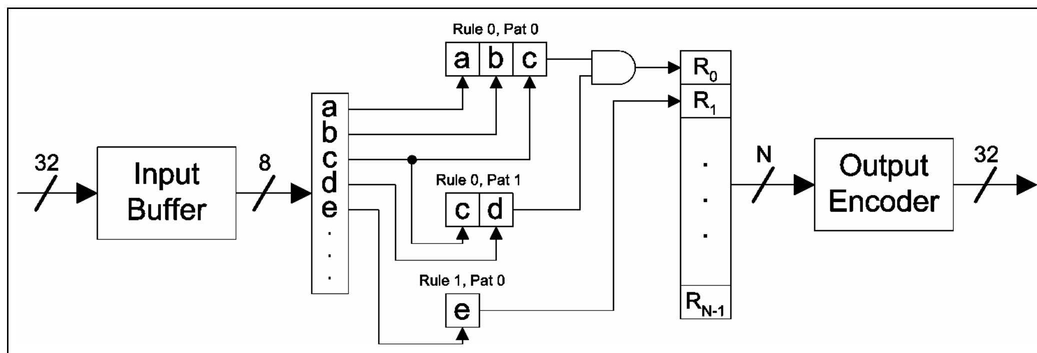


Figure 9: Data path of the FPGA pattern matching processor

3.2 High-Throughput Circuit Design

The previous section introduced optimizations to NFA pattern matching circuit design techniques and showed how they improve the density and throughput of designs that process one input character per cycle. In this section, the optimizations are extended to improve the efficiency of high-throughput circuits that process multiple input characters per clock cycle.

Figure 10 shows the implementation of a circuit using the shared decoder technique to process four input characters per clock cycle. The design requires four character decoders (not shown in the figure), each decoding a different input character. A wire label of the form c_i represents the match signal output of the ith decoder associated with the character code c. Each of the rows in the circuit is an NFA that matches the target pattern starting at one of the four possible offsets. The columns of flip-flops are pipeline registers. The OR of the match signal outputs from all rows represents the match signal for the target pattern.

In general, each FPGA logic element (LE) in a circuit using the shared decoder design can implement up to a four-input gate and a flip-flop, so a single LE can match from one to four characters. Thus, the number of LEs used to implement i character comparisons is $\left\lceil \frac{i}{4} \right\rceil$, which simplifies to $\frac{i}{4}$ when i is a power of two. The total number of LEs used to implement each stage of the NFA pipeline for all i offsets is $\left\lceil \frac{i}{4} \right\rceil \cdot i$ or, when i is a power of two, $\frac{i^2}{4}$.

A similar circuit could be implemented based on the distributed comparator approach. As mentioned in Section 3.1, two LEs are required to implement each character comparison with this technique. Therefore, the number of LEs used for i comparisons is $2i$, and the number used in each stage of i offsets is $2i^2$. Based on these equations, the character decoder approach uses 8 times fewer LEs for circuits that process more than one character per cycle. It also requires 8 times fewer routing connections. The character decoder design has $n \cdot i$ connections, while the distributed comparator design has $8n \cdot i$ connections.

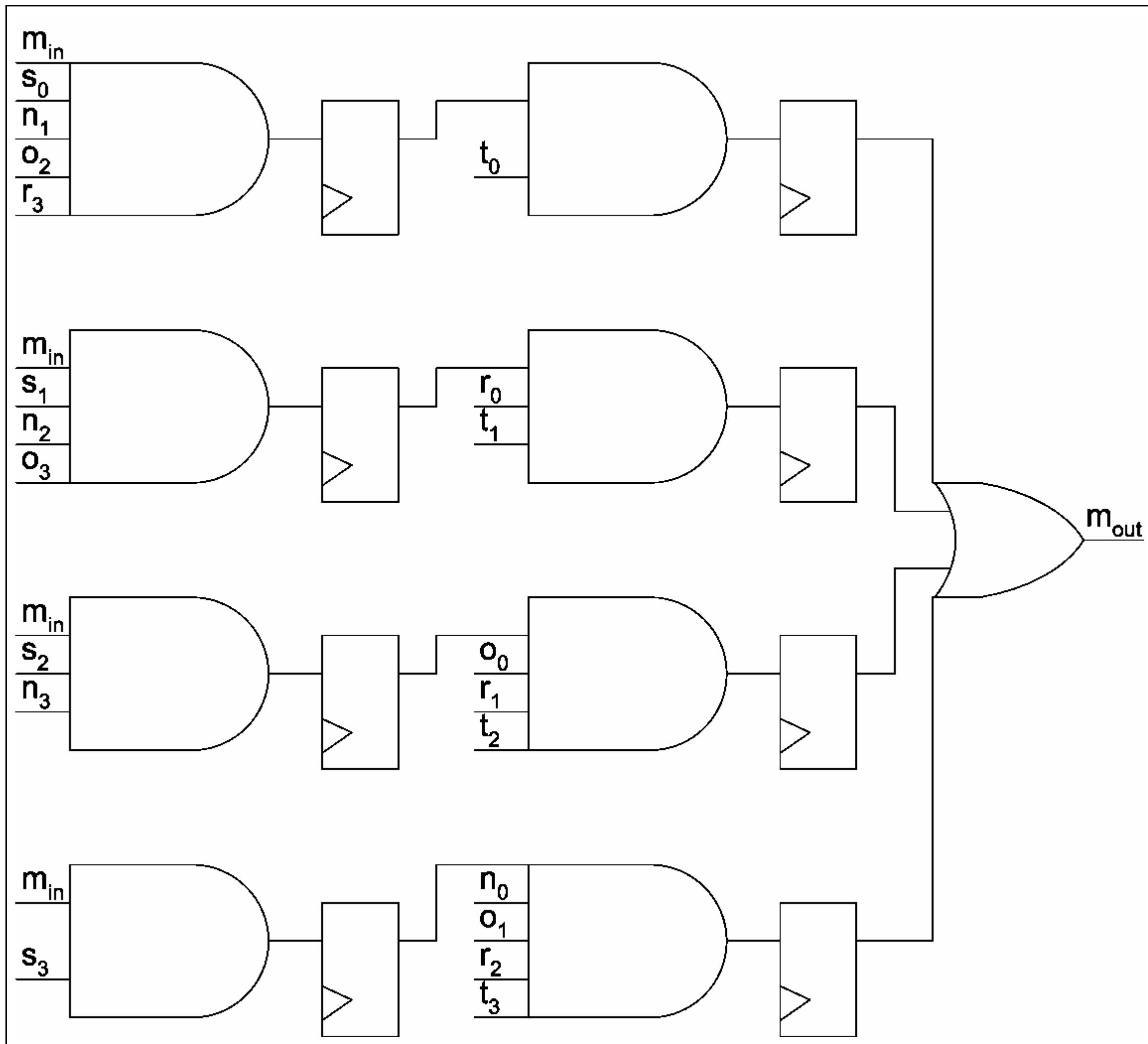


Figure 10: Four-way parallel decoder NFA circuit for pattern "`snort`"

3.3 Support for Complex Patterns

This section discusses some extensions to NFA notation and the corresponding circuit representations that allow for more complex regular expressions to be represented efficiently in reconfigurable logic. The implementation of these extensions allows the FPGA design to support all the pattern matching options of the Snort rule language [15].

3.3.1 Case Insensitivity

The case-sensitivity of comparisons is important for intrusion detection. For case-insensitive network applications, it is crucial that the NIDS analyze patterns in the same way to prevent the evasion technique of using non-standard capitalization. Case-sensitivity can also be used to help avoid false positives. The Snort *nocase* rule option indicates that a pattern should be matched using case-insensitive comparisons; if it is not present in a rule, then case-sensitive comparisons are performed.

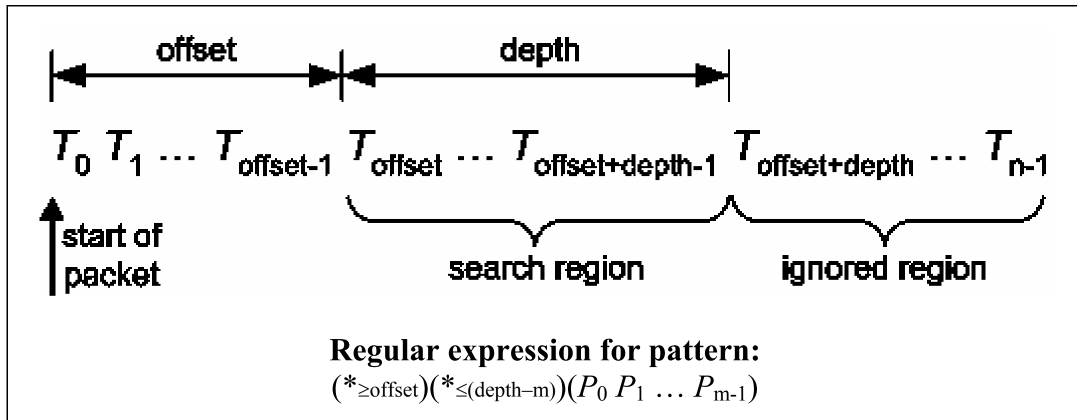
A case-insensitive character comparison can be achieved by performing two comparisons—one for the lower case character and one for the upper case character. In logic, this would translate into two character match units with their outputs routed into an OR function to generate the case-insensitive match signal. However, there is a more efficient method that accomplishes a case-insensitive comparison using a single character match unit. This technique takes advantage of the fact that the upper-case and lower-case ASCII codes for each letter differ by only one binary digit. Using the distributed comparator approach, a case-insensitive comparison can be implemented by having two one values in the LUT for the high-order bits of the character. Using the shared decoder approach, a case-insensitive comparison is implemented by taking the OR of the match sig-

nals for the lower-case and upper-case characters as the current match input. With either approach, resources are saved by eliminating the need for a second character match unit.

3.3.2 Bounded-length Wildcards

The use of bounded-length wildcards allows ordering and spacing information to be specified for multiple sub-patterns within an attack. This makes it possible for a single rule to detect multiple variations of a well-documented attack by instructing the NIDS to look for the invariant parts of an attack while ignoring other parts of the data. Another potential benefit is a reduction in false positives by restricting the searching to a sub-section of a packet based on knowledge of the application protocol.

In the Snort language, the *`offset`* and *`depth`* rule options are used to specify a search region (a range of allowable pattern positions) relative to the beginning of a packet. Figure 11 illustrates the usage of these options. The *`distance`* and *`within`* rule options are functionally identical, but their values are specified relative to the end of the previous pattern in the rule. A wildcard sequence with a lower-bounded length is used to implement *`offset`* and *`distance`*, while a sequence with an upper-bounded length is used to implement *`depth`* and *`within`*.



Regular expression for pattern:

$(*_{\geq \text{offset}})(*_{\leq (\text{depth}-\text{m})})(P_0\ P_1\ \ldots\ P_{\text{m-1}})$

Figure 11: Use of position rule options

Earlier work has shown how to implement wildcards for zero or more characters [11] and zero or one characters [13] using NFA circuits. This section shows how to implement wildcards whose lengths are specified by an arbitrary lower bound or upper bound. First, some notation must be introduced. In regular expressions, $(*_{\geq n})$ is used to denote any character sequence with a minimum length of n characters and $(*_{\leq n})$ is used to denote any character sequence with a maximum length of n. In NFA transition diagrams ε^* is used to represent a character that matches the same characters as ε, but whose transition cannot be eliminated in the conversion to logic, and $\neg(c)$ is used to label a transition that is taken for any character other than the character c. Examples of NFAs using this notation and the corresponding circuits that implement them are shown in Figure 12 and Figure 13.

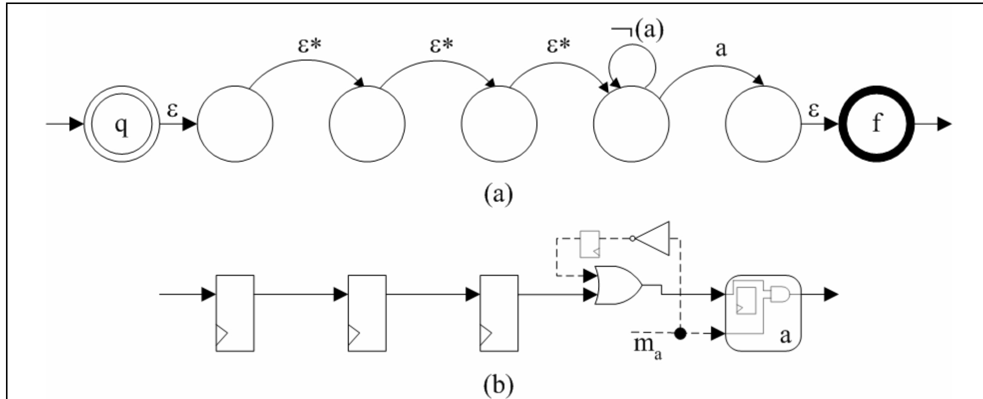


Figure 12: Wildcard with lower-bounded length
(a) NFA and **(b)** circuit representation of the regular expression $(*_{\geq 3})(a)$

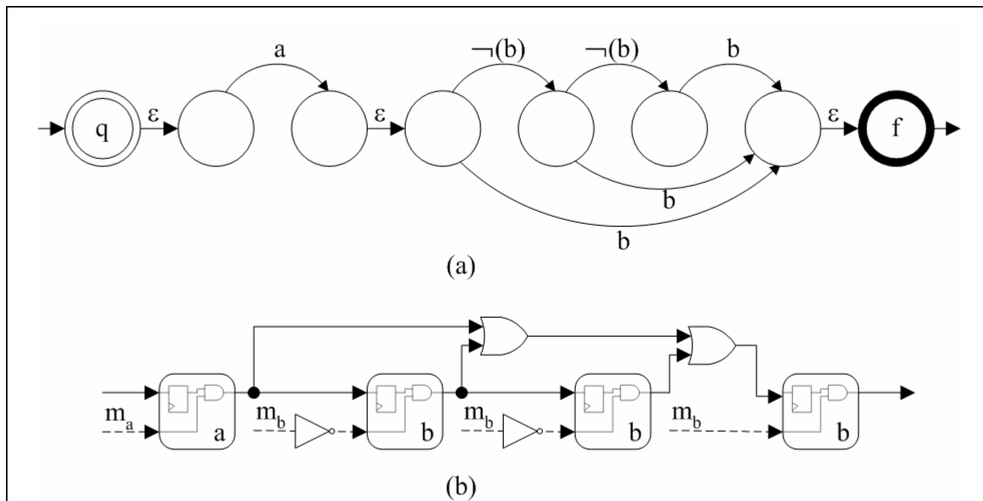


Figure 13: Wildcard with upper-bounded length
(a) NFA and **(b)** circuit representation of the regular expression $(a)(*_{\leq 2})(b)$

3.4 Approximate Pattern Matching

Sometimes it is desirable to allow a small amount of mismatching in the pattern matching process. This is useful for detecting an attack pattern that is expected to contain some variable content, but the exact variations are unknown or too numerous to list. It can also help detect new exploits that are similar to known exploits. Regular expressions with bounded-length wildcards and approximate matching are complementary techniques; the former is applicable to patterns with predictable variation, while the latter is suitable for patterns with more uncertainty.

Formally, approximate matching is known as the k-differences problem. Given a pattern P of length m and a text string T of length n, the task is to find any character sequence in T that differs from P by at most k characters. This problem has been studied in various contexts, including the development of a bit-parallel simulation of an NFA in [16] and its adaptation to a misuse detection system for a multi-user computer in [17].

This thesis presents an NFA circuit capable of detecting approximate matches where the data may contain character substitutions, insertions, or deletions. Figure 14 depicts an NFA for the pattern "`abcd`" allowing two or less differences. The notation used is based on that in [16]. The NFA has $(m+1)(k+1)$ states named using ordered pairs of the form (i,j), where $i \in [0,k]$ and $j \in [0,m]$. State (0,0) is the initial state and there are $k+1$ final states: $(0,m)$, $(1,m)$ … (k,m). Transitions between states are labeled with the character that enables them. Unlabeled transitions are enabled for any character in the alphabet. As drawn in Figure 14, horizontal transitions indicate character matches, vertical ones are character insertions, solid diagonals represent character substitutions, and dashed diagonals are used for character deletions.

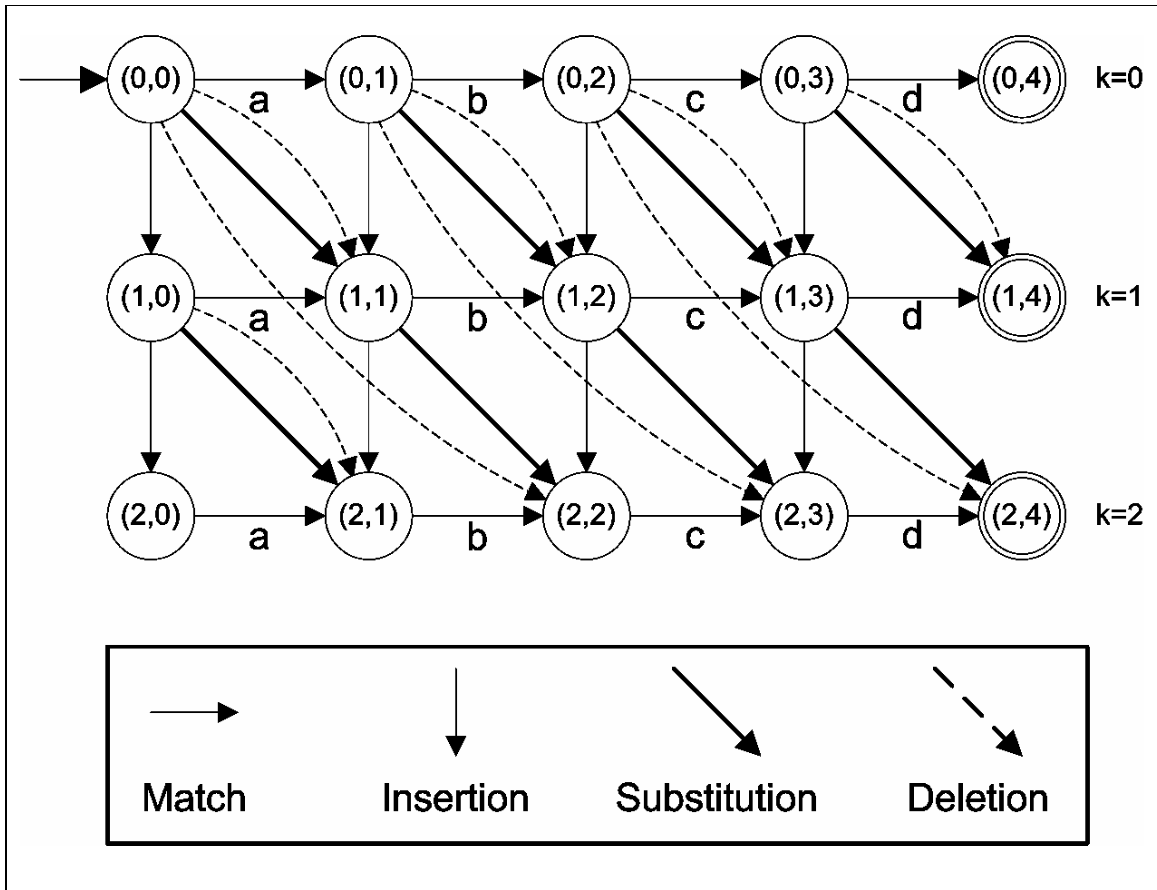


Figure 14: NFA for "`abcd`" with $k \leq 2$

Figure 15 shows how the NFA in Figure 14 is implemented in a circuit. Notation similar to the NFA is used to illustrate the different types of transitions—thin lines for insertions, bold lines for substitutions, and dashed lines for deletions. Notice that the states in the column for j=0 do not need to be implemented. State (0,0) is not stored because it is always equal to the value of input to the NFA. The other states in the column can only be reached by insertions before the first character. In NIDS pattern matching, by default, any number of characters may occur before the start of the pattern. Therefore, these characters are not considered insertions and the associated states can be ignored.

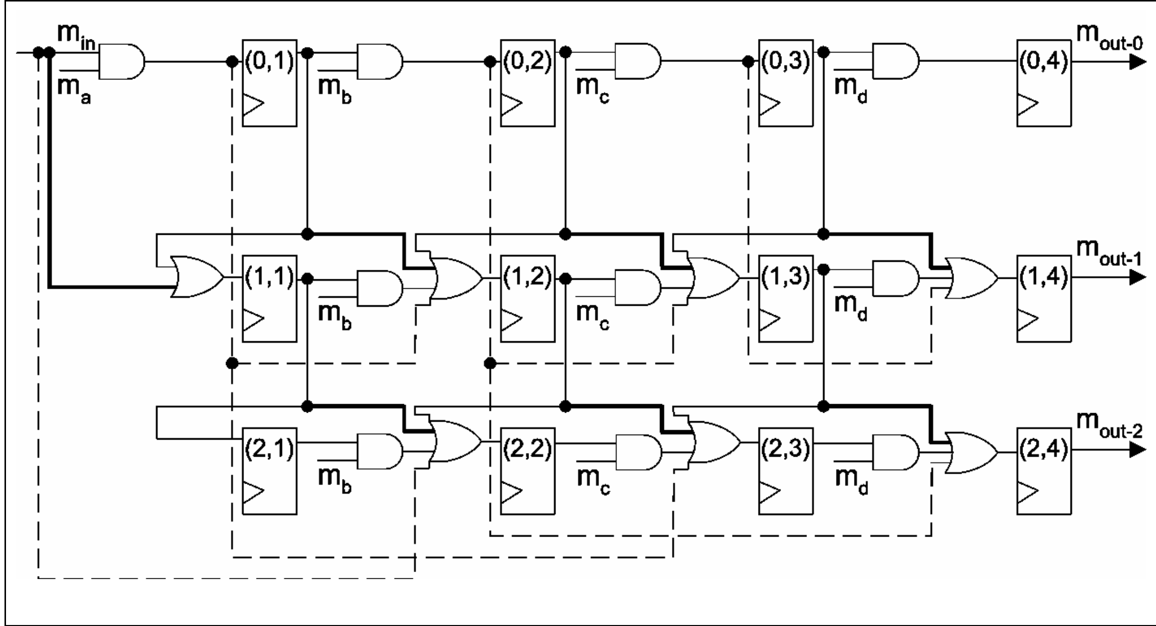


Figure 15: Circuit for "`abcd`" with $k \leq 2$

The circuit has k+1 outputs corresponding to the final states of the NFA. It is possible for multiple outputs to signal a match, but only the closest match result should be saved. This is achieved by routing the outputs through a priority encoder with the signal for k=0 given the highest priority. The encoded k value is returned to the NIDS analysis engine with the rest of the match results.

A simple analysis of the state diagram in Figure 14 shows its complexity to be $O(k(m+1))$. In other words, increasing k by one increases the total number of states by m+1. Similarly, the circuit area scaling factor is $O(km)$.

3.5 Protocol Analysis

Many ASCII-based Internet protocols (e.g. HTTP, FTP, SMTP) use a similar format. The basic structure consists of a command, followed by whitespace, then followed by one or more arguments, and finally terminated with a newline character. The efficiency and

robustness of a NIDS can be improved by decoding this format and analyzing different portions independently. A good example of this is found in the processing of HTTP requests. The command portion of the request, in which most attacks are found, may contain a couple hundred bytes, while the data portion, which is usually benign, may be several thousand bytes long. By constraining pattern matching to only the command portion, the NIDS can increase throughput while eliminating false alarms for data that looks like attacks. This functionality is implemented in the Snort rule language with the `uricontent` option. The Universal Resource Indicator (URI) portion of the HTTP header occurs after a method command and is followed by the HTTP version:

```
<method> <URI> <HTTP-version>
```

The pattern-matcher decodes the header line by looking for all of the possible method patterns followed by any number of whitespace characters. After a method match is found, all the pattern matchers for `uricontent` patterns are enabled. These pattern matchers are subsequently disabled whenever the next white space character is reached. The same structure can also be used to detect buffer overflow attempts by generating an alert whenever the length of an argument exceeds the allowable size. This design is shown in Figure 16.

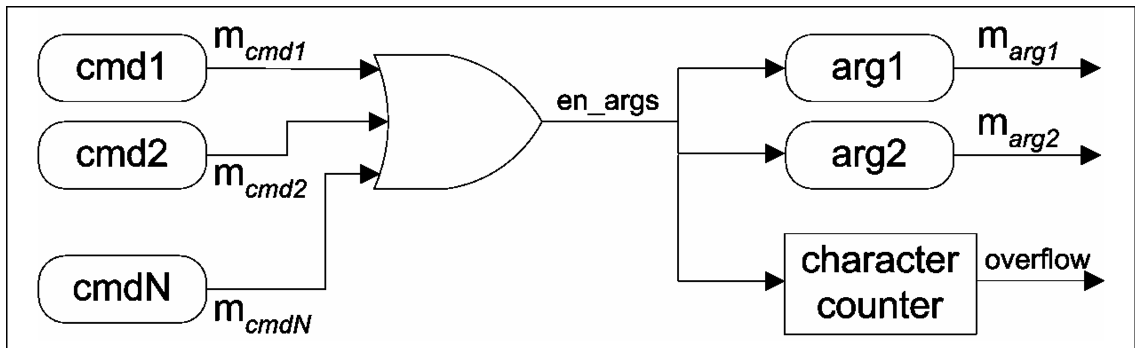


Figure 16: Protocol analysis

3.6 Automated Circuit Generation

Software has been developed that translates a Snort rule file into a circuit description for matching its content strings following all the semantics of the rule options. The tool is written in Java and consists of two main components—a rule file parser and a circuit generator. The parser converts Snort rules into an internal representation that is used as input to the circuit generator. The circuit generator uses the Java Hardware Description Language (JHDL) [18] to specify FPGA circuit components and connect them to implement the pattern matchers. Several circuit generators have been written that use different algorithms for designing circuits.

3.7 System Integration

Integrating the FPGA pattern matcher with the HardIDS system described in Section 2.2 required modifications to some existing components as well as the development of new hardware and software components. The details of the work are discussed here.

Since the FPGA and IXP are physically located on different boards, all communication must take place over the PCI system bus. Shared data is stored in single-ported SRAM on the FPGA board, which means that synchronization using locking operations is required. There are four SRAM banks that can be independently locked by either the FPGA or the IXP.

The communication between the FPGA and the IXP consists of two main actions—the IXP sending packets to the FPGA and the FPGA sending match results to the IXP. The data transfers in the IXP to FPGA direction (downstream) are much larger than those in the FPGA to IXP direction (upstream). Therefore, two SRAM banks are used for downstream data and one bank is used for upstream data. The FPGA and IXP coordinate

locking the downstream banks so that the FPGA can read packets from one bank while the IXP writes packets to the other bank.

Software has been developed for the IXP StrongARM to lock an SRAM bank on the FPGA and write data into the bank using direct memory access (DMA) transfers. FPGA circuit modules have been developed to lock a downstream SRAM bank, read packets from the bank into an on-chip buffer, send a character stream through the pattern matching modules, and write the match results into the upstream SRAM bank. Finally, software has been developed for the StrongARM to read the FPGA match results from the upstream bank.

The Snort software running on the StrongARM has been modified to use the FPGA's pattern matching results instead of running Snort's pattern matching algorithm. Changes have been made to the Snort detection engine to combine the RTN matches from the microengines with the pattern matches from the FPGA. For each matching RTN, the OTN list is traversed and the rule options are checked like normal. Whenever a content option is reached, Snort consults the FPGA match results rather than calling the software pattern matching routine.

4 EVALUATION

4.1 Comparison of Circuit Design Approaches

To provide a basis for comparing different pattern matching circuit design approaches, several circuit generators were developed and tested. One circuit generator was designed to use the brute-force algorithm. Two circuit generators were developed that produce NFA circuits using an algorithm based on distributed characters similar to that described in [13] and Section 1.3.3. The produced circuits differ from the referenced design in that they have a match signal output for each rule. The first distributed comparator generator implements a full pattern comparator for each pattern, while the second generator uses a prefix tree to reduce redundant circuitry for patterns with common prefixes. Also, three circuit generators were implemented that produce NFA circuits based on the shared decoder approach developed in this work. The first generator creates a full pattern comparator for each pattern, the second combines patterns using a prefix tree, and the third uses a prefix tree and supports the bounded wildcard options. Finally, a parallel decoder NFA circuit generator was developed that matches four characters per clock cycle.

All of the different designs were tested to determine their FPGA logic resource usage and supported clock rate. Every design performed both case-sensitive and case-insensitive comparisons as indicated by the Snort rules. The circuits were pipelined to process at least one input character every clock cycle. Each of the designs was tested with various-sized sets of rules from the Snort 2.0 rule database. The set with all of the default rules provided in the software distribution contained 17,537 characters. For all tests, the JHDL tools were used for synthesis, and the Xilinx Foundation tools were used for mapping, placement, and routing.

4.1.1 Capacity Scalability

Tests were performed to determine the relative capacity scalability of all the design approaches that process a single character per clock cycle. The test platform used was a Xilinx Virtex-1000 FPGA, which is a one-million-gate equivalent chip. The same compilation parameters were used for every experiment, and included a target clock rate of 100 MHz, which is the maximum frequency supported by the FPGA platform. The designs were verified to produce correct output when running on the hardware.

The speed and area results for the designs are presented in Table 1. The numbers show that all of the designs except the distributed comparators with prefix tree met the speed goal of 100 MHz as long as the logic element usage was less than 95 percent. The reason that this design could not achieve high clock speeds is that the prefix tree introduces multiple levels of high fan-out match signals, which place more demand on the limited routing resources that must be shared with the character broadcast signals. The prefix tree does not limit the speed of the shared decoder design because there are plenty of routing resources available due to the more efficient implementation of the character match units.

The distinguishing property between the designs is their character capacity, which varies significantly among the approaches. This is clearly illustrated in Figure 17, which plots the logic element usage against the number of pattern characters. All of the designs have approximately linear increases in logic usage as the number of characters is increased. The important message portrayed by the graph is the relative slopes of the lines, which indicate the scalability of the designs. The use of a prefix tree improves efficiency somewhat due to the elimination of duplicate logic, but it is apparent that the use of a shared character decoder is the key to providing the scalability necessary to achieve ma-

jor gains in capacity. Another point indicated by the graph is that support for wildcard options adds a constant overhead to the shared decoder with prefix tree implementation.

The throughput of a design is calculated by multiplying the clock frequency by the data width (8-bits). For a design running at 100 MHz, the throughput is 800 Mb/s. The latency for each data packet is a determined by a fixed overhead and the size of the packet (n). For each packet, there is a 21-cycle setup time, an n-cycle match time, and a 39-cycle output time. For a design running at 100 MHz, the latency is 1.2 μs for a 64-byte packet and 15.6 μs for a 1500-byte packet. However, due to the pipelined design, the output overhead can be hidden with a sequence of packets because the input processing for a packet can begin as soon as the matching stage for the previous packet is complete and the match results have been sent to the output stage.

Table 1: Speed and area comparison of different design approaches (Virtex-1000)

Number of Chars	Brute Force		Comparators		Comparators with Prefix Tree		Decoder		Decoder with Prefix Tree		Decoder with Prefix Tree & Wildcards	
	Area (Slices)	Freq (MHz)	Area (Slices)	Freq (MHz)	Area (Slices)	Freq (MHz)	Area (Slices)	Freq (MHz)	Area (Slices)	Freq (MHz)	Area (Slices)	Freq (MHz)
2,001	>100%	-	39%	100.8	30%	83.3	16%	100.6	17%	101.7	20%	100.4
4,012	-	-	75%	100.9	52%	73.9	30%	101.4	25%	102.1	38%	100.3
7,996	-	-	-	-	99%	67.1	52%	100.6	42%	101.1	54%	100.1
17,537	-	-	-	-	-	-	99%	82.3	80%	100.1	96%	76.9

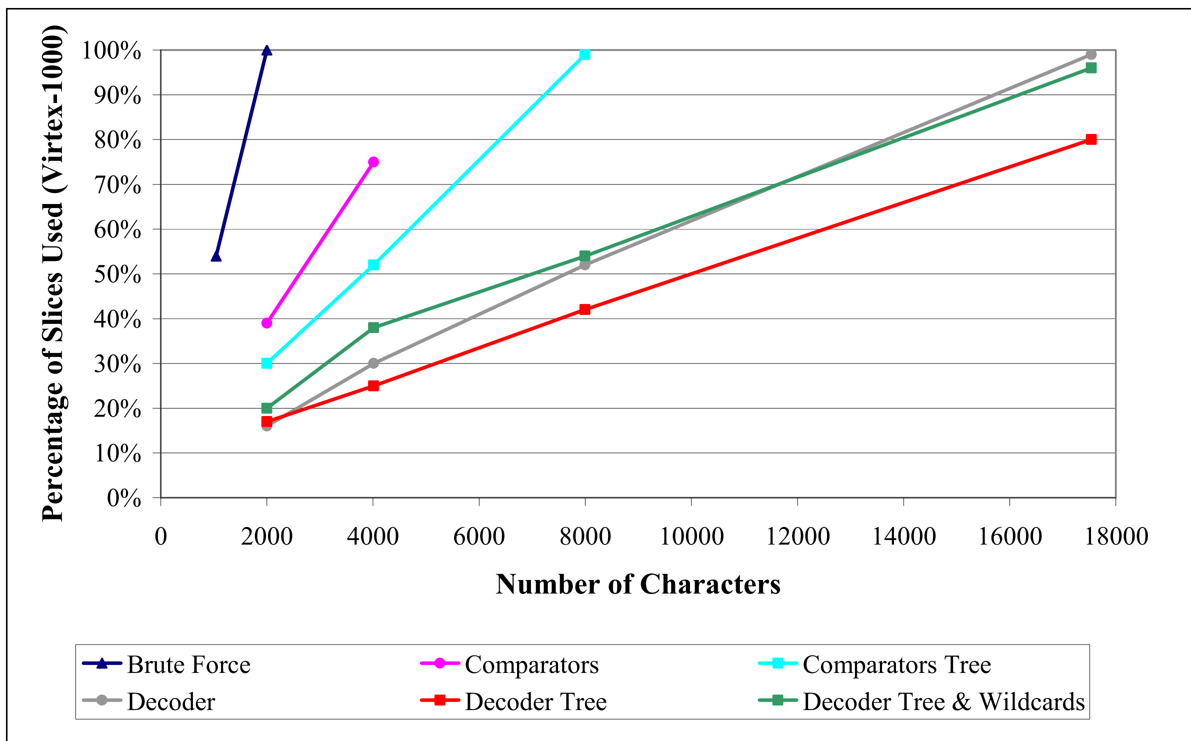


Figure 17: Capacity comparison of different design approaches (Virtex-1000)

4.1.2 Throughput Scalability

With the availability of FPGA devices with more available logic resources than are needed to implement a full set of rules, it is interesting to explore the tradeoffs associated with processing multiple input characters simultaneously at the expense of increasing the amount of logic required for each pattern matcher. To demonstrate the throughput scalability of the character decoder design, tests were performed using a Xilinx Virtex2-6000 FPGA. Circuits were generated for processing one character per clock cycle and four characters per clock cycle. Table 2 presents the area and throughput results from these tests. A comparison of the numbers for the two designs indicates that increasing the input width from one to four characters results in approximately a four times increase in area and throughput. This linear scalability is expected to continue for larger input widths, allowing larger FPGAs to support higher throughput.

Table 2: Area and throughput for different input widths (Virtex2-6000)

Number of Characters	1 character (8 bits) per cycle			4 characters (32 bits) per cycle		
	Area (Slices)	Freq (MHz)	Throughput (Mbps)	Area (Slices)	Freq (MHz)	Throughput (Mbps)
2,001	5%	250.2	2002	17%	234.1	7491
4,012	8%	246.4	1971	28%	207.8	6650
7,996	14%	227.6	1821	44%	200.8	6426
17,537	28%	192.0	1536	81%	189.9	6077

4.2 Comparison with Previous Work

This section compares the results of this research with the results of previous work on FPGA pattern matchers for network security applications. The chosen bodies of work represent the highest-performing implementations of each algorithmic approach—brute-force, DFA, distributed comparator NFA, and shared decoder NFA. The metrics used for comparison are throughput and capacity. The throughput (in Megabits per second) of a design is calculated by multiplying the amount of input data (in bits) processed per cycle by the maximum clock frequency (in Megahertz). The capacity of a design is the number of characters that can be programmed into a given FPGA device. For comparison purposes, a device-neutral metric called logic elements per character (LEs/char) is used. This figure is determined by dividing the total number of logic elements used in a design (including the input and output circuitry) by the sum of the lengths of all patterns programmed into the design. A logic element is the fundamental unit of FPGA logic and consists of a four-input look-up table and a flip-flop. Since throughput and capacity generally have an inverse relationship, any comparison of designs must consider both metrics. Here, a value called *Performance* is defined as throughput times density. Density is the character density, or the reciprocal of LEs/char. *Performance* increases as throughput and density increase and decreases as throughput and density decrease. Therefore, a design with higher *Performance* provides a better tradeoff between throughput and area, or, in other words, a smaller increase in area as throughput is increased.

Table 3 presents a comparison summary of previous work with this work. The results show that the character decoder circuits have the best Performance for both 8-bit and 32-bit input widths.

Table 3: Comparison with Previous Work

Category	Work	Input Width	Device	Freq (MHz)	Throughput (Mbps)	LEs	Chars	LEs/char	Performance
Brute Force	**Cho, Navab, Mangione-Smith [9]**	32	Altera EP20K	90.0	**2,880**	17,000	1,611	**10.6**	**272.9**
Brute Force	**Sourdis, Pnevmatikatos [10]**	32	Virtex-1000	171.0	**5,472**	8,132	489	**16.6**	**328.9**
		32	Virtex2-1000	344.0	**11,008**	8,132	489	**16.6**	**661.7**
		32	Virtex2-6000	252.0	**8,064**	47,686	2,457	**19.4**	**415.5**
DFA	**Moscola, Lockwood, et. al. [12]**	8	VirtexE-2000	37.0	**296**	2,318	420	**5.5**	**53.6**
		32	VirtexE-2000	37.0	**1,184**	8,134	420	**19.4**	**61.1**
Comparator NFA	**Franklin, Carver, Hutchings [13]**	8	Virtex-1000	30.9	**247**	20,618	8,003	**2.6**	**96.0**
		8	VirtexE-2000	52.5	**420**	20,618	8,003	**2.6**	**163.0**
		8	VirtexE-2000	49.5	**396**	40,232	16,028	**2.5**	**157.8**
Decoder NFA	**Clark, Schimmel**	8	Virtex-1000	100.1	**801**	19,698	17,537	**1.1**	**712.9**
		8	Virtex2-6000	192.0	**1,536**	19,698	17,537	**1.1**	**1367.5**
		32	Virtex2-6000	189.9	**6,077**	54,742	17,537	**3.1**	**1946.7**

4.3 Complexity of Approximate Matching

The logic complexity of approximate matching circuits was measured experimentally. Three sets of Snort rules containing approximately 2000, 4000, and 8000 characters were used. For each set, a test was run with approximate matching applied to all the patterns for the following values of k: 0, 1, 2, and 4. The results are shown in Figure 18. The dashed lines show the theoretically-predicted values. As expected, the design scales near the rate of km.

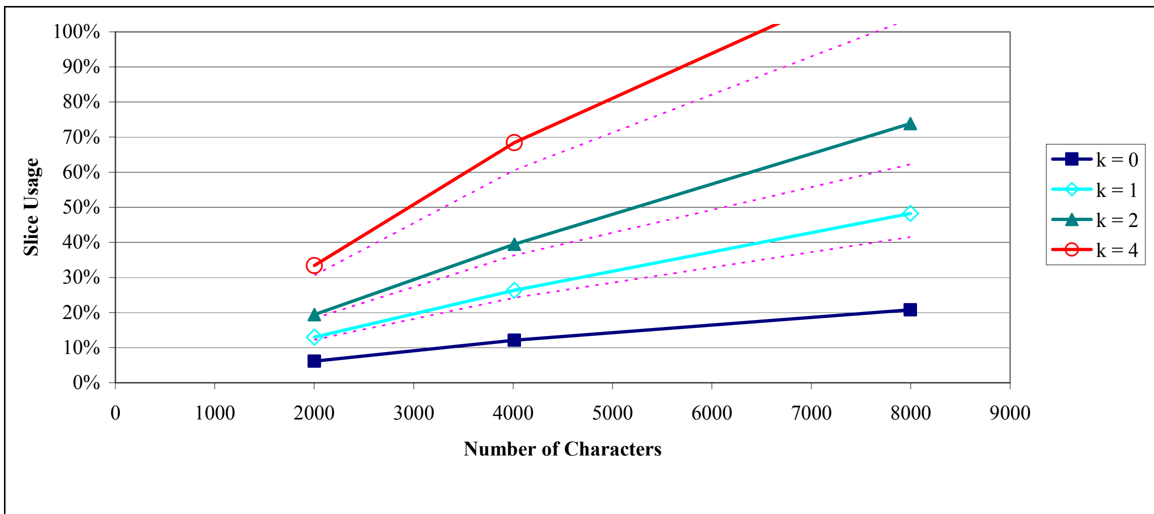


Figure 18: Complexity of approximate matching circuits (Virtex2-6000)

REFERENCES

[1] D.E. Knuth, J.H. Morris, and V.R Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, Vol 6, No 2, pp. 323-350, June 1977.

[2] R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, Vol 20, No 10, pp. 66-72, Oct. 1977.

[3] G. Davies and S. Bowsher, "Algorithms for Pattern Matching," *Software—Practice and Experience*, Vol 16, No 6, pp. 575-601, June 1986.

[4] Alfred V. Aho and Margaret J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search", *Communications of the ACM*, Vol 18, No 6, pp. 333-340, June 1975.

[5] Mike Fisk and George Varghese, "Fast Content-Based Packet Handling for Intrusion Detection," *Technical Report UCSD CS2001-0670*, May 2001.

[6] C. Jason Coit, Stuart Staniford, and Joseph McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *DARPA Information Survivability Conference and Exposition*, June 2001.

[7] R.P.S. Sidhu, A. Mei, and V.K. Prasanna, "String Matching on Multicontext FPGAs using Self-Reconfiguration," *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 217-226, Feb. 1999.

[8] B. Gunther, G. Milne, and L. Narasimhan, "Assesing Document Relevance with Run-time Reconfigurable Machines," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 10-17, Apr. 1996.

[9] Young H. Cho, Shiva Navab, and William H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," *Proceedings of 12th International Conference on Field Programmable Logic and Applications*, 2002.

[10] Ioannis Sourdis and Dionisios Pnevmatikatos, "Fast, Large-Scale String Match for a 10 Gbps FPGA-based Network Intrusion Detection System," *Proceedings of the 13th International Conference on Field Programmable Logic and Applications*, Sept. 2003.

[11] R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching using FPGAs," *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. Apr. 2001.

[12] James Moscola, John Lockwood, Ronald P. Loui, Michael Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 31-38, Apr. 2003.

[13] R. Franklin, D. Carver, and B.L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 111-120, Apr. 2002.

[14] Martin Roesch, "Snort – Lightweight Intrusion Detection for Networks," *USENIX LISA Conference*, Nov. 1999.

[15] Martin Roesch and Chris Green, "Snort User's Manual". Available at http://www.snort.org/docs/writing_rules/.

[16] Ricardo Baeza-Yates and Gonzalo Navarro, "Faster Approximate String Matching," *Algorithmica* 23(2), pp. 127-158, 1999.

[17] J. Kuri and G. Navarro, "Fast Multipattern Search Algorithms for Intrusion Detection", *Proceedings of String Processing and Information Retrieval (SPIRE 2000)*, pp. 169-180, 2000.

[18] P. Bellows and B.L. Hutchings, "JHDL—An HDL for Reconfigurable Systems," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 175-184, Apr. 1998. Software available at http://www.jhdl.org.