In Proceedings of *IEEE International Conference on Field-Programmable Technology (FPT)*, Tokyo, Japan, December 2003.

A Pattern-Matching Co-Processor for Network Intrusion Detection Systems

Christopher R. Clark and David E. Schimmel[†]
School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA, USA
{cclark, schimmel} @ ece.gatech.edu

Abstract

This paper explores the design and analysis of an FPGA module that implements pattern-matching functionality for the network intrusion detection problem. The specific features of the pattern-matcher include support for complex regular expressions and approximate matching with bounded substitutions, insertions, and deletions. A module generator is presented that utilizes non-deterministic finite automata to dynamically create efficient circuits for matching patterns specified with a standard rule language. The logic complexity and performance of the generated circuits is measured and analyzed. Results indicate our techniques yield circuits that are more than twice as dense as other reported designs, while maintaining the throughput necessary for processing at gigabit line speeds and beyond. The FPGA pattern-matching processor is integrated with other hardware and software components to form a complete network intrusion detection system.

Keywords: network intrusion detection, pattern matching, non-deterministic finite automata (NFA), FPGA, network processor

1. Introduction

Network intrusion detection systems (NIDS) have become critical components of network security systems. This is due, in part, to the availability of automatic attack tools, which has led to a rapidly growing rate of reported security incidents [1]. This trend, coupled with the fact that network traffic is increasing faster than computer performance [2], places an overwhelming burden on NIDS. Current NIDS are struggling to keep up with traffic rates above 100 Mb/s, while 1 Gb/s networks are becoming common and 10 Gb/s networks are being deployed. It is clear that the gap between network traffic rates and NIDS analysis rates must be addressed. Various previous works have studied the use of reconfigurable hardware to accelerate network security applications [3,4]. In this paper, we focus on speeding up and enhancing pattern matching for very large data sets.

On our existing hardware platform, the performance of our FPGA design exceeds the I/O capabilities of the 1Gbps Ethernet and PCI interfaces. As a consequence, we do not focus on further improvements in pattern matching throughput in this paper. However, we are currently investigating variations on our current designs that will scale the performance of our FPGA pattern matcher to 10 Gbps Ethernet rates and beyond. These results will appear in a later paper.

The remainder of this paper is structured as follows: section 2 provides some background information and related work; section 3 describes the FPGA module design; section 4 presents the features of the reconfigurable pattern-matcher; section 5 discusses integration of the pattern-matcher with a NIDS; section 6 evaluates the performance of the system; section 7 provides concluding remarks.

2. Background

One of the most computationally-intensive tasks performed by rule-based NIDS, such as Snort [5], is pattern-matching on packet content [6]. Despite improved software pattern-matching algorithms [6,7], pattern-matching is still the limiting factor in the analysis of high-speed traffic. The goal of our research is to eliminate this bottleneck by offloading all the pattern-matching tasks to a reconfigurable FPGA co-processor.

The task of pattern-matching in NIIDS consists of comparing a large number of known patterns against a stream of small data sets (packets). This differs from most classical research in FPGA pattern-matching, which compares short patterns entered at run-time against large, pre-processed data sets. A pattern-matching approach well-suited to NIDS was developed by Sidhu and

[†] This work was supported in part under NSF Grant 9876573, and by a grant from Intel Corporation.

Prasanna. In [8], they present a method of designing circuits for matching large regular expressions based on nondeterministic finite automata (NFA). In [9], Franklin et. al. combined multiple Snort patterns into a single regular expression and created a corresponding NFA circuit. An interesting contribution of their work was to store the patterns in a prefix tree before generating the circuit, which led to a logic resource savings when some of the patterns had common prefixes.

An optimized NFA circuit design methodology was shown in [10] that provided a 2-fold reduction in logic resource usage and an 8-fold reduction in routing resource usage over the previously-best results. As compared to [9], the more-efficient circuits allowed over twice as many pattern characters to be stored in the same device and still supported a higher clock frequency. These significant improvements in scalability were achieved by eliminating redundant information distributed to each character detector and by developing a hardware-efficient version of the character detector. Rather than broadcasting the fully encoded character to each detector module, this approach factored out the decoding operation and then distributed the decoded signals to each character detector. In this paper, we extend our preliminary work in [10] by adding support for more pattern-matching functions and integrating the FPGA with a fully-functional NIDS implementation.

3. FPGA Design

3.1. Data Path

A block diagram showing the data path of the pattern-matching module is shown in Figure 1. 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.

3.2. Module Generator

We have developed software 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) [11] to specify FPGA circuit components and connect them to implement the pattern matchers. The JHDL tool suite is used for functional simulation and then to convert the circuit to an EDIF netlist. Then, Xilinx Foundation is used to compile the design to the targeted FPGA device.

4. Pattern-Matcher Features

This section describes the features supported by our pattern-matching co-processor. Although these features were developed for their value in NIDS applications, all (except protocol analysis) are applicable to NFA pattern-matching circuits in general and can be used in other applications. Our system adopts the rule specification language from the open-source NIDS software, Snort [5],

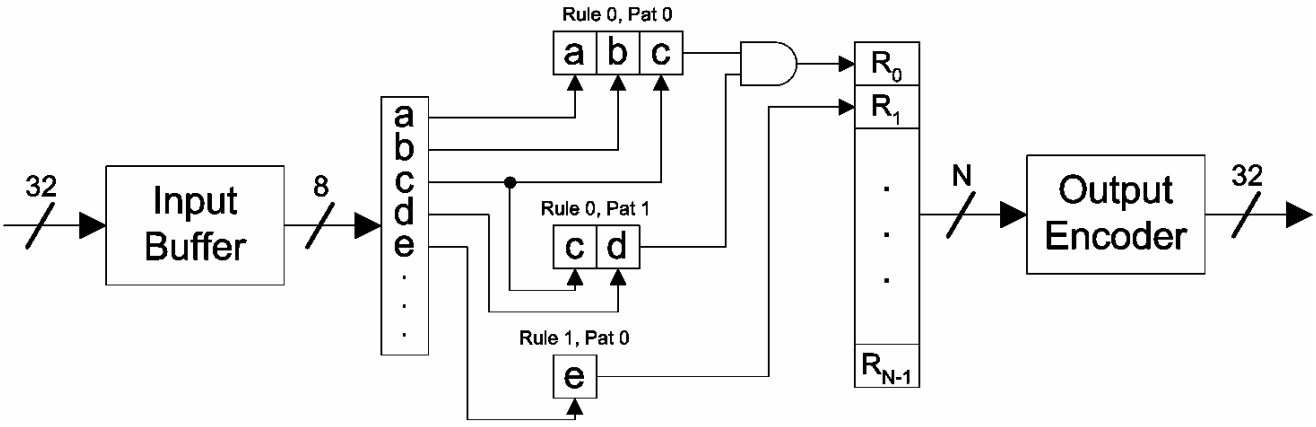


Figure 1. Data path of the FPGA pattern-matching module

```
alert tcp $EXTERNAL_NET any ->
        $SMTP_SERVERS 25
(msg:"SMTP HELO overflow attempt";
 flow:to_server,established;
 content:"HELO "; offset:0; depth:5;
 content:!"|0a|"; within:500;)
```

Figure 2. Example Snort rule

and supports a superset of the pattern-matching functionality in Snort version 2.0.

An example of a Snort rule is shown in Figure 2. This rule will trigger an alert for an attempted buffer overflow attack on an SMTP server. The part of the rule before the parentheses is called the rule header, and the parts of the rule inside the parentheses are called the rule options. Each of the rule options is a Boolean predicate, and a rule match occurs if the conjunction of the predicates is true.

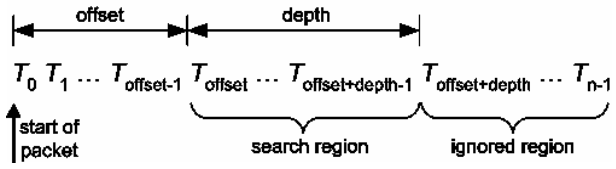
4.1. Case Sensitivity

The case-sensitivity of comparisons is important for intrusion detection. For case-insensitive 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. Case-insensitive comparisons are performed by taking the logical OR of the decoder's output signals for the upper and lower case character.

4.2. 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. Our pattern-matcher supports the specification of wildcard sequences with either an upper-bound or lower-bound on their length.

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 3 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:

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

Figure 3. Use of position rule options

4.3. Approximate 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*, we want 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 a non-deterministic finite automation (NFA) in [12] and its adaptation to a misuse detection system for a multi-user computer in [13].

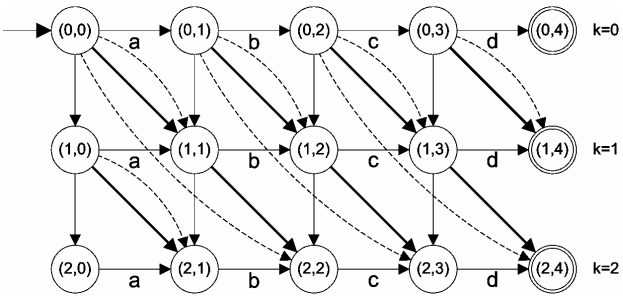


Figure 4. NFA for "`abcd`" with $k \leq 2$

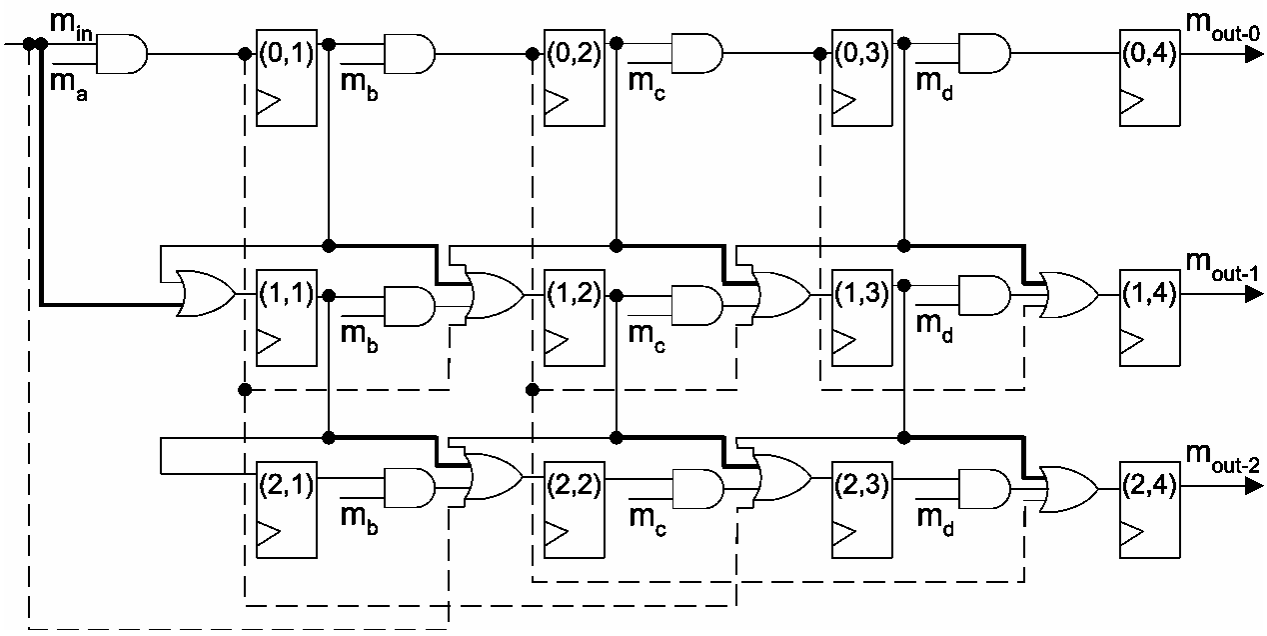


Figure 5. Circuit for "abcd" with $k \leq 2$

In this paper, we develop an NFA circuit capable of detecting approximate matches where the data may contain character substitutions, insertions, or deletions. Figure 4 depicts an NFA for the pattern "`abcd`" allowing two or less differences. The notation used is based on that in [12]. 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 4, horizontal transitions indicate character matches, vertical ones are character insertions, solid diagonals represent character substitutions, and dashed diagonals are used for character deletions.

Figure 5 shows how the NFA in Figure 4 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, we allow any number of characters to occur before the start of the pattern. Therefore, these characters are not considered insertions and the associated states can be ignored.

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 4 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 $O(km)$.

4.4. Protocol Analysis

Many ASCII-based Internet protocols (eg. 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

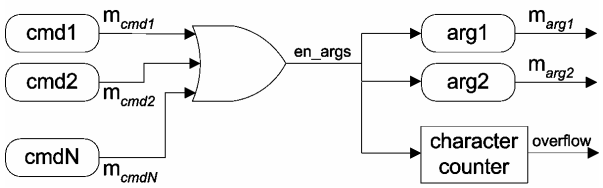


Figure 6. Protocol analysis

(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 6.

5. NIDS System Integration

5.1. Hardware Platform

This work is being carried out as part of an effort to design and develop a complete network intrusion detection system implemented using embedded hardware components. A prototype based on an Intel IXP network processor (NP) and a Xilinx FPGA has been developed. The architecture of this system is shown in Figure 7. The NP is a system-on-chip that contains a StrongARM CPU core running a Linux operating system and several packet processors, or microengines. Network connectivity is enabled through one or more 1 Gbps Ethernet ports. The NP and FPGA share a dedicated 32-bit PCI bus operating at 66 MHz. The NIDS board can be installed in a PC using a standard connector attached to the host PCI bus.

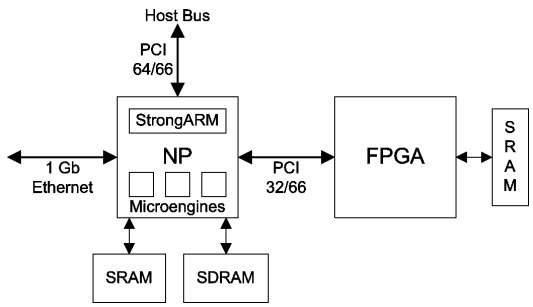


Figure 7. Hardware NIDS platform

5.2. Data Flow

The flow of data through the various components of the intrusion detection system is shown in Figure 8. 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.

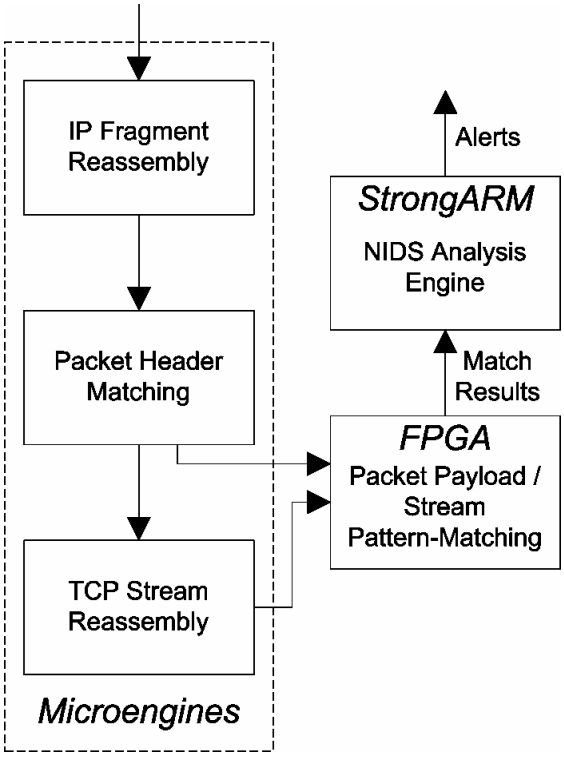


Figure 8. NIDS Data flow

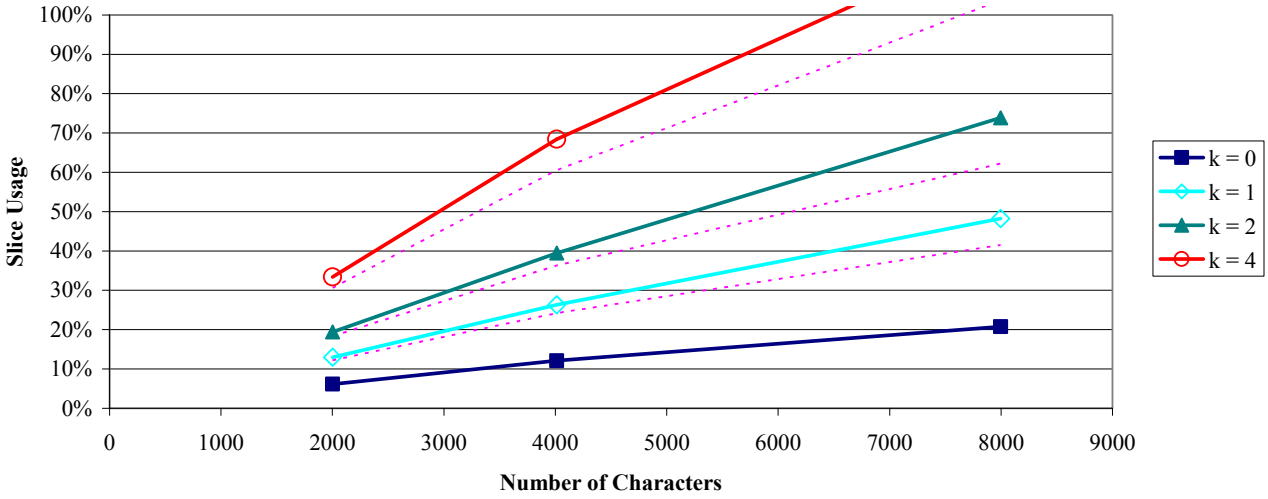


Figure 9. Complexity of approximate matching circuits (Virtex2-6000)

6. Evaluation

6.1. 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 9. The dashed lines show the theoretically-predicted values. As expected the design scales near the rate of *km*.

6.2. Hardware Performance

For comparison purposes, we implemented and tested two approaches to building NFA circuits. The first, which we call the distributed comparator approach, was based on the design used in [8] and [9]. The second, which we call the shared decoder approach, used our design originally presented in [10]. In addition, each approach was tested both with and without a prefix tree structure that enables the sharing of circuitry for patterns with common prefixes.

To illustrate the scalability of the different approaches, we ran tests with the same rule sets as above plus the entire Snort 2.0 rule set, which contained 17,537 characters. The FPGA resource usage for each design is plotted in Figure 10. As the figure shows, the shared decoder approach allowed more the double the number of pattern characters to be programmed into the same FPGA. The throughput of the decoder design was also measured and is shown in Table 1. The throughput is calculated by multiplying the clock frequency by the data width (8-bits). 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 for the previous packet is complete.

Table 1. Performance of shared decoder with prefix tree

Number of Characters	Virtex-1000			Virtex2-6000		
	Area (Slices)	Freq (MHz)	Throughput (Mbps)	Area (Slices)	Freq (MHz)	Throughput (Mbps)
2,001	17%	118.9	951	5%	250.2	2002
4,012	25%	114.5	916	8%	246.4	1971
7,996	42%	101.1	809	14%	227.6	1821
17,537	80%	100.1	801	28%	192.0	1536

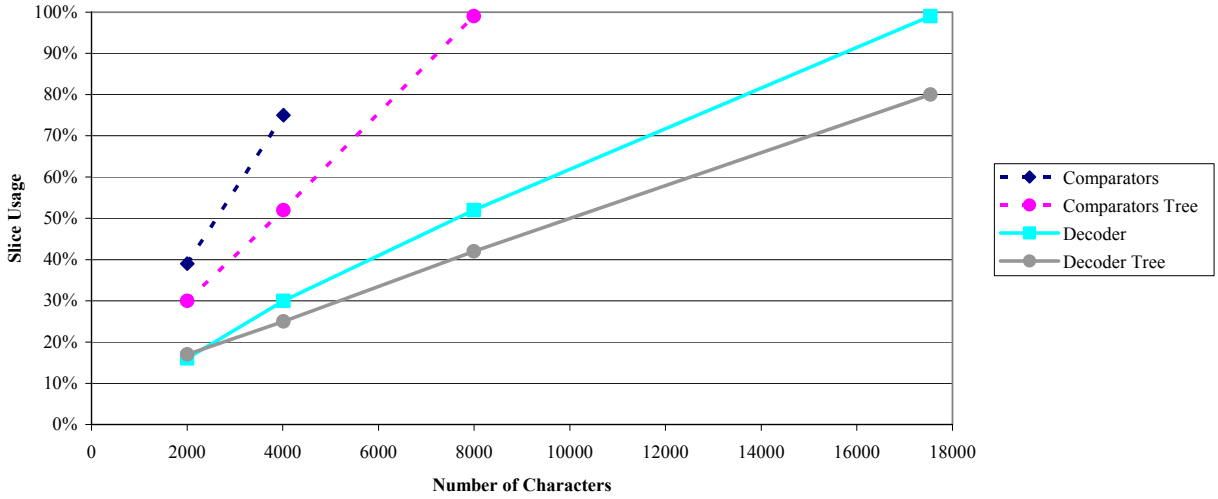


Figure 10. Comparison of distributed comparator and shared decoder approaches (Virtex-1000)

7. Conclusion

We have presented a high performance, high efficiency design for concurrently matching a large number of patterns against a large data set. The design and accompanying module generator software support a rich specification feature set including approximate matching and complex predicates. We have performed a detailed study of area and throughput for our design in both exact and approximate matching scenarios. Results show that our current design is quite scalable, and fits easily within a Virtex-1000 with the entire Snort ruleset loaded. In a Virtex2-6000, throughput is nearly 2 Gbit/s. We have also demonstrated that the approximate matcher scales nicely with the specified bounds on the number of character differences.

References

[1] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stonesr, "State of the Practice of Intrusion Detection Technologies," Technical Report CMU/SEI-99-TR-028, 1999.

[2] L.G. Roberts, "Beyond Moore's Law: Internet Growth Trends," *IEEE Computer*, pp. 117-119, Jan 2000.

[3] Marc Necker, Didier Contis, and David Schimmel, "TCP-Stream Reassembly and State Tracking in Hardware," *Proceedings of FCCM 2002*, pp. 286-287, Apr 2002.

[4] James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall", *Proceedings of IEEE FCCM 2003*, April 2003.

[5] Martin Roesch and Chris Green, "Snort User's Manual". Online: http://www.snort.org.

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

[7] C. Jason Coit, Stuart Staniford, and Joseph McAlerney, "Towards Faster String Matching for Intrusion Detection," *DARPA Information Survivability Conference*, June 2001.

[8] R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching using FPGAs," *Proceedings of IEEE FCCM 2001*, Apr. 2001.

[9] R. Franklin, D. Carver, and B.L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *Proceedings of IEEE FCCM 2002*, pp. 111-120, Apr. 2002.

[10] Christopher Clark and David Schimmel, "Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns", *Proceedings of FPL 2003*, Sep. 2003.

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

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

[13] 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.