Optimization of Regular Expression Pattern Matching Circuits on FPGA

Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

Abstract

Regular expressions are widely used in Network Intrusion Detection System (NIDS) to represent patterns of network attacks. Since traditional software-only NIDS cannot catch up to the speed advance of networks, many previous works propose hardware architectures on FPGA to accelerate attack detection. The challenge of hardware implementation is to accommodate the regular expressions to FPGAs of the large number of attacks. Although the minimization of logic equations has been studied intensively in the CAD area, the minimization of multiple regular expressions has been largely neglected. This paper presents a novel architecture allowing our algorithm to extract and share common sub-regular expressions. Experimental results show that our sharing scheme significantly reduces the area of regular expression circuits.

1. Introduction

Regular expressions are widely used in Network Intrusion Detection System (NIDS) to represent network attack patterns. The NIDS is used to recognize and detect network attacks, especially in the application layer, that general firewalls cannot find. As soon as any malicious packet is matched to an attack pattern, the NIDS notifies the system and takes the appropriate actions. Due to the rapid increase of network attacks and data traffic, traditional software-only NIDS, which sequentially matches input packets against attack patterns, may be too slow for networking needs.

In contrast to software-only NIDS, many studies [1][2][3][4][5] proposed hardware architectures for accelerating attack detection. These hardware architectures are mostly implemented in FPGA because FPGA allows for updating new attacks patterns. Sidhu and Prasanna [1] proposed to construct an NFA (Nondeterministic Finite Automaton) from a regular expression to perform string matching. Hutchings, Franklin and Carver [2] developed a module generator that combined common prefixes to reduce FPGA area. In contrast to NFA approaches, a content matching server [3] was developed to automatically generate Deterministic Finite Automatons (DFAs) to search for regular expressions. Based on the Knuth-Morris-Pratt (KMP) algorithm, Baker and Prasanna [4] proposed a linear-array, pipelined, two-comparators and buffered string-matching architecture that provided instantaneous reconfiguration and better scalability.

One of the main challenges of hardware implementation is to accommodate the large number regular expressions to FPGAs. Most previous works proposed novel architectures that translated each regular expression pattern to one circuit module. Then, input strings are fed into corresponding circuit modules of all regular expressions in parallel to detect all attacks. However, one to one hardware implementation of a regular expression can lead to cost-inefficient designs that cannot deal with the ever-increasing bandwidth and number of attacks. Therefore, it is important to develop new methodology to minimize large multiple regular expressions. Although the minimization of logic equations has been studied intensively in the CAD area, there is very little research in the minimization of multiple regular expressions.

The following example illustrates the difficulty of minimizing regular expressions. Consider two simple regular expression patterns “PassWinDirUserGate” and “PassSysDirNetGate.” Figure 1 shows a simplified regular expression circuit where the top five blocks match the first pattern and the bottom five blocks match the second pattern. Each block compares a substring and asserts the output signal once the substring matches the desired pattern. For example, the first block (highlighted) compares the pattern “Pass.” Once the first block succeeds in matching, the following block is activated by triggering the control signal “en.” It is easy to find that both “Pass” blocks can be shared, as shown in Figure 2 [2]. Although the common infix sub-pattern “Dir” exists, the corresponding hardware blocks cannot be shared directly, as they are in Figure 3. Because the block “Dir” can be triggered either by “Win” or “Sys,” the string “PassSysDirUserGate” may wrongly be recognized as a match at the output of the top block.

3-9810801-0-6/DATE06 © 2006 EDAA

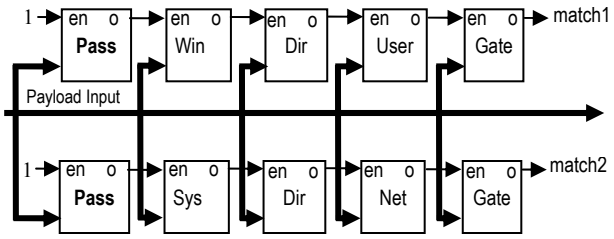


Figure 1. Original circuits

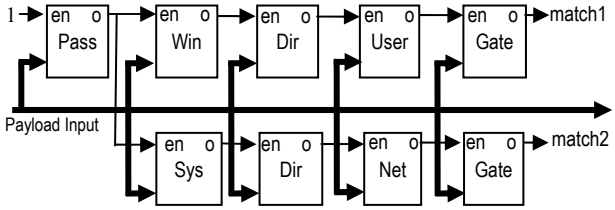


Figure 2. Sharing prefix common sub-patterns

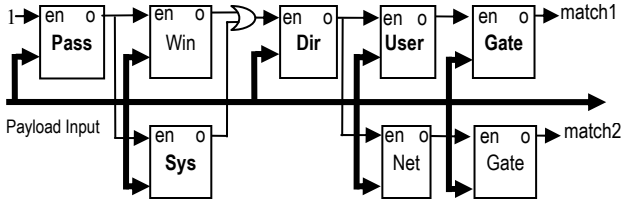


Figure 3. An erroneous implementation to share infix Dir

In this paper, we present a novel architecture which allows our algorithm to extract and share as many common sub-regular expressions as possible. Additionally, in order to construct the regular expression patterns of Snort [6] and Trend Micro, we develop five basic NFA components to support Perl-compatible regular expressions (PCRE). We obtain a significant area reduction on both patterns. For Snort patterns, our experimental results show 48% in area reduction on average. We also obtain 20% in area reduction on average in the patterns of Trend Micro. The paper is organized as follows: Section 2 introduces the regular expressions for attacks' description. Then, Section 3 demonstrates our sharing scheme. Section 4 demonstrates the hardware implementation and finally, experimental results and conclusions are given in Sections 5 and 6, respectively.

2. Regular expressions for attacks' description

Regular expressions are a common way to express attack patterns. In Snort, two types of regular expression are used to describe attack patterns. The first type defines exact string patterns such as *Backdoor*'s pattern, `"Ahhhh My Mouth Is Open.`" In Snort, about 87% of rules belong to this type. The second type consists of meta-characters such as anchor (`^` and `$`), alternation (`|`), and quantifier (`*` and `?`). For example, the rule for detecting the Oracle Web Cache attack is written as

`alert tcp any -> (pcre:"^GET[^s]{432} ";…).`

The string "`^GET[^s]{432}`" in the "pcre" field represents a complex pattern where "`^`" denotes "the beginning of a line", and the "`GET [^s]{432}`" denotes that the successive 432 characters after "`GET`" cannot contain "s." The Snort has about 1,777 rules for detecting a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and OS fingerprinting attempts.

3. Minimization of regular expression circuits

In the introduction, we describe the difficulty of sharing *infix* sub-pattern where the common sub-pattern occurs in the middle of a pattern. Similarly, we cannot directly share *postfix* sub-pattern where the common sub-pattern occurs at the tail of a pattern. If the two outputs, match1 and match2, of the original circuit are merged directly, the circuit cannot differentiate the pattern being matched from the other whenever the match out is asserted.

In order to solve the problems caused by directly sharing infix and postfix, we propose a new architecture that can memorize the path that the trigger signal passes through. Under specific constraints, our approach supports the sharing of common infix and postfix sub-patterns. In addition, the proposed architecture is not only fit for the exact string matching, but also the complex regular expression patterns composed of meta-characters. The new architectures for sharing infix, postfix, and prefix are described as follows.

Given m regular expressions, $R_1, R_2, \ldots, R_m$, and assuming that all of them have the infix common sub-pattern, R_c, the m regular expressions can be represented as $R_{1pre}R_cR_{1post}$, $R_{2pre}R_cR_{2post}, \ldots$, and $R_{mpre}R_cR_{mpost}$, where the suffixes, *pre-* and *post-*, denote the prefix and postfix, respectively. To resolve the two problems caused by directly sharing common infix and postfix sub-patterns, we propose a novel sharing architecture in Figure 4 that allows the common sub-pattern R_c to be shared.

In Figure 4, two additional circuit blocks are inserted. The switch module is used to memorize where the trigger signal comes from, and then output control signals to DeMux (De-Multiplexer) to guide the output of R_c to the correct postfix circuit. If R_{1pre} is matched, the output signal of R_c will pass to R_{1post}. Similarly, if R_{2pre} is matched, only R_{2post} can be activated after R_c is matched. By this new architecture, the output of the common infix sub-pattern can be passed to the appropriate postfix circuit, preventing from the problem caused by directly sharing infix sub-pattern. Similarly, this new architecture can support the sharing of the common postfix sub-patterns. This can be done easily if all or some of the postfixes are eliminated in Figure 4.

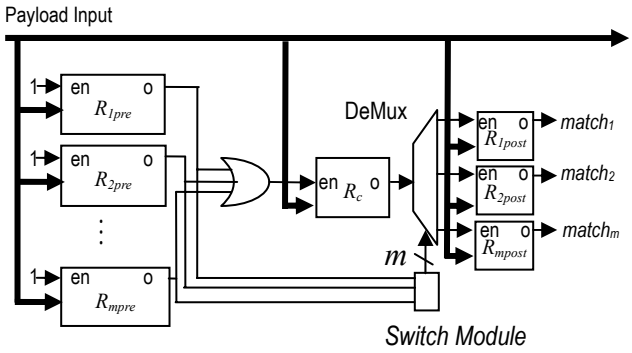


Figure 4. Sharing scheme for infix and postfix

Still, the new architecture has two constraints, which disallow certain special types of sharing. Common sub-pattern satisfies the following constraint cannot be shared by using the sharing architecture shown in Figure 4. Taking two regular expressions with a common sub-pattern R_c for example in Figure 5, the switch module behaves as a JK flip-flop. The two constraints are described as follows.

Constraint 1: For the m regular expressions in Figure 4, $\{R_{1pre}R_cR_{1post}, R_{2pre}R_cR_{2post}, \ldots, R_{mpre}R_cR_{mpost}\}$, the prefix R_{jpre} cannot be null for $j \in 1...m$.

Proof: Omitted.

For example, given two patterns, "abcdefgh" and "defpq," there exists a common sub-pattern "def," but it cannot be shared by directly applying our new architecture. In Figure 5, our architecture applies a constant 1 at the input of the highlighted OR gate. However, this will cause the match output of block "def" always pass to block "pq." Therefore, in this special case, we will skip the sharing of sub-pattern "def."

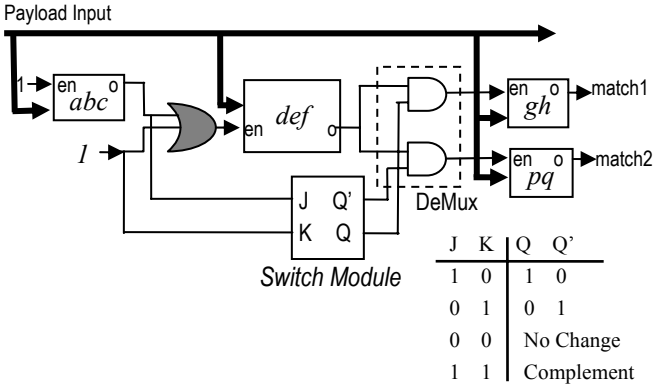


Figure 5. An example of constraint 1

Constraint 2: For the m regular expressions in Figure 4, $\{R_{1pre}R_cR_{1post}, R_{2pre}R_cR_{2post}, \ldots, R_{mpre}R_cR_{mpost}\}$, the R_c cannot be shared if $R_{jpre} \subset R_{kpre}R_c$, $\forall k \neq j$, $k, j \in 1\ldots m$.

Proof. Omitted.

For example, given two patterns, "abcdefgh" and "dedefpq," there exists a common sub-pattern "def." But in this case, our sharing architecture cannot be applied. As Figure 6 is shown, the block "de" is a sub-pattern of the block "abcdef." Suppose a string "abcdefgh" is fed, the trigger signal should be guided from the block "abc" through the block "def," to the block "gh," and then match1 should be asserted. But actually it fails because the outputs of the JK flip flop will be complemented when the string "abcde" is fed and the string "abcdefgh" will be missed. Again, when this special condition is recognized, our algorithm will skip the sharing to prevent an erroneous result.

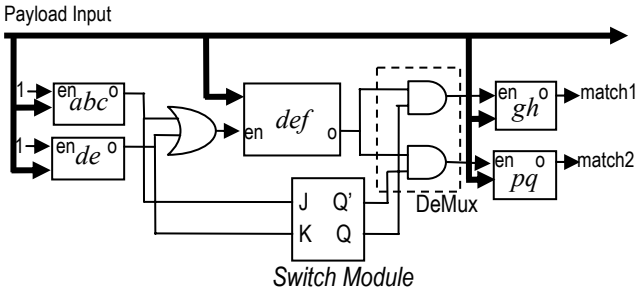


Figure 6. Example of constraint 2

4. Hardware implementation

4.1. Regular expression module generator

We develop a regular expression module generator that can explore the sharing of common prefix, infix and postfix sub-patterns. The flow diagram of our generator is shown in Figure 7. In the first stage, we obtain regular expression patterns from the pattern database. Then, common prefix sub-patterns are shared directly. After that, we recursively extract one common infix or postfix sub-pattern which has the largest *sharing gain* defined as follows. The sharing gain of a common sub-pattern is defined to be the number of characters in the sub-pattern multiplies by the number of regular expressions having the sub-pattern. For example, three regular expressions, "1Common1", "2Common2", and "3Common3" have the common sub-pattern "Common." The sharing gain of the common sub-pattern is 18=6*3 because "Common" has 6 characters and the number of regular expressions is 3. In our experiment, because sharing also has hardware overhead, we heuristically restrict the number of characters of a common sub-pattern to be more than two. The process of sharing continues until no common sub-pattern can be shared. Note that a shared common sub-pattern must succeed in passing the constraints described in Section 3. In the final stage, we generate the Verilog HDL code of the shared architecture.

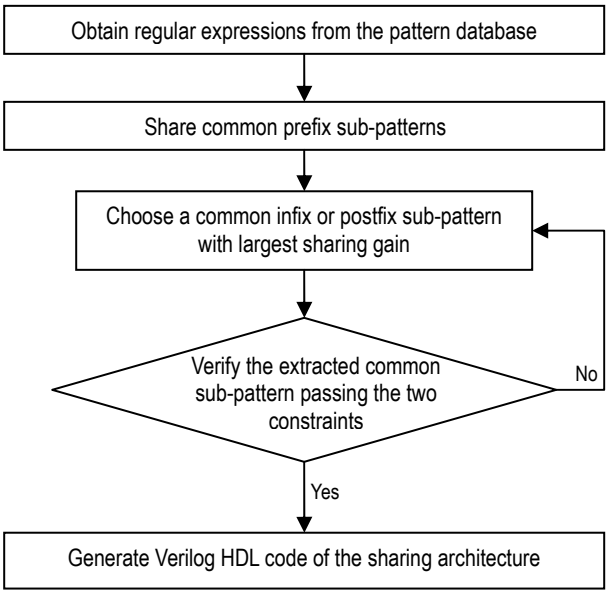
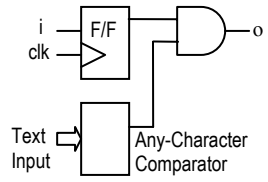


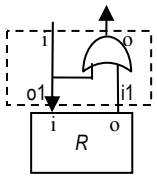
Figure 7: Flow of regular expression module generation

4.2. Basic components of NFA approach

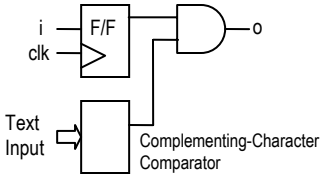
Figure 4 demonstrates our sharing architecture, of which each block can be constructed by the basic four NFA components, single-character matcher, union (|), concatenation, and Kleene-star (*), proposed by Sidhu and Prasanna [1]. The single-character matcher is used to match single character. In order to support Perl-compatible regular expressions (PCRE), we also develop five additional meta-character components, including *any-character* matcher (.), *complementing-character* matcher (^), question mark (?) quantifier, plus quantifier (+), and dollar sign anchor ($) (see Figure 8). The any-character matcher is used to match any input character (see Figure 8 (a)). The complementing-character matcher is used to match the characters outside of a range by complementing the set (see Figure 8 (c)). Similarly, given a regular expression, *R*, *R*? matches any string composed of zero or one occurrences of *R* (see Figure 8 (b)); *R*+ matches any string composed of one or more occurrences of *R* (see Figure 8 (d)). The dollar sign anchor ($) is used to match the end of a line, of which the ASCII code is hexadecimal 0D or 0A (see Figure 8 (e)). Most of the regular expression patterns in the Snort and Trend Micro pattern databases can be constructed with these basic components. For example, the NFA circuit constructed form the regular expression, ab? • [^c]d+, is shown in Figure 9.



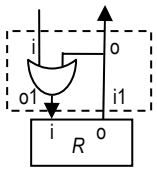
(a) Any-character matcher (•)



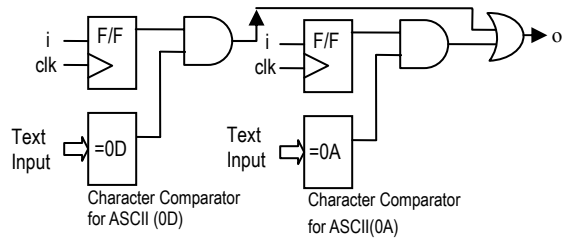
(b) Question mark (?) quantifier(dashed box)



(c) Complementing–character matcher (^)



(d) Plus quantifier (+) (dashed box)



(e) Dollar sign anchor ($)

Figure 8. Logical structures for the proposed meta-character components

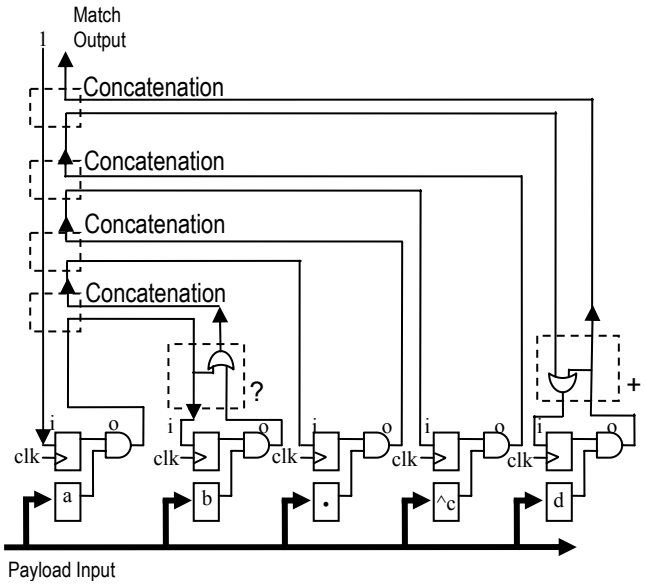


Figure 9: Implementation of NFA for ab? • [^c]d+

5. Experimental results

We implement the algorithm shown in Figure 7 and apply to the regular expression patterns from Snort and an industry company, Trend Micro. The results are compared with the approach of sharing only common prefixes as in [2]. Table 1 lists the experimental results on seven Snort rule sets. The area comparison is made with all circuits being synthesized by the commercial tool, Xilinx ISE7.1i, where the target FPGA is Xilinx Virtex XCV2000E consisting of 19,200 slices.

We perform experiments on seven sets of regular expressions from Snort and three sets from Trend Micro. The name of the set, the number of patterns, and the number characters are shown in the 1st, 2nd and 3rd columns, respectively. For each set of regular expressions, we first construct an NFA circuit for a regular expression pattern and then put all the NFA circuits in parallel. The resulting area is shown in the 4th column. We then apply the approach of sharing common prefixes to the same rule set. The area and the percentage of improvement are shown in the following two columns, respectively. Finally, we apply our sharing scheme to the same rule set and report the area and the percentage of improvement in the last two columns. For example, the area to implement the Snort Oracle rule set, which has 138 patterns with 4,674 characters**,** is 3,060 slices (see the 1st row of Table 1). The area reduces to 2,011 slices after applying the technique of sharing common prefixes. The area reduction is 34%, as compared with the original design. Using our sharing scheme, the area to implement the same rule set is 912 slices. And the area reduction achieves 70%.

Table 1: The comparison among different approaches on Snort rule sets

Rule Set	# of Patterns	# of Characters	Original Design	Sharing Prefix [2]		Our Sharing Scheme	
			Area (Slices)	Area (Slices)	Area reduction. (%)	Area (Slices)	Area Reduction (%)
Oracle	138	4,674	3,060	2,011	34%	912	70%
Backdoor	57	599	355	343	3%	315	11%
Sql	44	1,089	714	442	38%	365	49%
Web-iis	113	2,047	1,362	1,213	11%	932	32%
Web-php	115	2,455	1,683	1,397	17%	887	47%
Web-misc	310	4,711	2,906	2,506	14%	1,789	38%
Web-cgi	347	5,339	3,092	2,548	18%	1,667	46%
Total	**1,124**	**20,914**	**13,172**	**10,460**	**21%**	**6,867**	**48%**

Table 2: The comparison among different approaches on industrial rule sets

Rule Set	# of Patterns	# of Characters	Original Design	Sharing Prefix [2]		Our Sharing Scheme	
			Area (Slices)	Area (Slices)	Area reduction. (%)	Area (Slices)	Area Reduction (%)
Worstcase	173	6,465	5,804	5,653	3%	4,660	20%
Combined	322	12,950	11,034	10,833	2%	8,686	21%
Normal	357	13,152	11,171	10,956	2%	8,953	20%
Total	**852**	**32,567**	**28,009**	**27,442**	**2%**	**22,299**	**20%**

The traditional technique of sharing common prefixes can only have 21% of area reduction for Snort rule sets on average. The proposed approach, however, achieves 48% of area reduction, which can share common prefixes, infixes and postfixes efficiently.

In addition, the area reduction becomes less significant for the industrial rule sets. Table 2 shows the results of applying different approaches. In this table the area reduction is only 2% on average when applying the traditional common prefix sharing. Using our approach, the improvement on area can achieve 20%. The experimental results demonstrate the area efficient of our algorithm for both benchmark rule sets and realistic industrial rule sets.

6. Conclusion

Regular expressions are widely used in Network Intrusion Detection System to represent network attack patterns. To accommodate large number of regular expressions to FPGAs, area reduction of regular expression pattern matching circuits is very important. In this paper, we presented a novel architecture allowing our algorithm to extract and share common prefix, infix, and postfix sub-regular expressions. Under specific constraints, both the common infix and postfix sub-patterns can be extracted and shared efficiently. Additionally, in order to support Perl-compatible regular expressions (PCRE), we also developed five important meta-character components. An automatic generation tool is also presented to cost-effectively extract the common sub-pattern for FPGA implementation. The experimental results show that our sharing scheme can significantly reduce the area of the regular expression circuits both for the Snort and industrial realistic regular expression rule sets.

7. Acknowledgements

The authors would like to thank the following experts of Trend Micro Inc., Ming Deng (Group Project Manager), Sarah Chin (Project Manager), Chris Lo (QA Manager), Vic Lo (Development Manager), Kenneth Kuo (Development Manager), Viking Ho (Sr. Engineer), Porpoise Chiang (Project Lead), Ronaldo Mier (QA Project Lead), and Kent Chiang (Engineer) for their constructive inputs. This work was supported in part by NSC under contract 94-2220-E-007-038.

References

[1] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, Apr. 2001, pp. 227-238.

[2] B.L. Hutchings, R. Franklin and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *Proc. of the10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, Sep. 2002.

[3] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *Proc. of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*, Apr. 2003.

[4] Z. K. Baker, V. K. Prasanna, "Time and area efficient pattern matching on FPGAs," in *Proc. of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, Feb. 2004, pp. 223-232.

[5] Young H. Cho and William H. Mangione-Smith, "A Pattern Matching co-processor for Network Security," in *Proc. of the DAC 2005*, June, 2005.

[6] M. Roesch. Snort- lightweight Intrusion Detection for networks, in *Proceedings of LISA99, the 15th Systems Administration Conference*, 1999.

[7] Monther Aldwairi, Thomas Conte, and Paul Franzon, "Configurable String Matching Hardware for Speeding up Intrusion Detection," *ACM SIGARCH Computer Architecture News*, vol. 33, No. 1, March 2005.