Optimization of Pattern Matching Circuits for Regular Expression on FPGA

Cheng-Hung Lin, *Student Member, IEEE*, Chih-Tsun Huang, *Member, IEEE*, Chang-Ping Jiang, and Shih-Chieh Chang, *Member, IEEE*

***Abstract*—Regular expressions are widely used in the network intrusion detection system (NIDS) to represent attack patterns. Previously, many hardware architectures have been proposed to accelerate regular expression matching using field-programmable gate array (FPGA) because FPGAs allow updating of new attack patterns. Because of the increasing number of attacks, we need to accommodate a large number of regular expressions on FPGAs. Although the minimization of logic equations has been studied intensively in the area of computer-aided design (CAD), the minimization of multiple regular expressions has been largely neglected. This paper presents a novel sharing architecture allowing our algorithm to extract and share common subregular expressions. Experimental results show that our sharing scheme significantly reduces the area of pattern matching circuits for regular expression.**

***Index Terms*—Finite automata, field-programmable gate array (FPGA), intrusion detection, pattern matching.**

I. Introduction

REGULAR expressions are widely used in the network intrusion detection system (NIDS) to represent attack patterns. The NIDS is used to recognize and detect network attacks that general firewalls cannot find, especially in the application layer. As soon as any malicious packet is identified to contain an attack pattern, the NIDS notifies the system and takes appropriate actions. Due to the rapid increase of network attacks and data traffic, traditional software-based NIDS, which sequentially matches input packets against attack patterns, will become inadequate for networking needs due to its slowness.

In contrast to software-based NIDS, many studies proposed hardware architectures for accelerating attack detection. These hardware architectures are mostly implemented on field-programmable gate array (FPGA) because FPGAs allows updating for new attack patterns. Sidhu *et al.* [1] proposed to construct a nondeterministic finite automaton (NFA) from a regular expression to perform string matching. Hutchings *et al.* [2] developed a module generator that shared common prefixes to reduce the circuit area on FPGA. Clark *et al.* [3] made excellent area and throughput by adding predecoded wide parallel inputs to traditional NFA implementations. Cho *et al.* [5] compressed the hardware size by reusing the subcomponents of reconfigurable discrete logic filter. Baker *et al.* [7] presented a predecoded shift-and-compare architecture to reduce the area. In contrast to NFA approaches, a content matching server [9] was developed to automatically generate deterministic finite automatons (DFAs) to search for pattern matching. Baker *et al.* [10] proposed a novel linear-array string matching architecture providing better scalability and reconfiguration, and more hardware utilization.

One of the main challenges of hardware implementation is to accommodate a large number of regular expressions to FPGAs. Most previous works proposed novel architectures that translated each regular expression pattern to one circuit module. Then, payloads are broadcasted to the multiple regular expression circuits to detect attacks. However, one-to-one hardware implementation of regular expressions can lead to cost-inefficient designs that cannot deal with the ever-increasing number of attacks. Therefore, it is important to develop a new methodology to minimize the circuit area of the large number of regular expressions. Although the minimization of logic equations has been studied intensively in the area of computer-aided design (CAD), 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." Fig. 1 shows a simplified regular expression circuit where the top five concatenated blocks are used to match the first pattern and the bottom five concatenated blocks are used to match the second pattern. Each block compares a substring and outputs a logical high once the substring matches the desired pattern. For example, the first block (highlighted) compares the pattern "Pass." Once the first block matches the substring "Pass," it outputs a logical high and activates the successive block by triggering the control signal "en." One of the powerful techniques to reduce the area is to perform circuit sharing. The subexpression "Pass" in the front position of the regular expression is called the prefix. It is easy to find out that both patterns have common prefixes "Pass." Therefore, it can be shared to reduce the area [2]. However, there exist more opportunities in sharing common subexpressions in the middle position, the infixes, and the common subexpressions in the tail position, i.e., the suffixes. On the other hand, sharing common infix and suffix requires much more complex consideration. Consider the same example in Fig. 2. Although these two patterns have a common infix "Dir," the corresponding hardware blocks cannot be shared directly as shown in Fig. 3. Because the block "Dir" will trigger both the block "User" and block "Net," the

Manuscript received April 4, 2006; revised October 20, 2006. This work was supported by the Ministry of Economic Affairs of Taiwan under 96-EC-17-A-01-S1-038.

C.-H. Lin, C.-T. Huang, and S.-C. Chang are with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan, R.O.C. (e-mail: brucelin@nthucad.cs.nthu.edu.tw; cthuang@cs.nthu.edu.tw; scchang@cs.nthu.edu.tw).

C.-P. Jiang is with SpringSoft, Inc., Hsinchu 30013, Taiwan, R.O.C. (e-mail: flat@nthucad.cs.nthu.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2007.909801

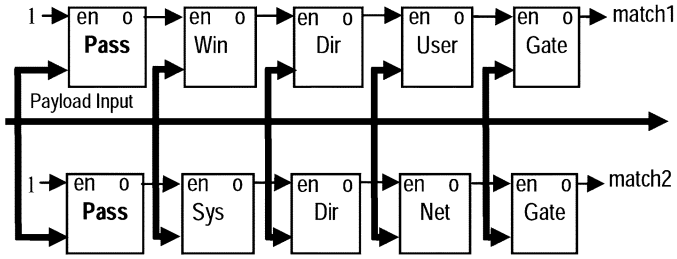


Fig. 1. Original circuits.

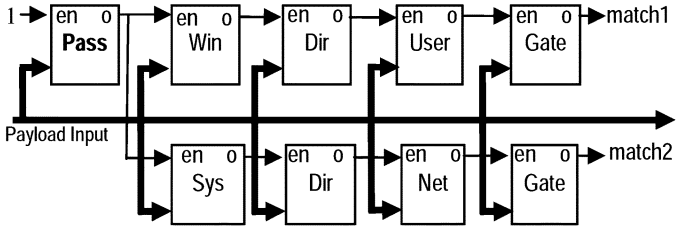


Fig. 2. Sharing common prefix subexpressions.

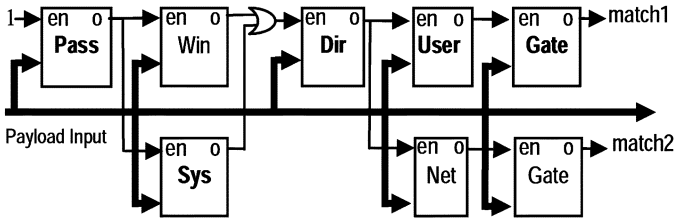


Fig. 3. Erroneous implementation to share common infix "Dir."

string "PassSysDirUserGate" may be mistaken as a match at the output of the upper blocks, i.e., the match1 signal. The erroneous result is so-called the "false positive."

In this paper, we propose a novel sharing architecture which allows our algorithm to extract and share as many common subregular expressions as possible. Additionally, in order to construct the regular expression patterns of Snort [11] and from the industry company, Trend Micro, we develop five basic NFA components to support Perl-compatible regular expressions (PCRE). Furthermore, we integrate the predecoding approach proposed by Clark [3] to our algorithm, called the integration approach. We replace the distributed comparators with 8-to-256 character decoders. The experimental results show that the integration approach can achieve an average of 28% in the area reduction on the Snort rule sets and 38% on the patterns from Trend Micro. Besides, although our approach is aimed to optimize the area on FPGA, the circuit delay is also improved because our algorithm can reduce the fanout load of inputs and thus the predecoding approach can alleviate its routing complexity. The integration approach achieves an average of 22% in delay reduction on the Snort rule sets and 22% on the Trend Micro rule sets. The results show that our approach is very efficient when combined with the predecoding approach for the area and timing optimization.

This paper is organized as follows. Section II discusses the previous works. Section III introduces the regular expressions for attacks' description. In Section IV, we present our sharing architecture. Section V demonstrates the NFA hardware implementation and Section VI discusses our regular expression module generator. Finally, Sections VII and VIII give the experimental results and conclusions, respectively.

II. Previous Works

In this section, we review several previous works in this area. Sidhu *et al.* [1] first proposed a simple and fast algorithm to construct an NFA for a given regular expression and used it to process text characters. Subsequently, Hutchings *et al.* [2] implemented a module generator that can extract patterns form the Snort rule sets, and generate regular expression to match all the extracted patterns.

In order to reduce the area, many strategies are proposed for reducing the redundancy through predesign optimization. Clark *et al.* [3] proposed the predecoding approach which replaced the distributed comparators with a shared 8-to-256 character decoder and extended the approach to a scalable bandwidth system [4]. By adding predecoded wide parallel inputs to traditional NFA implementation, the area can be effectively reduced. Cho *et al.* [5] proposed a high-speed rule-based multilayer inspection firewall system by large, pipelined comparators, and then presented a methodology which reduced the number of comparators by finding identical alignments between other unattached patterns [6]. The preprocessing takes advantage of the shared alignments and allows for the 32-bits architecture. Baker *et al.* [7] presented a methodology which integrated rule-based graph creation and min-cut partitioning, allowing efficient multibyte comparisons and partial matches, and then adopted the predecoded shift-and-compare architecture to reduce comparator size and routing [8]. Besides, J. Moscola *et al.* [16] presented an implementation of a high-performance network application layer parser in FPGA, of which the 8-to-256 decoder is applied to pattern matcher for decreasing the routing resource. Sourdis *et al.* [12] adopted a scalable, low-latency architecture by employing full-width comparators for the search.

Except for the NFA approach, Moscola *et al.* [9] proposed a multigigabyte pattern matching system by demultiplexing a TCP/IP stream into multiple substreams and spreading the load over several parallel matching units constructed by the DFA. Based on Knuth–Morris–Pratt (KMP) algorithm, Baker *et al.* [10] proposed a linear-array string matching architecture using a buffer with two-comparators that provided instantaneous reconfiguration and better scalability. Cho *et al.* [17] presented a high performance pattern matching co-processor, a RAM-based design which stores the state transitions in programmable RAMs.

Instead of matching fixed characters per cycle, the CAM-based solution can match the entire pattern at once when the pattern is shifted past the CAM. Gokhale *et al.* [13] proposed a CAM-based solution to perform parallel search at a high speed. Sourdis *et al.* [14] advocated the use of predecoding for CAM-based pattern matching to reduce the area. Besides, Yu *et al.* [15] presented a ternary content addressable memory (TCAM)-based multiple-pattern matching which can handle complex patterns, correlated patterns, and patterns with negation. However, as compared with standard memories, CAM is costly in terms of design complexity, area overhead and power consumption.

A more recent hash-based approach was proposed to utilize Bloom filter for deep packet inspection. Dharmapurikar *et al.* [18] proposed a hashing-table lookup mechanism utilizing

parallel bloom filters to enable a large number of fixed-length strings to be scanned in hardware. Lockwood *et al.* [19] proposed an intelligent gateway based on Bloom filter that provides Internet worm and virus protection in both local and wide area networks.

III. 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 the 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 metacharacters such as anchor (^ and $), alternation (|), and quantifier (* and ?). For example, the rule for detecting the Oracle Web Cache attack is written as

$$\texttt{alert tcp any} \rightarrow (\texttt{pcre: } \texttt{"}\hat{}\texttt{GET}[\hat{}\texttt{s}]\{432\}\ \texttt{"};\ldots).$$

The string "$\texttt{GET}[\hat{}\texttt{s}]\{432\}$" in the "pcre" field represents a complex pattern, where "^" denotes "the beginning of a line," and the "$\texttt{GET}[\hat{}s]\{432\}$" denotes that the successive 432 characters after "GET" cannot contain "s." The Snort has about 1777 rules for detecting a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and OS fingerprinting attempts.

Given a regular expression $P = p_1p_2, \ldots, p_{m-1}p_m$, we say a partial expression, $p_1p_2, \ldots, p_k$ is a *prefix* of P if $k < m$, a partial expression, $p_jp_{j+1}, \ldots, p_k$ is an *infix* of P if $j > 1$ and $k < m$, and a partial expression $p_jp_{j+1}, \ldots, p_m$ is a *suffix* of P if $j > 1$. For example, let the expression P be "networking." The partial expression "net" is a prefix, "work" is an infix, and "ing" is a suffix of P.

IV. Minimization of Regular Expression Circuits

Among a set of regular expressions, there may exist common subexpressions. If the common subexpression is a prefix, [2] shows a way to share prefix subexpressions. However, there exist more opportunities in sharing common infixes and suffixes but they require more complex consideration than the sharing of common prefixes. In the introduction, we have shown the difficulty of sharing common infixes. The difficulty is mainly due to the needs to differentiate which attack's regular expression is matched. In other words, we need to know exactly which regular expression of attack is matched. In Section IV-A, we first describe the issue incurred by directly sharing common suffixes. In Section IV-B, we propose a new architecture which can share common infixes and suffixes without the differentiation problem. However, the new architecture creates a new problem called the critical section problem. In Section IV-C, we will discuss in detail how the critical-section problem occurs, and also our approach to prevent the critical-section problem.

A. Sharing Common Suffixes

Consider two regular expressions "PassWinDirUserGate" and "MainSysRootNetGate," and both of which have the common suffixes "Gate." A direct but erroneous implementation is to share the hardware block for recognizing "Gate" in Fig. 4. The main problem is that when the output of last block

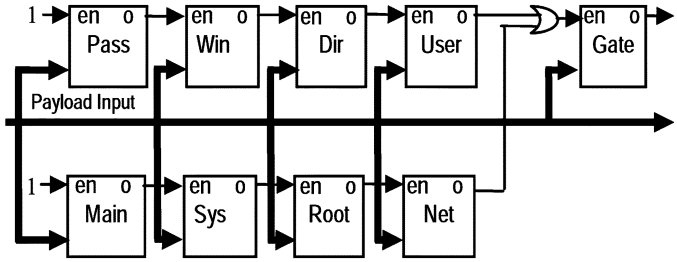


Fig. 4. Erroneous implementation to share the common suffix "Gate."

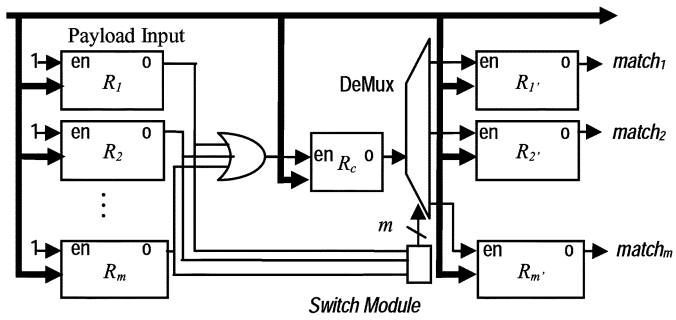


Fig. 5. Sharing architecture for infix and suffix.

is asserted, there is no way to differentiate which virus regular expression is matched by only one output. This is similar to the *differentiation problem* when sharing common infixes directly.

B. Novel Sharing Architecture

In this section, we propose a new sharing architecture to resolve the differentiation problem. Consider m regular expressions which all have a common infix R_c. The m regular expressions can be represented as concatenation forms, $R_1R_cR_{1'}, R_2R_cR_{2'}, \ldots$, and $R_mR_cR_{m'}$. In Fig. 5, the *switch module* in parallel with the common infix R_c is used to memorize which prefix triggers the infix R_c, and the DeMux (demultiplexer) is used to guide the output of R_c to trigger the corresponding successive blocks.

We now illustrate the sharing architecture using an example. Given two regular expressions with a common infix R_c in Fig. 6, the *switch module* can be implemented by a JK flip-flop. The outputs of prefix blocks R_1 and R_2 are connected to the inputs of the JK flip flop. When the R_1 is matched and its output o is asserted a cycle to trigger the R_c via the OR gate, the JK flip flop will memorize this state because $J = 1$ and $K = 0$, the output $Q = 1$, the output $Q' = 0$. The state will be kept even the output o of the R_1 is restored to 0 afterward. As $Q = 1$ and $Q' = 0$, the output o of R_c will be guided to trigger the successive $R_{1'}$ via the DeMux. In contrast, the JK flip flop will guide the output of R_c to trigger the successive $R_{2'}$ if the R_2 triggered the R_c. Note that the prefix blocks, R_1 and R_2, cannot trigger the shared block R_c simultaneously. Otherwise the outputs of JK flip-flop will be complemented each cycle and the R_c may trigger a wrong successive block. By applying the new sharing architecture, the shared block can trigger the proper successive block according to the storage of switch module, preventing the differentiation problem caused by directly sharing common infixes. Similarly, the new sharing architecture can support the sharing of common suffixes.

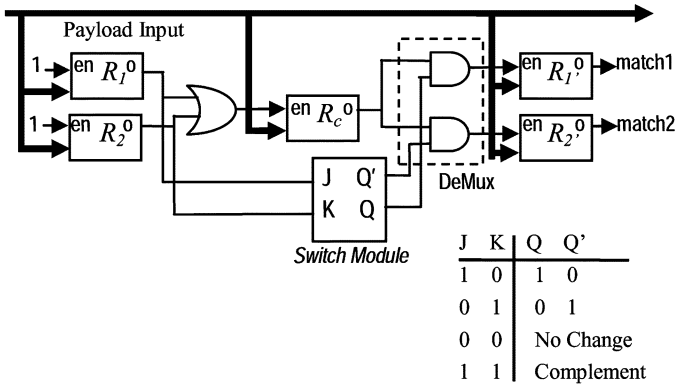


Fig. 6. Two patterns share common infix R_c.

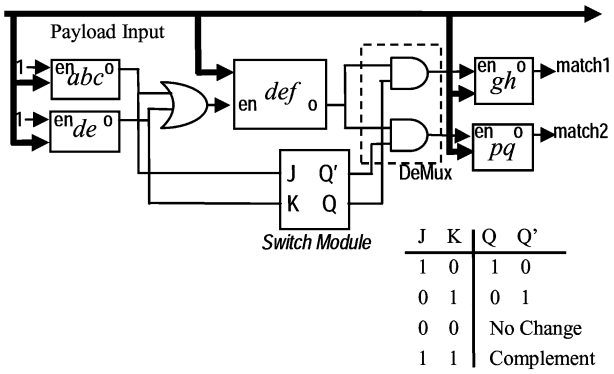


Fig. 7. Example of critical section problem.

C. Critical-Section Problem in the Sharing Architecture

In Fig. 6, whenever a prefix block, R_1 block, or R_2 block completes the matching of a subexpression, its output will trigger the shared block R_c via the OR gate. Note that the shared block R_c may require more than one cycle to process its expression matching. During the process, it is possible that R_c is triggered again. Since the switch module only memorizes one and the only one triggering source, it is important that other prefix blocks cannot trigger the shared block until the shared block completes the expression matching. This is similar to the critical section problem in the operating system, where the critical section can only be entered once.

Fig. 7 shows an example of two patterns, "abc**def**gh" and "de**def**pq," using our sharing architecture to share the common infix "def." Supposing a string "abcdefgh" is fed into the circuit, the circuit should identify that the string is matched with the pattern "abcdefgh." But, it fails because when the string "abc" is fed, the prefix "abc" is matched and the critical section "def" is triggered first while the switch module memorizes the source of triggering signal comes from the prefix "abc." Then, when the string "**de**" is fed, the prefix "de" is matched and the critical section is retriggered while the shared block does not complete expression matching yet. The retriggering to the critical section will complement the state of switch module and cause the output of "def" to trigger "pq." In other words, the retriggering to the shared block leads to the failure of matching the successive "gh" correctly. We can see that the two triggers of "de" are not mutual exclusive in time and the critical section problem arises. Therefore, when similar patterns like this example are detected, our algorithm will avoid the sharing of the common parts to prevent critical section problem.

Some shared blocks may have the critical section problem while unshared blocks do not have. One way to prevent the critical section problem is to avoid the sharing when it is possible to have the critical section problem. In the following theorem, we show a necessary condition for the critical section problem. Therefore, we can safely share the common subexpressions without the critical section problem if the necessary condition does not satisfy.

Definition: An expression, R' is called the ***cross-subexpression*** of R_1R_2 if R' is not a subexpression of R_1 and R' is a subexpression of R_1R_2.

For example, given two expressions $R_1 =$ "abc" and $R_2 =$ "def," expression "*cd*" is a cross-subexpression of R_1R_2 because "cd" is not a subexpression of "abc," but a subexpression of "abcdef." Similarly, expressions "cde," "cdef," "bcd," "bcde," and "bcdef" are all cross-subexpressions of R_1 and R_2.

Let R_c be a common subexpression of two regular expressions $P_1 = R_1R_c$ and $P_2 = R_2R_c$.

Theorem: If R_c has the critical section problem, either R_1 is a cross-subexpression of R_2R_c, or R_2 is a cross-subexpression of R_1R_c.

Proof: The critical section problem arises when the shared block is triggered again before completing the expression matching. Suppose the shared block is triggered because earlier inputs matched to R_1 and is currently processed to check if subsequent inputs are matched to R_c. In order for the shared block to be triggered by R_2, R_2 must be a cross-subexpression of R_1R_c. Similarly, supposing the shared block is triggered by R_2 and is currently processed to check if subsequent inputs matched to R_c. In order for the shared block to be triggered by R_1, R_1 must be a cross-subexpression of R_2R_c. As long as R_1 or R_2 is a cross-subexpression, the critical section problem will happen. ■

V. Regular Expression to NFA Hardware Implementation

In this section, we describe the hardware implementation of a regular expression. The NFA approach [1] has shown four basic NFA components: single-character matcher, concatenation, union (|), and Kleene-star (*) in Fig. 8. The hardware for matching a normal regular expression can be constructed by connecting the four basic NFA components. In order to support the regular expression patterns of Snort [11] and Trend Micro, we develop another five NFA basic components to support Perl-compatible regular expressions (PCRE), as shown in Fig. 9. These components include *any-character* matcher $(\cdot)$, *complementing-character* matcher $(\hat{})$, question mark quantifier (?), plus quantifier $(+)$, and dollar sign anchor (\$). The any-character matcher is used to match any input character [see Fig. 9(a)]. The complementing-character matcher is used to match the characters outside of a range by complementing the set [see Fig. 9(c)]. Similarly, given a regular expression $R, R?$ matches any string composed of zero or one occurrences of R [see Fig. 9(b)]. $R+$ matches any string composed of one or more occurrences of R

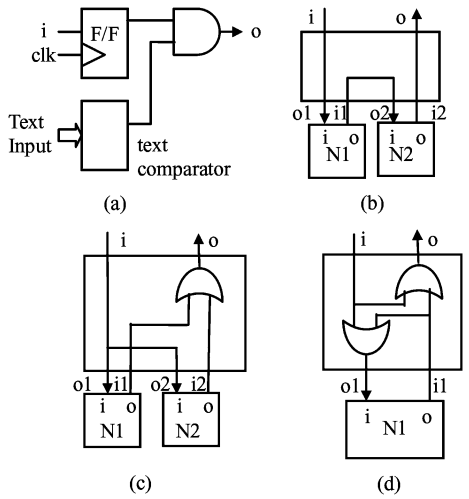


Fig. 8. Four basic NFA components [1]: (a) single-character; (b) concatenation; (c) union; (d) kleene-star.

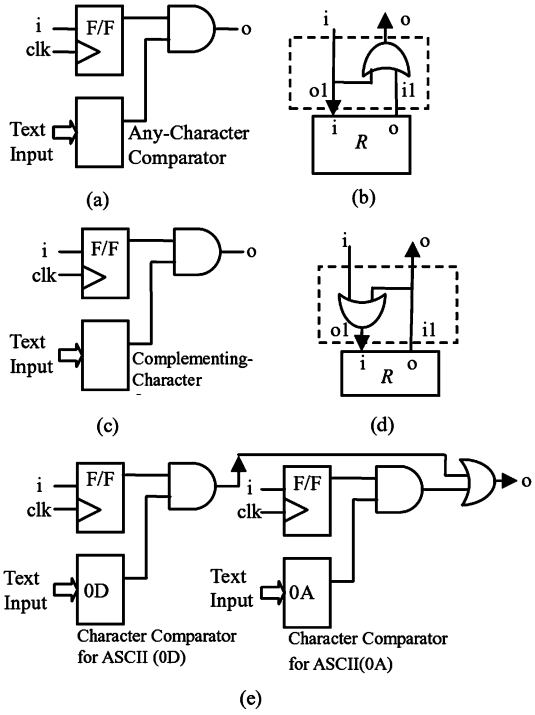


Fig. 9. New NFA components to support PCRE: (a) any-character matcher $(\cdot)$; (b) questionmark $(?)$ quantifier (dashed box); (c) complemting-character $()$ matcher; (d) plus quantifier $(+)$ (dashed box).

[see Fig. 9(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 Fig. 9(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? $\cdot[\hat{}c]d+$, is shown in Fig. 10.

VI. Regular Expression Module Generator

In order to automatically extract and share the common subexpressions and convert them to NFA hardware components, we develop a regular expression module generator that can explore

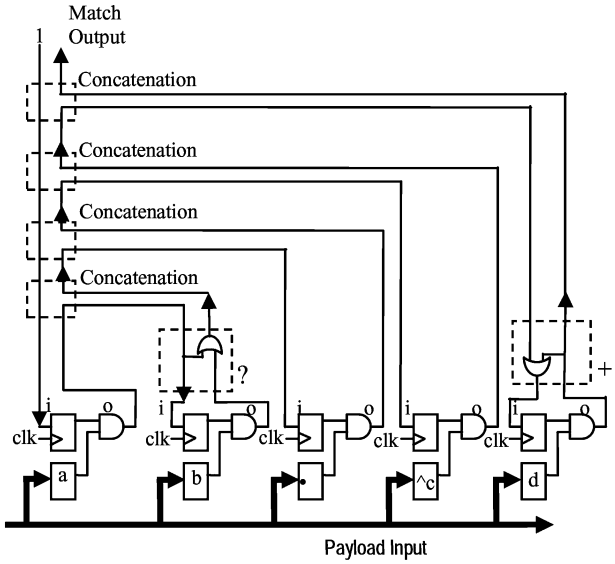


Fig. 10. Implementation of NFA for ab? $\cdot[\hat{}c]d+$.

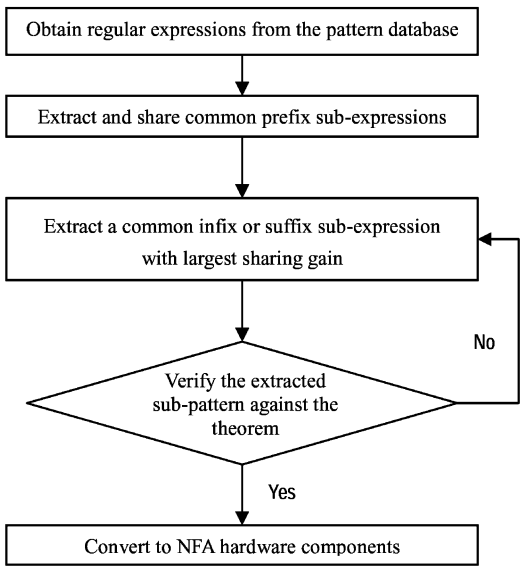


Fig. 11. Flow of regular expression module generation.

the sharing of common prefix, infix, and suffix subexpressions. The flow diagram of our generator is shown in Fig. 11. In the first stage, we obtain regular expression patterns from the pattern database. Then in second stage, common prefix subexpressions are shared directly. In the third and fourth stages, we recursively extract one common infix or suffix subexpression which has the largest *sharing gain* defined as follows. The sharing gain of a common subexpression is defined to be the number of characters in the subexpression multiplies by the number of regular expressions having that subexpression. For example, three regular expressions, "1Common1," "2Common2," and "3Common3" have the common subexpression "Common." The sharing gain of the common subexpression is $6 \times 3 = 18$ because "Common" has 6 characters and the number of regular expressions is 3. In our experiment, because the sharing also has hardware overhead, we heuristically fine-tune the sharing gain according to the results of area. The stages three and four continue until no common

TABLE I
EXPERIMENTAL RESULTS AMONG DIFFERENT APPROACHES ON SNORT RULE SETS

Rule Set	# of Characters	Original Design			Sharing Prefix [2]			Decoder [3]			Decoder tree [3]			Our Sharing Architecture			Integration Approach			
		Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Throughput (Mbps)
Oracle	4,659	1,210	3.9	9.84	1,185	3.9	8.75	1,313	3.5	7.97	1,294	3.6	7.87	898	5.2	8.11	995	4.7	6.18	1294
Web-iis	2,047	1,099	1.9	9.87	1,097	1.9	9.22	1,004	2.0	7.97	1,029	2.0	7.81	957	2.1	8.30	874	2.3	6.92	1155
Web-php	2,455	1,355	1.8	9.93	1,370	1.8	8.59	1,192	2.1	7.92	1,196	2.1	7.89	918	2.7	8.29	904	2.7	6.91	1158
Web-misc	4,711	2,603	1.8	9.90	2,323	2.0	9.86	2,118	2.2	7.98	2,331	2.0	7.91	2,441	1.9	9.70	1,993	2.4	6.99	1145
Web-cgi	5,339	2,720	2.0	9.94	2,883	1.9	9.95	3,184	1.7	8.00	2,691	2.0	7.97	1,653	3.2	9.68	1,620	3.3	7.73	1035
subset1	14,896	8,795	1.7	11.53	10,750	1.4	9.97	7,366	2.0	10.86	7,403	2.0	9.60	7,072	2.1	9.90	6,850	2.2	8.99	890
subset2	9,318	5,054	1.8	9.97	5,652	1.6	9.93	4,665	2.0	9.94	4,666	2.0	9.97	3,446	2.7	9.88	3,488	2.7	7.95	1007
subset3	21,813	13,311	1.6	9.99	12,777	1.7	9.96	9,988	2.2	9.99	10,107	2.2	9.95	9,371	2.3	10.33	9,462	2.3	9.55	837
Total rule	24,214	16,546	1.5	9.93	16213	1.5	9.99	11,909	2.0	13.49	11,865	2.0	9.94	10766	2.2	9.99	10,919	2.2	9.97	803
Average	8,155	4,518	2.1	10.10	4,755	1.7	9.58	3,854	2.1	9.35	3,840	2.1	8.77	3,345	2.4	9.35	3,273	2.5	7.91	1065
RATIO		100%	100%	100%	105%	83%	95%	85%	103%	93%	85%	103%	87%	**74%**	118%	**93%**	**72%**	121%	**78%**	

TABLE II
EXPERIMENTAL RESULTS AMONG DIFFERENT APPROACHES ON TREND MICRO RULE SETS

Rule Set	# of Characters	Original Design			Sharing Prefix [2]			Decoder [3]			Decoder tree [3]			Our Sharing Architecture			Integration Approach			
		Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Area (Slices)	char / slice	post-layout Minimum period (ns)	Throughput (Mbps)
Worstcase	6,465	7,752	0.8	10.69	8,843	0.7	9.88	5,048	1.3	6.40	5,054	1.3	7.28	6,229	1.0	7.97	3,961	1.6	7.31	1095
Combined	12,950	9,678	1.3	9.99	9,877	1.3	9.96	9,187	1.4	9.96	9,208	1.4	9.94	7,667	1.7	9.99	7,387	1.8	8.98	891
Normal	13,152	13,571	1.0	11.71	12,440	1.1	9.96	9,611	1.4	8.98	9,622	1.4	11.25	8,295	1.6	9.97	7,753	1.7	8.97	892
Average	10,856	10,334	1.1	10.80	10,387	1.0	9.93	7,949	1.4	8.45	7,961	1.4	9.49	7,397	1.5	9.31	6,367	1.7	8.42	950
RATIO		100%	100%	100%	101%	99%	92%	77%	130%	78%	77%	130%	88%	**72%**	140%	**86%**	**62%**	162%	**78%**	

subexpression can be shared. Note that a shared common subexpression must not cause the critical section problem described in Section IV-C. In the final stage, we convert the regular expression patterns to the NFA hardware components.

VII. EXPERIMENTAL RESULTS

We implement the algorithm shown in Fig. 11 and apply to the regular expression patterns from Snort and an industry company, Trend Micro. The results are compared with the approaches of sharing only common prefixes as in [2] and sharing decoder [3], [4]. All circuits are synthesized by the commercial tool, Synplify Pro 7.7.1 and placed and routed by Xilinx ISE 8.1i, where the target FPGA is Xilinx Virtex2 XC2V6000 consisting of 33 792 slices.

In addition to total rules of Snort, we also implemented six different approaches on the largest eight subsets of regular expressions from Snort and three sets from Trend Micro for our experiments. The first approach is the traditional NFA approach [1]. The second proposed in [2] extended the first approach by adding a prefix tree to share common prefixes. The third is to share character decoder, called "decoder" approach [3]. The fourth is to share character decoder with prefix tree, called "decoder tree" approach [3]. The fifth is based on our original algorithm and the sixth is an integration of our algorithm and "decoder tree" approach, called integration approach.

Table I lists the comparison of characters, area, and delay among different approaches on Snort rule sets, and Table II on industrial rule sets of Trend Micro. The name of the set and the number of characters are shown in the first and second columns. The number of area, character per slice, and minimum period of original circuit are shown in the third, fourth, and fifth columns. The results of sharing common prefixes [2] are shown in the

sixth, seventh, and eighth columns. The results of "decoder" approach are shown in the ninth, tenth, and eleventh columns. The results of "decoder tree" approach are shown in the twelfth, thirteenth, and fourteenth columns. The results of our sharing architecture are shown in the fifteenth, sixteenth, and seventeenth columns. Finally, we apply the integration approach to the same rule set and report the number of area, character per slice, minimum period, and throughput in the last four columns. For example in the first row of Table I, the Snort Oracle rule set has 4674 characters. The area of the original design on FPGA is 1210 slices and the character per slice is 3.9. The minimum period after place and route process is 9.84 ns. Applying the technique of sharing common prefixes [2], the area, character per slice, and minimum period are reduced to 1185, 3.9 slices, and 8.75 ns. Applying the "decoder" approach, the area, character per slice, and minimum period are 1313 slices, 3.5 and 7.97 ns. Applying the "decoder tree" approach, the area, character per slice, and minimum period are 1294 slices, 3.6, and 7.87 ns. Applying our sharing architecture, the area, character per slice, and minimum period are 898 slices, 5.2, and 8.11 ns. By integrating our sharing architecture with the "decoder tree" approach, the area, character per slice, minimum period, and throughput are 995 slices, 4.7, 6.18 ns, and 1294 Mb/s.

The experimental results show that the integration approach on the Snort rule sets can achieve an average of 28% in area reduction and the reduction is 38% on industrial rule sets of Trend Micro. The integration approach has the best area reduction than previous approaches. The results show that our approach is very efficient when combined with the predecoding approach for the area minimization.

Furthermore, although our approach is aimed at optimizing the area on FPGA, the circuit delay is also improved because the sharing architecture can reduce the fan-out load of the payload input and alleviate the routing complexity. The integration approach achieves an average of 22% in delay reduction both on Snort rule sets and industrial rule sets of Trend Micro.

VIII. Conclusion

Regular expressions are widely used in the NIDS to represent attack patterns. To accommodate a large number of regular expressions to FPGAs, the area reduction of the pattern matching circuits is very important. In this paper, we present a novel sharing architecture allowing our algorithm to extract and share common prefixes, infixes, and suffixes. Under specific condition, both the common infix and suffix subexpressions can be extracted and shared effectively. Additionally, in order to support Perl-compatible regular expressions (PCRE), we also developed five important NFA components. An automatic generation tool is also presented to cost-effectively extract the common subexpressions for FPGA implementation. The experimental results show that our sharing architecture can significantly reduce the area of the pattern matching circuits both for the Snort and industrial realistic regular expression rule sets. In addition, the results show that our approach is very efficient when combined with the predecoding approach for the area minimization.

Moreover, because our sharing architecture can effectively reduce the fan-out load of the text inputs and alleviate the routing complexity, the circuit delay is also improved a lot.

Acknowledgment

The authors would like to thank the following experts of the Trend Micro Inc., M. Deng (Group Project Manager), S. Chin (Project Manager), C. Lo (QA Manager), V. Lo (Development Manager), K. Kuo (Development Manager), V. Ho (Sr. Engineer), P. Chiang (Project Lead), R. Mier (QA Project Lead), and K. Chiang (Engineer) for their constructive inputs.

References

[1] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. 9th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2001, pp. 227–238.

[2] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. 10th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2002, pp. 111–120.

[3] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns," in *Proc. 11th ACM/SIGDA Int. Conf. Field-Program. Logic Appl. (FPL)*, 2003, pp. 956–959.

[4] C. R. Clark and D. E. Schimmel, "Scalable parallel pattern matching on high speed networks," in *Proc. 12th Ann. IEEE Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2004, pp. 249–257.

[5] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized hardware for deep network packet filtering," in *Proc. 10th ACM/SIGDA Int. Conf. Field-Program. Logic Appl. (FPL)*, 2002, pp. 452–461.

[6] Y. Cho and W. H. M. Smith, "Deep packet filter with dedicated logic and read only memories," in *Proc. 12th Ann. IEEE Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2004, pp. 125–134.

[7] Z. K. Baker and V. K. Prasanna, "A methodology for the synthesis of efficient intrusion detection systems on FPGAs," in *Proc. 12th Ann. IEEE Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2004, pp. 135–144.

[8] Z. K. Baker and V. K. Prasanna, "High-throughput Linked-Pattern Matching for Intrusion Detection System," in *Proc. Symp. Architecture Netw. Commun. Syst.*, 2005, pp. 193–202.

[9] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *Proc. 11th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2003, pp. 31–38.

[10] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on FPGAs," in *Proc. ACM/SIGDA 12th Int. Symp. Field Program. Gate Arrays*, 2004, pp. 223–232.

[11] M. Roesch, "Snort- lightweight Intrusion detection for networks," in *Proc. 15th Syst. Administration Conf. (LISA)*, 1999, pp. 229–238.

[12] I. Sourdis and D. Pnevmatikatos, "Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion detection system," in *Proc. 11th ACM/SIGDA Int. Conf. Field-Program. Logic Appl. (FPL)*, 2003, pp. 880–889.

[13] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards gigabit rate network intrusion detection," in *Proc. 12th Ann. ACM/SIGDA Int. Conf. Field-Program. Logic Appl. (FPL)*, 2002, pp. 404–413.

[14] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," in *Proc. 12th Ann. IEEE Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2004, pp. 258–267.

[15] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM," in *Proc.12th IEEE Int. Conf. Netw. Protocols (ICNP)*, 2004, pp. 174–183.

[16] J. Moscola, Y. H. Cho, and J. W. Lockwood, "Implementation of network application layer parser for multiple TCP/IP flows in reconfigurable devices," in *Proc. 16th Int. Conf. Field Program. Logic Appl. (FPL)*, 2006, pp. 1–4.

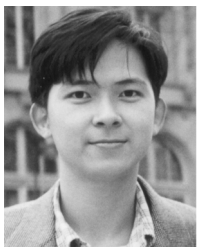
[17] Y. H. Cho and W. H. Mangione-Smith, " pattern matching co-processor for network security," in *Proc. 42nd Des. Autom. Conf. (DAC)*, 2005, pp. 234–239.

[18] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *Proc. 11th Symp. High Performance Interconnects*, 2003, pp. 44–53.

[19] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks, "Internet worm and virus protection in dynamically reconfigurable hardware," in *Proc. Military Aerosp. Program. Logic Device (MAPLD)*, 2003, p. E10.

Cheng-Hung Lin (S'06) received the B.S. and M.S. degrees in industrial technology education from National Taiwan Normal University, Taiwan, R.O.C., in 1994 and 1997, respectively. He is currently pursuing the Ph.D. degree in computer science from the National Tsing Hua University, Hsinchu, Taiwan, R.O.C.

His current research interests include network intrusion detection and related computer-aided design (CAD) techniques.



Chih-Tsun Huang (S'98–M'01) received the Ph.D. degree in electrical engineering from the National Tsing Hua University (NTHU), Hsinchu, Taiwan, R.O.C., in 2000.

He is currently an Assistant Professor with the Department of Computer Science, NTHU, where has been since 2004. His research interests include security and error-correction VLSI designs, core-based SOC/IP designs, VLSI/SOC design and test, and embedded memory testing and repair.

Prof. Huang was a recipient of the Best Paper Award of the 2003 IEEE Asia and South Pacific Design Automation Conference (ASP-DAC) and the Special Feature Award of the 2003 ASP-DAC University LSI Design Contest.

Chang-Ping Jiang received the B.S. and M.S. degrees in computer science from the National Tsing Hua University, Hsinchu, Taiwan, R.O.C., in 2004 and 2006, respectively.

In 2006, he joined the Springsoft, Inc, Taiwan, R.O.C., where he is currently an Engineer with the Research and Development Group 1. His research interests include network intrusion detection, pattern matching circuit, and related computer-aided design (CAD) techniques.

Shih-Chieh Chang (S'92–M'95) received the B.S. degree in electrical engineering from the National Taiwan University, Taiwan, R.O.C., in 1987, and the Ph.D. degree in electrical engineering from the University of California, Santa Barbara, in 1994.

He is currently a Professor and Vice Chairman with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, R.O.C. From 1995 to 1996, he worked with Synopsys, Inc., Mountain View, CA. From 1996 to 2001, he joined the faculty with the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan, R.O.C. His current research interests include logic synthesis, functional verification for SoC, and noise analysis.

Dr. Chang was a recipient of a Best Paper Award at the 1994 Design Automation Conference.