2010 24th IEEE International Conference on Advanced Information Networking and Applications

The Cost Effective Pre-Processing based NFA Pattern Matching Architecture for NIDS

Yeim-Kuan Chang, Chen-Rong Chang, and Cheng-Chien Su
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, 701, Taiwan
{ykchang, p7697446, p7894104}@mail.ncku.edu.tw

***Abstract*—Network Intrusion Detection System (NIDS) is a system which can detect network attacks resulted from worms and viruses on the Internet. An efficient pattern matching algorithm plays an important role in NIDS. There have been many proposed methods for pattern matching algorithms. Traditionally, the multi-character NFA that is capable of matching multiple characters per cycle can be built by duplicating entire circuit of 1-character architecture. In this paper, we propose a pre-processing based architecture to improve the original multi-character architecture. The design of the proposed architecture and its implementation in FPGA are described in details. Our simulation results show that the proposed architecture performs better than all the existing Brute-Force based approaches in terms of the throughput and the slice utilization. Specifically, the proposed architectures of 2-character and 4-character designs can achieve the throughputs of 4.68 and 7.27 Gbps and the slice utilization of 2.86 and 2.10 in terms of char/slice, respectively.**

Keywords-Intrusion detection; Finite state machine; Multi-character; Pattern matching

I. INTRODUCTION

A large number of malicious attacks and worms spread on the Internet every day. As a result, many networks are vulnerable to the attacks. Network Intrusion Detection System (NIDS) is a detection system which can detect malicious attacks and protect the network systems. The pattern matching algorithm plays an important role in NIDS. Traditionally, pattern matching has software-based and hardware-based solutions. Software-based solutions have their limits in system processing speed. So, there have been several proposed FPGA-based hardware solutions. The deterministic finite automata (DFA) and non-deterministic finite automata (NFA) are typical methods for the pattern matching architectures.

The deterministic finite automata (DFA) approach uses a state machine to track partial pattern matches across clock cycles. For this reason, it is possible to match complex regular expression using this technique. A DFA will take in a string of input character. In DFA definition, a DFA can have only one active state. An advantage of a single active state is a compact state encoding, which allows for efficient context switches that are useful in certain applications. Non-deterministic finite automata (NFA) approach can reduce the transition complexity by allowing multiple active states. NFA has a balance of logic and state that maps well to current architecture, allowing them to achieve compact density.

The most popular real pattern sets are from the open source software such as Snort [10] for intrusion detection and ClamAV [1] for anti-virus. The requirements can be concluded to be those matching the variable-length, multiple patterns and on-line processing of all packet inspection systems.

The rest of the paper is organized as follows: In section II, we review and summarize the related work. In section III, we present our scheme. In section IV, we propose our multi-character architecture and describe the solution of the false positive that may be incurred by our simple design. In section V, we present the multi-character simulation results with a summary of our design and the comparison of our scheme with other similar projects. Finally, in section VI, we present the conclusions.

II. RELATED WORK

Since the throughput of hardware-based solutions is much higher than the software-based ones, many FPGA based implementations are proposed for network intrusion detection in recent years. In this section, we will briefly introduce some pattern matching architectures in previous works.

The brute-force (BF) approach compares the pattern with the packet payload (called input string or text) for each possible substring relative to the beginning of the packet payload. The BF algorithm compares a character in the pattern and a character in the text from left to right. In case of a match or mismatch, it shifts only one position to the right. Take input string “aabbcda” and pattern “bcd” as an example. In the first attempt, the first characters from both input string and pattern are compared. Since they don’t match, the search is shifted one position to the right. The second attempt also results in a mismatch, and the search is shifted one position to the right. In the third attempt, the first characters match, but the second characters don’t match. The search is again shifted one position to the right. The fourth attempt results in a match, and the searched pattern is found in the text. The Brute-Force algorithm has O(*nm*) worst case time complexity, where *n* is the length of text and *m* is the length of pattern.

Figure 1(a) shows the BF architecture whose pattern is “Processor” where the nine blocks are character comparators. Each character comparator compares a 1-byte character of text per clock cycle and outputs TRUE signal if the input character

1550-445X/10 $26.00 © 2010 IEEE
DOI 10.1109/AINA.2010.42

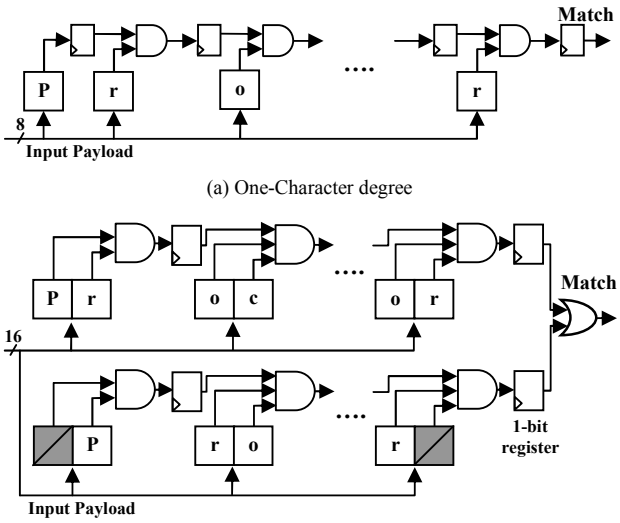


Figure 1. The BF architecture.

matches the desired character of comparator. The output signal of comparator is connected to an AND gate with output signal that were detected in the previous cycle. The final match signal is obtained from the last AND gate. For example, if the input text "Processor" arrives in order, the first comparator matches "P" and output "1" to the flip-flop. In the next clock cycle, the second comparator matches "r" and the output signal combined with the output signal at previous stage by AND gate. The AND gate outputs TRUE to the next flip-flop. Finally, we will get a match signal at ninth clock cycle.

Traditionally, Non-deterministic Finite Automata (NFA) represents each state by a pipelined stage. The longer length of the pattern, the more slice of the circuit is needed. Similarly, if we want to match multiple characters in the pattern per cycle, the cost of hardware will be increased exponentially, compared to single character design. There are many researches focusing on BF in the literature [4, 5, 7, 8, 9, 11, 14, 15]. An example of the BF of multi-character architecture is shown in Figure 1(b). In [4], the proposed architecture uses a multi-character method augmented by a pre-decoder. Figure 1(b) is two-character NFA architecture. However, these proposed methods must duplicate the circuit of one-character architecture. The result is that the multiple-character architecture must match all possible substrings of the input payload and the input character may be shifted by one or more positions in input payload. In order to match all possible substrings of the payload, the circuit must express all kinds of input types. The duplication of the single-character architecture results in the reduced slice utilization of the circuit.

The authors in [5] improved the BF architecture and proposed a hardware-based pattern match architecture by employing a multi-character processor array. The proposed multi-character processor array is a parallel and pipelined architecture which can process multiple characters of the input text per clock. We proposed a regular design for multi-character architecture [5]. In our design, the details of each processor element (PE) are the same. It is easy to implement and increase multi-character degree due to the regularity of the

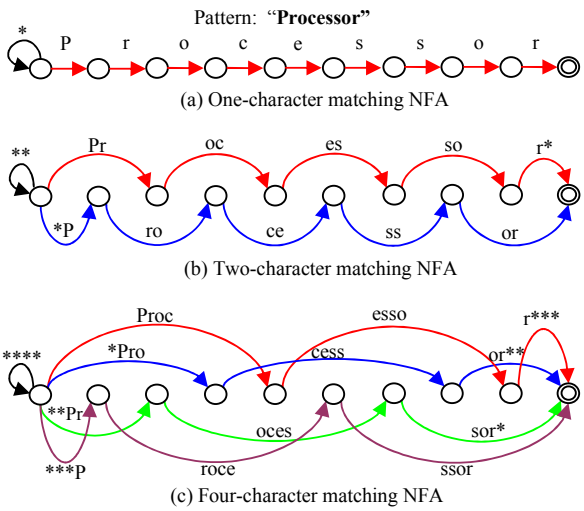


Figure 2. Transition chain of multi-character matching NFA.

PE. In general, it is a trade-off of throughput and area cost. Throughput of the architecture is decreased as area cost is increased. Therefore, we provide a precise method to design multi-character architecture for processing *N* bytes per cycle. The important addition what we have said about our design is we reduce 83% of the computations compared with the brute-force approach.

In [8] the authors have proposed the idea of pattern infix sharing to reduce the number of slices per pattern match engine across many similar patterns. Hutchings *et al.* [7] proposed a method to reduce the area cost of BF approach. They share the circuits of common prefix in different pattern. But this technique is not useful on FPGA. Based on the simulation results by using FPGA software, the area cost of sharing prefix is similar to the cost of BF approach. Processing multiple input characters per cycle is needed in order to improve the throughput. The BF matching module can be scaled by simply widening the bus and adding duplicate modules. But the architecture is not regular and wastes too many computations when comparing characters.

One of the early exact string matching algorithms of Automata design approach is the Aho-Corasick [2] algorithm. The Aho-Corasick algorithm locates all occurrences of any keywords in a text string. It works in constructing a finite state pattern matching machine from all of the keywords, and then using the pattern matching machine to process the payload string in a single pass. The state machine starts from an empty root node. Each pattern to be matched adds states to the machine, starting at the root and going to the end of the pattern. The state machine is then traversed and failure pointers are added to indicate any disconnection between two states. The time complexity of Aho-Corasick algorithm is linear in the size of the input.

Brodie et al. [3] also improve the throughput by processing the multiple characters at each clock cycle. They converted the regular expression patterns into DFAs and implemented them with pipelined FSM structures specially designed for regular expression matching. They also use a complicated alphabet encoding scheme and a transition table compression to reduce

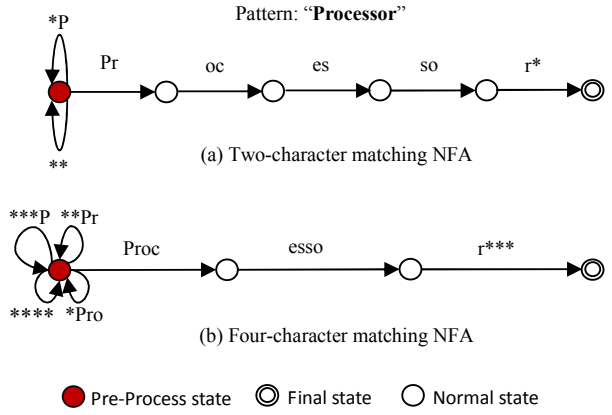


Figure 3. Reduce transition edge and stage for *Pre-Process State*.

the exponentially increasing number of states.

Tan *et al.* [13] proposed a bit-split state machine by splitting Aho-Corasick state machine. The proposed algorithm works by separating the set of the patterns into group and building a small state machine for each group. Each state machine recognizes a subset of the patterns from the rule set. The disadvantage of Aho-Corasick method is that building a state machine requires an exponential number of states. This results in large amount of storage. Tan *et al.* [13] split the state machine into a new set of many binary state machines. The advantage of this method is that binary state machines can be run independently the number of states is reduced.

III. PROPOSED SCHEME

All transition states in traditional NFA graph are matched against the input character simultaneously. We find the chain with the least transition edge length from the initial state to a finish state in transition graph of NFA. The chain is called "*transition chain*" which has an equal number of states as the pattern length. The NFA is implemented in a pipelined architecture, and each state of transition chain is a pipeline stage. For example, with the pattern "Processor", we implement the pipeline architecture as in Figure 2(a). The pipeline architecture is equal to Brute-Force architecture of Figure 1(a). However, one-character matching NFA pipeline architecture implementation is simple in hardware, but its throughput is disappointing. In order to improve the throughput, we usually use the multi-character matching NFA architecture. We can build the transition graph of multi-characters NFA and find the multiple transition chains in the graph. Figure 2(b) is the same example that finds two transition chains in two-characters matching NFA. Each chain must be implemented as a pipeline circuit of two-character matching NFA as shown in Figure 1(b).

Although two-character matching NFA pipeline architecture improves the throughput, it doubles the number of character comparators. The multi-character matching NFA increases transition edges because there are many possible substrings in the input string that will match the pattern. To reduce the doubled cost of the character comparators, we propose to use a pre-processing based scheme as follows. Take a 2-character case as an example. In the original 2-character NFA architecture, there are two sets of 1-character NFA circuits, one is called the non-shift condition and the other is called shift condition. We try to avoid using the shift condition NFA by detecting if the input string can be matched by the shift condition NFA. Then, we can adjust the input string in the pre-processing unit to make it non-shift condition.

TABLE I. PRE-PROCESS STATE OPERATIONS

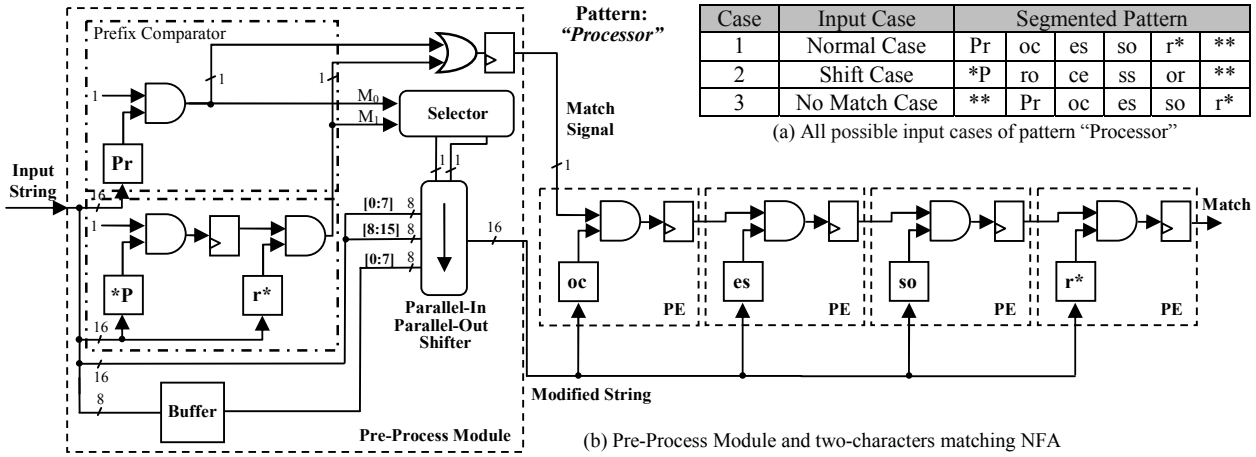
Input payload "ABCProcessor"				
Cycle	2-character input	Pre-Process State output string	Combined sub-string	Store sub-substring
1	"AB"	"AB"	No action	"B"
2	"CP"	"CP"	No action	"P"
3	"ro"	"Pr"	"P" and "r"	"o"
4	"ce"	"oc"	"o" and "c"	"e"
5	"ss"	"es"	"e" and "s"	"s"
6	"or"	"so"	"s" and "o"	"r"
7	"**"	"r*"	"r" and "*"	"*"

We propose a *pre-processing* based multi-character matching transition graph as shown in Figure 3 in which the initial state is built as a pre-processing state. We use two-character matching NFA of the Figure 3(a) as an example to explain how the proposed architecture works. We assume input string is "**ABCProcessor"**. The 2-character input is "**AB**" in the first cycle which does not match any character. In the second cycle, the 2-character input is "**CP**" which partially matches the prefix of the pattern "***P**". In this cycle, the pre-process state will hold partial matching string with "P". In the next cycle, the 2-character input is "**ro**". The pre-process state will combine the 2-character inputs in the current and previous cycles and send combined string "**Pr**" to next state for matching. At the same time, the leftover character "**o**" is held for waiting for the 2-character input in the next cycle. We summarize the operations in all the cycles in Table I. The differences between the traditional NFA and pre-processing based NFA are as follows. In traditional multi-character scheme, total transition NFA edges increase as the number of input characters increases. And the number of states keeps the same. In the proposed pre-processing based transition graph, the number of the states becomes approximate *1/n* of the traditional NFA if the number of input characters is *n*. The transition edge of each stage is only one. When we increase the number of input characters, the pre-processing based NFA keep as simple as the 1-character NFA.

IV. PROPOSED ARCHITECTURE ON FPGA

In this section, we implement the proposed architecture in FPGA. We design a *Pre-Process Module* (PPM) which can handle the operations needed in the pre-processing state. In order to effortlessly explain the proposed *PPM* design, we use the same example described in previous section. For example in Figure 4(a), the *PPM* can handle the matching operations of three cases in which *PPM* should indicate a control signal.

In case 1, the input string exactly matches the pattern "**Processor**" at each clock cycle. For initial cycle, the packet payload will match "**Pr**". In the second and three cycles, the "**oc**" and "**es**" have been matched in this case. For simplifying

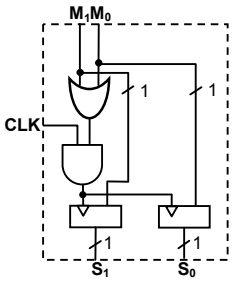


Case	Input Case	Segmented Pattern					
1	Normal Case	Pr	oc	es	so	r*	**
2	Shift Case	*P	ro	ce	ss	or	**
3	No Match Case	**	Pr	oc	es	so	r*

(a) All possible input cases of pattern "Processor"

(b) Pre-Process Module and two-characters matching NFA

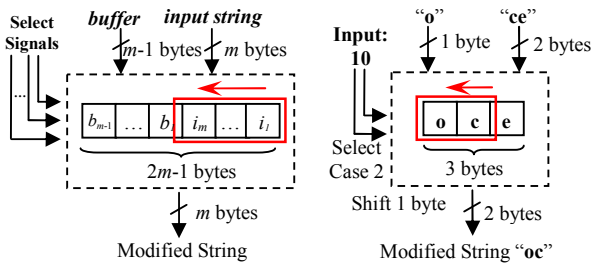
Figure 4. The proposed architecture.



Characteristic Table

S_1	S_0	M_1	M_0	S_1(t+1)	S_0(t+1)
0	0	0	0	Indeterminate	
0	1	0	0	0	1
1	0	0	0	1	0
1	1	0	0	Indeterminate	
*	*	0	1	0	1
*	*	1	0	1	0
*	*	1	1	Indeterminate	

Figure 5. Selector.



(a) The *m*-byte Shifter (b) The pattern"Processor" (*m*=2)

Figure 6. Parallel-In Parallel-Out Shifter.

description of the proposed architecture later, case 1 is called the "*Normal Case*".

The pattern has been cut by *m*-character, where *m* is the number of input characters per cycle. In case 2, the input string matches *k*-byte suffix characters in first cycle and the non-match (*m*−*k*)-bytes prefix characters are wildcard. The case 2 is called "*Shift Case*" which may have a (*m*−*1)*-byte shift in pattern. In the example of case 2 of Figure 4(a), *m* is 2 and *k* is 1. And the case 3 is called "*No Match Case*" which does not match any string in initial cycle.

The proposed pre-processing based NFA architecture is constructed by two circuits as shown in Figure 4(b). The left one is the *PPM* and the right one is the two-character matching NFA pipeline circuit. Each comparator can contain two characters. *PPM* uses the input string to determine which case the current input belongs. After *PPM* determines the case, it will output shifted or non-shifted string to the next stage, which is called *modified string* in our architecture.

A. The Detailed Design of Pre-Process Module

The *PPM* is constituted by four components that are *prefix compactor*, *selector*, *buffer*, and *parallel-in parallel-out (PIPO) shifter*.

- The *prefix compactor* is mainly used to compare pattern prefix. When the number of input character is *m*, prefix compactor needs to consider *m* possible shift matching conditions. So, we need *m* prefix compactors. As shown in Figure 4(a), *m* is 2, *PPM* have two prefix compactors, that are "Pr" and "*P"+"r*". If input string is "*P" that "P" and "r" are separated in two continuous cycles, prefix compactor can identify shifted matching condition. Prefix compactor results will be output to the selector.
- The *selector* is a special component that is designed for *PPM* to be used to control PIPO shifter output. If we use the results of prefix compactor to control PIPO shifter, single shifter is not enough to satisfy our requirement. Some mistakes may occur in our architecture, explained as follows. When *prefix comparator* matches a pattern prefix, *PPM* needs to have a component to record if it is a shifted or non-shifted condition. We design the selector for this purpose. The circuit of Figure 5 is the detailed design of the selector. We use flip-flop registers to hold on the signal of the match. The *m*-characters have *m* flip-flop registers. When *m* is 2 as in Figure 5, two registers are needed. As shown in the Characteristic Table of Figure 5, when CLK is triggered, S_0(t+1) and S_1(t+1) are the output signals of the selector. S_0 and S_1 are the output signals in the previous cycle. When M_1 or M_0 is set because of pattern prefix is matched at the current cycle, the *selector* will know to output shifted or non-shifted control signal to *buffer* at the next cycle.
- The *buffer* stores the least significant *m – 1* characters of m-character input string in the current cycle. In other words, when input string is "i_m ... i_1", the *buffer* will store "b_{m-1} ...b_1". If *m* is 2, then buffer just stores the last one character of input string, as shown in Figure 4(a). Buffer outputs the stored *m – 1* characters of the pervious cycle to PIPO shifter which in turn outputs characters to the right side NFA circuit.

n – Pattern length.
m – Multi-character per cycle.
W(i,m) – Subset of pattern which use to compare H(m) where i is index of sub-pattern, and m is length of sub-pattern.
H(m) – Recursive Prefix.

```
Algorithm SearchPrefixPattern(P, m, n)
{
01  H(m) = {p1, p2..., pm};
02  W(i,m) = { pi , pi+1,..., pi+m-1};
03  while (m ≦ n) {
04      for (i = 1 to n-m) {          // Search for each sub-pattern.
05          if (H(m) == W(i,m)){      // Exist same prefix pattern.
06              m=m+1;                // Add one character to prefix.
07              break;
08          }
09      } // end for
10      if (i >= n-m)                 // Return the recursive prefix.
11          return H(m);
12  } // end while
13  return NULL;
}
```

Figure 7. Algorithm for searching the recursive prefix.

- The *Parallel-In Parallel-Out Shifter* (PIPO shifter) is a (*2m–1*)-to-*m* multiplexer, as shown in Figure 6(a). PIPO shifter gets its input data from *buffer* and the m-character input string and select appropriate substring by signals from *selector*. PIPO shifter behaves like the sliding window. Figure 6(b) is the example of two-character PIPO shifter. Assume the initial three cycles have input string "*P", "ro" and "ce", respectively. In the third cycle, PIPO shifter includes the current input string "ce" and temporary input "o". The selector selects case 2 that matches prefix pattern. So sliding window will shift left 1 byte and the output modified string is "oc".

B. The False Positive of Pro-Process Module

In our architecture, we allow all possible matched cases in *PPM*. In Figure 4, we can exactly match this pattern "Processor" from this exercise. The m-character per cycle can have m+1 match cases. Each segmented pattern in *PPM* may have at most m-bytes length. This design can match the most of patterns in rulesets.

But if we have a pattern "PreProcessor", and the pattern can be segmented to "Pr", "eP", "ro", "ce", "ss", and "or" for each cycle. The prefix sub-pattern are "Pr" and "*P" in *PPM*. Assume we have a payload input "RootPreProcessor". There have a special case which may lead to a false positive in our architecture. From the above exercise, the input strings are "Ro" and "ot" in the first two cycles. At the third cycle, the input string is "Pr". The selector will trigger case 1 from its multiplexer. This selected state has been held on since the trigger time. In our design, the multiplexer selects the type of possible cases. In the next several cycles, the suffix of the pattern will be matched. But in the forth cycle, the input payload is "eP", the multiplexer has changed the select signal. This action will be effect the path to match sub-pattern. At the later cycles, the modified payload would not match next sub-pattern.

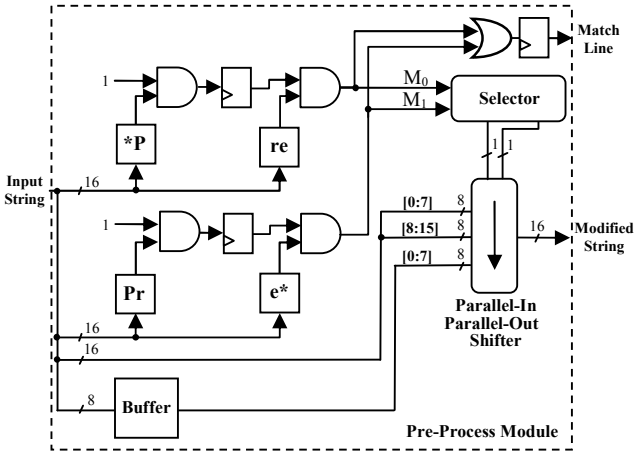


Figure 8. Recursive prefix "pre" of *PPM* where the pattern is "PreProcessor". In two-character match (m=2), the original prefix "Pr" may has false positive to cause mismatch. It should be find out recursive prefix in it.

The reason that results in this problem is that the prefix sub-pattern may repeat in the inner part of the pattern. We call this repeated prefix as *recursive prefix*. This may confuse the match case of multiplexer. In the example above, prefix 'Pr' is the recursive prefix. Let the maximal recursive prefix be the longest possible recursive prefix of the pattern. One way to prevent the false positive problem is to be aware if the maximal recursive prefix is matched or not. Specifically, we have to determine the maximal recursive prefix (say $p_1, p_2...p_r$) and add extra circuit to determine if $p_1, p_2...p_r, p_{r+1}$ is matched or not. Figure 8 shows the design of $m = 2$ for the pattern 'PreProcessor' in which we replace the shift registers of the original design, 'Pr' and '*P'+'r*', with '*P'+'re' and 'Pr'+'e*'. In general, we perform the following steps. We assume that the n-byte pattern is $P = p_1, p_2...p_n$ and m-character per cycle where $m<n$. The prefix pattern bases the length of m-byte which is $H(m)= \{p_1, p_2..., p_m\}$ and $H(m) \subset P$. We want to find out the recursive prefix pattern in P. So we also define $W(i, m)=\{p_i, p_{i+1}, p_{i+2}..., p_{i+m-1}\}$ which is the subset of pattern. The necessary condition which we want to satisfy is $H(m) \neq W(i,m)$. The method to find out the condition is to increase i of $W(i,m)$ sequentially for $i<n-m$. When we find out any sub-pattern and don't satisfy the necessary condition, we increase m by one and continue to perform the same steps until the condition $H(m) \neq W(i,m)$ is satisfied.

We now give the detailed algorithm *SearchPrefixPattern* for finding the maximal recursive pattern prefix in Figure 7. *SearchPrefixPattern* receives, as the input, the n-byte pattern and m-character input string and return the maximal recursive prefix of the pattern.

V. PERFORMANCE EVALUATION

In this section, we present the results of hardware simulation implemented in Xilinx 10.1i. The simulation for each pattern set was synthesized, placed, and routed on the Virtex5 XC5VLX85 [16] chip where the package and speed are FF676 and -3, respectively. The pattern sets are selected from

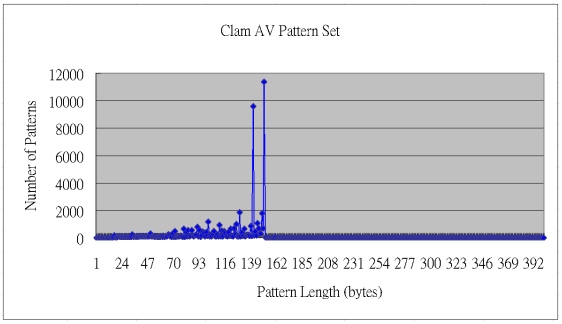


Figure 9. Pattern length of ClamAV set.

TABLE II. EXPERIMENTAL RESULTS ON INPUT *N* CHARACTER PER CLOCK

# of Chars	Input Chars (bits)	Proposed Pre-Process NFA Architecture						
		Slice	# of Register	# of LUT	Char/ Slice	Period (ns)	Throughput (Gbps)	Performance
928	16	360	592	973	2.58	2.70	5.93	0.96
	32	406	301	1105	2.28	3.50	9.13	0.65
1903	16	642	1,173	1,845	2.96	4.41	3.63	0.67
	32	692	568	2,039	2.75	3.49	9.16	0.79
3582	16	1,258	2,224	3,564	2.85	3.41	4.69	0.83
	32	1,380	1,070	4,189	2.60	4.14	7.72	0.62
8417	16	3,025	5,235	8,397	2.78	3.34	4.79	0.83
	32	4,127	3,328	12,171	2.04	3.99	8.01	0.51
16028	16	5,626	9,690	15,374	2.86	3.42	4.68	0.84
	32	7,635	6,550	30,387	2.10	4.40	7.27	0.48

TABLE III.COMPARSION OF PREVIOUS WORKS

Design	Device	Input Chars	# of Chars	Slice	# of Register	# of LUT	Char/Slice	Throughput (Gbps)	Performance
Our design	Virtex5-LX85T	16	16,028	5,626	9,690	15,374	2.86	4.68	0.84
		32		7,635	6,550	30,387	2.10	7.27	0.48
Chang et al. [5]	Virtex5-LX85T	8	1,796	581	1,757	1,773	3.09	2.19	0.85
		16	16,028	7,221	15,640	15,736	2.21	4.67	0.65
		32		15,531	15,943	48,767	1.03	6.27	0.20
Norio Yamagaki et al. Multi-Character NFA [15]	Altera Stratix II EP2S180	16	15,506	n/a	11,640	12,072	n/a	2.67	n/a
		32		n/a	14,765	11,327	n/a	4.00	n/a
Clark et al. NFA decoder [4]	Virtex2-8000	8	7,996	8,852	n/a	n/a	0.90	2.20	0.25
			17,537	17,239	n/a	n/a	1.01	2.02	0.26
		32	7,996	20,500	n/a	n/a	0.39	7.30	0.09
			17,537	37,740	n/a	n/a	0.46	7.00	0.10
Sourdis et al. Discrete Comparators [11]	Virtex2-6000	32	2,457	23,843	n/a	n/a	0.05	8.06	0.01
Hutchings et al Sharing prefix[7]	Virtex2-6000	8	2,008	2,331	n/a	n/a	0.86	0.40	0.04
			4,003	4,375	n/a	n/a	0.92	0.35	0.04
			8,003	10,309	n/a	n/a	0.78	0.25	0.02

ClamAV (Version 0.92) [1] which contains 54351 static patterns. Figure 9 plots the distribution of the pattern lengths varying from 4 bytes to 392 bytes. The average pattern length is 120 bytes. To evaluate whether the proposed implementation performs well or not, we could perform the simulations based on the following issues:

Input Chars (bits): Each character is an 8-bit width data. If we could process more characters per cycle, we might have better throughput.

Slice: Slice is the FPGA resource in Xilinx FPGA chip. The number of logic elements in a slice is dependent on the FPGA device. Number of slices represents the area cost. In Virtex-5, each FPGA slice contains four LUTs and four flip-flops.

Clock period: The clock period is the speed of the maximum critical path in FPGA. The period can be obtained from the synthesis report of Xilinx software. The smaller is clock period, the faster is implementation.

Throughput: Throughput = Input bits / Clock period. So decreasing the clock period or processing more characters per cycle will get better throughput.

Characters/Slice: The "Characters/Slice" indicates the average number of characters that can be implemented by a slice.

In general, clock period and area cost are trade-off. The clock period of the architecture decreases as the area cost increases. These two metrics are both considered when we compare the simulation results with different existing pattern matching architectures. We use a new metric "Performance" defined to be the area cost divided by implementation speed as follows.

$$Performance = \frac{Area\ Cost}{Speed} = \frac{Char/Slice}{Period}$$

The Table II shows the experiment results. The # *of chars* (the number of characters) is the total number of characters in all the patterns of five pattern sets. The numbers of characters in these pattern sets are from 928 to 16028. Two-character and four-character designs are simulated for each pattern set. The # *of registers* and # *of LUTs* show the utilization of registers and LUTs in our architecture. In this table, we can see that the more number of characters in each pattern sets, the more slice has been used. But we can also observe the # *of register*. The 4-character has lower number of register than 2-character design in each pattern sets. This is because our architecture doesn't

duplicate the same circuit of 1-character NFA to support the matching of all possible substring of the pattern. The increase of *Input Chars*, the # of register will be decrease. Another advantage is that our proposed *PPM* architecture also can reduce design complexity. The only additional cost is the *PPM* whose cost increases as the number of characters input in each cycle increases.

The Table III summarizes the performance comparison between related works and our design. We focus on the Char/Slice metric. Our design has the best throughput in all approaches. We also can see that our design has higher throughput and performance than Brute-Force and Norio Yamagaki *et al.* [15]. The result is that our design has lower hardware complexity. This may effects FPGA synthesis tools to perform the result during placement and routing the circuits.

VI. Conclusion and Future Work

In this paper, we proposed a novel pre-processing-based pattern matching architecture, and implemented it in FPGA. The advantage is that the proposed architecture has a higher slice utilization and a lower hardware complexity. The simulation results show that the proposed architecture performs better than the existing approaches that also are based on Brute-Force scheme, in terms of the throughput and the slice utilization. Specifically, the proposed architectures of 2-character and 4-character designs can achieve the throughputs of 4.68 and 7.27 Gbps and the Char/Slice of 2.86 and 2.10, respectively.

References

[1] Clam Anti-Virus signature database, www.clamav.net.

[2] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliography search," *Comm. Of the ACM*, vol 18, no.6, pp.333-340, 1975.

[3] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching, " In Proceedings of the 33nd Annual International Symposium on Computer Architecture (ISCA'06), pp. 191−202, 2006.

[4] C.R. Clark and D.E. Schimmel, "Scalable pattern matching for high speed networks," In Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines(FCCM '04), pp. 249-257, 2004.

[5] Yeim-Kuan Chang, Ming-Li Tsai and Yu-Ru Chung , "Multi-Character Processor Array for Pattern Matching in Network Intrusion Detection System," In Proceedings of the 22th IEEE International Conference on Advanced Information Networking and Applications (AINA'08), pp. 991-996, 2008.

[6] Young H. Cho, W.H. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," In Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines(FCCM '04), pp. 125-134, 2004.

[7] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," In Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computer (FCCM'02), pp. 111–120, 2002.

[8] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang, "Optimization of Pattern Matching Circuits for Regular Expression on FPGA," *IEEE Transactions On Very Large Scale Integration Systems* (VLSI'07), vol. 15, no. 12, pp. 1303-1310, 2007.

[9] R. Nidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," In Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), pp. 227-238, 2001.

[10] Snort: The Open Source Network Intrusion Detection System, www.snort.org.

[11] I. Sourdis, D. Pnevsmatikatos, "Fast, Large-Scale String Match for a 10 Gbps FPGA-based Network Intrusion Detection System, "In Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03), pp. 880-889, 2003.

[12] Lin Tan, Timothy Sherwood, "Architecture For Bit-Split String Scanning in Intrusion Detection," IEEE mirco., pp. 110-117, 2006.

[13] Lin Tan, Timothy Sherwood, "A High Throughput String Matching Architecture for Intrusion detection and Prevention," In Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05), pp. 112-122, 2005.

[14] Yi-Hua E. Yang, Weirong Jiang and Viktor K. Prasanna, "Compact Architecture for High-Throughput Regular Expression Matching on FPGA", in proceedings of the 4th ACM/IEEE symposium On Architecture For Networking And Communications Systems (ANCS'08), pp.30-39, 2008.

[15] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya, "High-Speed Regular Expression Matching Engine Using Multi-Character NFA," In Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08), pp. 697-701, 2008.

[16] Xilinx Virtex-5 Plarform FPGAs: Detailed description. http://www.xilinx.com/support/documentation/virtex-5.htm.