Multi-Character Processor Array for Pattern Matching in Network Intrusion Detection System

Yeim-Kuan Chang, Ming-Li Tsai and Yu-Ru Chung
Department of Computer Science and Information Engineering
National Cheng Kung University
{ykchang, p7694157, p7695101}@mail.ncku.edu.tw

***Abstract*—Network Intrusion Detection System (NIDS) is a system developed for identifying attacks by using a set of rules. NIDS is an efficient way to provide the security protection for today's internet. Pattern match algorithm plays an important role in NIDS that performs searches against multiple patterns for a string match. Pattern matching is a computationally expensive task. Traditional software-based NIDS solutions usually can not achieve a high-speed required for ever growing Internet attacks. In order to satisfy high-speed packet content inspection, hardware-implementable pattern match algorithm is required. In this paper, we propose a hardware-based pattern match architecture that employs 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 stream per cycle. The proposed architecture can reduce a lot of unnecessary computations and thus it is power efficient. We use Snort pattern sets and DEFCON packet traces to perform our simulations. Our experiment results show that, with a 3-character processor array, we can reduce 83% of the computations compared with the brute force approach.**

Keywords: intrusion detection, pattern matching, Snort, processor array

I. INTRODUCTION

With the growth of the Internet, large number of viruses and malicious probes spread every day. Many network users are vulnerable to attacks. Traditionally, networks have been protected using firewalls that monitor and filter network traffic. Firewalls usually examine the packet headers to determine whether or the packets are allowed to pass through or dropped. For example, if a user attempts to connect to a disallowed port, such as port number 80 (http), the connection is rejected. However, firewalls are not effective to protect networks from worms and viruses. Today, the most commonly used defense strategy is to use end-host based solutions that rely on security tools, such as antivirus software. The main drawback of these approaches is the inability to protect thousands of hosts in less than an hour.

Network intrusion detection systems (NIDS) are utilized to detect malicious attacks and protect Internet system. The intrusion detection system are growing in popularity because they provide an efficient protection to attacks. The NIDS differs from a firewall in that it needs to scan both the headers and the payloads of each incoming packet for thousands of suspicious patterns. By inspecting both packet headers and payloads to identify attack signatures, NIDS is able to discover whether malicious attacks or hackers are attempting to intrude.

Because most of the known attacks can be represented with patterns or combinations of multiple sub-patterns, pattern matching has become a performance bottleneck in intrusion detection system. Current NIDS pattern databases consist of thousands of patterns and thus searching against them is a computationally expensive task. Traditionally, software-based NIDS may be overloaded when the packet arrival rate becomes high. To keep up with the high-speed networks, a hardware-based NIDS implementation is generally needed.

The rest of the paper is organized as follows: In section 2, we review and summarize the related works. In section 3, we present the proposed multi-character processor array design. The performance evaluation results are presented in section 4. Finally, we present the conclusions.

II. RELATED WORK

In the past few years, several interesting algorithms and techniques have been proposed for multiple-pattern matching in the content of network intrusion detection. Current software-based NIDS cannot meet the bandwidth requirement of a multiple-gigabit network. Hence, several hardware-based solutions have been proposed to solve the problem. In this chapter, we divide common pattern matching solutions for intrusion detection system into four categories: software-based, FGPA-based, Bloom filter based, TCAM based, and brute force approaches.

A. Software-based solutions

Knuth-Morris-Pratt (KMP) [8] and Boyer-Moore (BM) [3] are the most well-known single-pattern matching algorithms. Assume the length of the pattern is m and the length of text is n. The KMP algorithm [8] utilizes a precomputed table to prevent redundant comparisons. KMP reduces the worst case running time from $O(n \times m)$ to $O(n+m)$. The precomputed table, or π-table, tells the automata which pattern character to match against next. That is, the π-table indicates that, when a mismatch occurs, we know which position of the pattern to continue the matching process without starting the matching over from the beginning of the pattern. The time complexity of BM algorithm is sub-linear of $O(n/m)$ in the best case, and $O(n)$ in the average case . However, the performance of the BM algorithm depends on the characters in the input string and the pattern. The worst-case complexity to find all occurrences in a text needs approximately $3 \times n$ comparisons regardless whether the text contains a match or not. Hence the worse-case time complexity is $O(n \times m)$. Current Snort implementations use Boyer-Moore algorithm because BM has the best average-case performance. The Aho-Corasick(AC) [1] is designed for multiple-pattern matching. By pre-processing the patterns and we build a finite state machine in AC. AC algorithm can process the input text in a linear time, so the searching complexity of the algorithm is linear in the

1550-445X/08 $25.00 © 2008 IEEE
DOI 10.1109/AINA.2008.119

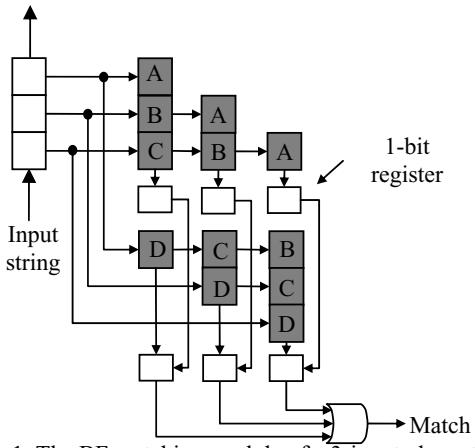

Figure 1. The BF matching module of a 3-input-character design for pattern 'ABCD'.

length of the input text. Although the AC algorithm is theoretically independent of the pattern set size, it will become slow for a large pattern set in practice because of the worse cache locality in accessing a large transition table.

B. FPGA solutions

With emergence of new worms and viruses, the rule set is constantly updated. Therefore, traditional intrusion detection systems must be capable of being re-programmed. Software implementation on general purpose processors can not meet the performance requirement of the intrusion detection systems. Many hardware-based algorithms have been proposed, of which many solutions are based on Field Programmable Gate Arrays (FPGAs). FPGAs can be programmed for fast pattern matching due to their exploitation of reconfigurable hardware capability and their ability for parallelism. In recent years, there are several researches on FPGA pattern-matching for networks intrusion detection system. The three main design approaches that have been described are brute-force, deterministic finite automata (DFA), and non-deterministic finite automata (NFA).

The *brute force* (*BF*) approach compares the pattern with the packet payload for each possible shift relative to the beginning of the packet payload. At each clock cycle, the window of the input stream compares with the pattern stored in comparators and then slides forward by one byte. All the partial matches that were detected in the previous cycle are combined by an *AND* gate to produce the final match result. This brute force approach requires *O(ms)* comparisons, where *m* is the pattern size and *s* is the input stream size. The performance of this module is only dependent on the bus width and clock rate.

Since the sliding window is advanced by one byte at each cycle results, the throughput of BF approach is still poor, which is linear to the packet payload size. Processing multiple input characters per clock cycle is needed in order to improve the throughput of matching module. The BF matching module can be scaled by simply widening the bus and adding duplicate modules [4]. Figure 1 shows a BF matching module with a 3-input-character design for pattern *"ABCD"*. Each module consists of several 1 through 3 comparators in each stage of the pipelined register.

The drawbacks of this approach are high area cost and large number of unnecessary computations. There are $m \times n$ comparators in an n-character BF module for the pattern of length m. Sharing common comparators were proposed in [4], [9] to reduce the logic area cost. By eliminating duplicated comparators, the logic cost is reduced by 25~50 percent. However, sharing comparators result in complex and long wire design which will in turn decrease the clock rate of the entire design. Besides, the redundant computations are still serious. A large number of comparisons are required for dealing with multiple characters, as shown in Figure 1, because all possible combinations of characters are compared against the input string in parallel.

The deterministic finite automata (DFA) approach [2,7,10] uses a state machine to track partial pattern matches across clock cycles. A DFA will take in a string of input character. For each input character it will then transition to a state given by following a transition function. There is one and only one transition to a next state for each pair of state and input character. For this reason, it is possible to match complex regular expressions using this technique. By definition, a DFA can have only one active state. This requires complex state transition logic and may necessitate a state machine with a large number of states. However, due to their more complex logic, DFA circuits provide lower throughput than brute-force designs and have similar character density.

The non-deterministic finite automata (NFA) is a finite state machine where for each pair of state and input character there may be several possible next states. This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined. The NFA approach reduces the transition logic complexity by allowing multiple active states. Since it is based on a state machine, this technique can support regular expressions. NFA circuits have a balance of logic and state that maps very well to current FPGA architectures, allowing them to achieve higher character density. Sidhu et. al. [9] proposed an approach based on nondeterministic finite automaton (NFA) for regular expression matching.

C. Parallel Bloom Filters

Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element or string is a member of a set. False positives are possible, but false negatives are not. Give a string S, the bloom filter computes k hash functions on it, producing k hashing values ranging from 1 to m. The filter then set k bits in an m-bit vector at the addresses corresponding to the k hash values.

Dharmapurikar *et al.* [5] proposed a multiple-pattern matching solution using parallel bloom filters. The proposed scheme builds a bloom filter for each possible pattern length. The Bloom filter engine reads as input a data stream that arrives at the rate of one byte per clock cycle. Each different pattern length that requires a separate bloom filter is a limit factor. Especially when dealing with very long virus definitions. Furthermore, this scheme needs to inspect every single byte of packet payload so that the maximum throughput is around 2.46Gbps.

D. TCAM solutions

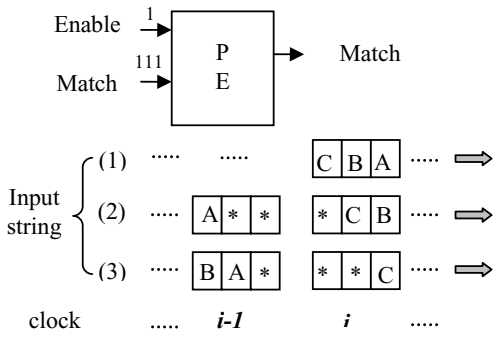

Figure 2. Three possible cases that PE should indicate a match for pattern *P='ABC'* at clock cycle *i*

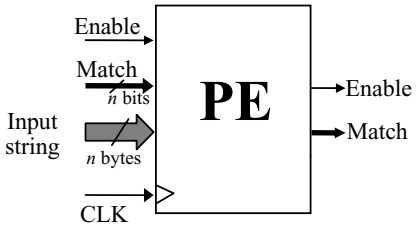

Figure 3. The proposed processing element.

Fang Yu *et al*.[11]proposed a TCAM-based pattern matching algorithm for handling both short patterns and long patterns. If a pattern is shorter than TCAM width, then we put it into TCAM and pad it with don't care bits. A pattern longer than TCAM width will split into several sub-patterns: the first TCAM width prefix pattern and remaining suffix patterns. There are three data structures to be stored in memory for matching long patterns: pattern table, partial hit list, and matching table

To match a pattern, a window of characters from input string is looked up in the TCAM. Then, the result is stored in a temporary table. The window is moved forward by a character and the look up is executed again. At every stage, the approximate partial match table entry is taken into account to verify if a complete pattern matched

III. PROPOSED METHOD

A. Proposed Processing Element Design

In this section, we improve the brute force multi-character architecture design by using the concept of processor array. In the proposed multi-character processor array architecture of degree *n*, each *processing element* (*PE*) can handle *n* characters per clock cycle. The proposed processor array architecture is based on the following observations: instead of enabling all PEs each of which is compared against a window of input data, we only enable some PEs that are necessary to obtain the final match signal. As a result, unnecessary comparisons are significantly reduced and design complexity is also simplified.

B. Details of Processing Element

Our processing element is designed to detect a match of *n*-byte sub-pattern against packet payload. There are *n* possible cases that PE should indicate a match for the input string by using the matching results at the current

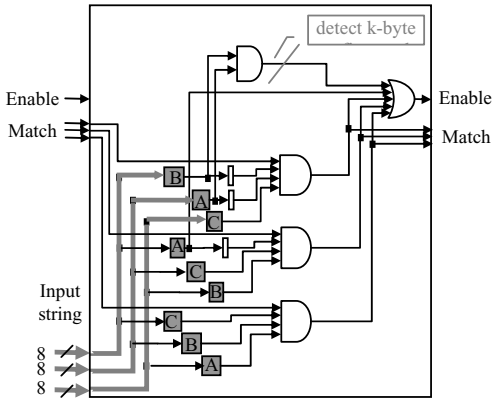

Figure 4. Details of PE with degree 3.

and last clocks. Consider the example in Figure 2 in which each PE of degree 3 stores pattern *'ABC'*. The initial value of two inputs, 'Enable' and 'Match', are set to true. There are three cases that PE should indicate a matching signal:

(1) A window of input string is *'ABC'* at the current clock cycle,

(2) A window of input string are *'BC*'* at the current clock cycle and *'**A'* at the previous clock cycle, where '*' means 'don't care'.

(3) A window of input string are *'C**'* at the current clock cycle and *'*AB'* at the previous clock cycle.

In case (1), the input string exactly matches the pattern *'ABC'* and then a match signal is output immediately. In the cases (2) and (3), the input string matches *k*-byte prefix and (*m*–*k*)-byte suffix of pattern *'ABC'* at the preceding clock and at the current clock, respectively. For simplifying our description later, case (1) is called '*exact PE match*' and the other two cases are called '*partial PE match*'. In order to keep the previous comparison result, shift registers are used to delay the signals of partial PE matching results by one clock cycle.

The PE of degree *n* consists of n^2 8-bit comparators and $n\times(n-1)/2$ SRL16 shift registers. Each *n*-byte pattern is stored in the corresponding comparators of a PE. If pattern is shorter than *n*, we assign a designated value, known as wildcard, to the remaining bytes. If pattern is longer than *n*, we cut it into multiple *n*-byte sub-patterns. It should be noted that the last sub-pattern may be less than *n* bytes.

Figure 3 shows the block diagram of our processing element, which has three inputs and two outputs. A PE is active if and only if its input 'Enable' is set. The *n*-bit matching result is used to identify the *n* possible matching offsets between then-byte input string and sub-pattern.

To illustrate how the *n*-bit match output is set, we assume that the *n*-byte input string is $W = w_1 \ldots w_n$ and the *n*-byte sub-pattern in PE is $PE = p_1 \ldots p_n$. At clock cycle *i*, the bit *k* of the *n*-bit match output denoted by *Match_out*[*k*] is set to 1 if $w_1 \ldots w_{n-k} = p_{k+1} \ldots p_n$ and *Match_in*[*k*] = 1 for $k \in \{1, \ldots, n-1\}$ and *Match_out*[*n*] is set to 1 if $w_1 \ldots w_n = p_1 \ldots p_n$ and *Match_in*[*n*] = 1. Notice that the match input signal *Match_in*[*k*] resulting from $w_{n-k+1} \ldots w_n = p_1 \ldots p_k$ at clock $i-1$ was delayed one clock by the shift register and thus can be used at clock *i*.

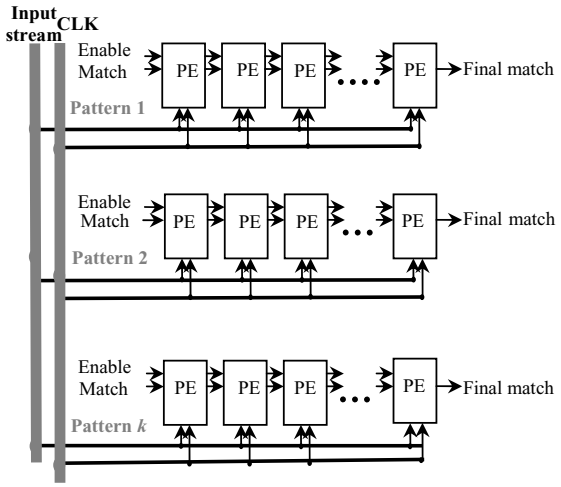

Figure 5 A pipeline stage

Each of the n possible sub-pattern matching results of PE and the corresponding 1-bit 'Match' input are combined with an *AND* gate to produce the 1-bit 'Match' output. The 'Enable' output is set if the input string matches the prefix of sub-pattern or the 'Match' output is set, shown as follows.

$$\text{Enable-out} = \begin{cases} \textbf{1:} \text{ Match output is set or} \\ w_{n-k+1}w_{n-k} \cdots w_n = p_1p_2 \cdots p_k, \\ \text{where } k \in \{1, \ldots, n-1\} \\ \textbf{0:} \text{ otherwise} \end{cases}$$

If the PE detects any bit of matching result is true or detects the last k-byte input string matches k-byte prefix of sub-pattern, the output 'Enable' will be set. The output signal of each PE is pipelined to the next PE. Figure 4 shows the details of PE of degree $n = 3$.

Processor arrays are designed to execute mathematical operations on multiple data elements simultaneously. They can achieve high performance by a huge replication of simple processing elements. Processor array is a simple, regular, and modular structure for implementing iterative pattern matching algorithms [6]. This section describes how we design multi-character processor array for pattern matching. Our proposed architecture has the properties of modularity and regularity. Two properties of our architecture that apply to processor array are pipeline and parallel techniques and regular multi-character matching modules.

C. Architecture

We first give some necessary notations as follows

n: Multi-character degree of our processing element.

m: The length of pattern $P = p_1 p_2 \ldots\ldots p_m$

s: Packet payload size. The input text $T = t_1 t_2 \ldots\ldots t_s$

W: a window of input string $W = w_1 w_2 \ldots\ldots w_n$

Our proposed architecture uses pipeline and parallel techniques to deal with a set of patterns. It enables high-performance and easily configurable design. For a pattern P, one or several PEs executed in a pipelining fashion are organized as a processor array according to pattern length and multi-character degree of PE. The n-byte input stream is compared against n^2 comparators in parallel. Moreover,

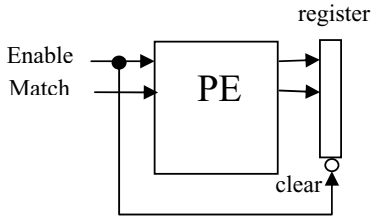

Figure 6. A pipelined PE.

all processor arrays designed for a certain pattern are executed in parallel. Since the design of PE is scalable, we can find a balance between area cost and throughput according to different requirements.

The proposed pipeline and parallel architecture for a set of patterns is shown in Figure 5. The processor array consists of $\lceil m/n \rceil$ PEs of degree n. The last PE in processor array indicates the final match of a pattern. The output 'Enable' of the previous pipeline stage is forwarded as the enable input of the next PE. If the 'Enable' input of PE is set, the PE would be active by a rising clock edge. Two inputs 'Enable' and 'Match' of first PE in processor array are always set. The window of input stream is broadcast on the input bus to all PEs, and then is shifted by n bytes per clock cycle. Because the partial matching signal is delayed in a PE, the final match result of processor array may delay one clock cycle. The details will be described in next section.

At each pipeline stage, the outputs of PE are stored into the registers, and they are feed to next PE by a rising clock edge. The data in a register is set to false at first except the first one. If the input 'Enable' of PE is false, we must unset the register to avoid enabling next PE that are fed with old data from register. Figure 6 shows the details for a pipeline stage.

Another advantage of our proposed processor array is that we provide a computationally efficient architecture. By disabling some PEs that had no effect to final matching result, a large number of computations can be saved. We observe that it is rare for an incoming packet to fully match more than hundred of patterns. On the other hand, most PEs for those un-matching patterns that never occur in packet payload can be disabled efficiently. The concept of our saving-computation design is based on the following observation:

Two matching conditions, 'exact PE match' and 'partial PE match' in a PE were described in section II-B, may occur. We have the corresponding strategies to each of them. The PE never generates a 'partial PE match' at current clock if input string does not match k-byte prefix of sub-pattern at last clock, where $0 < k < n$. The pattern P is divided into several sub-patterns and those sub-patterns are stored in PEs. If the result of comparing n-byte input string against first sub-pattern of P neither exact match nor k-byte prefix match, it implies that a real match for pattern P will not occur from locations $b, b+1, \ldots, b+n-1$, where b is location of first byte in input string. Thus, we can disable the following PEs to save unnecessary computations. Otherwise, we must enable the following PEs to detect the final match.

By extending this concept to other PEs, our processor array achieves a computationally efficient architecture.

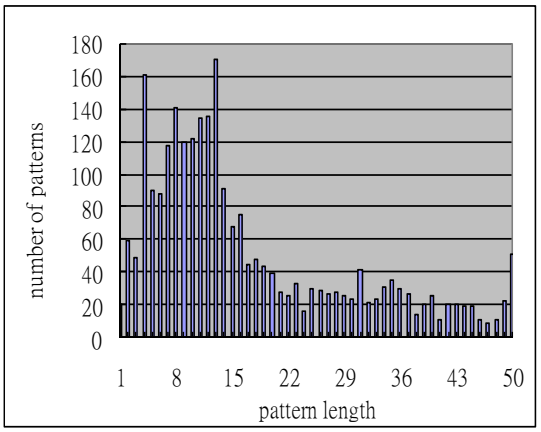

Figure 7. Distribution of the string lengths in the Snort database

The details of our computation saving technique working on processor array are described as follows:

Given a pattern $P = p_1 p_2 \ldots\ldots p_m$, and input text $T = t_1 t_2 \ldots\ldots t_s$. We assume multi-character degree of our PE is n. the pattern P is divided into $\lceil m/n \rceil$ sub-patterns, and those sub-patterns are stored in the $\lceil m/n \rceil$ corresponsive PEs (PE_1, PE_2, ….., $PE_{\lceil m/n \rceil}$). The inputs 'Enable' and 'Pre-Match' of all PEs are unset except the first one before starting to search. In the search phase, there are four cases when we compare a window $W = w_1 w_2 \ldots\ldots w_n$ of input text against sub-pattern of PE_g at clock cycle i, where we assume the location of w_1 is b ($w_1 = t_b$) and $1 \le g \le \lceil m/n \rceil$.

(1) **Exact match:**

The window W matches sub-pattern PE_g exactly, where $w_1 w_2 \ldots w_n = p_{g1} p_{g2} \ldots p_{gn}$. It implies that a real pattern match may occur if the input 'Pre-Match' of PE_g is true and the subsequent $m-(g\times n)$ bytes of input string are equal to $p_{(g+1)1} \ldots p_{(g+1)n}, \ldots, p_{(m/n)1} \ldots p_{(m/n)n}$. Therefore, if the input 'Match' of PE_g is true in this case, the outputs 'Enable' and 'Match' of PE_g are set. On the contrary, the outputs of PE_g are unset.

(2) ***k*-byte prefix match:**

The k-byte suffix of W match k-byte the prefix sub-pattern, where $w_{(n-k+1)} w_{(n-k+2)} \ldots\ldots w_n = p_{g1} p_{g2} \ldots\ldots p_{gk}$. It implies that a real pattern match may occur if subsequent $m-(g\times n)+(n-k)$ bytes of input string are equal to $p_{g(k+1)} \ldots p_{gn}$ $p_{(g+1)1} \ldots p_{(g+1)n}, \ldots, p_{(m/n)1} \ldots p_{(m/n)n}$. Because those k-byte matching results are delayed one clock by shift registers, the partial PE match may occur at next clock cycle. Therefore, we must enable next PE even if the input 'Match' of PE_g is false.

(3) ***(n-k)*-byte suffix match:**

The $(n-k)$-byte prefix of W matches $(n-k)$-byte suffix of sub-pattern, where $w_1 w_2 \ldots w_{n-k} = p_{g(n-k+1)} p_{g(n-k+2)} \ldots p_{gn}$. This case is similar to case (1). It implies that a real pattern match may occur if the input 'Match' of PE_g is true and the subsequent $m-(g\times n)$ bytes of input string are equal to $p_{(g+1)1} \ldots p_{(g+1)n}, \ldots, p_{(m/n)1} \ldots p_{(m/n)n}$. Therefore, if the input 'Match' of PE_g is true in this case, the outputs 'Enable' and 'Match' of PE_g are set. On the contrary, the outputs of PE_g are unset. The difference between cases (1) and (3) is the final matching result will delay one clock.

(4) **Mismatch:**

If the cases (1), (2), and (3) were not discovered, a real pattern match will not occur from current input string. Therefore, the next PE is unset to save unnecessary computations.

If the cases (1), (2), and (3) were not discovered, a real pattern match will not occur from current input string. Therefore, the next PE is unset to save unnecessary computations.

Table 1. Area cost for different multi-character degrees.

	n=3	*n=4*	*n=5*
# of comparators per PE	9	16	25
# of shift registers per PE	3	6	10
# of PEs required for Snort	15,733	12,020	9,882
Avg # of PEs for a pattern	6.2	4.7	3.9

Table 2. Computation reduction rates of various traces with $n = 3$.

	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
Computation reduction rate	83%	83%	83%	83%	83%

Table 3. Computation reduction rates for different multi-character degree.

	n=3	*n=4*	*n=5*
Computation reduction rate	83%	78%	74%

IV. Performance Evaluation

We evaluate the efficiency of our multi-character processor array architecture using two metrics: area cost and computation-saving rate. We used the Snort rule set and DEFCON8[13] packet traces for our experiments. We used the Snort [12] version of 2.4 for our experiments. The total number of unique patterns in Snort is 2535. The pattern length distribution is shown in Figure 7. The average length of patterns is 17 bytes and the maximum length of patterns is 364 bytes.

Since our PE design is scalable, we considered three multi-character degrees: 3, 4, and 5 to evaluate our architecture. As we know that increasing the multi-character degree of PE increases both the throughput and area cost per PE. Thus, we can find a balance between performance and cost.

A. Analysis of area cost

In this section, we analyze the area cost by real Snort patterns. Suppose we have a total of k patterns, each with m_i bytes. If we set the multi-character degree of PE is n, then each pattern will be cut into $\lceil m_i/n \rceil$ PEs. Each PE consists of n^2 8-bits character comparators and $n(n-1)/2$ SRL16 shift registers. Therefore, we need total $\sum \lceil m_i/n \rceil \times n^2$ comparators and $\sum \lceil m_i/n \rceil \times n(n-1)/2$ shift registers in our architecture.

Our Snort rule set contains 2535 unique patterns and total 44,416 characters. The average length of patterns is 17 bytes and the maximum length of patterns is 364 bytes. For our experiments, the value of n should be smaller than six to achieve the best benefit of cost and performance. If we use 3 as multi-character degree of PE, we need total 15,733 PEs in our entire system and have seven PEs on

average for a pattern. Table 1 shows the area cost for different multi-character degrees. The results show that entire cost will increase significant as *n* increases.

B. Computation reduction

In order to measure the computation reduction rates, we utilize five DEFCON8 packet traces for our experiments. We compare our computation reduction techniques against the general approach that enables all of the PEs during the search phase. The general *n*-character approach requires *O(ms)* comparisons, where *m* is the pattern size and *s* is the input string size. Our approach can reduce a large amount of computations by disabling certain PEs that had no effect on final results. First, we calculated the computation reduction rate by using different DEFCON8 packet traces and multi-character degree *n=3*. Table 2 shows that the average computation reduction rate is about 83%, which means we only enable 2675 PEs per clock cycle. This result explains that the computation reduction rate is stable for different packet traces.

We also calculate the results with various values for *n*. Since the Snort pattern set contains many short patterns and average length of patterns is 17 bytes, a large value of *n* will diminish the effectiveness of our computation reduction technique and increase the cost of PE. Therefore, we set *n* to be 3, 4, and 5. Table 3 shows that the computation reduction rate will decrease as *n* increases.

V. CONCLUSION

In this paper, we proposed an effective pattern matching approach for high-speed network. We used Snort pattern set and DEFCON packet traces to evaluate the performance. The approach has modular and computation reduction properties that can reduce about 83% computation against brute force approach. The PE design is simple and flexible. Thus, choosing *n* is a tradeoff between cost and performance.

REFERENCES

[1] A. Aho and M. Corasick, “Efficient string matching: An aid to bibliographic search,” Communications of the ACM, vol. 18, no. 6, pp.333-343, June 1975.

[2] M. Aldwairi, T. Conte, and P. Franzon. “Configurable string matching hardware for speeding up intrusion detection.” SIGARCH Comput. Archit. News, 33(1):99.107, 2005.

[3] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” Communications of the ACM, vol. 20, no 10, pp.762-772, Oct. 1977.

[4] Young H. Cho, W.H. Mangione-Smith, “Deep packet filter with dedicated logic and read only memories,” in Proceedings of the 12th IEEE Symposium of Field-Programmable Custom Computing Machines, 2004.

[5] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull and J. W. Lockwood, “Deep Packet Inspection using Parallel Bloom Filters”, IEEE Micro, vol. 24, no. 1, pp. 52-61, Jan. 2004.

[6] Fayez Gebali, A. N. M. Ehtesham Rafiq, “Processor Array Architectures for Deep Packet Classification,” IEEE Trans. Parallel Distrib. Syst. 17(3): 241-252 (2006)

[7] Moscola J, Lockwood J, Loui RP, Pachos M. Implementation of a content-scanning module for an Internet firewall. In: Pocek KL, ed. Proc. of the 11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), pp.31-38, 2003.

[8] D. E. Knuth, J. H. M. Jr., and V. R. Pratt, “Fast pattern matching in strings,” SIAM J. Comput., vol. 6, no. 2, pp. 323–350, June 1977.

[9] R. Sidhu 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), pages 227–238, Rohnert Park, CA, USA, May 2001.

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

[11] F. Yu, R. H. Katz, T. V. Lakshman, “Gigabit Rate Packet Pattern-Matching Using TCAM,” in Proceeding of 12th IEEE International Conference on Network Protocols (ICNP’04), Berlin, Germany, Oct. 2004, pp. 174-183.

[12] *Snort-the de Facto Standard for Intrusion Detection/Prevention,* [Online] www.snort.org

[13] DEFCON. http://www.shmoo.com/, http://cctf.shmoo.com/data/cctf-defcon8/