PAPER

PAMELA: Pattern Matching Engine with Limited-Time Update for NIDS/NIPS*

Tran Ngoc THINH[†a)], **Surin KITTITORNKUN**[†], ***Nonmembers***, ***and*** **Shigenori TOMIYAMA**[††], ***Member***

SUMMARY Several hardware-based pattern matching engines for network intrusion/prevention detection systems (NIDS/NIPSs) can achieve high throughput with less hardware resources. However, their flexibility to update new patterns is limited and still challenging. This paper describes a *PAttern Matching Engine with Limited-time updAte* (PAMELA) engine using a recently proposed hashing algorithm called *Cuckoo Hashing*. PAMELA features on-the-fly pattern updates without reconfiguration, more efficient hardware utilization, and higher performance compared with other works. First, we implement the improved parallel exact pattern matching with arbitrary length based on Cuckoo Hashing and linked-list technique. Second, while PAMELA is being updated with new attack patterns, both stack and FIFO are utilized to bound insertion time due to the drawback of Cuckoo Hashing and to avoid interruption of input data stream. Third, we extend the system for multi-character processing to achieve higher throughput. Our engine can accommodate the latest Snort rule-set, an open source NIDS/NIPS, and achieve the throughput up to 8.8 Gigabit per second while consuming the lowest amount of hardware. Compared to other approaches, ours is far more efficient than any other implemented on Xilinx FPGA architectures.

key words: *Cuckoo Hashing, dynamic update, pattern matching, FPGA, NIDS/NIPS*

1. Introduction

Nowadays, illegal intrusion is one of the most serious threats to network security. Network Intrusion Detection/Prevention Systems (NIDS/NIPSs) are designed to examine not only the headers but also the payload of the packets to match and identify intrusions. Most modern NIDS/NIPSs apply a set of rules that lead to a decision regarding whether an activity is suspicious. For example, Snort [1] is an open source network intrusion detection and prevention system utilizing a rule-driven language. As the number of known attacks grows, the patterns for these attacks are made into Snort signatures (pattern set). The simple rule structure allows flexibility and convenience in configuring Snort. However, checking thousands of patterns to see whether it matches becomes a computationally intensive task as the highest network speed increases to several gigabits per second (Gbps).

Manuscript received September 9, 2008.

Manuscript revised January 16, 2009.

[†]The authors are with the Department of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok, 10520 Thailand.

[††]The author is with the Department of Embedded Technology, School of Information and Telecommunication Engineering, Tokai University, Tokyo, 108–8619 Japan.

*The previous version of this paper was presented at International Conference on Field-Programmable Technology 2007.

a) E-mail: tnthinh@cse.hcmut.edu.vn

DOI: 10.1587/transinf.E92.D.1049

To improve the performance of pattern matching in Snort, various implementations of field programmable gate array (FPGA) systems have been proposed. These systems can simultaneously process thousands of patterns relying on native parallelism of hardware so that their throughput can satisfy current gigabit networks. The drawback of hardware-based systems, however, is the flexibility. With emergence of new worms and viruses, the rule set must be frequently updated. For recently proposed FPGA-based NIDS/NIPSs [2]–[14], adding or subtracting a few rules requires recompilation (synthesizing, placing and routing) and reconfiguration of some parts or the entire design. Although reconfiguration is one of the advantages of SRAM-based FPGA; this process can take several minutes to several hours to complete. Today, such latency may not be acceptable for most networks when new attacks are released frequently. Moreover, the measure of throughput per area should be used in all hardware implementations. It is necessary to build a system that can achieve high-throughput per area with rapid rule set update.

Based on a novel Cuckoo Hashing [15], we implemented an FPGA-based architecture of variable-length pattern matching in our previous paper [16]. Unlike most previous FPGA-based systems, ours [16] can update the static pattern set without reconfiguration thanks to Cuckoo Hashing. In this paper, we propose a system named *PAttern Matching Engine with Limited-time updAte* (PAMELA) that extends and improves some disadvantages of [16]. Based on our analysis, the main drawback of Cuckoo Hashing is time consuming that can make the rule update time unlimited. We propose to use a stack to prevent rehashing and a FIFO to buffer the incoming data while updating the pattern set. Based on our theoretical analysis and simulation results, the insertion time of a new pattern is limited to 17 microseconds at 200 MHz clock frequency. As a result, a new rule set can be updated to PAMELA on line and on the fly. The power of high throughput processing is addressed in this paper. With the extension for multi-character processing, the current system can sustain higher throughput than the previous one. Moreover, the linked-list method is improved to detect the continuous matches of long patterns; performance analysis, more experimental verification and comparison are carried out. With several improvements, PAMELA can save 30% of the area compared with the best system, and the throughput can achieve up to 8.8 Gbps for 4-character processing.

The rest of our paper is organized as follows. In Sect. 2, some previous hardware implementations of static pattern

matching and Cuckoo Hashing are presented. Section 3 proposes the architecture of PAMELA engine. Next, performance analysis of PAMELA is discussed in Sect. 4. The extension for multi-character processing is mentioned in Sect. 5 before the experimental results on FPGA are presented in Sect. 6. Finally, future works are suggested in the conclusion.

2. Background and Related Works

2.1 Pattern Matching on NIDS/NIPS

For a line speed of gigabit network, a variety of FPGA approaches of NIDS/NIPS have been proposed, for example: shift-and-compare; state machine such as Nondeterministic/Deterministic finite automata (NFA/DFA), Aho-Corasick algorithm; and finally hashing. Firstly, some of the shift-and-compare method are [2], [3]. They apply parallel comparators and deep pipelining on different, partially overlapping, positions in the incoming packet. The simplicity of the parallel architecture can achieve high throughput when compared to software approaches. The drawback of these methods is the high area cost. To reduce the area cost and achieve a high clock rate, many improvements are proposed. The work [5] is extended from [2] to share common substrings. Predecoded shift-and-compare architectures ([7], [17]) convert the incoming characters to bit lines to reduce the size of comparators. A variation in tree-based optimization ([8], [9]) divides the pattern set into partitions to share similar characters resulting in excellent area performance.

The next approach exploits state machines (NFAs/DFAs) [4], [18]. The state machines can be implemented on hardware platform to work all together in parallel. By allowing multiple active states, NFA is used in [18] to convert all the Snort static patterns into a single regular expression. Moscola et al. [4] recognized that most of minimal DFAs content fewer states than NFAs, so their modules can automatically generate the DFAs to search for regular expressions. Like the shift-and-compare implementations, the predecoded method is also used in [6] to improve area performance of NFAs. The main advantage of regular expression format as compared with static pattern one is that a single regular expression can describe a set of static patterns by using meta-characters with special meaning. As a result, recently, a special format of regular expressions such as Perl Compatible Regular Expressions (PCRE) is added in Snort [1] instead of static patterns, and some new works tried to improve on PCRE matching [19], [20]. However, most of these systems suffer scalability problems, i.e. too many states consume too many hardware resources and long reconfiguration time.

Another approach of the state machine method used for static pattern matching is the Aho-Corasick algorithm [21]. By modifying this algorithm on hardware, the implementations in [22]–[24] can get high performance. Aldwairi et al. [22] partitioned the rule set into small ones according to the type of attacks in Snort database. The state machine in [23] is split into smaller FSMs which can run in parallel to improve memory requirements. This bit-split FSM can fit over 12k characters of Snort rule set to 3.2 Mbits memory at 10 Gbps on ASIC implementation and can update new rules in the order of seconds with no interruption. Nonetheless, its FPGA implementation [24] can achieve lower throughput rate while using larger memory.

Finally, hashing approaches [10]–[14] can find a candidate pattern at constant look up time. The authors in [11], [14] use perfect hashing for pattern matching. Although their system memory usage is of high density, the systems require hardware reconfiguration for updates. Papadopoulos et al. proposed a system named HashMem [12] system using simple CRC polynomials hashing implemented with XOR gates that can use efficient area resources than before. For the improvement of memory density and logic gate count, they implemented V-HashMem [13]. Moreover, V-HashMem is extended to support the packet header matching. However, these systems have some drawbacks: 1) To avoid collision, CRC hash functions must be chosen very carefully depending on specific pattern groups; 2) Since the pattern set is dependent, probability of redesigning system and reprogramming the FPGA is very high when the patterns are being updated; 3) By using glue logic gates for simplicity, the long patterns processing is also ineffective for updating.

On the other hand, Dharmapurikar et al. propose to use Bloom Filters to do the deep packet inspection [25]. Unlike other hashing approaches mentioned above, the pattern update process can be done easily without reprogramming FPGA. A Bloom Filter with multiple hash functions up to 35 probes is used to check whether or not a pattern is member of the set. Nevertheless, its main problem is due to false positive matches, which requires extra cost of hardware to confirm the match.

2.2 Cuckoo Hashing

Cuckoo Hashing is proposed by Pagh and Rodler [15] as an algorithm for maintaining a dynamic dictionary with constant lookup time in the worst case scenario. The algorithm utilizes two tables, T_1 and T_2, of $m = (1 + \epsilon)n$ cells each, where ϵ is some constant ($\epsilon > 0$), n is the number of elements (strings). Cuckoo Hashing does not need perfect hash functions that is very complicated if the set of stored elements dynamically changes under the insertion and deletion. Given two hash functions, h_1 and h_2, mapping from universe U to $[m]$, a pattern x can be exactly stored in either cell $T_1[h_1(x)]$ or $T_2[h_2(x)]$ but not both. So, lookup of x requires at most two positions. Deletion procedure can also run in worst case constant time.

Pagh and Rodler described a procedure to insert a new pattern x in expected constant time. For the first time, x is always placed into cell $T_1[h_1(x)]$. If this cell is empty, the insertion is complete; if it is occupied by a pattern y which necessarily satisfies $h_1(x) = h_1(y)$, then x is put in cell $T_1[h_1(x)]$ anyway, and y is kicked out. Then, y is put into

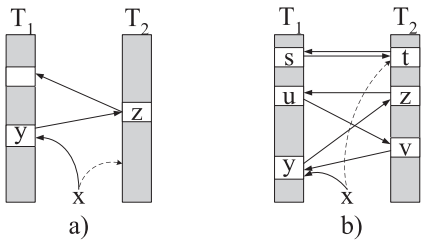

Fig. 1 Original Cuckoo Hashing [15] a) A pattern x is successfully inserted by moving y and z. b) A pattern x cannot be accommodated and a rehash is required.

the cell $T_2[h_2(y)]$ of the second table similarly, which is also possibly occupied by another pattern z with $h_2(y) = h_2(z)$. In this case, z is placed in cell $T_1[h_1(z)]$, and the process repeats until the pattern can be placed in an empty cell as in Fig. 1 a. However, it can be seen that the "cuckoo process" may not terminate as Fig. 1 b. As a result, the number of iterations is limited by a bound M chosen beforehand. In this case every entry must be rehashed with two new hash functions.

3. FPGA-Based Pattern Matching Engine in NIDS/NIPSs Using Cuckoo Hashing

For pattern matching in NIDS/NIPS, the patterns are searched on the incoming data (packets). The matched pattern can occur anywhere as the longest substring. This problem is called multi-pattern matching with a set of patterns $P = \{p_1, p_2, \ldots, p_n\}$ and the incoming data T. Normally, the pattern set is preprocessed and built in a system. The incoming data is entered into a FIFO to compare with all of patterns. As a result, Cuckoo Hashing is a good candidate for multi-pattern matching with constant lookup time. Furthermore, dynamic update for the pattern set does not affect the performance of lookup. Figure 2 a shows an overview of multi-pattern matching engine using FPGA-based Cuckoo Hashing. Each module named *Cuckoo* L_i can process patterns of i characters long.

In order to process at the network speed in Gbps, we have to construct Cuckoo Hashing module for every pattern length from $L_{min} = 1$ up to L_{max} characters. The priority circuit then selects the longest pattern if multi matches happen. However, L_{max} can grow up to hundreds in most of NIDS/NIPS systems. Thus, we build the Cuckoo Hashing modules for short patterns with the maximum length L_{max_s} according to the most distribution of patterns in NIDS/NIPS. In Snort pattern set, L_{max_s} gets the best value of 16 characters. For longer patterns, we can break them into shorter segments so that we can insert those segments to the Cuckoo modules of short patterns. We then use simple address linked-lists to combine these segments later. Figure 2 b shows our optimized architecture for pattern matching.

3.1 FPGA-Based Cuckoo Hashing Module

In order to increase memory utilization, we build up a hashing module for each pattern length and use indirect storage. Small and sparse hash tables contain indices of patterns which are the addresses of a condensed pattern-stored table. Our approach is also to change the lookup to parallel processing. The insertion can be changed for better selection of the available space in both hash tables. With some improvements in architecture, ours can take full advantages of hardware.

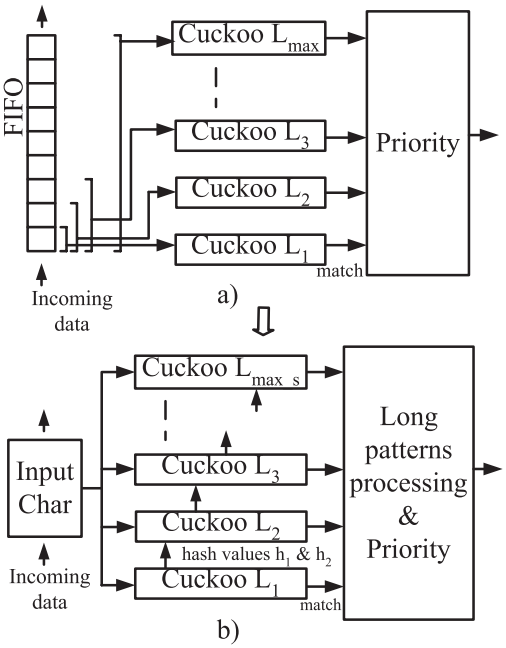

Fig. 2 FPGA-based pattern matching engine in NIDS/NIPSs using Cuckoo Hashing. a) General Model b) Optimized Hardware Model.

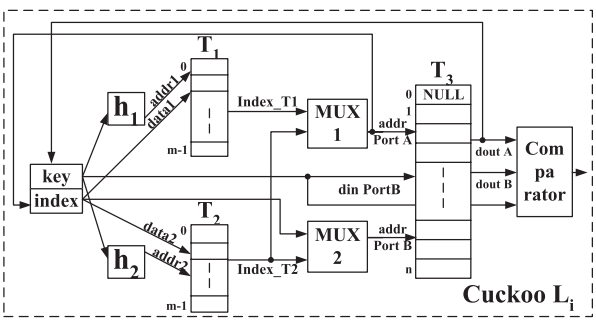

Fig. 3 FPGA-based Cuckoo Hashing module with parallel lookup. Tables T_1 and T_2 store the key indices; Table T_3 stores the keys.

The architecture of a FPGA-based Cuckoo Hashing module as shown in Fig. 3 consists of three tables. Two index tables (hash tables) T_1 and T_2 are single-port SRAMs and a pattern-stored table T_3 is a double-port SRAM for concurrent processing. Hash functions can be changed if they are required to rehash. Two registers named *key* and *index* are the pattern to be searched for and the memory address of that pattern in T_3, respectively. Besides, two multiplexers are used to select addresses of T_3. Finally, a comparator is used for exact matching of *key* with two candidate patterns from T_3.

3.1.1 Parallel Lookup of an Incoming Pattern

By using parallel and multi-phase pipeline architecture, PAMELA can look up patterns every clock cycle. Figure 4 is the pseudo-code of parallel Cuckoo lookup function. A pattern x is hashed by two hash functions concurrently. Then, the values of two hash functions are used as the addresses of two index tables. After that, the outputs of two index tables are used as the addresses of T_3. Finally, to determine the match, the outputs of T_3 are compared with x.

3.1.2 Dynamic Pattern Insertion and Deletion

The insertion of a new pattern x can be described in Fig. 5. We have some improvements as compared with the original Cuckoo Hashing. We consider both tables to reduce the insertion time. If one of the outputs of two index tables is empty (NULL), the index of x is inserted into T_1 or T_2 and the insertion is complete. If the outputs of both T_1 and T_2 are not NULL, we insert the *index* into the table with less number of patterns. At the same time, the "kicked-out" *index*_T_1 (*index*_T_2) and its data from T_3 will be written into the *index* and *key* registers to start the hashing process. Then, M, $[3log_{1+\epsilon}m]$ [15], is decreased and the key value is hashed by hash function h_2 (h_1). The output data will be checked for whether the value is NULL. If it is NULL, the process ends with successful insertion. On the other hand, the process is continued by taking in turns hashing from h_2 (h_1) to h_1 (h_2). The worst case happens when M decreases to zero. Hence, a rehash is required. Two new hash functions h_1 and h_2 are issued by a pseudo-random number generator.

For deletion, the algorithm in Fig. 6 is as simple as the lookup process. If the lookup succeeds, $MUX1$ will select exactly one of the outputs of two index tables to write into the index register. We then write NULL into table T_3 at the address pointed by the index register. After that, we reset the index register to NULL and write it into appropriate T_1 or T_2. The deletion process results in some "holes" in T_3. When the number of "holes" is greater than a threshold, the rehash for rearranging of T_3 can be implemented.

```
function lookup(x)
  select index_T1 in MUX1  and index_T2 in MUX2;
     index_T1 = T1(h1(x));              // phase1+2
     index_T2 = T2(h2(x));
     doutA = PortA(index_T1);           // phase3
     doutB = PortB(index_T2);
     return(doutA = x or doutB = x); // phase4
end
```

Fig. 4 Pseudo-code of parallel Cuckoo lookup algorithm.

```
procedure insert(x)
    if (lookup(x)) return;
    select index_T1 in MUX1  and index in MUX2;
    PortB(index) = x;
    if(index_T1 == NULL){
        T1(h1(x))= index;   return;}
    else if(index_T2 == NULL){
        T2(h2(x))= index;   return;}
    select index_T1 or index_T2 in MUX1
        //depending on balance of hash table
    loop MaxLoop times  // "Cuckoo process"
        if (select index_T1 in MUX1){
            key = PortA(index_T1);
            index = index_T1;
            select index_T2 in MUX1;}
        else{ // (select index_T2 in MUX1)
            key = PortA(index_T2);
            index = index_T2;
            select index_T1 in MUX1;}
        index_T1 = T1(h1(x));
        index_T2 = T2(h2(x));
        if (select index_T1 in MUX1){
            if(index_T1 == NULL) return;
            doutA = PortA(index_T1);}
        else{ // (select index_T2 in MUX1)
            if(index_T2 == NULL) return;
            doutA = PortA(index_T2);}
    end loop
    rehash(); insert(x);
end
```

Fig. 5 Pseudo-code of parallel Cuckoo insertion algorithm.

3.1.3 Recommended Hash Function

The hash function of choice greatly affects the performance of the system. A fast way of generating a class of universal hash function that is hardware-friendly and tabulation based method [26], is defined as follows:

$$H_t(x) = a_t[0][x_0] \oplus a_t[1][x_1] \oplus \ldots \oplus a_t[n-1][x_{n-1}] \quad (1)$$

A table contains a 2-D array of random numbers in the hashing space. A key x is a string of n characters, $x_0 x_1 .. x_{n-1}$, and the hash value is calculated by bit-wise exclusive-or ($\oplus$) a sequence of values $a_t[i][x_i]$, which is indexed by each byte value of x_i and position of i in the string. The drawback is that the size of random table is very large and depends on the key length.

Another class of simple hash function for hashing character strings named *shift-add-xor* (SAX) [27] utilizes only the simple and fast operations of shift, exclusive-or and addition.

$$H_i = H_{i-1} \oplus (S_L(H_{i-1}) + S_R(H_{i-1}) + c_i) \quad (2)$$

Two operators S_L and S_R denote the shift left and right, respectively. The symbol c_i is the character i^{th} of string and H_i is an intermediate hash value after examination of i characters. The initial value H_0 can be generated randomly. The main advantage of SAX as compared with random-table is very few hardware resources required. To generate the new

```
procedure delete(x)
    if (!lookup(x)) return;
    key = NULL;
    index = index_T1 or index_T2;
          //depending on the lookup result
    PortB(index) = key;
    T1(h1(x))= NULL or T2(h2(x))= NULL;
    return;
end
```

Fig. 6 Pseudo-code of parallel Cuckoo deletion algorithm.

SAX hash function in case of rehashing, we only need to change the value of H_0 by a simple pseudo-random circuit LFSR [28]. Therefore, SAX is a suitable choice for PAMELA.

3.1.4 Hardware Optimization for Cuckoo Module

We can significantly reduce large amount of hardware by exploiting accumulative characteristic of SAX hash function. From Eq. (2), to calculate hash value of an incoming pattern with length i characters in the hash module i^{th}, the requisite inputs are the hash values of $i-1$ characters calculated beforehand in the hash module $(i-1)^{th}$ and the i^{th} character. Therefore, the values of previous hash module can be reused for the next hash module. As shown in Fig. 2 b, two hash values h_1 and h_2 of *Cuckoo* L_{i-1} are fed into *Cuckoo* L_i $(2 < i \leq L_{max\_s})$. In addition, the FIFO for the incoming data is also replaced by only a one-character shift register.

Nevertheless, our hardware optimization can increase the probability of rehash on the system. For example, when a rehash by collision happens at Cuckoo module i, new functions h_1 and h_2 of module i make the inputs of module $i+1$ changed and the hash values of module i+1 will be incorrect. The process is going on recursively up to module L_{max_s}. As a result, the rehash can be forced from Cuckoo module $i+1$ to module L_{max_s}. This thing, however, only affects the insertion process. This trade off is good enough because the first priority is fast lookup with smaller hardware.

3.2 Matching Long (>16-Character) Patterns

We break the longer (> 16-Character) patterns into variable-length segments of 1 to $L_{max\_s} = 16$ characters. The above Cuckoo Hashing modules can then be used for matching these individual segments. After that, these segments of a long pattern are combined to a chain that can be implemented by simple linked-list technique. The data structure for storing linked-lists is a table named T_4 whose depth is the depth of T_3 multiplied by the number of Cuckoo modules. Each address represents one segment and its content is an address of next segment in the same long pattern.

For more details, we describe the technique using a simple example. We assume that string *"abcdefghij"* is a long pattern that is broken into smaller segments with proportion 3 : 3 : 2 : 2 as in Fig. 7 a. Segments 1 and 2 are hashed and stored in T_3 of *Cuckoo* L_3 at address $1FFh$ and $1FEh$ while segments 3 and 4 are hashed and stored in T_3 of *Cuckoo* L_2 at address $1A4h$ and $1A5h$. We combine the addresses in $T_3 s$ together with the number of Cuckoo modules to link these individual segments. If the depth of T_3 in every module and the number of Cuckoo modules are 512 and 16 then the bit number is 13 for representing a position of individual segment: 9 least significant bits for representing the address of T_3 and 4 most significant bits for representing the number of Cuckoo modules. The segment addresses of *"abcdefghij"* in T_4 are $7FFh$, $7FEh$, $5A4h$, and $5A5h$ as shown in Fig. 7 b. The matching process is described as the

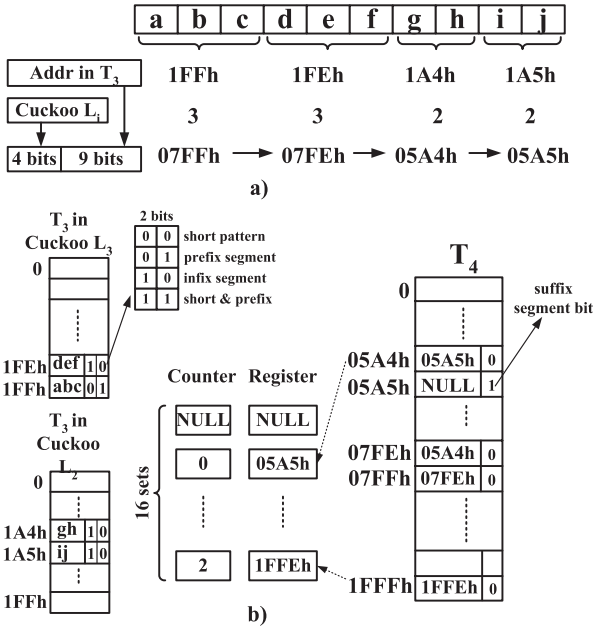

Fig. 7 Matching long patterns. a) Example of breaking a long pattern "abcdefghij". b) How to store a long pattern in table T_4 as a linked-list.

following. When we get a match for segment *"abc"*, we have address $7FFh$ whose content in T_4 is $7FEh$. After 3 clock cycles, if the mach segment is *"def"*, the process is continued by jumping from $7FFh$ to $7FEh$. Otherwise, the detective process finishes without match and resets for other patterns. Similarly, from $7FEh$, if the next match segment after 2 clock cycles is *"gh"*, address $5A4h$ is considered. Finally, if the last segment *"ij"* is detected 2 next clock cycles later then the content of address $5A5h$ is read and the match of this pattern is reported.

Two more bits are added in every entry of T_3 as in Fig. 7 b. They are used to describe a string which can be the short pattern, the first segment of long pattern called *prefix segment*, the body of long pattern called *infix segment*, or even the short pattern which is also the prefix segment. In case of the end of long pattern called *suffix segment*, we can determine it by checking the content of its address in T_4 whether it is NULL. However, if a long pattern is a prefix substring of another one then it cannot be detected. For example, a long pattern *"abcdef"* can be the prefix substring of *"abcdefghij"*. To improve this case, we add one more bit in table T_4. When this bit is active, a suffix segment can also be an infix segment of another pattern. In other words, the content of a suffix segment in T_4 can be a pointer to another address.

The above paragraphs describe the method for holding one long pattern per time. Each time, if the system detects one segment with length L, it has to wait for the next segment in L clock cycles. During this time, if other matches happen then the system cannot detect them. For example, the incoming data is *"...abcdefghijk..."* and we have 2 long patterns *"abcdmnpq"* and *"bcdefghi"* in the pattern set. We assume that these patterns are broken into seg-

ments *"abcd", "mnpq"* and *"bcde", "fghi"*, respectively. In clock cycle t, if the matched string is *"abcd"* then the system waits for the next match in 4 clock cycles. However, the next match for the first pattern does not happen in this case. Meanwhile, in clock cycle $t + 1$, the system cannot detect the segment *"bcde"* that is the prefix for the second pattern.

To detect all segments of long patterns while matching in every clock cycle, we use $L_{max\_s} = 16$ down counters and registers for storing the match values read from T_4; the length of segment and its content in T_4 are stored in an available counter and register, respectively. When a segment match happens and it is a prefix segment, its position in T_4 is considered. If it is an infix or a suffix of current long pattern candidates, it must be simultaneously compared to all registers whose counters are currently zero values to determine the unique address of T_4. Because the length of segments can reach at L_{max_s}, the maximum number of current long pattern candidates is L_{max_s}.

We use an automatic generator to partition the long patterns in pattern set. Figure 8 is the pseudo-code of long pattern insertion. First, we consider long patterns as the incoming strings pass through the engine for lookup. If a short pattern in the engine is the longest prefix of a candidate long pattern then we break the long pattern such that its first segment is the same length as the short pattern. We mark the short pattern which is also the prefix segment to avoid checking again. After that, we break the rest of the long pattern into halves if the length of the rest is greater than L_{max_s} or the number of patterns in the same table T_3 is greater than a certain threshold Th, somewhat arbitrarily set less than the size of hash table, to avoid the high probability of rehash. Otherwise, the rest can be a suffix segment. The partitioning can continue if any segment matches with a string in the engine.

```
procedure insert_long(x)
    sh: short pattern in table T3;
    prefix(k,x): prefix with length k of x;
    suffix(k,x): suffix with length k of x;
    //  lookup the longest prefix of x
    if(lookup(x)) return;
    if(lookup(sh=prefix(k,x)) and sh without "011"){
        //short & prefix segment
        change the end of sh to "011";
        call part(suffix(k(x-sh),x),"010","110");}
    else
        call part(x,"001","110");
end

procedure part(x,pre_id,suf_id)
    L: length of x;
    #T3(L): #patterns in T3 of length L;
    Th_L:threshold of length L;
    if(L > L_max_s or #T3(L) > Th_L or lookup(x)){
        part(prefix(L-L/2,x),pred_id,"010");
        part(suffix(L/2,x),"010",suf_id);}
    else{
        if(pre_id=="001")      //prefix segment
            insert(x) with the end "001";
        else if(suf_id=="110") //suffix segment
            insert(x) with the end "110";
        else                   //infix segment
            insert(x) with the end "010";}
end
```

Fig. 8 Pseudo-code of long pattern insertion algorithm.

4. Performance Evaluation: Theoretical Analysis and Simulation

4.1 Theoretical Analysis

In this subsection, we will analyze the time to lookup and insert a pattern. Based on the worst case scenario, we propose a method for bounding insertion time of a new pattern. In addition, two common area metrics of SRAM-based FPGA, memory and logic gate utilization are also explored to show the efficiency of PAMELA engine. To facilitate the theoretical analysis, some main notations are listed in Table 1.

4.1.1 Lookup Time and Insertion Time

The time to process a pattern of length L (characters or bytes) is a function of the number of cycles needed to calculate the hash values T_{hash}, and the number of cycles needed to access memory including hash tables $T_{mem_{1,2}}$ and storage-pattern table T_{mem_3}. We assume that the time needed to access every memory table is constant and $T_{mem_{1,2}} = T_{mem_3} = T_{mem}$. Our engine takes one character per clock cycle, so $T_{hash} = L$. The equation for the processing time T_{proc_L} is expressed as

$$T_{proc_L} = T_{hash} + 2T_{mem} = L + 2T_{mem} \tag{3}$$

For pattern lookup, we need one more stage to compare the candidate pattern after reading out of table T_3 with the part of the incoming data. With the time of the comparison T_{mtch}, the equation for the lookup time is calculated as follows.

$$T_{lu_L} = T_{proc_L} + T_{mtch} = L + 2T_{mem} + T_{mtch} \tag{4}$$

The insertion time T_{in_L} of a pattern x with length L can include some shuffles of other patterns in the hash table. Let h

Table 1 Summary of main notations used in the performance analysis.

Symbol	Units	Description
L	bytes	The length of a pattern
T_{hash}	cycles	The time to calculate the hash values
T_{mem}	cycles	The time to access any memory table
T_{proc_L}	cycles	The time to process a pattern of length L
T_{mtch}	cycles	The time of the comparison
T_{lu_L}	cycles	The lookup time of a pattern
T_{in_L}	cycles	The insertion time of a pattern
$T_{in\_w_L}$	cycles	The worst case insertion time
T_{lu_long}	cycles	The time for lookup of a long pattern
T_{in_long}	cycles	The time for insertion of a long pattern
S_S	bytes	The size of the stack
$T_{in\_u_L}$	cycles	The unsuccessful insertion time
T_{part_L}	cycles	The online partitioning time
S_F	bytes	The size of the FIFO
U_{mem}		The average memory utilization
R_{LC}		The ratio logic gates of the general architecture and the optimized architecture

denote the number of shuffles then the average time of successful insertion is expressed as: $T_{in\_avg_L} = (1 + h) \times T_{proc_L}$. At the best case, successful insertion of a pattern requires no shuffle of other patterns in hash tables, $T_{in\_best_L} = T_{proc_L}$.

The worst case can happen as h reaches M and the rehash occurs. According to [15], probability of this case is $O(1/n^2)$ when M is $3log_{\epsilon+1}m$. Before the rehashing process begins, the hash tables have to clear every occupied space. The reset time is T_{reset}. If N_L is the number of insertions for all patterns of length L until the successful insertion for pattern x then the time of rehash is $N_L \times T_{proc_L}$. In addition, according to the accumulative characteristic of SAX hash function, the rehashes are also required for patterns of lengths from $L+1$ to L_{max_s}. Finally, the worst-case insertion time of a pattern can be calculated as

$$T_{in\_w_L} = M \times T_{proc_L} + \sum_{i=L}^{L_{max\_s}} (N_i \times T_{proc_i} + T_{reset}) \qquad (5)$$

where T_{proc_i} denotes the processing time of pattern with length i and N_i denotes the number of insertion time of all patterns with length i.

For long patterns, we consider three more parameters: the time to access table T_4, $T_{mem_4} = T_{mem}$; the time to compare and connect segments, $T_{connect}$; and the number of segments of a pattern, s. We express the time for lookup and insertion of a long pattern as follows.

$$T_{lu\_long} = (T_{lu_L} + T_{mem} + T_{connect}) \times s \qquad (6)$$

$$T_{in\_long} = (T_{in_L} + T_{mem} + T_{connect}) \times s \qquad (7)$$

4.1.2 Limited-Time Update

As the analyzed worst case of insertion in Eq. (5), updating time of a pattern can be lengthy. For real-time protection, some practical systems can be vulnerable. To avoid this weakness, the basic solution is that we build the duplicate modules or separate modules for updating only [10], [23]. However, this solution consumes a lot of hardware resources and require re-compiling some parts of the system. We propose a simple and fast method that needs minimum hardware based on the capability of breaking a pattern into segments.

The details of our solution can be described as follows. If the worst case happens as a new pattern is inserted in the system then we report unsuccessful insertion instead of rehashing the hash tables. However, at that time, some positions of the hash tables are modified. Thus, we add a *stack* to trace the insertion process. Every step of insertion process is stored in the stack. If M is reached then we copy the traces from the stack back to the hash tables to restore the system. For illustration, we use a simple example as in Fig. 9. We insert a new element *"4"* into hash tables that stored *"0, 2"* at addresses *3, 0* of T_1 and *"1, 3"* at addresses *4, 0* of T_2. At every step of collisions, we store the old information of "kick-out" elements in the stack including of the address in

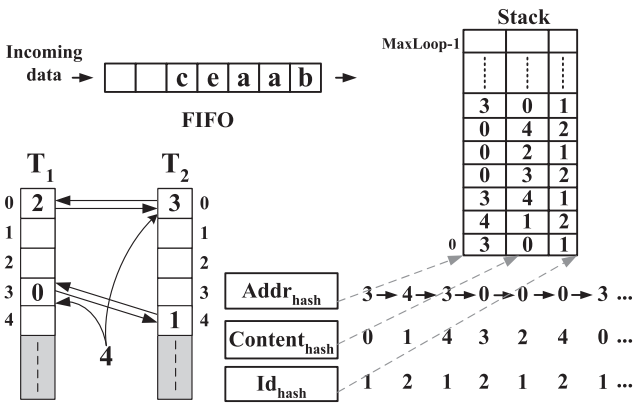

Fig. 9 Example of Limited-time pattern update. A *stack* traces the insertion process, the old information of "kick-out" elements, including of the address in hash table $Addr_{hash}$, the content at this address $Content_{hash}$, and the order number of hash table Id_{hash}. A *FIFO* buffers the incoming data as updating patterns.

hash table $Addr_{hash}$, the content at this address $Content_{hash}$, and the order number of hash table Id_{hash}. As in Fig. 9, insertion process is infinite. Therefore, we restore steps one by one from the stack to the hash tables until the stack is empty with the last element of unavailable space being *"4"*. We can calculate the size of the stack as follows

$$S_S = M \times \lceil Addr_{hash} + Content_{hash} + Id_{hash} \rceil_{byte} \qquad (8)$$

In Eq. (8), the symbol $\lceil \ldots \rceil_{byte}$ denotes rounding to byte of the bit sum of fields in the stack.

Next, we consider the new pattern as a long pattern and break it into segments to re-insert to the system. This process is recursive until the successful insertion occurs. We can prove that the update process of a string (pattern or segment) is limited by following paragraphs.

After un unsuccessful insertion of string less than or equal L_{max_s}, the string must be broken into halves. Note that the partitioning of long patterns greater than L_{max_s} is preprocessed off-line before updating. The online partitioning is slightly different from off-line method as mentioned in Sect. 3.2. We do not consider whether the short pattern in the engine can be the prefix segment to save time. So the unsuccessful insertion time and the partitioning time can be calculated as follows.

$$T_{in\_u_L} = M \times T_{proc_L} + M = M(T_{proc_L} + 1) \qquad (9)$$

$$T_{part_L} = 2T_{lu_{L/2}} + 2T_{in_{L/2}} \qquad (10)$$

The unsuccessful insertion time equals to the sum of insertion time and back tracking time. The partitioning time includes the lookup and insertion time of two halves. The worst case insertion time can be expressed by the sum of the unsuccessful time and the partitioning time.

$$\begin{aligned} T_{in\_w_L} &= T_{in\_u_L} + T_{part_L} \\ &= T_{in\_u_L} + 2T_{lu_{L/2}} + 2T_{in_{L/2}} \end{aligned} \qquad (11)$$

Let us assume that $L_{max\_s} = 2^K$ and the shortest segment is one-character which is always successful, so the insertion time $T_{in_1} = T_{in\_w_1}$.

Lemma 1. *The worst case insertion time of a new string of length $L_{max\_s} = 2^K (K \geq 1)$ is limited by the following.*

$$T_{in\_w_{2^K}} \leq \sum_{i=0}^{K-1} 2^i (T_{in\_u_{2^{K-i}}} + 2T_{lu_{2^{K-i-1}}}) + 2^K T_{in\_w_1}$$

Proof. The proof is mathematical induction using Eq. (11).

- *Basis*: for $K = 1$, according to Eq. (11), it is true.

$$\begin{aligned} T_{in\_w_2} &= T_{in\_u_2} + 2T_{lu_2} + 2T_{in_1} \\ &\leq T_{in\_u_2} + 2T_{lu_2} + 2T_{in\_w_1} \end{aligned}$$

- *Inductive step*: we assume that the result holds for some unspecified value of K.
  We must be shown that the result holds for $K + 1$, that is:

$$\begin{aligned} T_{in\_w_{2^{K+1}}} &= T_{in\_u_{2^{K+1}}} + 2T_{lu_{2^K}} + 2T_{in_{2^K}} \\ &\leq T_{in\_u_{2^{K+1}}} + 2T_{lu_{2^K}} \\ &\quad + 2\Bigg[\sum_{i=0}^{K-1} 2^i (T_{in\_u_{2^{K-i}}} + 2T_{lu_{2^{K-i-1}}}) \\ &\quad + 2^K T_{in_{2^K/2^K}} \Bigg] \\ &\leq \sum_{i=0}^{K} 2^i (T_{in\_u_{2^{K+1-i}}} + 2T_{lu_{2^{K-i}}}) \\ &\quad + 2^{K+1} T_{in\_w_1} \end{aligned}$$

  Hence, the proof. □

From Eqs. (9), (3) and (4):

$$\begin{aligned} T_{in\_w_{2^K}} \leq \sum_{i=0}^{K-1} 2^i [M(2^{K-i} + 2T_{mem} + 1) \\ + 2(2^{K-i-1} + 2T_{mem} + T_{mtch})] + 2^K T_{in\_w_1} \end{aligned} \quad (12)$$

Moreover, for on-the-fly update without interrupting the data stream, we can use a *FIFO* to buffer data stream. While a pattern is being updated, the data stream enters the FIFO. After the successful or unsuccessful update of a pattern, PAMELA will get data from the FIFO. The next update can continue as soon as the FIFO is empty. Because the network load is not 100% at all time, this on-the-fly update process is practical. If the line rate network is one byte per clock cycle, the size in byte of FIFO equals to the upper bound of a string insertion.

We assume that all time parameters in Eq. (12) consuming one clock cycle, $K = 4$, and $M = 30$ so we have the maximum successful insertion time of a pattern less than 3440 clock cycles as follows.

$$\begin{aligned} T_{in\_w_{2^K}} &\leq \sum_{i=0}^{K-1} [2^K (M + 1) + 2^i (3M + 6)] + 2^K \\ &\leq 109M + 170 = 3440 = S_F \end{aligned} \quad (13)$$

where S_F is the size of the FIFO.

4.1.3 Hardware Utilization

In the direct storage method, the number of elements per hash table size named *load factor* presents the efficiency of memory utilization. In Cuckoo Hashing, load factor is less than 0.5 to guarantee success of pattern insertion. If we define n_{avg} as the average number of patterns and m_{avg} as the average size of hash table in each Cuckoo module, then the average load factor α is given by the formula $\alpha = n_{avg}/(2 \times m_{avg})$

In our indirect storage method, the hash tables only store the indices of patterns using a few hardware resources, and most of resources are for pattern storage. Therefore, the memory utilization metric of our system has to take into account the condensed table T_3. If we assume the size in bit of T_3 is equal to the number of bits of pattern storage required, then it is the result of n_{avg} in a module multiplied by the average number of bits $(8 \times L_{avg})$ per pattern of this module. The number of bits to encode the address of storage table T_3 is $\lceil log_2 n_{avg} \rceil$, where $\lceil \rceil$ denotes rounding up. Finally, the average memory utilization U_{mem} is calculated as

$$U_{mem} = \frac{n_{avg} \times \lceil log_2 n_{avg} \rceil + n_{avg} \times 8 \times L_{avg}}{2 \times m_{avg} \times \lceil log_2 n_{avg} \rceil + n_{avg} \times 8 \times L_{avg}} \quad (14)$$

In Eq. (14), U_{mem} is the ratio of the patterns and their indices with the total sizes of T_1, T_2 and T_3. Figure 10 shows the effect of U_{mem} based on load factor α. With the maximum value of α of 0.5, U_{mem} is approximately 0.88 when $m_{avg} = 512$, $n_{avg} \simeq 500$ and $L_{avg} \simeq 8$; we name it *PAMELA-1*. As α reduces by half to 0.25, U_{mem} slightly reduces and equals to 0.72 when $m_{avg} = 1{,}024$; we name it *PAMELA-2*. To consider lower value of load factor, it is not interesting in due to wasting too many memory resources. Therefore, we only select these two systems for further testing. If the balance between the high flexibility of update and the area is the first priority then *PAMELA-2* is selected. Otherwise, hardware-efficient *PAMELA-1* is the choice. Next section will show the practical comparison of two systems.

Besides the memory utilization, the logic gate (logic

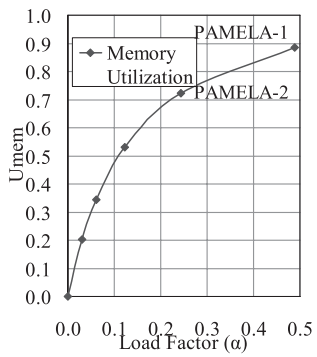

Fig. 10 Memory Utilization vs. Load Factor. *PAMELA-1* has Memory Utilization U_{mem} of 0.88, *PAMELA-2* has U_{mem} of 0.72.

cell) utilization is also our interest. We assume that if a character needs p_{LC} logic gates for implementing hash function as in Fig. 2 a then each pattern with length L needs $L \times p_{LC}$ logic gates. With $L_{max\_s} = 16$ modules, the number of logic gates is very large for the general architecture. However, as in Fig. 2 b, our optimized architecture only uses p_{LC} logic gates for each module. The notation R_{LC} to represent the ratio between logic gates of the general architecture and the optimized architecture is

$$R_{LC} = \frac{\sum_{L=1}^{L_{max\_s}} (L \times p_{LC})}{L_{max\_s} \times p_{LC}} = \frac{L_{max\_s}(L_{max\_s}+1) \times p_{LC}}{2 \times L_{max\_s} \times p_{LC}} = \frac{(L_{max\_s}+1)}{2} \quad (15)$$

If L_{max_s} is 16 then the saving ratio is over 8 times for implementing the hash functions.

4.2 Performance Simulations

4.2.1 Off-Line Insertion of Short Patterns

On Dec 15, 2006, there were 4,748 unique patterns with 64,873 characters in Snorts rule set. The distribution of the pattern lengths in Snort database is from 1 up to 109 characters. Fortunately, 65% of total numbers of patterns are up to 16 characters. Therefore, we build the Cuckoo Hashing modules for short patterns which are less than or equal to $L_{max\_s} = 16$ characters according to this fact.

For practical comparison, we implement hash functions with patterns of the lengths from 2 to 16 characters. For pattern length of one character, we directly match the patterns to save the hardware. In all experiments from now on, the number of trials is 1,000. We define a parameter named *%Rehash* as the following equation to determine the possibility of rehash happening in every trial in every pattern length.

$$\%Rehash = \frac{the\ number\ of\ rehashes}{the\ number\ of\ trials} \times 100\% \quad (16)$$

Figure 11 presents the number of insertions of Cuckoo Hashing with three hash functions: SAX, random table and CRC in which the size of index table is 512. Although CRC is a polynomials function, it is a suitable candidate for testing due to its significantly cheap implementation on hardware [29]. Three names, *SAX_hard*, *Tab_hard* and *CRC_hard*, are the FPGA-based systems whose architectures are changed as in Sect. 3.1 of this paper with patterns inserted in balancing.

The results in Fig. 11 show that the FPGA-based systems have the number of insertions less than the original systems by 20% and the performance of SAX hash function is close to that of random table. The results in Fig. 11 also show that the SAX and random table functions are significantly more efficient than CRC function with very small *%Rehash* of less than 5%. With the index table size of 512,

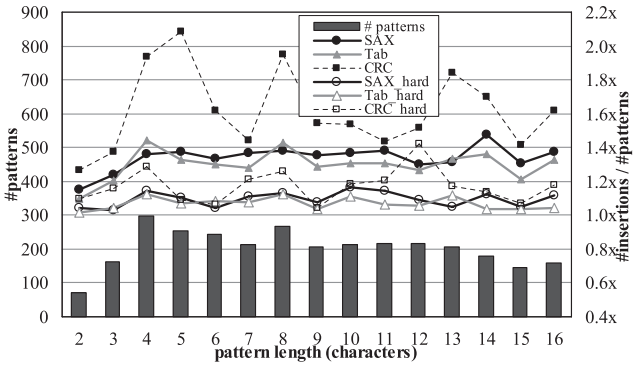

Fig. 11 The number of insertions of various hash functions vs. pattern length (characters). Bar graphs are the numbers of patterns. Line graphs are the ratio of numbers of insertions over numbers of patterns. Index (hash) table size is 512. The number of trials is 1,000.

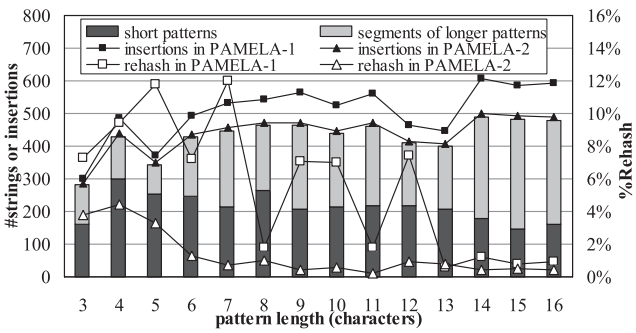

Fig. 12 The number of insertions and %Rehash after addition of longer patterns vs. pattern lengths (L). *PAMELA-1* and *PAMELA-2* have the index table sizes of 512 and 1,024, respectively. Both systems are based on SAX hash function and our improved architecture. The number of trials is 1,000.

the average load factor of index table is about 1/4. The remaining space can be used for fitting the segments of patterns with lengths of over 16 characters.

4.2.2 Off-Line Insertion of Long Patterns

We break 1,643 long patterns of Snort rule set into over 3,500 segments of lengths from 3 to 16 characters as the described technique in Sect. 3.2. By sharing the prefixes with short patterns in the engine, the number of unique segments reduces about 12%. Let *string* denote the short pattern and the segment of long pattern. Totally, there are 6,136 strings of pattern set. Note that the distribution of segments, Th, is also considered to make the string number less than 512 in every length. This condition helps reduce the number of block RAMs of FPGA for implementing T_3 as well as T_1 and T_2.

The design can be parameterized with different table depths of 512 and 1,024 entries as *PAMELA-1* and *PAMELA-2*, respectively as defined in the previous subsection. These systems are used to evaluate the trade off between hardware utilization and performance of insertion. Both systems are based on SAX hash function and the FPGA-based Cuckoo architecture. Figure 12 shows the

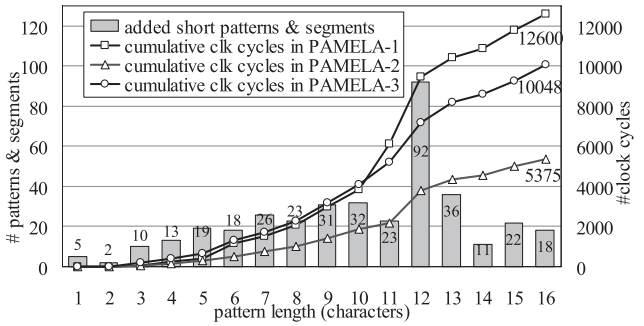

Fig. 13 The average insertion time (clock cycles) for inserting 381 new strings (patterns & segments). *PAMELA-3* is extended from *PAMELA-1* by adding a stack and a FIFO for limited-time and uninterruptible update. The number of trials is 1,000.

number of insertions as well as number of strings in every length after adding segments of long patterns. In every length of string, the number of insertions in *PAMELA-2* is just greater than the number of strings slightly. Unfortunately for *PAMELA-1*, the number of insertions is about 16% greater than the number of strings. *%Rehash*es of both systems are showed in Fig. 12 to explain why the number of insertions increases in *PAMELA-1*. Due to high memory utilization, the collision (*%Rehash*) increases as the number of patterns increases, approximately 12% as compared with 4% of *PAMELA-2*.

4.2.3 Dynamic Update for New Patterns

Snort rule database must be updated to handle the new attacks. Normally, the web site of Snort generates new rules every one or two weeks. As a result, PAMELA can also be rapidly and easily updated to avoid vulnerability.

To demonstrate the dynamic update ability of PAMELA, we accumulated the latest Snort pattern set on May 14, 2007 to 5,026 unique patterns and 68,266 characters. Based on the pattern set on Dec 15, 2006, there were 293 unique and new patterns consisting 3,476 characters and 15 unique patterns consisting 83 characters deleted. Note that all new patterns are searched in the system before inserting. Within these 293 new patterns, 65 patterns are longer than 16 characters. These long patterns are broken off-line into segments as described in Sect. 3.2. Finally, only 381 unique short patterns and segments are inserted.

The average insertion time in number of clock cycles of each PAMELA system is compared in Fig. 13. To determine M, we can assign $\epsilon = 1$, $m_{avg} = 1{,}024$ for all systems; thus the result of M is 30. Due to bigger table size of 1,024 entries, *PAMELA-2* takes 5,375 clock cycles to insert without rehashing. In *PAMELA-1*, the numbers of clock cycles at pattern lengths of 6, 11, 12 and 15 characters rapidly increase due to the high penalty of rehashing. With *%Rehash* from 0.6%–2.9%, it takes 12,600 clock cycles to insert that is about 2.5 times as compared with *PAMELA-2*. *PAMELA-3* is extended from *PAMELA-1* by adding a stack and a FIFO for limited-time and uninterruptible update. According to Eqs. (8) and (13), the sizes of the stack and FIFO are 90 and 3440 bytes, respectively. Some new patterns with lengths of 6, 11, 12 and 15 characters are broken into segments with lengths of 3, 5–8 characters as they are unsuccessfully inserted into the index tables. Then, the segments are inserted into the modules with corresponding lengths. Some segments with length of 6 are continually broken again due to high collision but the total of partitioning times is only 2. Although the number of strings needed for insertion increases up to 390, the number of clock cycles including partitioning time in *PAMELA-3* reduces to 10,048. For 15 deleted patterns, the process is fast with around 100 clock cycles for each system.

Table 2 Dynamic update comparison for a pattern.

System	Clock Freq. (MHz)	Update time (μs)	Additional hardware requirement
PAMELA-1 (index table size:512)	200 (assumed)	0.21	No
PAMELA-2 (index table size:1,024)		0.09	No
PAMELA-3 (extended from *PAMELA-1*)		0.17	A stack & FIFO for uninterruptible update
Bit-split AC [23]	N/A	$\sim 10^6$	A temporary module
Rom+CoProc [10]	260	$\sim 1/3 \times 10^3$	A co-processor

We assume that *PAMELA-1* to *PAMELA-3* are clocked at 200 MHz which is less than our synthesized results shown later and can be achieved on many common Virtex FPGA boards. With 5,475 to 12,700 clock cycles, the update time is about 28 to 64 microseconds (μs). Table 2 shows the comparison of the dynamic update time of some systems. [23] shows that the system can update non-interrupting data stream by using a temporary module for updating only and the duration is less than one second for compiling and updating. [10] uses a co-processor to update with estimated time of 10 milliseconds (ms) for 30 new patterns. Thus we can show the approximate time for one pattern is 1/3 ms. We use no additional hardware in *PAMELA-1* and *PAMELA-2* of simulation systems. As a result, to add 293 new patterns, the average insertion time for a pattern is 19–43 clock cycles; and with added the delete time, the average time for updating one pattern is only 18–42 clock cycles, about 0.09–0.21 μs on 200 MHz FPGA systems. If the limited-time and uninterruptible update system is required then *PAMELA-3* is used; the insertion time of a new pattern is limited in 3440 clock cycles ($\sim$17 μs). These results show that PAMELAs are very efficient for updating pattern set without lengthy FPGA reconfiguration time.

5. Extension for Multi-Character Processing

To increase the throughput, PAMELA can be easily extended for multi-character processing. Instead of reading one character per clock cycle, the system shifts N characters into an input buffer every clock cycle The system has N blocks corresponding to N characters needed to process. In-

side the block, each Cuckoo module receives one character from the pre-determined address of input buffer and the hash value from the previous module. Figure 14 is an example of our system to process 4-character per clock cycle. Due to higher addressing complexity, we need to determine the address of input buffer that each Cuckoo Module is connected to. In each block j, if the address connected to the previous module is $A_{i-1,j}$ ($2 \le i \le L_{max\_s}$) and contents character x_k then the address connected to the next module, $A_{i,j}$, has to shift $N-1$ from $A_{i-1,j}$ and the next module processes the next character. For example, if the string is "...*abcd*...", each time we shift $N = 4$ characters, the previous module consumes the character $'a'$ at address $A_{i-1,j}$ then the next module consumes the character $'b'$ at address $A_{i-1,j}+(N-1)$ one clock cycle later. In general, the address in the buffer connected to Cuckoo i of block j is:

$$\begin{aligned} A_{i,j} &= A_{i-1,j} + (N-1) = A_{i-2,j} + 2(N-1) \\ &= \ldots = A_{1,j} + (i-1)(N-1) \end{aligned} \tag{17}$$

From Eq. (17), we can see that when $N = 1$, the address of any module coincides with the first address. So it is correct with our one-character design. We can also calculate the size of input buffer from Eq. (17). The last address is $A_{L_{max\_s},1}$ when $i = L_{max\_s}$. If the lowest address is zero then $A_{1,1}$ is $N-1$. The size of the buffer can be calculated as follows.

$$\begin{aligned} S_{Buffer} &= A_{L_{max\_s},1} + 1 \\ &= A_{1,1} + (L_{max\_s} - 1)(N-1) + 1 \\ &= L_{max\_s}(N-1) + 1 \end{aligned} \tag{18}$$

The resources for storing patterns in multi-character processing scheme can be significantly reduced by sharing SRAMs together. If the number of ports of SRAMs are two times of the number of processed characters, T_3 can be shared for Cuckoo modules that process the same pattern length. Moreover, the processing of long pattern can be also carried out only one table T_4 for the whole system.

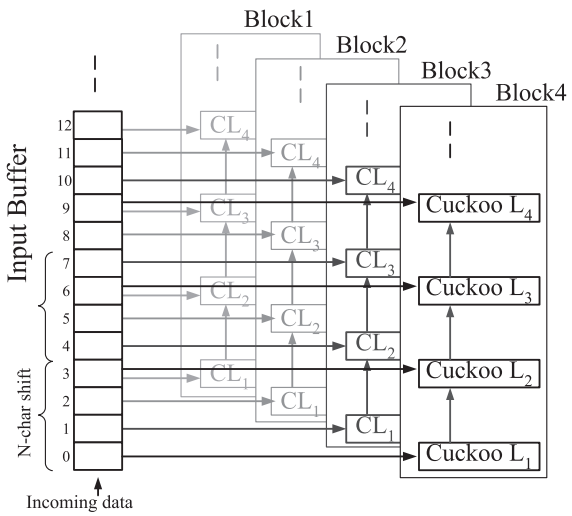

Fig. 14 PAMELA for N-character processing ($N = 4$). Cuckoo Modules are connected to the determined addresses of input buffer.

6. FPGA Implementation Results

Our design is developed in Verilog hardware description language and by Xilinx's ISE 8.1i for hardware synthesis, mapping, and placing and routing. The target chips are some major Xilinx FPGA chips such as Virtex2, Virtex-Pro and Virtex-4. To reduce the number of memory blocks in FPGA, we can implement two index tables in the same block RAM; T_1 is in a low addresses part and T_2 is in a high addresses part. The block RAM of Xilinx FPGA can be configured as dual-port mode that can be accessed concurrently.

Based on our parallel pattern matching engine described earlier, we can measure the cost based on the numbers of block RAMs and logic cells in Table 3. With the maximum capacity of 18 kbits, block RAMs can be programmed as $18\text{K} \times 1$ bit to 512×36 bits, in various depth and width configurations. Hence, we can configure both tables T_1 and T_2 with size $1{,}024 \times 9$ bits for each as *PAMELA-2* mentioned above. Since the number of patterns in each Cuckoo module is less than 500, we can set the depth of T_3 as 512. The width of T_3 can be determined by the pattern length added 2 more bits for controlling long patterns. For table T_4, the width of T_4 is equal to the number of its address bits plus 1 bit for suffix segment of long pattern. For this reason, the size of T_4 is $2^{13} \times 14$. For the short patterns of one character, we can directly compare to save hardware resources. The other parts of the system use logic cell of FPGA. While SAX functions use few resources of logic cells, the most consuming parts are the comparators. Totally, we use only 62 block RAMs and 3,220 logic cells to fit 68,266 characters of the entire rule set on the XCV4LX100 FPGA chip. To update without interrupting the incoming data, three more block RAMs are used to implement the stack and FIFO. For multi-character processing, since current Xilinx FPGA chips only have 2-port block RAM, we have to duplicate the pattern in T_3s. In addition, T_4s can only process 2 lookups at the same clock cycle. Some components such as the controls, counters, registers, LFSR, etc. are also shared inside the system.

Table 4 shows the comparison of our synthesized systems with other recent FPGA systems. Two metrics, logic

Table 3 Logic and memory cost of components in Virtex-4.

Component	Quantity	Block RAMs	Logic cells	Note
Tables T_1 & T_2	15×2	15	0	2 tables in a BRAM
Table T_3	16	39	0	The BRAM numbers for lengths 1 to 4: 1; 5 to 8: 2 9 to 13: 3; 14 to 16: 4
Table T_4	1	8	0	
SAX hashes	15×2	0	840	
Multiplexer	15×2	0	300	
Comparator	32	0	1,320	16 for long patterns
Counter, Priority, Reg, LFSR, etc.		0	760	
Total		**62**	**3,220**	

Table 4 Performance comparison of FPGA-based systems for NIDS/NIPS.

System	Device (Xilinx)	bits/ cycle	Freq. (MHz)	No. chars	No. LCs	Mem (kbits)	LCs/ char	Mem per char (bits)	T-put (Gbps)	PEM
	XC4VLX100	**8**	**285**		**3,220**	**1,116**	**0.047**	**16.74**	**2.28**	**10.29**
	XC4VLX100	**16**	**282**		**6,120**	**2,070**	**0.090**	**31.05**	**4.51**	**10.92**
	XC4VLX100	**32**	**275**		**11,980**	**4,140**	**0.175**	**62.10**	**8.80**	**10.70**
PAMELA-2	**XC2VP20**	**8**	**272**	**68,266**	**3,266**	**1,116**	**0.048**	**16.74**	**2.18**	**9.79**
	XC2V6000	**8**	**223**		**3,266**	**1,116**	**0.048**	**16.74**	**1.78**	**8.03**
	XC2V6000	**16**	**218**		**6,212**	**2,070**	**0.091**	**31.05**	**3.49**	**8.42**
PAMELA-2 (with a FIFO and a stack)	**XC4VLX100**	**8**	**285**	**68,266**	**3,233**	**1,170**	**0.047**	**17.55**	**2.28**	**9.91**
V-HashMem [13]	XC2VP30	8	306	33,613	2,084	702	0.060	21.39	2.49	8.60
HashMem [12]	XC2V1000	8	250	18,636	2,570	630	0.140	34.62	2.00	4.01
	XC2V3000	16	232		5,230	1,188	0.280	65.28	3.71	3.86
PH-Mem [11]	XC2V1000	8	263	20,911	6,272	288	0.300	14.10	2.11	4.71
	XC2V1500	16	260		10,224	306	0.490	14.98	4.16	6.44
ROM+Coproc [10]	XC4VLX15	8	260	32,384	8,480	276	0.260	8.73	2.08	5.90
Prefix Tree [9]	XC2VP100	8	191	39,278	12,176	0	0.310	0	1.53	4.93
PreD-CAM [7]	XC2V3000	8	372	18,036	19,854	0	1.100	0	2.98	2.70
	XC2V6000	32	303		64,268	0	3.560	0	9.70	2.72
FPGA-based Bit-Split [24]	XC4FX100	8	200	16,715	4,514	6,000	0.270	184	1.60	0.39
PreD-NFA [6]	XC2V8000	32	219	17,537	54,890	0	3.130	0	7.00	2.24

cells per character (LCs/char) and SRAM bits per character (bits/char), are used to evaluate the efficiency of FPGA utilization. For state machine approach, we just show some interesting implementations that are commonly used for static pattern matching only. We can see that the hashing systems [10]–[13] are almost better than state machine [6], [24] and compare-and-shift ones [7], [9] in term of hardware utilization. As compared to V-HashMem [13], PAMELAs are 30% fewer in LCs/char and bits/char. Note that V-HashMem supports header matching with less hardware. On the contrary, with 6,500 strings inserted into the system, the storage capacity of synthesized PAMELA can support 1500 more strings without additional hardware resource. As compared to [10], [11], the Block RAM usage of our architecture is 1.22–2.07 times greater than theirs, but our logic cell usage is significantly smaller than theirs by 4.3–5.2 times. In summary, PAMELAs are the most efficient ones in using logic cells of FPGA at a cost of 0.047-0.175 LCs/char. In addition, the memory usage of our architecture is of high density (16.74–62.10 bits/char) and is acceptable as compared to other systems.

For throughput comparison, our throughput (T-put) can vary from 1.78–8.8 Gbps depending on the kinds of FPGA chips and the number of characters processed per clock cycle. Some works can also process multi-character at very high throughput up to 10 Gbps, especially shift-and-compare architectures. However, the area cost is high. Therefore, a Performance Efficiency Metric (PEM) is used as the ratio of throughput in Gbps to the logic cell per each pattern character for performance evaluation.

$$PEM = \frac{Throughput}{\frac{No.Logiccells + \frac{Membytes}{12}}{No.Characters}} \quad (19)$$

Assuming that the cost of 12 bytes block RAMs is equivalent to a logic cell [30], Eq. (19) takes into account both block RAMs and logic cells area metrics for fair comparison [14] between the memory-based systems and the logic gate-based systems. As PEMs in the range of 8.03–10.92, PAMELAs are the best of the FPGA-based hashing systems, far better than the shift-and-compare systems by at least two times, and better than the state machine systems over three times.

7. Conclusion

A pattern matching engine based on Cuckoo hashing for NIDS/NIPS named PAMELA is proposed. PAMELA engine can update the new patterns rapidly in the orders of microseconds. PAMELA can also guarantee that the data stream cannot be interrupted during pattern updates by adding both small stack and FIFO. According to the implementation results, the performance of PAMELA is the best when compared with other previous systems and the achievable throughput can be up to 8.8 Gbits/s. The current scheme of PAMELA is also scalable enough to process N characters every clock cycle. Since PAMELA requires no reconfiguration at all, this engine can be applied on ASIC at much higher performance than FPGA.

Acknowledgments

This work was supported by the AUN-SeedNet Program of JICA.

References

[1] http://www.snort.org, "SNORT official website."
[2] Y.H. Cho, S. Navab, and W.H. Mangione-Smith, "Specialized hardware for deep network packet filtering," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.452–461, 2002.
[3] I. Sourdis and D.N. Pnevmatikatos, "Fast, large-scale string match for a 10gbps FPGA-based network intrusion detection system," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.880–889, 2003.

[4] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.31–38, 2003.
[5] Y.H. Cho and W.H. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.125–134, 2004.
[6] C.R. Clark and D.E. Schimmel, "Scalable pattern matching for high speed networks," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.249–257, 2004.
[7] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.258–267, 2004.
[8] S. Yusuf and W. Luk, "Bitwise optimised CAM for network intrusion detection systems," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.444–449, 2005.
[9] Z.K. Baker and V.K. Prasanna, "High-throughput linked-pattern matching for intrusion detection systems," Proc. ACM/IEEE Symp. on Architecture for Networking and Communications Systems, pp.193–202, 2005.
[10] Y.H. Cho and W.H. M.-Smith, "Fast reconfiguring deep packet filter for 1+ gigabit network," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.215–224, 2005.
[11] I. Sourdis, D.N. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.644–647, 2005.
[12] G. Papadopoulos and D.N. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.39–44, 2005.
[13] D. Pnevmatikatos and A. Arelakis, "Variable-length hashing for exact pattern matching," Proc. Int. Conf. on Field-Programmable Logic and Applications, pp.1–6, 2006.
[14] I. Sourdis, D.N. Pnevmatikatos, and S. Vassiliadis, "Scalable multi-gigabit pattern matching for packet inspection," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.16, no.2, pp.156–166, 2008.
[15] R. Pagh and F.F. Rodler, "Cuckoo hashing," J. Algorithms, vol.51, no.2, pp.122–144, 2004.
[16] T.N. Thinh, S. Kittitornkun, and S. Tomiyama, "Applying Cuckoo hashing for FPGA-based pattern matching in NIDS/NIPS," Proc. Int. Conf. on Field-Programmable Technology, pp.121–128, 2007.
[17] Z.K. Baker and V.K. Prasanna, "Time and area efficient pattern matching on FPGAs," Proc. ACM/SIGDA Symp. on FPGAs, pp.223–232, 2004.
[18] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, pp.111–120, 2002.
[19] J.C. Bispo, I. Sourdis, J.M. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," Proc. Int. Conf. on Field-Programmable Technology, pp.119–126, 2006.
[20] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," Proc. ACM/IEEE Symp. on Architecture for Networking and Communications Systems, pp.127–136, 2007.
[21] A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to bibliographic search," Commun. ACM, vol.18, no.6, pp.333–340, 1975.
[22] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," SIGARCH Comput. Archit. News, vol.33, no.1, pp.99–107, 2005.
[23] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," Proc. IEEE Symp. on Computer Architecture, pp.112–122, 2005.
[24] H.J. Jung, Z.K. Baker, and V.K. Prasanna, "Performance of FPGA implementation of bit-split architecture for intrusion detection systems," Proc. Reconfigurable Architectures Workshop, 2006.
[25] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J.W. Lockwood, "Deep packet inspection using parallel bloom filters," Symposium on High Performance Interconnects, pp.44–51, 2003.
[26] L. Carter and M.N. Wegman, "Universal classes of hash functions," J. Comput. Syst. Sci., vol.18, no.2, pp.143–154, 1979.
[27] M.V. Ramakrishna and J. Zobel, "Performance in practice of string hashing functions," Proc. Int. Conf. on Database Systems for Advanced Applications, pp.215–224, 1997.
[28] http://www.xilinx.com/bvdocs/appnotes/xapp211.pdf, "Xilinx application note."
[29] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and CRC's over real data," IEEE/ACM Trans. Netw., vol.6, no.5, pp.529–543, 1998.
[30] T. Sproull, G. Brebner, and C. Neely, "Mutable codesign for embedded protocol processing," Proc. Int. Conf. on Field Program Logic and Applications, pp.52–56, 2005.

Tran Ngoc Thinh received the B.E. degree from the Ho Chi Minh City University of Technology, Vietnam, in 1999, and the M.E. from the King Mongkut's Institute of Technology Ladkrabang, Thailand, in 2006, all in computer engineering. He is currently pursuing the Ph.D. degree in computer engineering from the King Mongkut's Institute of Technology Ladkrabang, Thailand. His research interests include Network security and bioinformatics on reconfigurable devices.

Surin Kittitornkun received his Ph.D. and MS.EE. from University of Wisconsin-Madison, USA in 2002 and 1997, respectively. His research interest includes high performance/parallel computing in FPGA. He was awarded Gerald Holdridge Course/Lab Development Teaching Assistant Award in 2002. Prior to that he was an intern at Motorola Incorporation, Schaumburg, Illinois. He is now an Assistant Professor at the Department of Computer Engineering, Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand.

Shigenori Tomiyama received the B.S. and M.S. degrees from Tokai University, Kanagawa, Japan, in 1966 and 1968 respectively. He received the Ph.D. degree in Engineering from Tokai university in 1988. In 1968, he joined Department of Communications Engineering, Tokai University, as an assistant professor. He has been a Professor since 1988. He is currently a professor with the Department of Embedded Technology, Tokai University. His research interests include system characteristics approximation, digital filter design and FPGA. He is a member of IEEE.