Computers and Security, 2010 (Elsevier Journal)

Efficient Hardware Support for Pattern Matching in Network Intrusion Detection

Nitesh B. Guinde and Sotirios G. Ziavras
Electrical and Computer Engineering Department
New Jersey Institute of Technology, Newark, NJ 07102, USA

Abstract— Deep packet inspection forms the backbone of any Network Intrusion Detection (NID) system. It involves matching known malicious patterns against the incoming traffic payload. Pattern matching in software is prohibitively slow in comparison to current network speeds. Due to the high complexity of matching, only FPGA (Field-Programmable Gate Array) or ASIC (Application-Specific Integrated Circuit) platforms can provide efficient solutions. FPGAs facilitate target architecture specialization due to their field programmability. Costly ASIC designs, on the other hand, are normally resilient to pattern updates. Our FPGA-based solution performs high-speed pattern matching while permitting pattern updates without resource reconfiguration. To its advantage, our solution can be adopted by software and ASIC realizations, however at the expense of much lower performance and higher price, respectively. Our solution permits the NID system to function while pattern updates occur. An off-line optimization method first finds common sub-patterns across all the patterns in the SNORT database of signatures [14]. A novel technique then compresses each pattern into a bit vector, where each bit represents such a sub-pattern. This approach reduces drastically the required on-chip storage as well as the complexity of pattern matching. The bit vectors for newly discovered patterns can be generated easily using a simple high-level language program before storing them into the on-chip RAM. Compared to earlier approaches, not only is our strategy very efficient while supporting runtime updates but it also results in impressive area savings; it utilizes just 0.052 logic cells for processing and 17.77 bits for storage per character in the current SNORT database of 6455 patterns. Also, the total number of logic cells for processing the traffic payload does not change with pattern updates.

Keywords— Field-Programmable Gate Array (FPGA), Pattern Matching, Network Intrusion Detection (NID), SNORT database.

1. Introduction

There have been many computer network attacks in recent times which were difficult to detect based only on packet header inspection. Deep packet inspection of the payload is needed to detect any application level attack. In the area of NID systems, new vulnerabilities are identified on a daily basis and appropriate rules are developed for defense. These rules may represent either new signatures or changes to existing ones. From October 2007 to August 2008, 1348 new SNORT rules were added while 8170 rules were updated (on a daily or weekly basis). The most recent 2.8 version of July 29th, 2009 contains 15,730 rules that involve 6455 distinct patterns of sequential characters; our evaluation uses this version of SNORT. It becomes obvious that robust NID systems should handle pattern updates (including additions, deletions and editions) without taking them off-line. Signature matching is also relevant to virus detection techniques that look for the presence of specific command sequences (of characters) inside a program [19]. Although we focus on pattern matching for NID, our approach can be extended for virus detection as well where new signatures are added almost daily.

The majority of deep packet inspection systems that try to identify malicious signatures employ pattern matching software running on general-purpose processors. The Boyer-Moore [21] and Aho-Corasick [24] string matching algorithms have been adopted in NID research. The Boyer-Moore algorithm performs matching from right to left by aligning the pattern to be matched with the input stream in such a way that the rightmost character of the pattern matches with the stream. It continues matching from right to left, and if a mismatch is encountered, then it skips all the characters upto the next alignment of rightmost characters. The Aho-Corasick algorithm builds a finite state machine from keywords (i.e., chosen pattern pieces) and processes the input text strings in a single pass. The work in [22] presented a multi-pattern matching algorithm combining the one-pass approach of Aho-Corasick with the skipping feature of Boyer-Moore. Tuck et al. [23] take a different approach to optimizing Aho-Corasick by incorporating bitmap and path compression to reduce storage. However, these software approaches do not adapt well for hardware realizations even though their database of rules can be updated quite easily. Their major disadvantage is the sequential software-driven matching process which is very slow. Thus, the pattern matching process cannot keep up with fast network speeds; as a result, some packets may be dropped while others may not be inspected at all. Existing hardware-based solutions, FPGA or ASIC, on the other hand have the potential to match network speeds but often suffer from flexibility issues related to database updates. FPGAs often match network speeds at the cost of complete system reconfiguration for pattern updates. The time penalty for complete system synthesis can be on the order of several hours, while the penalty for full FPGA reconfiguration can be many milliseconds/seconds [9]. Also, reconfiguration can be a tedious process involving digital-circuit redesign to support new rules. Therefore, complete system reconfiguration is not prudent for 24/7 active networks.

Common FPGA-based NID approaches aim to minimize the consumed area, match the network speed and rarely reduce the time for updates. The majority of them embed specialized state machines where each state represents an input sequence of known characters; state transition information is stored in a location pointed to by the next incoming character [4, 7]. Only a few papers [1-3, 20] discuss flexible solutions that do not require FPGA reconfiguration when adding new patterns. Our pattern matching solution attempts to minimize the consumed chip area while operating at a high speed and also providing for reconfiguration-less runtime pattern updates. A quantitative comparison with the majority of these approaches is included in our results and comparisons section (Table II). A quantitative comparison with [20] appears later in Section 2.

In other related work, Baker et al. [5, 6] applied graph-theoretic techniques to partition the rule set into groups based on common character sequences; this approach reduces redundant searches across patterns and consequently the required area consumption. Similarly, our pattern preprocessing step first looks for common sub-patterns in the pattern set. We break the patterns into variable-length sub-patterns and also encode their positions in the original patterns. The ultimate objective for our designed circuit is to create a RAM address based on the incoming stream of characters. If a malicious pattern is present then this address points to a value exclusive to the respective pattern. This process reduces the search area to just one location.

To compress the stored information, a bit vector is created for each sub-pattern to denote its location in the entire set of malicious patterns. The resulting dramatic compression in pattern storage is due to the fact that a single bit now represents an entire sub-pattern. Also, this approach ultimately condenses character-based pattern matching into position-based bit-vector matching, a very efficient process. Applying simple AND-SHIFT operations on these bit vectors, complete pattern detection is possible without the need for rigid state machines.

2. Related Work

The terms table and RAM are used interchangeably in this paper. The capabilities of FPGAs have recently improved tremendously [15-17] so they are now frequently used by NID systems. Sidhu et al. [4] proposed a straightforward algorithm to construct non-deterministic finite automata (NFA) representing given regular expressions. Hutchings et al. [7] implemented a module to extract patterns

from the SNORT rule set [14] and then generated their regular expressions for NFA realization. Lin et al. applied minimization to the regular expressions for resource sharing [13]. To reduce data transfer widths, an 8-bit character decoder provides 256 unique outputs; various designs [5, 6, 7, 8, 9] were implemented. Since these designs hard-code the patterns into the FPGA fabric, runtime updates are forbidden without complete FPGA reconfiguration. Content-addressable memories (CAMs) that support updates were proposed by Gokhale et al. [10]. Sourdis et al. [11] applied pre-decoding with CAM-based pattern matching to reduce the consumed area. Yu et al. [12] used ternary content-addressable memory (TCAM) for pattern matching. TCAM is a CAM with three possible states for a stored bit, namely '0', '1' and 'x' (don't care). However, CAM approaches require large amounts of on-chip memory and have high power consumption since multiple comparators are activated in parallel; they represent unfavorable choices for large rule sets.

The lookup mechanism presented in [1] employs a hash table and several Bloom filters for a set of fixed-length strings to be searched in parallel by hardware. All of the herein cited FPGA-based schemes do not produce false positives, except for [1]. Since the majority of traffic is not normally malicious, an incoming packet can be checked using Bloom filters that never generate false negatives. However, the system can be overloaded. More specifically, positive responses by the Bloom filters require that the packet header be sent off-line to distinguish between a true positive and a false alert. Off-line processing is then very slow; also, the system can be easily attacked by overloading it with false positives. The CRC in [3] reduce the number of logic cells and the memory space. Patterns are first decomposed into varying-length fragments (for a maximum of 17 characters). They use a wide input, hashing a fixed number of characters from the 17-character input stream separately for different length fragments and then look up for the fragments in separate RAMs. Their approach limits compression opportunities due to actual storage of wide patterns into the memory for final comparison. The work in [20] applied Cuckoo hashing scheme. It uses varying-length sub-patterns and supports runtime updates. It yields a good compression in terms of stored bits and logic cells per character. However, if a collision shows up while inserting a pattern, Cuckoo attempts to recalculate the hashing key. When the number of recalculation iterations is maxed out, signifying that a key cannot be generated for distinct placement, rehashing is needed for all the keys, including those for sub-patterns stored previously. This process may then suffer from unpredictable penalties. In contrast, our design has higher flexibility in resolving collisions faster (Section 4).

The Cho et al. [2] pattern matching co-processor facilitates updates. Modules that detect sub-patterns forward the respective sub-pattern indices to state machines registering state transitions for contained patterns. Our design employs a first-stage component similar to that in [2], where the hashing of fixed-length character streams can identify sub-patterns. However, our design employs fewer logic resources and has smaller memory consumption per character in the SNORT database than all of these designs. Another major advantage of our design is that the pattern matching module does not normally need to increase in size with an increase in the number of malicious patterns.

3. Our Method

3.1. Pre-Processing

Assume a database of known malicious patterns and the need to design an FPGA-based pattern matching engine that can facilitate runtime updates without the need for hardware reconfiguration. This reliable engine should never produce false positives. Without loss of generality, we will test our implementation with the complete set of signatures in the SNORT database [14]. Initially we split patterns of length greater than a preset *Max_Fragment_Length* number of characters into fragments (i.e., sequences of at most Max_Fragment_Length characters). Although the longest pattern in SNORT contains 213 characters, 80% of the SNORT patterns contain up to 24 characters. This fragmentation should create fragments of length less than or equal to this value. It will be shown in this paper that this fragmentation reduces the size of our design considerably. From now on, the term "original pattern" or "O_Pattern" will denote a pattern in pattern set before fragmentation. The term "pattern" and "fragment" will denote

patterns from the new pattern set obtained after fragmentation of O_Pattern.

For our example here we assume that Max_Fragment_Length is 16. Fig. 1 shows sample original pattern set with patterns denoted by *O_Pattern 1* to *O_Pattern 6*. O_Pattern 1, which contains 18 characters, is fragmented into two patterns (patterns 1 and 7) having 9 characters each. We normally try to fragment the end of a O_Pattern into two equal halves. If O_Pattern is more than twice the Max_Fragment_Length, then we split the pattern in such a way that we produce fragments of lengths Max_Fragment_Length characters each (except for the last two tail fragments which will have almost identical lengths in terms of number of characters). For example, if a pattern contains 33 characters then its three produced fragments will normally have lengths of 16, 9 and 8 characters, respectively. This approach targets adequate processing time for the detection of the last two tail fragments. However, in some special cases we do not follow this fragmentation rule; these cases are presented in Section 4.1.

Original pattern set before fragmentation	**Pattern set after fragmentation**
O_Pattern 1: executemalware.exe	Pattern 1: executema
O_Pattern 2: usernametoolong	Pattern 2: usernametoolong
O_Pattern 3: Badcommand	Pattern 3: Badcommand
O_Pattern 4: Passwords	Pattern 4: Passwords
O_Pattern 5: commandlong	Pattern 5: commandlong
O_Pattern 6: codewords	Pattern 6: codewords
	Pattern 7: lware.exe

Fig. 1. Pattern sets before and after fragmentation.

After fragmentation, we move into the next stage of pre-processing. This stage involves two steps. The first step assigns distinct weights to all the ASCII characters. The second step generates two distinct *bit vector sets* for the known set of malicious patterns.

STEP 1 (WEIGHT ASSIGNMENT): The i^{th} ASCII character, for $0 \leq i \leq 255$, is assigned a unique *m-tuple of weights* represented by vector $W = \{weight_1, weight_2, \ldots, weight_m\}$; let *bw* be the number of bits in a weight element. These weight m-tuples are placed in a *character table* addressed to by the ASCII code of the character (an example is shown in Fig. 2).

Using these weight tuples we pre-calculate a *summation m-tuple* for each pattern in the new pattern set after fragmentation of the original set. Consider pattern "Badcommand" (pattern 3 in Fig. 1). We calculate the summation tuple for this pattern at static time in the following steps:

1) Split this new pattern into *groups of three contiguous characters*, except for the last tail group that is left with one character (three looks like an arbitrary number in this example; we discuss the choice of this number in another section):

"Bad" "com" "man" "d"

2) To derive the summation m-tuples for each of these character groups, apply the following position-weighted, element-wise summations involving the respective weight-tuples of constituent characters:

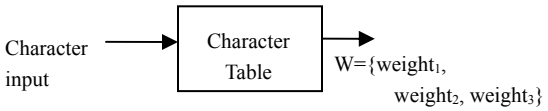
SUM("Bad") = W("B") + 2 * W("a") + 4 * W("d");
SUM("com") = W("c") + 2 * W("o") + 4 * W("m");
SUM("man") = W("m") + 2 * W("a") + 4 * W("n");
SUM("d") = W("d").

3) To derive the summation tuples for pattern 3 in Fig 1, apply the following element-wise summations involving the respective elements of weight tuples for the encompassed groups:

SUM("Badcommand") = SUM("Bad") + SUM("com") + SUM("man") + SUM("d").

This summation method is carried out on all the patterns. Fig. 3 shows the tuples for a chosen character table. The position of individual characters in the group of three is taken into account to create different sums (i.e., weight tuples) for patterns like "Badcommand" and "daBcommand" that contain identical, but permuted characters. However, sometimes we may have patterns with identical, but permuted groups of characters, like "Bad123" and "123Bad", which will result in identical sums since the position of groups is not accounted for in the final summation. This case will be identified in our pre-processing stage and will be dealt with by appropriately fragmenting one of them; e.g. "123Bad" could

be broken into “123B” and “ad”. There is no need for position-based summation for groups when producing the m-tuples of patterns since there is substantial flexibility in the selection of encompassed groups during pre-processing (as discussed in detail later on in this paper).



Character Table

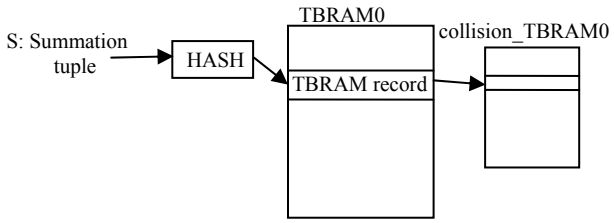
ASCII decimal value	Char	$weight_1$	$weight_2$	$weight_3$
0	NULL	3	1	1
.	.	.	.	.
.	.	.	.	.
65	A	5	3	5
66	B	2	2	2
67	C	3	6	7
.	.	.	.	.
.	.	.	.	.
255	ÿ	7	1	6

Fig. 2. Character table for m=3 and bw=3.

	Sum_1	Sum_2	Sum_3	Length of pattern
Pattern 1:	108	88	129	9
Pattern 2:	175	121	144	15
.	.	.	.	.
.	.	.	.	.
Pattern 7:	106	99	117	9

Fig. 3. Summation tuples for the pattern set in Fig. 1; SUM= (Sum_1, Sum_2, Sum_3).

Once we have pre-calculated the summation m-tuple of a pattern, we store the m-tuple in a table at a location which is produced by a hash function on this summation m-tuple. For our example we create up to 16 distinct tables to deal with 16 types of patterns containing one to 16 characters, respectively. However, since some patterns of certain lengths are lesser in population than others, we can store their summation m-tuples into one table instead of having separate tables for each of these lengths. In our example of Fig. 1, we place the summation m-tuples of 9-character patterns into TBRAM0 and the rest of the patterns containing 10, 11 and 15 characters into the common table TBRAM1. We choose the weights of characters in a way that ensures unique summation m-tuples for the patterns in a common TBRAM. In addition to the summation m-tuples, we also store two bit values, *start_fragbit* and *no_fragbit*, and a *collision_TBRAM_pointer* as shown in Fig. 4. If the start_fragbit is ‘1’ and no_fragbit is ‘0’ then this is the first fragment of a longer O_Pattern. If the start_fragbit is ‘0’ and no_fragbit is ‘1’ then the pattern is a complete O_Pattern with no fragmentation. Both start_fragbit and no_fragbit are ‘0’ if the pattern is a fragment of a longer O_Pattern but is not present at the start. If a pattern appears as the first fragment in a O_Pattern and as another fragment in another O_Pattern, then we change the chosen fragmentation during pre-processing in order to remove this case. There is a separate FRAM memory block in the O_Pattern match unit which is used to represent the sequences of fragments (i.e., patterns that constitute long O_Patterns). This is explained later in the O_Pattern match unit block section. A small collision_TBRAM stores the records of patterns that map to the same location in a TBRAM. The collision_TBRAM_pointer points to the first record in the collision list stored in the collision_TBRAM.



TBRAM Record: (Sum_1, Sum_2, Sum_3, start_fragbit, no_fragbit, Collision_TBRAM_pointer); Pattern 1 in Fig. 1 will have start_fragbit = '1'; Pattern 7 will have no_fragbit='0' and start_fragbit ='0'; Patterns 2 to 6 will have start_fragbit = '0' and no_fragbit='1'; "collision_TBRAM_pointer" points to the first record in a linear list of four other TBRAM records placed sequentially in case of collision.

Fig. 4. Summation m-tuples placed in TBRAM tables.

The maximum number of records per collision stored in collision_TBRAM varies with the implementation. According to our analysis, it suffices to set the maximum number of pattern collisions to five (one record in TBRAM and a maximum of four records stored sequentially in collision_TBRAM). If more records are mapped to the same location in a TBRAM, then we fragment patterns further to place them in exclusive locations in TBRAM (this process is explained later in Section 4.1).

STEP 2 (BIT/END VECTOR GENERATION): Our static-time pre-processing divides each pattern into contiguous sequences of 1- to N-character sub-patterns. We create two sets of sub-patterns for the same pattern set using N=3 and N=4. They are denoted as DSN3 and DSN4, respectively, and are handled exclusively without sharing. Please note that this splitting of patterns into sub-patterns is not connected to the grouping of three consecutive characters for calculating the summation m-tuples as explained earlier. . Fig 5.a and Fig. 5.b show the breaking of patterns into sub-patterns for N=3 and N=4, respectively. Identifying the position of sub-patterns in patterns is crucial to our algorithm.

Offset :	**1**	**2**	**3**	**4**	**5**
Pattern 1:	exe	cut	ema		
Pattern 2:	use	rna	met	ool	ong
Pattern 3:	Bad	com	man	d	
Pattern 4:	Pas	swo	rds		
Pattern 5:	com	man	dlo	ng	
Pattern 6:	cod	ewo	rds		
Pattern 7:	lwa	r	e.	exe	

(a)

Offset :	**1**	**2**	**3**	**4**	**5**
Pattern 1:	exec	ute	ma		
Pattern 2:	user	name	tool	ong	
Pattern 3:	Badc	omma	nd		
Pattern 4:	Pass	word	s		
Pattern 5:	comm	andl	ong		
Pattern 6:	code	word	s		
Pattern 7:	lwar	e.ex	e		

(b)

Fig. 5. (a) DSN3: The set of seven patterns from Fig. 1 separated into sub-patterns for N = 3; (b) DSN4: The set of seven patterns separated into sub-patterns for N = 4

Once all of the patterns have been separated into their sub-patterns, we store all distinct N-character sub-patterns into a table called GRP(N). If a sub-pattern appears multiple times, then only one position is reserved for this sub-pattern in GRP(N). Similarly, we create tables GRP(i), for i = 1, ..., N-1, where GRP(i) stores all of the i-character sub-patterns that appear in the patterns. We denote all of the GRP(i)'s, for i = 1, ..., N, collectively as GRP. A table may be empty if there is no sub-pattern of the corresponding length. Note that we could divide the patterns into sub-patterns of any number of characters from 1 to N, however we try to break the patterns such that we minimize the number of sub-patterns per pattern. This is not a rigid rule as we use exceptions when dealing with collisions. We discuss about such exceptions in Section 4.2. We create a *bit vector* (BV) and an *end vector* (EV) for every

sub-pattern in GRP. BV shows the position of the sub-pattern in all the patterns that contain it, excluding their tail. That is, if a particular sub-pattern appears only in the sub-pattern positions 2 and 4 of the same or two different patterns, then its BV will contain "010100….0". Multiple appearances of a sub-pattern in the same position of multiple patterns are registered only once in its BV. The EV vectors store information about the tails of patterns. If a sub-pattern forms the tail of a pattern, then it will contain '1' in the respective position of its EV vector.

For example, if two patterns exclusively end with a common sub-pattern in sub-pattern positions 2 and 3, respectively, then the EV vector of this sub-pattern will be "01100….0". BV is L bits long and EV is L+1 bits long, where L+1 is the maximum number of sub-patterns in a pattern. This is because EV will always store "1" for the tail. As we will see later, using the two sub-pattern sets DSN3 and DSN4 to derive their BV and EV vectors, we can detect the possible presence of a pattern match. This along with a pattern summation tuple match is then used to confirm a pattern match. Although we need only three bits for BV and four bits for EV in the case of DSN4, we have used L=4 for both DSN3 and DSN4 for flexibility in breaking a pattern into non-tail sub-patterns of less than N characters (sub-patterns at offsets 2 and 3 in pattern 7 of Fig. 5.a). This approach helps us to place sub-pattern records in exclusive memory locations as discussed later in Section 4.2 for the purpose of eliminating sub-pattern collisions. The BV's and EV's for the sub-patterns in DSN3 and DSN4 are shown in Fig. 6.a and Fig. 6.b., respectively.

GRP(3) TABLE

SP	BV	EV
exe	1000	00010
cut	0100	00000
ema	0000	00100
.	.	.
.	.	.
.	.	.
ewo	0100	00000

GRP(2) TABLE

SP	BV	EV
ng	0000	00010
e.	0010	00000

GRP(1) TABLE

SP	BV	EV
d	0000	00010
r	0100	00000

(a)

GRP(4) TABLE

SP	BV	EV
exec	1000	00000
lwar	1000	00000
.	.	.
.	.	.
.	.	.
code	1000	00000

GRP(3) TABLE

SP	BV	EV
ong	0000	00110
ute	0100	00000

GRP(2) TABLE

SP	BV	EV
nd	0000	00100
ma	0000	00100

GRP(1) TABLE

SP	BV	EV
s	0000	00100
e	0000	00100

(b)

Fig. 6. GRP tables for the patterns in Fig. 1, assuming (a) N=3 and L=4; (b) N=4 and L=4.

3.2. Run-time Pattern Detection

We have stored the summation m-tuples in the TBRAMs as explained above. We have generated the BV's and EV's. We also have the weight m-tuples for every character stored in the character tables. We will now look at the pattern detection unit. The detection unit is made up of the *Summation Block*, *Bit Detection Units*, *Pattern Match Unit and O_Pattern Match Unit*.

a) Summation Block: There are Max_Fragment_Length individual accumulation units, the same as the maximum length in characters of a pattern (i.e., fragment of an O_Pattern). For our example with Max_Fragment_Length =16 we have sixteen accumulation units, ACC1 to ACC16, as shown in Fig. 7, which receive the weight m-tuple as input from the character table for each arriving character and generate the summation m-tuples for the 16 possible patterns corresponding to the most recent character

arrivals. ACC1 always creates the summation m-tuple for one character while ACC2 creates the summation m-tuple for two characters, and so on. These sixteen accumulated values are then forwarded to the pattern match unit in parallel for every character input.

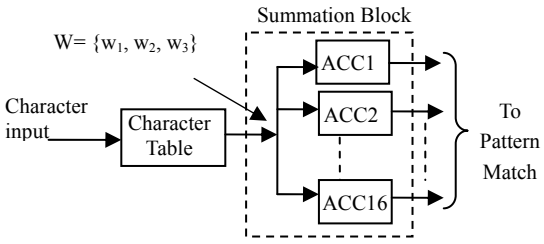


Fig. 7. Summation Block.

For an arbitrary complete stream "abcdefghij" at a cycle t, of ten consecutively arriving characters, where "a" is the first character, the accumulation units generate summation m-tuples in the following manner:

$ACC1_t$= W("j");
$ACC2_t$ = W("i") + 2*W("j") = $ACC1_{t-1}$ +2* W("j");
$ACC3_t$ =W("h") + 2*W("i") + 4*W("j") =$ACC2_{t-1}$ + 4*W("j");
$ACC4_t$ = $ACC3_{t-1}$ + W("j");
$ACC5_t$ = $ACC4_{t-1}$ + 2*W("j");
.
.
$ACC10_t$ = $ACC9_{t-1}$ +W("j");
$ACC11_t$, $ACC12_t$,…, $ACC16_t$ =0;

In the next clock cycle suppose a character "x" is inputted then the accumulations will now have the following result

$ACC1_{t+1}$= W("x") ;
$ACC2_{t+1}$ = $ACC1_t$ +2* W("x") ;
.
.
$ACC10_{t+1}$ = $ACC9_t$ +W("x")
$ACC11_{t+1}$ = $ACC10_t$ + 2*W("x")
$ACC12_{t+1}$,…, $ACC16_t$ =0;

b) Bit Detection Units: The input stream of characters arrive at the bit detection units one at a time in parallel. We have two bit detection units for the DSN3 and DSN4 generated vectors. We will denote the two detection units as BDN3 and BDN4, respectively. BDN3 has the GRP(1), GRP(2) and GRP(3) tables generated using N=3 (Fig. 6.a) whereas BDN4 has GRP(1), GRP(2), GRP(3) and GRP(4) tables generated using N=4 (Fig 6.b.). Every record in a GRP table contains the sub-pattern itself and the corresponding BV and EV vectors. Every bit detection unit has a shift register (window) of N characters that interfaces the input stream. Each cycle samples 1 to i consecutive characters in this window, where i is the total number of available characters (i=N for a full window); all possible sub-pattern matches are attempted against the N GRP tables. The input is hashed to generate addresses in the GRP tables, except for the GRP(1) table which can be addressed directly using the ASCII character. Thus, there are N-1 *hashing blocks* HB(i), for i = 2 to N, in both of the bit detection units, as shown in Fig. 8. The hashing implementation is simple and made up of XOR and ADD operations on the incoming input. The method we use to eliminate sub-pattern collisions in hashing is explained in Section 4.2. On a sub-pattern match, the respective BV and EV are forwarded from the GRP table to the AND-SHIFT-OR unit; otherwise a zero-vector ("000…0") is forwarded. The AND-SHIFT-OR unit has its own bit vectors, namely, *detection vector DV* and *end detection vector EDV*. The (L+1)-bit DV vector keeps track

of individual sub-pattern matches. The (L+1)-bit EDV vector detects pattern tails. The MSB of DV is originally set to '1' (in fact it is always '1') whereas the remaining bits are initialized to '0'. There are N different DV and EDV vectors which take care of a pattern starting at any of the N different offsets in the input stream of characters. These N vectors represent N different phases that repeat in a cyclic manner for every character input.

Pattern detection involves simple SHIFT, AND, OR and COMPARE operations on these binary vectors. The SHIFT operation is where the DV bit vector is right shifted (represented by >>) by one with a '1' entering into the MSB (equivalent to ORing of "1000…0"). In addition to DV and EDV, there is one *position bit vector PV* per DV that keeps track of the character offset in a partial pattern match. The bit detection units, BDN3 and BDN4, output two more *position bit vectors PVN3* and *PVN4*, respectively that keep track of a complete pattern match. PVs indicate the character offsets of patterns corresponding to a '1' in DV while PVN4 and PVN3 indicate the character offset of the pattern corresponding to a '1' in EDV. PV contains Max_Fragment_Length bits. The position of a '1' in PVN3 or PVN4 indicates the length of the pattern matched by the BDN3 or BDN4, respectively. Thus, an AND operation on these vectors indicates the length of the common, final pattern match which still needs to be verified by checking the accumulated sum in the TBRAMs.

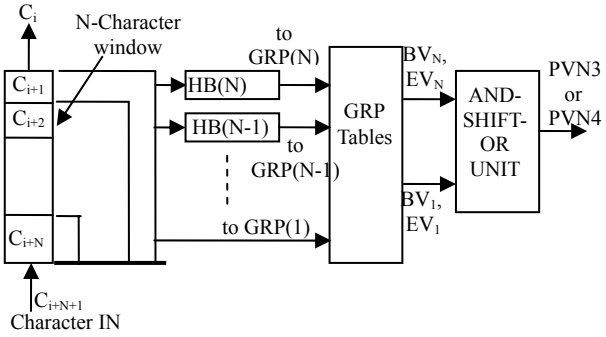


Fig. 8. Bit Detection Unit for N=4, 3.

Bit Detection Unit for N=4 (BDN4): For detection in BDN4 which has BV and EV generated using N= 4, there are four PV, DV and EDV vectors. The four DV vectors reflect on partial pattern matches, EDVs reflect on pattern tail matches and PVs reflect on the length of a pattern match (pseudocode to calculate all these bit vectors is shown in the Appendix). The DVs and EDVs are calculated using the following formulas (Note: to make BV and DV of equal length, a '0' is appended as the least significant bit of BV before performing all the AND and OR operations):

DV_1 = "100...0" OR ((DV_2 AND BV_3) OR (DV_3 AND BV_2) OR (DV_4 AND BV_1) OR (DV_1 AND BV_4)) >> 1); for offset (modulo) N = 1; i.e., offset=1, 5,…,etc
EDV1= ((DV_2 AND EV_3) OR (DV_3 AND EV_2) OR (DV_4 AND EV_1) OR (DV_1 AND EV_4)); for offset (modulo) N = 1; i.e., offset =1,5,…,etc

DV_2 = "100...0" OR ((DV_3 AND BV_3) OR (DV_4 AND BV_2) OR (DV_1 AND BV_1) OR (DV_2 AND BV_4)) >> 1); offset (modulo) N = 2; i.e., offset= 2, 6, 10,…, etc
EDV_2 = ((DV_3 AND EV_3) OR (DV_4 AND EV_2) OR (DV_1 AND EV_1) OR (DV_2 AND EV_4)); offset (modulo) N = 2; i.e., offset= 2, 6, 10,…, etc

DV_3 = "100...0" OR ((DV_4 AND BV_3) OR (DV_1 AND BV_2) OR (DV_2 AND BV_1) OR (DV_3 AND BV_4)) >> 1); offset (modulo) N = 3; i.e., offset=3, 7, 11, …, etc
EDV_3 = ((DV_4 AND EV_3) OR (DV_1 AND EV_2) OR (DV_2 AND EV_1) OR (DV_3 AND EV_4)); offset (modulo) N = 3; i.e., offset=3, 7, 11, …, etc

DV_4 = "100...0" OR (((DV_1 AND BV_3) OR (DV_2 AND BV_2) OR (DV_3 AND BV_1) OR (DV_4 AND

BV_4)) >> 1); offset (modulo) N = 1; i.e., offset= 4, 8, 12,…, etc
EDV_4 = ((DV_1 AND EV_3) OR (DV_2 AND EV_2) OR (DV_3 AND EV_1) OR (DV_4 AND EV_4)); offset (mod) N = 1; i.e., offset= 4, 8, 12,…, etc.
Offset in these formulas represents the position of a character in the input. At any offset only one DV-EDV-PV set is active. That is, we store the result of all the AND-OR operations in one DV-EDV-PV set, suppose DV_1-EDV_1-PV_1; then for the next character input DV_2-EDV_2-PV_2 will be active. This continues in a cyclic manner. This unit sends PVN4 to the pattern match unit (see the pseudocode in the Appendix for calculating PVN4). PVN4 contains '1' in the position representing that of the pattern being found if the active EDV is non-zero; otherwise, it is zero.

Bit Detection Unit for N=3 (BDN3): BDN3 is identical to BDN4 except that there are three PVs, DVs and EDVs since N = 3; hence it has three phases. It also forwards its own character offset pointer PVN3 to the pattern match unit.

c) Pattern Match Unit: This unit takes in the ACC inputs from the summation block and also the PV inputs from BDN3 and BDN4. If the EDVs from both the bit detection units are non-zero while having a common offset pointer (i.e., a non-zero bit in the same position of PVN4 and PVN3), then BDN3 and BDN4 have detected a possible pattern match of identical length starting at the same position. The pattern match unit forwards the appropriate summation m-tuple (in the above example it will forward ACC9 containing the summation m-tuple of pattern "Passwords") outputted by the summation block to the FIFO queues at the input of the TBRAM block.

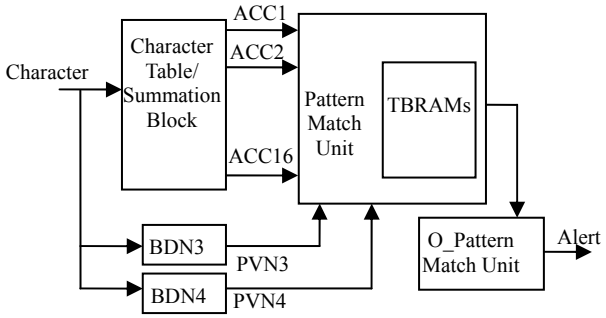


Fig. 9. Block diagram of complete pattern detection system.

Due to the pre-processing of the patterns, the PVN4 AND PVN3 operation will not generate a non-zero output every clock cycle. There will be a minimum gap of five clock cycles. No more than two bits of PV will ever be non-zeros, out of which only one can be because of a true pattern. The summation m-tuple is hashed and the output is used as an address for the TBRAM which is looked up to check if the summation m-tuple matches the pre-stored one at that location. We also look into the collision TBRAMs to see if there is a match. The veracity of a match is confirmed if there is a final match. The address of the TBRAM for a match, if found, is then forwarded to the O_Pattern match unit. The block diagram of our complete system is shown in Fig. 9.

d) O_Pattern Match Unit: This unit contains the FRAM block and uses appropriate delay cycles to join appropriately, matches of patterns that constitute an O_Pattern. The FRAM block is addressed by hashing the matched pattern address output of the TBRAMs. If the start_fragbit value of a pattern matched in TBRAM is '1' and no_fragbit is '0' , then this is the first fragment of a longer O_pattern, and thus the FRAM block is looked up to find out the remaining fragments of the O_pattern. The FRAM block consists of two RAMs (FRAM1 and FRAM2). FRAM1 stores address pointers to the locations in FRAM2 and the respective TBRAM addresses of the first fragment of fragmented O_patterns. FRAM2 stores the subsequent fragments' TBRAM addresses. If there is a match in FRAM1, then we access FRAM2 to fetch the subsequent fragments using the pointer to FRAM2. Using appropriate delay cycles we connect all the fragments in the O_Pattern match unit to match the longer

pattern. The data structure used for FRAMs is shown in Fig. 10.

The first node of a pattern is stored in FRAM1 and the others are stored in FRAM2. We set the maximum number of nodes which can have the same prefix fragment to four. If the number is more than four, then we move the final pattern concatenation process to software. We disconnect the link and make the "start_fragbit = 1" and "no_fragbit = 1" for that fragment in TBRAM which means that the joining of fragments for that O_Pattern is done at the higher layer i.e. the software in the host. Although in the SNORT database there exist quite a few O_Patterns with common prefixes, we do not come across in our experiments with common prefix fragments because of our choice of Max_Fragment_Length=24; this choice makes fewer O_Patterns to be fragmented since more than 80% of the O_Patterns have lengths less than or equal to 24 characters. Also, O_Patterns having common prefixes and containing more than 24 characters have different lengths and hence our rule of dividing the O_Pattern into fragments of almost equal lengths causes the later to contain different patterns (O_Pattern 5 in Fig. 10). Thus we do not normally have two or more O_Patterns with the same pattern prefix in FRAM2. If the O_Pattern is not fragmented ("start_fragbit = 0" and "no_fragbit = 1"), then the O_Pattern match unit forwards the TBRAM address directly to the higher software layer indicating a match.

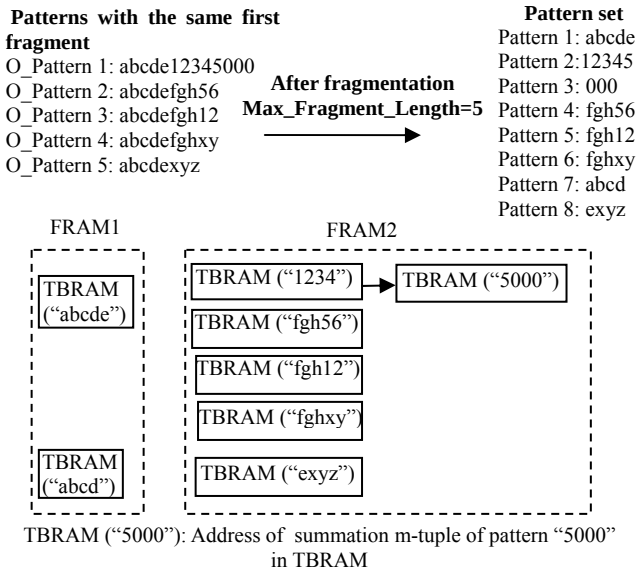


Fig. 10. FRAM Block data structure.

4. Eliminating Pattern Collisions and False Positives

4.1. Eliminating Pattern Collisions

A collision in the pattern RAM will show up if multiple patterns hash to the same location in the TBRAMs. As explained before, we keep a collision_TBRAM to take care of collisions. The number of collisions allowed is set at five. But if the list in the Collision_TBRAM is full with four summation m-tuples, then we fragment the pattern differently. Fig. 11 shows an example using pattern 3 and pattern 4 of Fig.1. Now if there is another pattern "abcdefghij" which has to be added at runtime and hashes to location 4 in TBRAM1, while the linear list is full with four records in it, then we break up the new pattern as per our convenience into appropriate fragments that remove this collision.

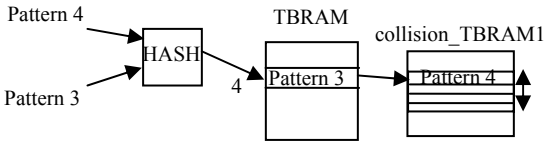


Fig. 11. Pattern collisions in TBRAM.

Some of the choices for fragmentation are ("abcdef" and "ghij", "abc" and "defghij", "abcd" and "efghij", and "abc", "defg" and"hij"). Note that this process also fragments the patterns. In Fig.1 our criterion of fragmenting an O_Pattern into patterns was the length. But now we further fragment patterns of the new pattern set (i.e., Fig. 1.b) to avoid collisions. Once this fragmentation is complete, we have new summation m-tuples for these newly formed patterns which are stored appropriately. These cases are rare since we can achieve an optimal placement of patterns at static time using appropriate values for the weight tuples; such a case can only be encountered for the addition of new patterns at run time.

4.2. Hashing and Eliminating Collisions for Sub-patterns

We use plain hash functions containing XOR and ADD operations to place the sub-patterns in the GRP RAMs. For the sake of efficiency, we use separate hash functions and RAMs for different GRP tables. There is no real need for hashing with GRP(1) due to the uniqueness of single characters that requires 2^8 (i.e., 256) distinct locations. Our hash functions apply simple operators to the input to generate an address; they do not need separate key inputs. We hash the character window containing N characters separately using N characters for GRP(N), N-1 characters for GRP(N-1) and so on as shown before. The important requirement for our design is that the GRP RAMs should output BV and EV ever clock cycle. Hence we should avoid any collisions in the GRP RAMs. We use a maximum of four hashing functions in a hash block HB(i), for i=2, …,N (this number varies depending on the size of the GRP RAM needed; we also want to take advantage of the Xilinx Block RAMs that come in 18 Kbit chunks). The outputs of the hash functions access the corresponding RAMs in the GRP(i) RAMs, for i= 2,…, N-1. Fig. 12 shows a hashing block for a GRP(3) RAM in bit detection unit BDN3. The four RAMs in the GRP(3) RAM take care of collisions.

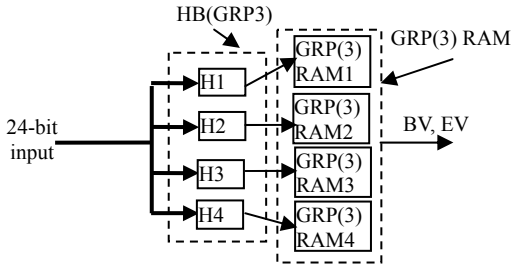


Fig. 12. Hashing Block of the GRP3 RAM in BDN3.

Since we have different length sub-patterns, we also have the additional flexibility of splitting the pattern into sub-patterns in such a way that avoids collisions. For example, if the arbitrary pattern "abcdef" has to be added, and the pattern split is "abc"-"def" and "abc" cannot be placed into any of the four GRP(3) RAMs because of collision (i.e., all four RAMs have no vacancy in the hashed location of "abc") then we try to change the way the pattern is split. For example, we can split it as {"a"-"bc"-"def"} or {"ab"-"c"-"def"} or {"a"-"bcd"-"ef"} or in some other way such that we can place the sub-pattern records in non-vacant locations in the GRP RAMs.

4.3. Eliminating False Positives for Patterns

Once the whole process of weight distribution and BV-EV generation for DSN3 and DSN4 is done, we check for pattern-related false positives with our method. If found at static time, we then take appropriate action to eliminate them. False positives will be possible only if there exists a fictitious pattern for which both EDVs (in BDN3 and BDN4) are non-zero, the AND operation of their position vectors (PVN3 and PVN4) produces non-zero resultant vector, and also the summation m-tuples are identical to a true pattern. Also, the length of the two patterns should fall into the same group; i.e., patterns of different lengths which are placed together and their summation m-tuples are stored in the same TBRAM. In Fig. 13.a, we show fictitious patterns which will generate non-zero EDVs. We can deduce from the Fig. 13.a that a fictitious pattern with non-zero EDV can be generated only if the first offset of a pattern in one set (suppose DSN4) has identical characters to the first offset and partial part or a whole part of the second offset in another set(suppose DSN3).

For example, the first pattern in Fig.1 for N=4 has “exec” at the first offset and the first pattern in BDN3 has “exe” at first offset. Now we have to search for a pattern (other than first one) in BDN3 which has a sub-pattern that starts with character “c” at the second offset. We find that pattern 3 has “com” at the second offset. Thus, if a string like “execommand” comes in, then we will get a non-zero EDV in BDN3 and BDN4, with identical PVs. We then have to make sure that the sum generated by such a fictitious pattern is not the same as any of the true patterns in TBRAM that stores the patterns of 10 characters (as “execommand”). This is possible with careful assignment of m-tuple weights. We can also counter this by fragmenting the pattern or by breaking the pattern into sub-patterns differently. Let us suppose that the first fictitious pattern generates a false positive and is difficult to avoid such a situation even by changing the weight m-tuples of the characters. We can easily avoid such a situation by fragmenting the original pattern 1 into two separate pattern fragments “exec” and “utema” of length 4 and 5, respectively. This is then broken into sub-patterns as shown in Fig 13. b. We can now see that the above given fictitious pattern cannot be generated with the new pattern set. Fragmenting the patterns again looks similar to the approach in Section 4.2; however, here we target the different problem of false positives. For the SNORT database, our sub-pattern creations for N=4 and N=3 along with the uniqueness of the summation m-tuples avoid false positives. This approach is only to add new patterns.

5. Results and Comparisons with Eatlier Work

5.1. Pre-processing and Simulation Results

All the patterns in the available SNORT rule set (version v2.8, July 29th, 2009) were chosen for analysis to prove the viability of our proposed pattern matching design. This version of SNORT has 6455 distinct patterns; the longest pattern contains 213 characters and the median length is 12 characters. We did some analysis to selecting Max_Fragment_Length. We can easily infer that the LUT usage in the target FPGA will increase with an increase in Max_Fragment_Length since the number of ACC units will increase. However as we increase Max_Fragment_Length the number of patterns that are fragmented will decrease since we can now provide pattern matching for a longer fragment.

Fig. 14 shows pre-processing results for various fragment lengths. The LUT usage for an implementation will almost remain constant irrespective of the number of patterns we add. Thus, we are more concerned with BRAM usage for higher compression. We have developed a method to select the value of Max_Fragment_Length, as it is unique to our implementation. Our objective is to fit the design in a single FPGA while also providing high utilization of resources towards yielding high performance. A value decrease produces shorter bit vectors, which require smaller computational blocks (i.e., comparators and units for logic operations). However, this decrease yields more fragments and consequently more summation m-tuples. Thus, a tradeoff is necessary. As discussed earlier, our analysis shows that 80% of the SNORT patterns contain up to 24 characters. As observed in Fig. 14, increasing the value of Max_Fragment _Length from 16 to 24 reduces considerably the memory consumption (1411 fewer fragments, resulting in substantially reduced summation m-tuples stored in BRAMs). In contrast, the change is much smaller as the value of Max_Fragment_Length is increased from 24 to 32. Also, there is a rather small difference in logic cell usage for the implementations where Max_Fragment _Length is

16 or 24. For Max_Fragment_Length=16, the design consumes 4733 flip-flops and 5132 LUTs. These numbers for Max_Fragment _Length=24 are quite similar, being 5162 flip-flops and 5569 LUTs. The pre-processing job on these patterns was carried out off-line using a C-program script. The script identifies the unique character sub-patterns, creates their corresponding sub-pattern records and assigns unique weight m-tuple to every ASCII character. It then calculates the summation m-tuple for every pattern. We use grouping of three consecutive characters for the summation. The difference between using three-character grouping and any other higher number of characters is that the summation m-tuple value will be higher in the latter case thereby requiring more bits per summation m-tuple. However, three-character grouping is sufficient to produce exclusive summation m-tuples in our case.

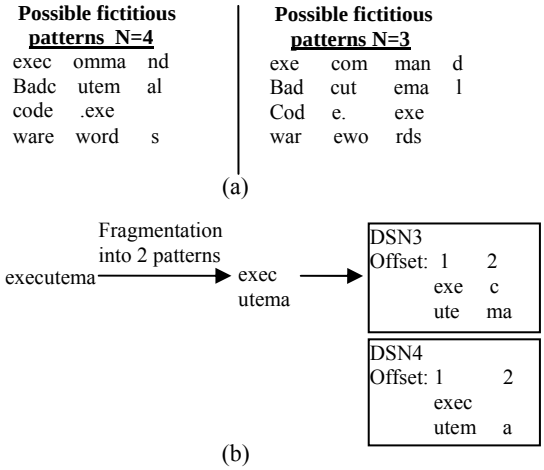
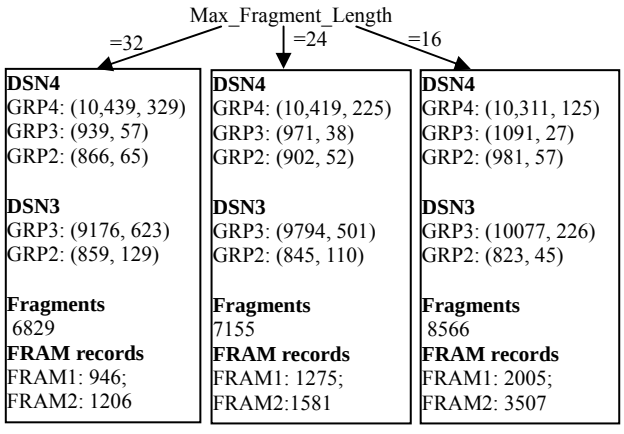


Fig. 13. (a) Fictitious patterns which generate non-zero EDV;
(b) Fictitious pattern prevention with fragmentation.



Note: The numbers in brackets show **(number of records, distinct BV and EV pointers)**

Fig. 14. Pre-processing for Max_Fragment_Length= 32, 24 and 16.

If two patterns belonging to the same TBRAM group have the same summation m-tuple, then we change the weight m-tuple value of the character and recalculate the summation m-tuples. If after a fixed number of attempts a solution is not obtained, we fragment the pattern and repeat the check. Once the check is done successfully, we generate pattern addresses such that we can place the pattern with no more than four collisions. These records are also kept in an off-line database to facilitate efficiency in future updates involving new patterns. To add new patterns, the available database information is com-

pared with the sub-patterns extracted from the new patterns. For each newly extracted sub-pattern that already exists in the database, its newly generated bit vectors are bitwise ORed with those of its identical sub-pattern in the database; the results are stored in the on-chip RAM as well as they are modified in the database. If a newly extracted sub-pattern is not present in the database, then the new sub-pattern along with its bit vectors and other relevant information are stored in the GRP table and the pattern RAM. The script could be run by the system administrator on the console. Complete pre-processing of the 6455 SNORT patterns on the host, that interfaces the FPGA board, consumes just 114 msec; the host has an Intel Pentium 4 processor running at 3 GHz, and having 1024KB total cache and 1GB RAM.

To test our design for future pattern additions, experiments were carried out in two parts. In Part I, we generated the sub-patterns, their respective vectors and the pattern addresses for 5834 patterns from the SNORT rule set. These patterns contained a total of 96,977 characters. In Part II, once the former GRP records were loaded into the on-chip RAMs and the design operated under normal working conditions, we enabled a modification of the already loaded set of patterns by adding the remaining set of 621 patterns. Information extracted for Parts I and II of our experiments is shown in Table I.

Table I. Pre-processing results for adding O_Patterns using Max_Fragment_Length = 24

(Number of)		**Part I**	**Part II**	**Total**
O_Patterns		5834	621	6455
Characters		96,977	8786	105,763
DSN4	GRP(4) records	9486	933	10419
	GRP(3) records	803	168	971
	GRP(2) records	776	126	902
	GRP(1) records	126	1	127
DSN3	GRP(3) records	9125	669	9794
	GRP(2) records	707	138	845
	GRP(1) records	170	2	172
Number of Fragments FRAM1		1217	58	1275
Number of Fragments FRAM2		1520	61	1581

5.2. VHDL System Synthesis/Implementation and Rule Insertion/Deletion

The synthesis and simulation of our design worked flawlessly. We implemented the design with Max_Fragment_Length 24 characters. We will discuss the parameters for the design with 24 characters for Max_Fragment_Length. The length of BV for BDN3 and BDN4 was set to 8 bits and EV was set to 9 bits. Also, our off-line experiments for weight assignments to m-tuples revealed that unique summation m-tuples could be carried out with m=3 and bw = 3 bits which will also give a unique summation m-tuple to patterns. While calculating summation we group three characters at a time. With the maximum value of 7 per weight tuple, a group can have a maximum value of 49 per group (7 +2*7 + 4*7). With a maximum of 8 groups possible, the largest possible summation weight requires 9 bits. For the 10,419 GRP(4) records, we deduced that there were only 225 distinct BV-EV combinations. Hence, we moved the BV-EV combination into a separate smaller RAM with 256 locations. Thus, instead of storing an 8-bit BV and 9-bit EV for every record, we stored only an 8-bit pointer per record which points

to this BV-EV combination and results in considerable memory savings. The same was done for other records of BDN4 and BDN3. Fig. 15 shows the chosen parameter values for system synthesis. For GRP(1), we do not need a BV and EV pointer since BV and EV are stored directly in the record. The maximum number of collisions allowed in TBRAM0 is set at 3 while in other TBRAMs it is set to 5.

We also grouped patterns of varying lengths into a single TBRAM in such a way that they are equally distributed in the TBRAMs. Fig. 15 shows the different TBRAMs and the grouping of patterns of different lengths placed in them. For pattern of lengths 1 to 3 characters and most of the 4-character patterns we use the GRP tables. Since they are already stored in one of these tables using a single bit field in the GRP RAM can tell us whether the GRP record is also a pattern of interest. Hence we do not need a separate TBRAMS for them. We used VHDL to program the architecture. BRAMs were used to store the GRP records and the summation triplets of the patterns. BRAMs were also used to weight triplets and FRAM records. It's a pipelined design with a latency of 21 clock cycles. The bit detection units have a latency of 11 clock cycles. The pattern match unit has a maximum latency of 5 clock cycles and the O_Pattern match unit has a maximum latency of 5 clock cycles.

DSN4:	DSN3
GRP(4) RAM: 14336 location RAM GRP(3) RAM: 1536 location RAM GRP(2) RAM: 1536 location RAM GRP(1) RAM: 256 location RAM record: (sub-pattern, BV-EVpointer) GRP(4): (32 bits, 8 bits) GRP(3): (24 bits, 6 bits) GRP(2): (16 bits, 6 bits) GRP(1): (BV,EV): (8 bits, 9 bits)	GRP(3) RAM: 12288 location RAM GRP(2) RAM: 1024 location RAM GRP(1) RAM: 256 location RAM record: (sub-pattern, BV-EVpointer) GRP(3): (24 bits, 10 bits) GRP(2): (16 bits, 8 bits) GRP(1): (BV,EV): (8 bits, 9 bits)
	FRAM1: four 512-location RAMs FRAM2: 2048 location RAM

TBRAM 0: patterns of lengths 4, 5, 6, 7, 8 and 9; total of 1639 patterns
TBRAM 1: patterns of lengths 10, 11, 12, 13 and 14; total of 1714 patterns
TBRAM 2: patterns of lengths 15, 16, 17 and 18; total of 1735 patterns
TBRAM 3: patterns of lengths 19, 20, 21, 22, 23 and 24; total of 1506 patterns

Fig. 15. Parameter values for system synthesis.

The hardware synthesis was done using Synplify Pro 9.1 as well as Xilinx ISE, with the parameters shown in Fig. 15. Our design was implemented on a Xilinx Virtex-II Pro XC2VP70 FPGA. For Max_Fragment_Length=24, it employs 102 18-Kbit BRAMs (Block RAMs), 5162 Flip Flops and 5569 LUTs, and operates at 300.1 MHz. A random pattern generator also interleaves patterns from the SNORT database. The design was tested in three phases. The first phase involved simulation of the VHDL code. The second phase focused on the post-synthesis output of the Xilinx synthesis and Synplify Pro tools. The third phase of testing involved the post-place and route output generated by the Xilinx Place and Route tools.

GRP(i) record(i = 2, 3, 4):
Word 1: Sub-pattern, BV-EV pointer
Word 2: Bit Vector, End Vector
GRP(1) record:
Word 1: Bit Vector, End Vector

Pattern Record:
Word 1: TBRAM record
Word 2: FRAM1 record (If pattern is fragmented)
Word 3: FRAM2 record

Fig. 16. Parameters needed to add a pattern.

The process to insert a new pattern is as follows. The Amirix Systems PCI-based FPGA board used in our implementation has a 64-bit data bus. Due to the dual-ported BRAMs in our design, and the

fact that reading and writing are independent of each other, BRAM updates can proceed while packets are being processed. We group various record fields for the 64-bit bus. The different words needed to add a GRP record are shown in Fig. 16. A 64-bit word can be loaded into the BRAM in one clock cycle. Weight tuples for all the 256 byte-character patterns, once assigned during the initial phase when the database is loaded, are not tampered with. We do all the placement of the patterns and fragmentation based on these weight values. New patterns will not be available in matching until the pattern RAM is updated (after all the involved sub-pattern records are updated). Hence, we add the sub-patterns and BV-EV pointers first followed by the BV-EV values, FRAM records, if any, and then the summation tuples. When a new pattern is added we only need to place the new GRP records, if any, or edit the old ones and also place the pattern sums and FRAM records if the pattern is a fragmented one.It takes up to two clock cycles to add as well as update a sub-pattern record in the GRP(4), GRP(3) or GRP(2) RAM. It takes up to one clock cycle to add as well as update a sub-pattern record in the GRP(1) RAM. Similarly, it takes one clock cycle to add the pattern summation tuples. Now, if an O_Pattern is fragmented, then we need to store the information of the pattern fragments (pattern addresses in the TBRAM as shown in Fig. 10) in the FRAMs. This will be equal to twice the number of fragments in terms of clock cycles (one for the summation m-tuple of the fragment and the other to store its address information in FRAM). The addition of the 621 new patterns takes 14.855 μsec. The details are shown in Fig. 17. In contrast, the work presented in [20] requires around 12,600 clock cycles, or 64 microseconds, to insert a reduced number of 381 patterns.

To remove a pattern, the process is as follows. The respective entry in the pattern RAM is first invalidated in a single clock cycle. Its constituent sub-pattern records are then accessed subsequently. For every sub-pattern, a two-dimensional linked list is kept on the host; the first dimension contains its BV bits whereas the second dimension contains the pointers to the patterns that contain it in the corresponding bit offset.

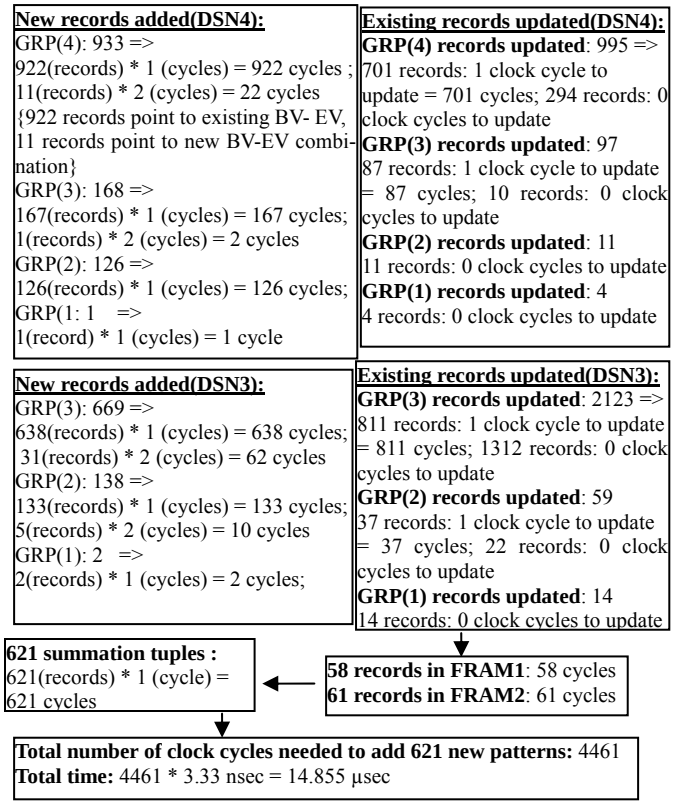


Fig. 17. Total time to add 621 new patterns.

Fig. 18 shows the list for sub-pattern "abc" (from DSN3) in a hypothetical set of patterns. This figure shows that "abc" is present in position 1 of pattern 1, and position 2 of patterns 2 and 13. Now assume the deletion of pattern 13 that contains this sub-pattern. Since "abc" is also present at the same offset in pattern 2, its BV will not be changed. However, to delete pattern 1, after the summation tuples for pattern 1 are invalidated in pattern RAM, we then delete the node for "abc" in position 1 (as shown in fig. 18. b). The other sub-patterns are removed in the same manner in subsequent clock cycles. To modify a pattern, we delete the old pattern and then add the modified pattern. The flexibility of updating or changing a pattern in the database without re-calculating hashing keys works to our advantage as compared to the approach in [20].

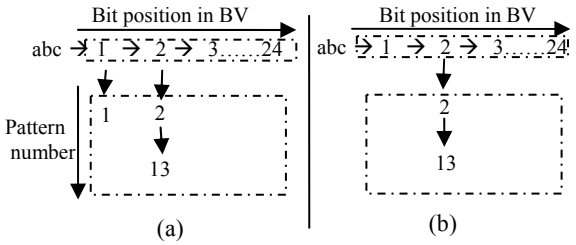


Fig. 18. Linked list for updates with sub-pattern "abc".

5.3. Comparison with Earlier Approaches

Table II shows a comparison with the most prominent efforts in the area of pattern matching with FPGAs or ASICs. The first three designs force complete reprogramming of the FPGA to load new malicious patterns and hence do not employ BRAM. The results assume an input channel of eight bits (i.e., the incoming rate is one character per clock cycle), thus providing a common platform for comparison. Our design is the most comprehensive so far as it employs the largest freely available SNORT database (as of July 29th, 2009).

The approach in [1] uses on-chip memory only for Bloom filter table realization. It stores all the patterns in slow off-chip RAM of several Megabytes capacity. Since Cho's [2] is an ASIC implementation, it has a large clock frequency at the cost of rigidity to updates. Also, another ASIC solution in [18] involves memory tiles where a 2-bit input selects one of four finite state machines. Although [18] does not list the number of patterns in the implementation, it contains a comparison with the design proposed in [11] that assumes 1466 rules with 18,031 characters. The work in [18] uses 3200 Kbits of memory, yielding a memory consumption ratio of 181.7 bits per character which is quite high compared to our design (see Table II). Also, our updating process is very simple as it does not require intricate knowledge of our design. An off-line script just simply creates sub-pattern records and pattern addresses. We can easily conclude that our design provides very substantial memory compression (i.e., in terms of stored bits per input character) compared to other methods that also facilitate runtime pattern updates. It also operates at a substantially high frequency and requires by far the least logic cell usage per character, while also yielding very high throughput. Finally, our analysis is comprehensive as it involved a larger number of SNORT rules than earlier approaches.

6. Conclusions

We presented a novel design for pattern matching with FPGAs that can be utilized by NID systems. This approach can also be exploited by other pattern-matching oriented applications, such as the detection of virus signatures. It is a memory-oriented, high-throughput, compression-based design that incorporates a simple pattern detection technique. It differs substantially from earlier approaches since it does not require long-distance routing of information inside the processing chip. Another major advantage of our approach is that it supports runtime updates for the set of stored patterns without a need to reprogram the FPGA. This is a necessity for NID systems operating at a 24/7 schedule as the database of stored malicious patterns may require frequent updates. Our evaluation was comprehensive,

involving a larger number of rules than earlier approaches.

Table II. Comparison with other designs (N/A: not available or not applicable).

Design, Year	FPGA Device	Patterns	Characters	MHz	Throughput (Gbps)	BRAM Memory (Kbits)	Logic cells/ Character	BRAM bits/ character
Baker [6], 2004 (no new rules)	Virtex-II Pro 100	361	8263	250	1.790	0	0.35	0
Sourdis [11], 2004 (no new rules)	Virtex-II 3000	1466	18,031	335	2.680	0	0.97	0
Clark [9], 2004 (no new rules)	Virtex 8000	1512	17,537	253	2.024	0	1.7	0
Gokhale [10], 2002	Virtex E 1000	N/A	640	N/A	2.180	24	15.19	37.5
Cho [2], 2005	ASIC	2107	22,340	893	7.144	864	0.5	38.6
Lockwood [1], 2006	Virtex-4	2259	N/A	250	1.96	94	N/A	N/A
Pnevmatikatos [3], 2006	Virtex-II Pro XC2VP30	2187	33,613	306	2.448	702	0.06	21.4
Our method, 2009 (Max_Fragment_Length=24)	Virtex-II Pro XC2VP70	6455	105,763	300.1	2.408	1836	0.052	17.77

REFERENCES

[1] S. Dharmapurikar and J. Lockwood, "Fast and Scalable Pattern Matching for Network Intrusion Detection Systems," *IEEE Journal Selected Areas Comm.,* Vol. 24, Oct. 2006, pp. 1781–1792.

[2] Y. Cho and W. Mangione-Smith, "A Pattern Matching Co-processor for Network Security," *Annual ACM/IEEE Design Automation Conference*, 2005.

[3] D. Pnevmatikatos and A. Arelakis, "Variable-Length Hashing for Exact Pattern Matching," *International Conference on Field Programmable Logic and Application*, Aug. 2006, pp. 1-6.

[4] R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching using FPGAs," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[5] Z. Baker and V.K. Prasanna, "Automatic Synthesis of Efficient Intrusion Detection systems on FPGAs," *14th Internation-al conference on Field Programmable Logic and Applications*, 2004.

[6] Z. Baker and V.K. Prasanna, "A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs," *12th IEEE Symposium Field-Program. Custom Computing Machines,* 2004.

[7] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *IEEE Symp. Field-Programmable Custom Computing Machines*, 2002.

[8] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System," *International Conference on Field Programmable Logic and Applications*, Lisbon, Portugal, Sept. 2003.
[9] C.R. Clark and D.E. Schimmel, "Scalable Parallel Pattern-Matching on High-Speed Networks," *IEEE Symp. Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004.
[10] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," *12th Conference Field Programmable Logic and Applications*, Montpellier, France, Sept. 2002, pp. 404–413.
[11] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-speed NIDS Pattern Matching," *12th Annual IEEE Symposium on Field Programmable Custom Computing Machines*, 2004, pp. 258–267.
[12] F. Yu, R. H. Katz, and T.V. Lakshman, "Gigabit Rate Packet Pattern Matching using TCAM," *12th IEEE International Conference on Network Protocols*, 2004, pp. 174–183.
[13] C. Lin, C. Huang, C. Jiang, and S. Chang, "Optimization of Pattern Matching Circuits for Regular Expression on FPGA," *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 15, Dec. 2007.
[14] SNORT® Open Source Network Intrusion Prevention and Detection System, http://www.snort.org
[15] X. Wang and S.G. Ziavras, "Performance Optimization of an FPGA-based Configurable Multiprocessor for Matrix Operations," *IEEE International Conference on Field-Programmable Technology*, Tokyo, Japan, Dec. 15-17, 2003.
[16] X. Xu and S.G. Ziavras, "A Coarse-grain Hierarchical Technique for 2-dimensional FFT on Configurable Parallel Computers," *IEICE Transactions on Information and Systems*, Vol. E89-D, No. 2, Feb. 2006, pp. 639-646.
[17] S.G. Ziavras, A. Gerbessiotis, and R. Bafna, "Coprocessor Design to Support MPI Primitives in Configurable Multiprocessors," *Integration, the VLSI Journal*, Vol. 40, No. 3, 2007, pp. 235-252.
[18] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *32nd Annual International Symposium on Computer Architecture*, June 2005, pp. 112–122.
[19] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," *12th USENIX Security Symposium*, 2003, Vol. 12.
[20] T.N. Thinh, S. Kittitornkun, and S. Tomiyama, "Applying Cuckoo Hashing for FPGA-based Pattern Matching in NIDS/NIPS," *International Conference on Field-Programmable Technology*, Dec. 2007, pp. 121-128.
[21] R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, 20(10), 1977, pp. 761–772.
[22] M. Fisk and G. Varghese, "Applying Fast String Matching to Intrusion Detection" *Technical Report in preparation, successor to UCSD TR*, CS2001-0670, Univ. of California, San Diego, 2001.
[23] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," *23rd Conference of the IEEE Communications Society (Infocomm)*, March 2004.
[24] A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, Vol. 18, No. 6, 1975, pp. 333-340.

APPENDIX

Calculation of PVN4, PVN3, PV, DV, EDV

- Assume Bit Detection Units BDN4 and BDN3 with N= 4 and N=3, respectively.
- DV and EDV are (L+1)-bit vectors.
- tempDV and tempEDV are temporary vectors of L+1 bits.
- tempPV, PVB and PVE are temporary vectors of Max_Fragment_Length bits
- One of the N-vector combinations of DV-EDV-PV is active in every clock cycle. It is the i-th vector in the following code for the i-th iteration of the FOR_i loop..
- The following loop is executed in every clock cycle.

```
FOR_i: for i= 1 to N loop
{
   tempDV= "000…0"; tempEDV= "000…0";
 // initialize vectors to 0
   PVB= "000…0"; PVE= "000…0";   tempPV= "000…0"
  //The following for loop takes care of the AND-OR operations in the DV and EDV equations
  FOR_N: for k=1 to N loop
   {
      if (k ≥ i)
//this if loop selects the appropriate BV and EV to be ANDed  with DVk
     X= N-k+i;
// X variable stores the appropriate subscript of BV, EV to be ANDed with DVk
    else
     X= i-k;
    end if;
    tempDV = tempDV OR (DVk AND BVX);
```

```
 //tempDV stores temporary value of active DV
     tempEDV=tempEDV OR (DV_k AND EV_X);
// tempEDV stores temporary value of active EDV
// the following program statements are used to update the length of the partially matched pattern due to the presence of sub-patterns.
      tempPV= "000…0";            // reset the temporary vector
    if (DV_k(0) AND BV_X(0)) ≠ 0
// this means that a possible first sub-pattern of a pattern is detected; length of the sub-pattern is X characters.
        tempPV(X) = '1';   // make that bit '1';
      else
        tempPV = "000..0";
      end if;

PVB= PVB OR tempPV;
 // assign the length of first sub-pattern to PVB.
tempPV= "000…0";                  // reset the temporary vector
// BV is an L-bit vector used to search for sub-patterns except tails;
//Now we look for the sub-patterns other than the first using the BV output

    FOR_DL: for v in 1 to L-1 loop
   {
      if (DV_k(v) AND BV_X(v)) ≠ 0
      {
// The following loop is used to move the offset position of the partial matched pattern by same number of bits as the matched sub-pattern length to the right, thus increasing the length of the partially matched pattern by X bits. The appropriate PV_k is shifted. The sub pattern is found if the above DV AND BV operation is non-zero.
        FOR_DV:  for m in v to N*v+N-1 loop
        {
             tempPV= "000…0"; // reset tempPV
             if (m+v< Max_Fragment_Length)
                tempPV(m+X) = PV_k(m);
             end if;
             PVB= PVB OR tempPV;
// store the calculation result in PVB
        }
        end for FOR_DV;
     }
     end if;
  }
 end for FOR_DL;

tempPV= "000…0";
// The calculations below are the same as the ones above except that we now get the length of a complete pattern match instead of partial matches; search for tails and the vector is moved by (length of the tail) bits. The resultant '1' in the PVE vector indicates the length of the complete pattern.
 if DV_k(0) AND EV_X(0) neq 0
      tempPV(X) = 1;
    else
      tempPV = "000..0";
 end if;
 PVE= PVE OR tempPV;
 FOR_EL: for v in 1 to L loop
 {
     if (DV_k(v) AND EV_X(v)) ≠ 0
     {
FOR_EV: for m in v to N*v+N-1 loop
              {
                 tempPV= "000…0";
                 if (m+v< Max_Fragment_Length)
                     tempPV(m+X) = PV_k(m);
                 end if;
                PVE= PVE OR tempPV;
              }
             end for FOR_EV;
```

```
            }
            end if;
    }
    end for FOR_EL;
}
end for FOR_N;

PV_i  = PVB;
//Assign the PVB which contains the length of the partial pattern matched to the active PV
DV_i= "100…0" OR (tempDV >>1);
// Perform shift and OR operations and assign the temporary vectortempDV to the active DV; Similarly, assign tempEDV to the active EDV
EDV_i= tempEDV;
// for BDN4 N=4 and hence the length of the matched complete pattern is given by PVN4 while in BDN3 it is given by BDN3. Hence, assign PVE to PVN4 for N=4 and to PVN3 for N=3, respectively.

PVN3=PVE; // In BDN3
PVN4 = PVE; // In BDN4
```