Novel FPGA-Based Signature Matching for Deep Packet Inspection

Nitesh B. Guinde and Sotirios G. Ziavras

Electrical & 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. Thus, only FPGA (Field-Programmable Gate Array) or ASIC (Application-Specific Integrated Circuit) solutions could be efficient for this problem. Our FPGA-based solution performs high-speed matching while permitting pattern updates without resource reconfiguration. An off-line optimization method first finds sub-pattern similarities across all the patterns in the SNORT database of signatures [17]. A novel technique then compresses each pattern into a bit vector where each bit represents such a sub-pattern. Our approach reduces drastically the required on-chip storage as well as the complexity of matching, utilizing just 0.05 logic cells for processing and 17.74 bits for storage per character in the current SNORT database of 6456 patterns.

1 Introduction

Many computer network attacks in recent times were difficult to detect based only on header inspection. Deep packet inspection of the payload is needed to detect application-level attacks. The rules for matching may represent either new signatures or changes to existing ones. From October 2007 to August 2008, 1348 SNORT rules were added while 8170 rules were updated (on a daily or weekly basis) [17]. Thus, robust NID systems should handle rule updates (including additions) without taking them off-line. Signature/pattern matching is also relevant to virus detection based on the presence of specific command sequences in a program [21]; 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. Although their database of rules can be updated easily, the pattern matching process cannot keep up with fast network speeds. FPGA-based solutions, on the other hand, have the potential to match network speeds; often at the cost of complete system reconfiguration for pattern updates. Complete system synthesis can take several hours; also, the penalty for full FPGA reconfiguration can be many milliseconds or seconds [10]. Therefore, complete system reconfiguration is not prudent for 24/7 active networks.

P. Samarati et al. (Eds.): WISTP 2010, LNCS 6033, pp. 261–276, 2010.
© IFIP International Federation for Information Processing 2010

Common FPGA-based NID approaches aim to minimize the consumed area, increase the speed and rarely reduce the reconfiguration time for updates. The majority embed specialized state machines where states represent sequences of input characters; state transition information is stored in a location pointed to by the next incoming character (see [5] and [8]). A few papers (see [1], [2], [3] and [22]) discuss 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 supporting runtime pattern updates without FPGA reconfiguration. In related work (see [6] and [7]), graph-theoretic techniques partition the rule set into groups based on common character sequences; they reduce redundant searches across patterns and the consumed area. Similarly, our pattern preprocessing step first extracts fixed-length sub-patterns. To compress the stored information further, we create a bit vector for each sub-pattern to denote its location in the set of malicious patterns. Thus, a single bit now represents an entire sub-pattern. Also, our approach ultimately condenses 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. The terms table and RAM are used interchangeably.

2 Related Work

The capabilities of FPGAs have recently improved tremendously. Sidhu et al. [5] proposed a straightforward algorithm to construct non-deterministic finite automata (NFA) representing given regular expressions. Hutchings et al. [8] implemented a module to extract patterns from the SNORT rule set [17] and then generated their regular expressions for NFA realization. Lin et al. applied minimization to regular expressions for resource sharing [16]. To reduce transfer widths, an 8-bit character decoder provides 256 unique outputs; various designs (see [6], [7], [8], [9] and [10]) were implemented. Since these designs hard-code the patterns into the FPGA, runtime updates are forbidden without complete FPGA reconfiguration. Content-addressable memories (CAMs) that support updates were proposed by Gokhale et al. [12]. Sourdis et al. [14] applied pre-decoding with CAM-based pattern matching to reduce the consumed area. Yu et al.[15] used ternary content-addressable memory (TCAM) for pattern matching. However, CAM approaches require large amounts of on-chip memory and have high power consumption and hence are unfavorable choices for large rule sets.

The lookup mechanism in Dharmapurikar et al. [1] employs a hash table and several Bloom filters for a set of fixed-length strings to be searched in parallel. It may produce false positives and also accesses a slow off-chip memory after a Bloom filter match. CRC functions in Pnevmatikatos et al. (see [3] and [19]) reduce logic cells and memory space. Patterns are first decomposed into varying-length fragments for storage. Input fragments are hashed and appropriate delay logic combines fragment matches into complete pattern matches. Thinh et al. applied Cuckoo hashing in pattern matching [22] with varying-length sub-patterns

while supporting runtime updates. It yields a good compression in terms of bits and logic cells per character. However, if a collision shows up while inserting a pattern, Cuckoo attempts to recalculate the hashing key. When recalculations are maxed out (i.e., a key cannot be generated for distinct placement), rehashing is needed for all the keys (including for already stored sub-patterns). This process may then suffer from unpredictable penalties.

A pattern matching co-processor [2] 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 fewer logic resources with smaller memory consumption per character than these designs. Another advantage of our design is that the pattern matching module does not need to increase in size with an increase in the number of malicious patterns.

3 Our Method

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 not produce false positives. Without loss of generality, we will test the implementation with the complete set of signatures in the SNORT database [17]. In summary, our method breaks up statically each pattern into fixed-length sub-patterns and then encodes the position of each sub-pattern in all of the encompassing patterns into a common bit vector. '1' in this vector represents the presence of the sub-pattern in the respective position of a pattern while '0' denotes otherwise. For each new sub-pattern match in the input, a '1' bit is stored into a detection vector. Bit-wise AND -SHIFT operations on this vector move the '1' with every new sub-pattern match. Another bit vector shows the position of each sub-pattern as a tail in one or more patterns. If a new sub-pattern match at the respective position can potentially represent the end of a pattern, then a hardware-based verification process is invoked to confirm the veracity of a complete pattern match. The entire process is described in the following sub-sections.

3.1 Pre-processing

Our static-time preprocessing divides each pattern into contiguous sequences of N-character sub-patterns; the only exception may be the sub-pattern in the tail of a pattern that may instead include from one to N-1 characters (if the number of characters in the pattern is not a multiple of N). We fix N before the separation process. Our analysis in Section 5 for the SNORT database shows that the best choice is N=3. Identifying the position of sub-patterns in patterns is crucial to our algorithm. 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). Similarly, we create tables GRP(i), for $i = 1, ..., N-1$, where GRP(i) stores all of the i-character sub-patterns that appear as tails in patterns. We denote all of the GRP(i)'s, for $i = 1, ..., N$, collectively as GRP. Let

L be the maximum sub-pattern offset for a given pattern set. We create a *bit vector*(BV) and an *end vector* (EV) for every sub-pattern in GRP. BV is (L-1) bits long and shows the position of the sub-pattern in all the patterns, except the tail, that contain it. 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". The EVs are L bits long and store information about pattern tails. If a sub-pattern forms the tail of a pattern, then it will contain '1' in the respective position of its EV vector. Members of GRP(i), for i= 1,, N-1, appear only as tails and hence require only an EV without the need for a BV. Every record is assigned a unique m-tuple of weights represented by vector $W = \{weight_1, weight_2, ..., weight_m\}$; let *bw* be the bits per weight. Assume the set of six patterns in Fig. 1 and their sub-pattern separation for $N = 3$. Fig. 2 shows the GRP tables created for these patterns assuming $N = 3, m = 3$ and $b_w = 8$. We can infer from Fig. 1 that $L = 6$. An m-tuple of weights is then calculated for each stored pattern by summing up weight-wise the m-tuples of its contained sub-patterns. The result is stored in a *pattern table* at the address denoted by the *pattern address*. These summation m-tuples of sub-patterns and patterns will eventually help the sub-pattern and pattern detection processes. The *baseaddress* field of a sub-pattern in GRP contains valid information only

Offset :	**1**	**2**	**3**	**4**	**5**	**6**
Pattern 1:	exe	cut	ema	lwa	re.	exe
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			

Fig. 1. A set of six patterns separated into sub-patterns for N = 3

GRP(3) TABLE

SP	BV	EV	W1	W2	W3	baseaddress	hash
exe	10000	000001	3	21	45	0	0
cut	01000	000000	79	101	17	0	0
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
rds	001000	001000	66	34	200	0	1

GRP(2) TABLE

SP	EV	W1	W2	W3	baseaddress	hash
ng	000100	44	6	9	3	0

GRP(1) TABLE

SP	EV	W1	W2	W3	baseaddress	hash
d	000100	193	182	2	0	0

Fig. 2. GRP tables for the patterns in Fig. 2, assuming N=3, L=6, m=3 and bw = 8

if it appears as a tail. Its value is added to the sub-pattern offset to generate a pattern address pointing to a location in the pattern table that contains weight summation m-tuples. Fig. 3 shows the summation m-tuples (i.e., triplets since $m = 3$) for the patterns in Fig. 1; their components are represented by Sum_1, Sum_2 *and* Sum_3. It also shows the address of the pattern summation tuples in the pattern RAM. Sub-pattern "ng" appears at offset 4 in the tail of pattern 5. Address 4 in the pattern table is already occupied by pattern 3 and the next available location has address 7. Hence, the baseaddress of "ng" is set to 3 (since $3 + 4 = 7$).

Pattern 1 sum: (3, 21, 45) + (79, 101, 17) + (19, 57, 211) + (61, 88, 121) + (11, 7, 1) + (3, 21, 45) = (176, 295, 440)

Pattern Num	Address in pattern RAM (baseaddress of tail + tail offset)	Sum1	Sum2	Sum3
1	0 + 6 = 6	176	295	440
2	0 + 5 = 5	207	519	603
3	0 + 4 = 4	264	427	203
4	0 + 3 = 3	278	135	305
5	3 + 4 = 7	135	130	139
Pattern Num	**Address in collision RAM**	**Sum1**	**Sum2**	**Sum3**
6	2	119	138	298

Fig. 3. The pattern table for the patterns in Fig. 1

If two or more patterns have different tails at the same offset, then the baseaddress and offset fields of their tail sub-patterns receive such values that their summation points to distinct/available locations in the pattern table. To minimize the size of this table, a modulo Z operation is used when adding fields, where Z is the size of the pattern table (Z=16 in this example). However, if two or more patterns have the same tail sub-pattern at the same offset, then a collision will result. To remove collisions, a smaller collision RAM is used in addition to the pattern RAM. Patterns 4 and 6 have a common tail "rds" at the same offset, thus the collision RAM is used to place pattern 6 as shown in Fig. 3. The *hash* field in the GRP table is used to separate the placement of pattern summations. The collision RAM is addressed by hashing the summation tuples and hash field is used to select the appropriate summation tuples as inputs to the hashing function. Since no two patterns generate identical summation m-tuples, this clause is used to select the appropriate order of tuples as inputs to the hashing function. In the worst case, we could use a pattern splitting method to resolve collisions (explained later in Section 3.4).

3.2 Runtime Detection of Malicious Patterns

A malicious pattern could start at any character offset in the input stream. Up to N characters at a time are investigated for known sub-patterns stored

in the GRPs. A shift register (window) of N characters interfaces the input stream. Each cycle samples 1 to n consecutive characters in this window, where n is the total number of available characters (n=N for a full window); sub-pattern matches are attempted against the N GRP tables. On a sub-pattern match, the respective sub-pattern record is forwarded from the GRP table to a detection unit; otherwise, zero is transmitted. N detection units can deal with the N possible character strings in this window. A sub-pattern record is made up of BV, EV, m-tuple Weights, baseaddress and the hash field.

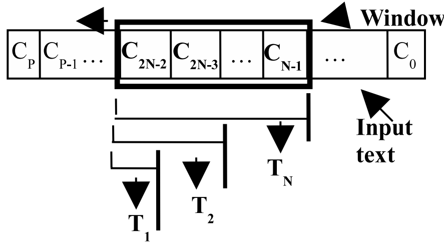

Fig. 4. Processing a P-character input with an N-character shifting window

Let C_1, C_2, ..., C_P be an input stream of P characters, as in Fig. 4. The characters enter the window from the left. The window is divided each time into N sub-windows containing from 1 to N characters; they are denoted by T_n, where n=1, ..., N. For example, if a full window contains characters C_{N-1} to C_{2N-2}, then sub-window T_N will contain characters C_{N-1} to C_{2N-2}, sub-window T_{N-1} will contain characters C_N to C_{2N-2}, and so on. Every sub-window's content is then looked up in the GRP tables for a match. If a sub-pattern match is found, then its GRP record is read out and forwarded to the appropriate detection unit; otherwise, zero is forwarded. Thus, in every cycle each detection unit receives either a GRP record or zero.

The N sub-pattern detection units are denoted by d_k, for k =1, ...,N. Every detection unit contains a *detection vector DV*, an *end detection vector EDV* and an *m-tuple ACC* of accumulated weights. The L-bit DV vector keeps track of individual sub-pattern matches; its MSB is originally set to '1' whereas the remaining bits are initialized to '0'. The $offset_d$ field shows the position of the only '1' in DV. The L-bit EDV vector is initialized to zero and detects a tail match. ACC is initially set to all zeroes. Pattern detection involves simple SHIFT, AND, COMPARE and ACCUMULATE operations on the binary vectors and weight m-tuples arriving from GRP. Consider any detection unit d (d=d_1, d_2, ...,d_N). Let EV_{GRP}, BV_{GRP} , W_{GRP} and $baseaddress_{GRP}$ represent arriving values from GRP for a sub-pattern sampled from window n. The detection unit then performs the following operations ("&" and ">>" denote concatenation and shift, respectively):

$$
\begin{aligned}
EDV_d &= DV_d \;\; AND \;\; EV_{GRP} && ; \\
DV_d &= \text{`1'} \,\&\, (DV_d \;\; AND \;\; BV_{GRP} >> 1) && \text{;if n=N, then the record} \\
& && \text{;will have BV} \\
&= DV_d && \text{;otherwise} \\
ACC_d &= ACC_d + W_{GRP} && \text{;if (DV}_d \text{ AND BV}_{GRP}) \neq 0 \\
& && \text{;and n = N} \\
&= 0 && \text{;if (DV}_d \text{ AND BV}_{GRP}) = 0 \\
& && \text{;and n = N} \\
&= ACC_d && \text{;if n} \neq \text{N} \\
Temp_d &= ACC_d + W_{GRP} && \text{;if EDV}_d \neq 0 \\
&= 0 && \text{;otherwise} \\
pattern\ address &= baseaddress + offset_d + 1 && \text{;if EDV}_d \neq 0
\end{aligned}
$$

If (DV_d AND BV_{GRP}) is non-zero, then the m-tuple of the incoming sub-pattern record is added to the existing ACC m-tuple; otherwise, ACC is reset to zero. Also, the $offset_d$ field is incremented if (DV_d AND BV_{GRP}) is non-zero and the sub-pattern record source is GRP(N). If EDV_d is non-zero, it signifies the presence of a pattern, and hence the incoming m-tuple is added to ACC and the resulting m-tuple is stored in the $Temp_d$ temporary m-tuple. $Temp_d$ must be compared with the pattern summation m-tuple in the pattern table for a match. The baseaddress of the tail sub-pattern that produced a non-zero EDV is then added to $offset_d$ ('1' is also added to take care of the tail sub-pattern match offset), an address is generated and the summation m-tuples stored in that location are then compared against the values in $Temp_d$. A match denotes the presence of a malicious pattern. Pattern matching takes place in the pattern verification unit that contains the pattern table. The input source to the overall detection unit varies with the window cycle e.g., if at one instance the detection unit receives an input from GRP(2), then at the next cycle it will receive input from GRP(3), and so on, until GRP (N) is reached after which the input source will be set again to GRP(1). Collision pattern RAM is also searched simultaneously for the summation tuple match by hashing the summation m-tuples using the hash field from the record. The hash field is used to select the inputs for hashing.

3.3 Appropriate Weight Distribution Prevents False Positives

Assume a tail sub-pattern that appears at the same offset *off* in a random input pattern and a GRP-stored malicious pattern. Also, each sub-pattern at offset i in this input, for i=1, 2, ..., off, appears at the same offset position in the set of stored patterns. A non-zero EDV value will be generated for this input. If the $Temp_d$ result is identical to the malicious pattern's weight summation m-tuple (sum_1, sum_2, ...,sum_m), then a false positive will be produced (the final decision

is based on a comparison of m-tuples). Hence, it is imperative to assign unique sub-pattern weights that do not produce a malicious pattern's summation m-tuple when permuting stored sub-patterns while preserving their offsets in the respective malicious patterns.

Our weight assignment process guarantees the prevention of false positives. To simplify the process, we select heuristically the values for bw and m in such a way that their product is much greater than the total number of bits required to assign unique m-tuples to all the sub-patterns and eventually unique accumulation m-tuples to all the patterns. We found out that the majority of SNORT patterns, around 67%, have lengths less than or equal to fifteen characters; in fact, around 40% have lengths less than or equal to nine. We first used the pattern length to order them in descending order. The sub-patterns appearing in patterns longer than fifteen characters were assigned weights on the higher side in order to produce very high summation m-tuples for these patterns. The sub-patterns appearing in patterns of up to nine characters were assigned weight values on the lower side in order to produce low pattern summation weight tuples. The remaining sub-patterns were assigned weight values in the mid range. There are many common sub-patterns in these three pattern groups. If a sub-pattern appears in a longer pattern as well as short patterns, then it is given a larger weight. The next step eliminates the possibility for runtime false positives. The process is based on each pattern's summation m-tuple and the offset (say, *off*) of its tail sub-pattern (say, *sp*). First, all possible "fictitious" patterns containing off sub-patterns are created, where: (1) their tail is sp, and (2) a sub-pattern at offset i (for i=1, 2, ...,off-1) also appears at the same offset i in a stored pattern (including the current pattern). The summation m-tuples of these "fictitious" patterns are then produced. If any summation m-tuple matches the original pattern's summation m-tuple, then a false positive could show up at runtime; to eliminate this possibility, the sub-pattern weights are then modified or the pattern is split using the pattern splitting method (explained later). Our weight assignment procedure reduces the possibility that sub-patterns in shorter patterns can affect longer patterns, and vice versa. Since we choose sufficient bit widths for sub-pattern and summation tuples, the complexity of this process is reduced dramatically. Let us now show an example with tail "rds" from patterns 4 and 6 in Fig. 1 to illustrate how "fictitious" patterns are created and how summation m-tuples for these patterns are generated. All "fictitious" patterns for this tail are shown in Fig. 5, along with the produced summation m-tuples (i.e., triplets since m=3). The summation triplets stored in the pattern table for patterns 4 and 6 differ from these triplets (see Fig. 3), thus false positives cannot be generated by this tail. Such a calculation of summation tuples is carried out for every tail in the database to check out if a false positive is possible. This process is carried out offline at static time.

3.4 Pattern Splitting

Let us now look at the extremely rare case where a single pattern's all sub-patterns show up in the input at the wrong offsets while its tail is still present

sub-pattern at offset=1	sub-pattern at offset=2	sub-pattern at offset=3	sum1	sum2	sum3
exe	cut	rds	18	156	262
exe	rna	rds	124	88	266
exe	com	rds	70	57	248
.	.	rds	.	.	.
.	.	rds	.	.	.
cod	man	rds	.	.	.

The valid summation triplets for the "rds" tail are (119, 138, 298), (278, 135, 305) and represent patterns 6 and 4, respectively, in Fig. 2.

Fig. 5. All possible "fictitious" patterns producing non-zero EDVs for tail "rds"

at the correct tail offset. Also, these sub-patterns appear in identical-with-this-input offsets in other patterns. Our static-time process then deals with this case as shown in the following small example with three patterns:

(1) "abc" "def" "123"; (2) "ssh" "abc" "465"; (3) "def" "tra" "678";

If the incoming flow contains "def abc 123", a false positive will be triggered for a pattern 1 match. We eliminate this possibility by splitting pattern 1 into two smaller patterns "abc d" and "ef1 23". Appropriate weights are then assigned to the modified set:

(1) "abc" "d"; (2) "ssh" "abc" "465"; (3) "def" "tra" "678"; (4) "ef1" "23";

The final decision for detecting the original pattern 1, which is now a combination of the new patterns 1 and 4, is moved to higher layers i.e. in software on the host. SNORT does not contain such patterns. This method could also be applied while placing summation tuples in the pattern and collision RAMs for patterns that cannot be placed successfully using hash field.

4 Hardware Implementation

We assume N=3 (i.e., a window with up to three bytes or 24 bits). The block diagram of our implementation is shown in Fig. 6.

Hashing: We found out from our analysis of the current SNORT that GRP(N) records are predominant and require a bigger RAM compared to the other GRP(i), for i=1, 2, ..., N-1. 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 C_3 that requires 2^8 (i.e., 256) distinct locations. Our hash functions apply simple XOR-ADD operations to the input to generate an address; they do not need separate key inputs. We use three RAMs per GRP table which are addressed in parallel using different functions which are a mix of one level and two level hashing. The details of hashing are out of scope here. If a pattern contains a sub-pattern which cannot be placed in any non-vacant position of the RAM, then we again resort to the pattern splitting method explained above.

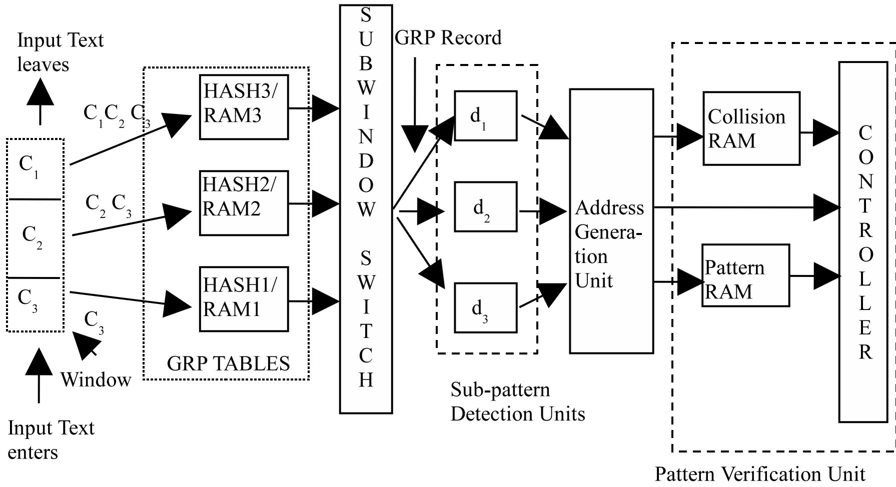

Fig. 6. Our architecture for N=3

Detection Unit: Detection is carried out using simple bitwise AND, SHIFT and ADD operations (i.e., accumulation operations) on bit vectors (DV, EDV) and weights (W). Our design reduces the problem complexity by applying compression to the data as it converts N consecutive characters (i.e., 8N bits) of a sub-pattern into a single bit in a vector representing a long pattern. In our current implementation, we represent any SNORT pattern with a 41-bit vector (the longest pattern in the current SNORT contains 122 characters whereas 3x41=123). Our design uses a simple pipelined structure where on every clock cycle bit vectors are used to potentially produce existing sub-pattern addresses and accumulated sums of weights. The bit vector of a sub-pattern coming from the sub-window switch block is bitwise ANDed with DV and then right shifted by one bit with a '1' entering from the left-hand side. DV is also ANDed with the incoming EV of the sub-pattern record and is stored in the 41-bit end detection vector (EDV). If the result of the AND operation between DV and BV is non-zero, then the weights associated with this record are accumulated into ACC. Otherwise, the accumulation registers are reset to zero. If EDV is non-zero, then the accumulated weights along with the pattern address are forwarded to the *controller block* (discussed below) to confirm the validity of a match. There are also four sets of offset_d counters, ACC m-tuples and Temp m-tuples in the offset count block to keep track of the position of up to four '1's in DV. One offset_d counter is initially enabled after being reset to the default value of 1. If the result of a (DV AND BV) operation is non-zero, the '1' in the MSB position of the result is shifted to the right and another '1' enters into the MSB from the left. This results in the first offset_d being incremented to 2 and the next offset_d being enabled after being reset to its default value of 1. Thus, the first offset_d counter keeps track of the '1' which is now in position two while the second

offset_d counter keeps track of the '1' currently in MSB. These counters are used to subvert special case attacks. Consider the four patterns:

(1) "abc" "123" "xyz" "klm" "8"; (2) "123" "xyz" "klm" "65";(3) "xyz" "klm" "ppp"; (4) "klm" "trs" "788" "23";

If we look closely, we can see that pattern 1 has commonalities with patterns 2, 3 and 4. However, they are not the same. For the input text "abc123xyzklmppp", we know that pattern 3 will be triggered. But we could also infer that at an instant there will be four '1' bits in DV at positions 1, 2, 3 and 4. These counters take care of such a scenario and the respective ACC and Temp m-tuples work correspondingly. It is highly unlikely to come across such a combination of patterns. We ran a script on SNORT and inferred that there could be at most two 1's in DV in a clock cycle. Thus, we need only two offset_d counters to subvert such attacks. We kept four counters for future updates in the SNORT signatures. Also, since we use drastically reduced logic in our implementation, adding more counters to the logic will not make a big difference to resource usage. We can always resort to our pattern splitting approach. Patterns longer than 123 characters can be broken up into smaller patterns for storage as explained for the case of false positives. For example, a pattern of 126 characters can be broken up into two patterns of length 123 and 3 characters, respectively, and we can then detect this pattern by using the condition that the first and second patterns should be matched consecutively by the same detection unit within three window shifts.

Address Generation unit: It employs a hash function and adders to generate the pattern address for the collision RAM by using the hash field and the summation tuples from the detection units. It also contains FIFOs to take care of non-zero EDVs from more than one detection units during the same clock cycle.

Controller: It gets the accumulated weight m-tuples from the sub-pattern detection blocks. It reads the respective values from the pattern RAM and compares them with the incoming values. If a match is found, then it informs the next layer.

5 Results and Comparison with Earlier Work

5.1 Pre-processing and Simulation Results

All the patterns in the available SNORT rule set (version v2.8, March 30th, 2009) were chosen for analysis to prove the viability of our proposed pattern matching design. This version of SNORT has 15,445 rules with 6456 distinct patterns; the longest pattern contains 122 characters and the median length is 12 characters. The pre-processing job on these patterns was carried out off-line using two C-program scripts. The first script identifies the unique character sub-patterns, and creates their corresponding sub-pattern records, hash keys, record addresses, and pattern addresses along with their summation m-tuples. This information is stored into the on-chip RAM. These records are also kept in an

off-line database to facilitate efficiency in future updates involving new patterns. To add new patterns, the second C-program script is run that differs from the first script only in that the available database information is compared 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 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 pattern RAM. These scripts could be run by the system administrator on the console. It is clear that a hardware implementation requires a fixed N. A trade-off is needed between the hardware complexity and the desired amount of on-chip data compression since L and the number of GRP records decrease with increases in N (hence, the memory consumption decreases). However, as we increase N the logic consumption increases since we need to include more detection units. We also need bigger switching fabric to forward the GRP vectors to the appropriate detection units in a cyclic manner. For a good choice of N, we studied the effects of N on the number of GRP records for SNORT. Since the longest pattern has 122 characters, we could easily obtain L by dividing 122 by N. The results are shown in Table 1.

Table 1. GRP table size when varying N for the SNORT database

N	L	GRP(5)	GRP(4)	GRP(3)	GRP(2)	GRP(1)	Total GRP records
3	41	-	-	10135	705	175	11015
4	31	-	11235	821	588	126	12770
5	25	11181	976	717	524	143	13541

Our hardware realization uses N=3 since it minimizes the number of GRP records and requires the least number of detection units. A choice of N<3 will not obviously have any benefits. The only drawback is the size of the bit vectors which will be 41 bits for the current set of SNORT rules. But this will be nullified by the width of sub-patterns to be stored in GRP.

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 weights and the pattern addresses for 6149 patterns from the SNORT rule set. In Part II, once the former GRP records were loaded into the on-chip RAMs and the design operated under normal conditions, we enabled a modification of the already loaded set of patterns by adding the remaining set of 307 patterns. Information extracted for Parts I and II of our experiments is shown in Table 2.

5.2 VHDL System Synthesis/Implementation

The synthesis and simulation of our design worked flawlessly for both parts of testing. We fixed the bit vectors to L=41 bits. Also, our off-line experiments for

Table 2. Pre-processing results for the SNORT database

Number of	Part I	PART II	Total
Patterns	6149	307	6456
Characters	100,800	4086	104,886
GRP(3) records	9842	293	10135
GRP(2) records	693	12	705
GRP(1) records	175	0	175

weight assignments to m-tuples revealed that unique summation tuples could be carried out with m=3 and bw=6 bits. Thus, the largest possible summation weight requires at least 12 bits per tuple (since $2^6 \times 41 < 2^{12}$). For the 10,135 GRP(3) records, we deduced that there were only 971 distinct BVs. Hence, we moved the BVs into a separate smaller RAM with 1024 locations. Thus, instead of storing a 40-bit BV for every record, we stored only a 10-bit pointer per record, which results in considerable memory savings. The same was done for EVs corresponding to only 137 distinct vectors, requiring an 8-bit pointer to a separate RAM. For GRP(2) records the total number of distinct EVs obtained was 142 which can be stored in a RAM of 256 locations. We used VHDL to program the architecture. BRAMs were used to store the GRP records and the summation triplets of the patterns. The hardware synthesis was done using Synplify Pro 9.1 as well as Xilinx ISE. Our implementation applies pipelining with a maximum delay of 31 clock cycles. The input arrives at the rate of one character per cycle. The input buffer can hold three bytes that are hashed to access the GRP RAMs. Our design was implemented on a Virtex II Pro (XC2VP70) FPGA. For bw=10, it employs 114 18-Kbit BRAMs (Block RAMs), 7538 Flip Flops and 6133 LUTs, and operates at 300.1 MHz. These numbers for bw=6 (our suggested implementation based on the results in Table 3), are 100 BRAMs, 6409 Flip Flops, 5321 LUTs and 300.3 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 involved the post-place and route output generated by the Xilinx Place and Route tools. 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. New patterns will not be available in matching until the pattern RAM is updated.

5.3 Comparison with Earlier Approaches

Table 3 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 patterns and hence do not employ BRAM. The results assume an input channel of eight bits, thus providing a common platform for comparison. Our design is the most comprehensive so far

as it employs the largest freely available SNORT database (of March 30th, 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. We can 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 updates. It also operates at a high frequency and requires the least logic cell usage per character, while also yielding very high throughput.

Table 3. Comparison with other designs (N/A: not available or not applicable)

Design, Year	FPGA Device	Patterns	Char-acters	MHz	Throu-ghput (Gbps)	BRAM Mem (Kbits)	Logic cells/ char	BRAM bits/ char
Baker[7], 2004	V2 Pro 100	361	8263	250	1.790	0	0.35	0
Sourdis[14], 2004	V2 3000	1466	18,031	335	2.680	0	0.97	0
Clark[10], 2004	Virtex 8000	1512	17,537	253	2.024	0	1.7	0
Gokhale[12], 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	V2 Pro XC2VP30	2187	33,613	306	2.448	702	0.06	21.4
Our method, 2008 (bw=6)	V2 Pro XC2VP70	6456	104,886	300.3	2.402	1818	0.050	17.74

6 Conclusions

We presented a novel design for pattern matching with FPGAs that can be utilized by NID systems. It is a memory-oriented, high-throughput 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 chip. Due to data compression, it yields further area savings related to

the required memory as well as the processing units. Another major advantage is that it supports runtime pattern updates without reconfiguring the FPGA.

Acknowledgements

Nitesh was supported in part by the NJIT Phonetel and Hashimoto Fellowships, and the NSF award 0924279.

References

1. Dharmapurikar, S., Lockwood, J.: Fast and Scalable Pattern Matching for Network Intrusion Detection Systems. IEEE Journal on Selected Areas in Comm. 24, 1781–1792 (2006)
2. Cho, Y., Mangione-Smith, W.: Pattern Matching Co-processor for Network Security. In: Annual ACM/IEEE Design Automation Conference (2005)
3. Pnevmatikatos, D., Arelakis, A.: Variable-Length Hashing for Exact Pattern Matching. In: International Conference on Field Programmable Logic and Application, pp. 1–6 (2006)
4. Wu, C., Wen, S., Huang, N., Kao, C.: A Pattern Matching Coprocessor for Deep and Large Signature Set in Network Security System. In: IEEE GlobeComm (2005)
5. Sidhu, R., Prasanna, V.K.: Fast Regular Expression Matching using FPGAs. In: IEEE Symposium on Field-Programmable Custom Computing Machines (2001)
6. Baker, Z., Prasanna, V.K.: Automatic Synthesis of Efficient Intrusion Detection systems on FPGAs. In: 14th International conference on Field Programmable Logic and Applications (2004)
7. Baker, Z., Prasanna, V.K.: A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs. In: 12th IEEE Symposium on Field-Programmable Custom Computing Machines (2004)
8. Hutchings, B.L., Franklin, R., Carver, D.: Assisting Network Intrusion Detection with Reconfigurable Hardware. In: IEEE Symp. Field-Programmable Custom Computing Machines (2002)
9. Sourdis, I., Pnevmatikatos, D.: Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System. In: International Conference on Field Programmable Logic and Applications, Lisbon, Portugal (2003)
10. Clark, C.R., Schimmel, D.E.: Scalable Parallel Pattern-Matching on High-Speed Networks. In: IEEE Symp. on Field-Programmable Custom Computing Machines, Napa Valley, CA (2004)
11. Cho, Y.H., Navab, S., Mangione-Smith, W.H.: Specialized Hardware for Deep Network Packet Filtering. In: 12th International Conference on Field Programmable Logic and Applications, Montpellier, France, pp. 452–461 (2002)
12. Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S., Hogsett, V.: Granidt: Towards Gigabit Rate Network Intrusion Detection Technology. In: 12th Conference on Field Programmable Logic and Applications, Montpellier, France, pp. 404–413 (2002)
13. Lockwood, J.W., Moscola, J., Kulig, M., Reddick, D., Brooks, T.: Internet Worm and Virus protection in Dynamically Reconfigurable Hardware. In: Military and Aerospace Programmable Logic Devices Conference, E10.M (2003)

14. Sourdis, I., Pnevmatikatos, D.: Pre-decoded CAMs for Efficient and High-speed NIDS Pattern Matching. In: 12th Annual IEEE Symposium on Field Programmable Custom Computing Machines, pp. 258–267 (2004)
15. Yu, F., Katz, R.H., Lakshman, T.V.: Gigabit Rate Packet Pattern Matching using TCAM. In: 12th IEEE International Conference on Network Protocols, pp. 174–183 (2004)
16. Lin, C., Huang, C., Jiang, C., Chang, S.: Optimization of Pattern Matching Circuits for Regular Expression on FPGA. IEEE Transactions on Very Large Scale Integration Systems 15 (2007)
17. SNORT® Open Source Network Intrusion Prevention and Detection System, http://www.snort.org
18. Roan, H., Hwang, W., Dan Lo, C.: Shift-Or Circuit for Efficient Network Intrusion Detection Pattern Matching. In: International Conference on Field Programmable Logic and Applications (2006)
19. Papadopoulos, G., Pnevmatikatos, D.: Hashing + Memory = Low Cost, Exact Pattern Matching. In: International Conference on Field Programmable Logic and Applications, pp. 39–44 (2005)
20. Tan, L., Sherwood, T.: A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In: 32nd Annual International Symposium on Computer Architecture, pp. 112–122 (2005)
21. Christodorescu, M., Jha, S.: Static Analysis of Executables to Detect Malicious Patterns. In: 12th USENIX Security Symposium, vol. 12 (2003)
22. Thinh, T.N., Kittitornkun, S., Tomiyama, S.: Applying Cuckoo Hashing for FPGA-based Pattern Matching in NIDS/NIPS. In: International Conference on Field-Programmable Technology, pp. 121–128 (2007)
23. Lee, T.: Hardware Architecture for High-Performance Regular Expression Matching. IEEE Transactions on Computers 58(7), 984–993 (2009)