ABSTRACT

HARDWARE SUPPORT FOR REAL-TIME NETWORK SECURITY AND PACKET CLASSIFICATION USING FIELD PROGRAMMABLE GATE ARRAYS

by
Nitesh Bhicu Guinde

Deep packet inspection and packet classification are the most computationally expensive operations in a Network Intrusion Detection (NID) system. Deep packet inspection involves content matching where the payload of the incoming packets is matched against a set of signatures in the database. Packet classification involves inspection of the packet header fields and is basically a multi-dimensional matching problem. Any matching in software is very slow in comparison to current network speeds. Also, both of these problems need a solution which is scalable and can work at high speeds. Due to the high complexity of these matching problems, only Field-Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC) platforms can facilitate efficient designs.

Two novel FPGA-based NID solutions were developed and implemented that not only carry out pattern matching at high speed but also allow changes to the set of stored patterns without resource/hardware reconfiguration; to their advantage, the solutions can easily be adopted by software or ASIC approaches as well. In both solutions, the proposed NID system can run while pattern updates occur. The designs can operate at 2.4 Gbps line rates, and have a memory consumption of around 17 bits per character and a logic cell usage of around 0.05 logic cells per character, which are the smallest compared to any other existing FPGA-based solution.

In addition to these solutions for pattern matching, a novel packet classification algorithm was developed and implemented on a FPGA. The method involves a two-field matching process at a time that then combines the constituent results to identify longer matches involving more header fields. The design can achieve a throughput larger than 9.72 Gbps and has an on-chip memory consumption of around 256Kbytes when dealing with more than 10,000 rules (without using external RAM). This memory consumption is the lowest among all the previously proposed FPGA-based designs for packet classification.

HARDWARE SUPPORT FOR REAL-TIME NETWORK SECURITY AND PACKET CLASSIFICATION USING FIELD PROGRAMMABLE GATE ARRAYS

by
Nitesh Bhicu Guinde

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Engineering

Department of Electrical and Computer Engineering

May 2010

APPROVAL PAGE

HARDWARE SUPPORT FOR REAL-TIME NETWORK SECURITY AND PACKET CLASSIFICATION USING FIELD PROGRAMMABLE GATE ARRAYS

Nitesh Bhicu Guinde

Dr. Sotirios Ziavras, Dissertation Advisor — Date
Professor of Electrical and Computer Engineering, NJIT

Dr. Roberto Rojas-Cessa, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Jie Hu, Committee Member — Date
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Alexandros Gerbessiotis, Committee Member — Date
Associate Professor of Computer Science Department, NJIT

BIOGRAPHICAL SKETCH

Author: Nitesh Bhicu Guinde

Degree: Doctor of Philosophy

Date: May 2010

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2010

- Master of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, December 2002

- Bachelor of Science in Electronics and Telecommunications Engineering,
  Goa Engineering College, Goa University, Goa, India, July 1999

Major: Computer Engineering

Presentations and Publications:

N. Guinde and S. G. Ziavras, "Novel FPGA-Based Signature Matching for Deep Packet Inspection," 4th Workshop in Information Security Theory and Practices: Security and Privacy of Pervasive Systems and Smart Devices, April 2010.

N. Guinde, X. Tang, R. Sutaria, S. G. Ziavras and C. N. Manikopoulos, "FPGA-based static analysis tool for detecting malicious binaries," 2nd IEEE International Conference on Computer and Automation Engineering, February 2010.

N. Guinde and S.G. Ziavras, "An Adaptable Platform for Network Intrusion Detection Systems," 8th New Jersey Universities Homeland Security Research Consortium Symposium, Princeton University, New Jersey, Dec. 5, 2008, poster presentation.

I dedicate this thesis to my parents, B. N. Guinde and Jyoti Guinde, and my lovely sister, Natasha. Without their support, patience, consideration and above all, love, this work would not have been possible.

ACKNOWLEDGMENT

I would like to thank my Professor Dr. Sotirios Ziavras for all the enthusiastic support and advice he has given me and most importantly for taking me under his wings after the sad demise of my previous professor, late Dr. Constantine Manikopoulos, who had generously extended his support towards me after my PhD qualifying exam. I would also like to thank my committee members: Dr. Roberto Rojas-Cessa, Dr. Edwin Hou, Dr. Jie Hu, and Dr. Alexandros Gerbessiotis.

I am very grateful to my cousin Dr. Meeta Naik and her husband Dr. Guruprasad Naik for the financial help they provided me with when needed the most, and also for being a support away from my home. I would also like to acknowledge my lovely and wonderful close-knit family back in Goa, including all my cousins, aunts and uncles. I would also like to extend my gratitude to my cousin, Shailesh Guinde, and also to Dilip, who are my real lifesavers. I would also like to thank all my friends and roommates here in US with whom I shared a special bonding.

Lastly, I feel myself privileged to have such wonderful parents and an awesome sister who have been very patient, understanding and really considerate during my doctorate-pursuing journey. They have been rock solid wall of emotional as well as financial support for me.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

FPGAs are characterized by low cost and short application development cycles. They also provide a right compromise between flexibility of re-programming and capability in operating at high speeds. Signature matching and packet classification are the most computationally expensive operations in NID systems that also demand very high processing speeds. In addition, these operations require new data uploads quite frequently. Thus, FPGAs present us with ideal platforms for these kinds of applications.

1.2 Objective

The objective of this dissertation is to provide a fast co-processor to the NIDS especially to perform the computationally intensive tasks of signature matching and header classification. Because of similarities of packet classification with header classification and since typically packet classification rules are significantly more than the header classification rules, the former is a more challenging problem. Hence it is implemented using a novel scheme on FPGAs.

This chapter first discusses the signature matching and then introduces the packet classification.

1.3 Introduction to Signature Matching

There have been many computer network attacks in recent times which were difficult to detect as the detection mechanism were based only on header inspection. Deep packet inspection of the payload is needed to detect application level attacks. 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 signatures. From October 2007 to August 2008, 1348 new SNORT rules were added while 8170 rules were updated (on a daily or weekly basis). Therefore, for the sake of security, decent NID systems should be able to handle rule updates (including additions) without taking them off-line. Signature matching is also relevant to virus detection where classic virus-detection techniques look for the presence of specific command sequences inside the program [24]. These command sequences basically form byte patterns. Thus, a pattern matching block becomes a critical component to subvert any virus attacks. The application of pattern matching part of this thesis is in NID systems but the approach could be extended to the virus detection area as well where new signatures are added almost daily.

There have been several works on pattern matching for deep packet inspection that attempt to identify malicious signatures. Most of the currently available deep packet inspection systems employ pattern matching software running on general-purpose processors. The Boyer-Moore [26] and Aho-Corasick [29] 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 up to 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. Fisk and Varghese [27] have presented a multi-pattern matching algorithm that combines the one-pass approach of Aho-Corasick with the skipping feature of Boyer-Moore. Tuck et al. [28] take a different approach to optimizing Aho-Corasick by incorporating bitmap and path compression to reduce storage. The advantage of software approaches is that the database of rules (i.e., the set of known malicious patterns) can be updated easily, if needed. 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 or seconds [10]. 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 [5, 8]. Only a few papers [1-3, 25] discuss flexible solutions that do not require FPGA reconfiguration when adding new patterns. The pattern matching solutions presented in this dissertation attempt 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 current approaches is included in this dissertation. A quantitative comparison with [25] appears later in Section 1.4. In other related work, Baker et al. [6, 7] 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.

This dissertation contains two pattern matching methods, fixed-length sub-pattern method and variable-length sub-pattern method. The ultimate objective for both the designed circuits 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.

1.4 Related Work (Signature Detection)

The terms table and RAM are used interchangeably in this dissertation. The capabilities of FPGAs have recently improved tremendously [19-21] so they are now frequently used by NID systems. 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 the regular expressions for resource sharing [16]. To reduce data transfer widths, an 8-bit character decoder provides 256 unique outputs; various designs [6, 7, 8, 9, 10] 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. [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. 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 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 by hardware. It may produce false positives and also accesses a slow off-chip memory after a Bloom filter match. The CRC functions in Pnevmatikatos et al. [3, 22] 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. Thinh et al. applied the Cuckoo hashing scheme in pattern matching [25]. Their design 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, the designs (both fixed-length and variable length) implemented in this dissertation have higher flexibility in resolving collisions faster. This will be explained in the later chapters.

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. The designs presented in this thesis employ a first-stage component similar to that in [2], where the hashing of fixed-length character streams can identify sub-patterns. However, both the designs utilize fewer logic resources and have smaller memory consumption per character in the SNORT database than all of aforementioned designs. Another major advantage in both the designs is that the pattern matching modules do not normally need to increase in size with an increase in the number of malicious patterns.

1.5 Introduction to Packet Classification

In general, Internet routers support packet forwarding of best-effort services in a first-come first serve basis, where the same amount of service is provided to any packet. However, the next-generation routers are required to provide different quality of services and supporting functions, such as admission control, resource reservation, per-flow queuing, and fair scheduling for different applications. Additional services also include Virtual Private Network (VPN) service, distributed firewalls, IP security gateways, traffic based billing, among others. The following example shows a case for differentiated services. Consider a small network shown in Figure 1.1.

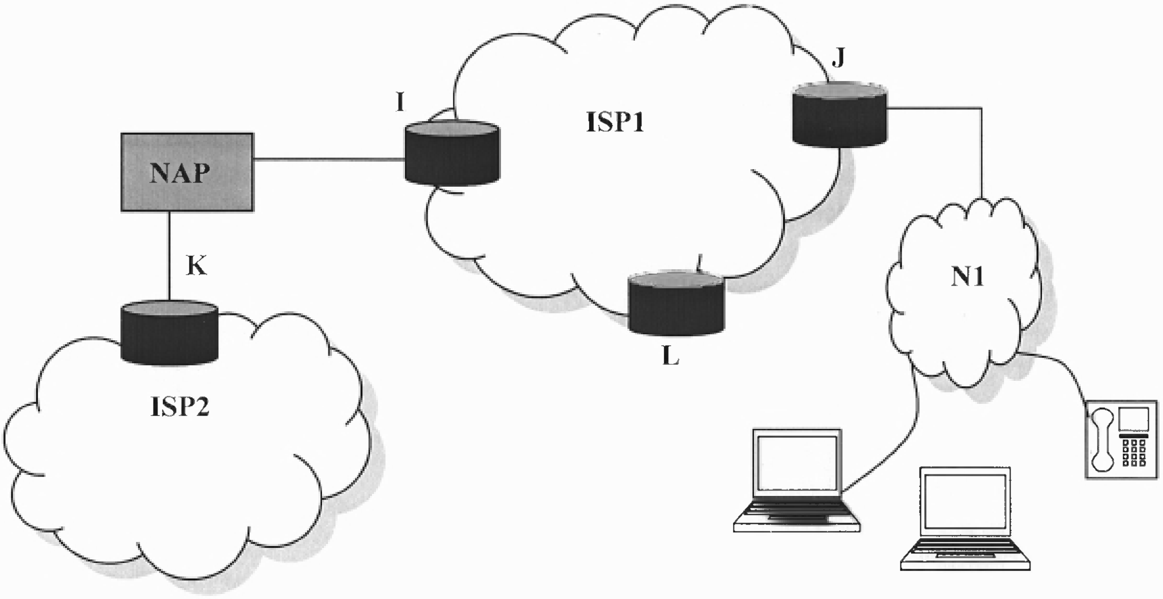


Figure 1.1 Example of a network where ISP (ISP1) is connected to a private network (N1) and other ISP network (ISP2).

Here ISP1 network is a service provider network with gateway routers I and J. ISP1 and ISP2 are connected via a network access point (NAP). Consider that ISP1 provides different services to its customer as shown in Table 1.1. Such enhancements require packet classification to determine the flow a packet belongs to based on one or more fields in the packet header. A flow usually is characterized by header fields source IP, destination IP, source port, destination port and protocol field of a packet. A per-flow service table in routers typically consists of a large number of rules arranged in an order where the best matching rule is placed first (at the top). A rule is a combination of *d* header fields (in most cases d equals five header fields) and an action associated to the combination. The action could be to drop the packet or to provide certain services, as an example.

Table 1.1 Sample packet forwarding services

Service	**Example**
Packet Filtering	Deny all traffic on interface
Accounting and billing	Treat all video traffic from network N1 on interface J as highest priority and perform accounting for the traffic
Policy routing	Send all voice-over-IP traffic arriving from N1 via interface L

In packet classification, three types of matching can be performed with these fields: 1) Exact matching: the fields in the header of the packet should match exactly the respective fields in the rule. 2) Prefix matching: a prefix and a prefix mask are provided by the rule for the longest match with fields in the packet. 3) Range matching: A range is provided by the rule and a match is found if the header information in the packet falls within this range. Due to the rapid growth in the size of rule sets for packet classification and increases in the link rate, multi-field based packet classification has become a fundamental challenge for high-speed designs. Of course, a software-oriented solution cannot satisfy the required processing speeds. Hence, efficient hardware implementations using ASIC devices or FPGAs have recently received substantial attention [31-34]. Some of the best packet classification algorithms targeting at the matching of d fields, where d>3, have O($log\ n$) time complexity at the cost of O(n^d) space, or O(($logn$)$^{d-1}$) search time at the cost of O(n) space, where n is the number of stored rules [46].

Researchers usually apply either an algorithmic or an architectural solution towards high performance in packet classification. However, sometimes hybrid approaches are pursued that involve a combination of these solutions. Such solutions are typically classified as *decomposition based* or *decision-tree based.* The former solutions search multiple fields in parallel and then combine the results, thus they are good candidates for hardware implementation. Decision-tree based solutions, like Hi-cuts [30], take a geometric view of the packet classification problem. The search space is reduced at each node of the decision tree based on information from one or more fields in the rule. A decision tree is built by choosing a dimension (i.e., a field) and also the number of cuts to make in the chosen dimension by using local optimization decisions. At every node a cut involves only one field. The cutting process is performed at each level and recursively on the children at the next level until the number of rules associated with each node at a level falls below a threshold.

FPGAs are bound by their limited number of resources, a disadvantage which is exacerbated further by continuous increases in rule-set sizes that beg for scalable hardware solutions. This implies the need for high on-chip rule-set compression that could ideally eliminate all off-chip memory accesses. Although packet classification has been recently researched by various groups, the ultimate goal of this dissertation is to achieve a comprehensive on-chip FPGA-based solution that does not require external memory accesses, while also being able to efficiently support more than 10,000 rules with a currently available FPGA device.

A novel approach where the rule-set is broken up at static time into combinations of smaller patterns involving two fields is proposed and implemented in this dissertation. The run-time system attempts to match pairs of fields at a time and these matching results are then combined to produce an address in an on-chip memory where a complete match, if present, can be verified. The rules are initially separated at static time into groups based on the presence of valid fields. A field is considered valid if its value is not "don't care." Using position-based vectors and partial matches, the groups are searched in parallel for the incoming packet headers and a running sum is generated that depends on the position vectors. This sum generates a unique address in the memory that verifies a match if this sum matches a pre-stored value. A final decision also involves rule priority. Compared to existing hardware-oriented solutions, this solution reduces drastically the on-chip memory consumption while also being able to match network speeds. It is also scalable, allowing the incorporation of many fields in the packet classification process.

1.6 Related Work (Packet Classification)

Lakshman and Stiliadis [43] proposed a scheme in which each rule is searched in parallel in every individual field using a prefix trie, and the result of the search is a bit vector (BV) where each bit represents a rule. A bit is set if the corresponding rule is matched in this field; it is reset otherwise. The intersection of all BVs via their bit-wise AND operation indicates the rule that matches the incoming packet. Although the scheme provides high throughput, the memory efficiency is very low, which makes it cumbersome for large rule sets. Baboescu et al. [42] employed the BV scheme assuming that for large rule-sets a packet can match not more than a few rules; a new aggregated bit vector scheme was devised using recursive aggregation of bit maps, and rule rearrangements for less memory space and higher speed. Ternary content addressable memories (TCAMs) [44, 45] also have been used in classification. Nevertheless, TCAMs are not scalable in terms of clock rate, power consumption or circuit area. Most of the TCAM-based solutions also have difficulty in converting ranges into prefixes. By combining TCAMs and the BV algorithm, Song et al. [40] presented the BV-TCAM architecture for packet classification. A TCAM is used for prefix or exact matches, whereas a multi-bit trie implemented as a Tree Bitmap [41] is used for source or destination port lookup. Their analysis involved 222 SNORT header rules.

Bloom filters [34, 37, 38] are popular due to their constant time requirements and low memory consumption. However, false positives are possible that require a secondary off-chip memory to check the authenticity of potential matches at a much slower rate. Trie-based schemes, like hierarchical tries, set-pruning tries [49] and grid-of-tries [50] work well for two-dimensional classifiers; however, as the number of dimensions increases their complexity increases too. Gupta and McKeown [47] and Taylor [48] have surveyed a large number of classification techniques. They concluded that very rarely will a packet match multiple rules. Using heuristics with real databases, Gupta and McKeown [47] developed the Recursive Flow Classification (RFC) algorithm that splits the packet header into many chunks of contiguous bits and then represents each one of them with a reduced number of action bits. A recursive process of action combining at run time yields the final action to be performed on the packet. The amount of storage for

RFC increases rapidly as the classifier size increases; it uses about 4 Mbytes to store 15,000 rules. Also, the heuristics-based HiCuts [30] approach has low memory requirements, consuming around 1 Mbyte for 1700 rules.

Taylor et al. [35] introduced Distributed Crossproducting of Field Labels (DCFL), a decomposition-based algorithm that employs independent search engines for different fields. They perform a distributed set membership query using a network of aggregation nodes; each query performs an intersection on the set of possible field combinations matched by the packet and the set of field combinations specified by filters in the filter set. Their design makes use of Bloom filters, thus it suffers from false positives. Papaefstathiou et al. [43] proposed a memory-efficient decomposition-based packet classification algorithm that employs multi-level Bloom filters to combine the search results from all the fields. Their FPGA implementation, called 2sBFCE [34], can support 4K rules with 178 Kbytes of memory consumption. However, the design takes 26 clock cycles on the average to classify a packet, resulting in a low throughput of 1.875 Gbps.

Hypercuts [36] is another heuristics-based approach similar to Hi-cuts except that it allows cuttings on multiple fields per step, thus resulting in a fatter and shorter decision tree. Jiang and Prasanna [31] modified the hypercuts algorithm to build a pipelined design in the form of a balanced tree. They reduce rule duplications in hypercuts by using an internal node to store the rules which are present in all of its children nodes and by also cutting precisely the range of to-be-matched fields. The design involves a tree pipeline that forms a decision tree and a rule pipeline that contains the rule lists for a node. Their implementation of 10K rules consists of 11 tree pipeline stages, 8 rule pipeline stages and a total of 12 rule pipelines. The memory consumption of their design is quite substantial, using around 407 Xilinx Virtex-5 Block RAM (BRAM) memories (36Kbits per BRAM) out of which 612 Kbytes are taken by the rule lists and the pipelined tree configuration.

The rest of the dissertation is organized as follows. Chapter 2 covers the discussion of the fixed-length sub-pattern method. Chapter 3 presents the variable-length sub-pattern method. Chapter 4 introduces the packet classification process. Concluding remarks along with a discussion on future work are finally presented in Chapter 5.

CHAPTER 2

SIGNATURE DETECTION USING FIXED-LENGTH SUB-PATTERNS

2.1 Fixed-length Method

A database of known malicious patterns is assumed and the objective is 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, the implementation is tested with the complete set of signatures in the SNORT database [17]. In summary, statically each pattern is broken up into fixed-length sub-patterns and then the position of each sub-pattern in all of the encompassing patterns is encoded 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.

2.1.1 Pre-Processing

The 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). N is fixed before the separation process. Section 2.3 shows the analysis for the SNORT database which confirms that the best choice is N=3. Identifying the position of sub-patterns in patterns is crucial to the algorithm. Once all of the patterns have been separated into their sub-patterns, all distinct N-character sub-patterns are stored into a table called GRP(N). Similarly, tables GRP(i), for i = 1,..., N-1, are created where GRP(i) stores all of the i-character sub-patterns that appear as tails in patterns. All of the GRP(i)'s, for i = 1,..., N, collectively denoted as GRP. Let L be the maximum sub-pattern offset for a given pattern set. A *bit vector* (BV) and an *end vector* (EV) are then created for every sub-pattern in GRP; each vector is L bits long. BV 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 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 $W = \{weight_1, weight_2, \ldots, weight_m\}$; let *bw* be the bits per weight. A set of six patterns is assumed and their sub-pattern separation for N=3 is shown in Fig. 2.1.

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			

Figure 2.1 A set of six patterns separated into sub-patterns for N = 3.

Fig. 2.2 shows the GRP tables created for these patterns assuming N=3, L=6, m=3 and bw=8. It can be inferred from Fig. 2.1 that L=6.

GRP(3) TABLE

SP	BV	EV	W1	W2	W3	baseaddress	hash
exe	100001	000001	3	21	45	0	0
cut	010000	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

Figure 2.2 GRP tables for the patterns in Fig. 2, assuming N=3, L=3, m=3 and bw = 8.

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 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. 2.3 shows the summation m-tuples (i.e., triplets since m=3) for the patterns in Fig. 2.1; their components are represented by *Sum1, Sum2* and *Sum3*. 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

Figure 2.3 The pattern table for the patterns in Fig. 2.1.

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. 2.3. 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, pattern splitting method to resolve collisions (explained later) can be used.

2.1.2 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.

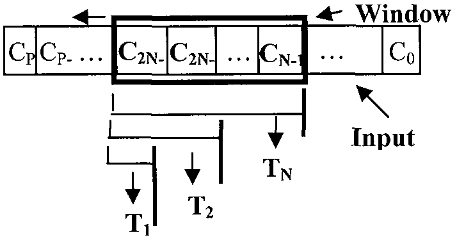


Figure 2.4 Processing a P-character input with an N-character shifting window.

Let C_1, C_2,..., C_P be an input stream of P characters, as in Fig. 2.4. 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 *offsetd* 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 ($d=d_1, d_2, \ldots, d_N$). Let EV_{GRP}, BV_{GRP} , W_{GRP} and $baseaddress_{GRP}$ represent arriving sub-pattern values from GRP. The detection unit then performs the following operations ("&" and ">>" denote concatenation and shift, respectively) where n is :

$EDV_d = DV_d$ AND EV_{GRP} ;

DV_d = '1' & (DV_d AND BV_{GRP} >> 1) ; if n=N, then the record will have BV

= DV_d ; otherwise

$ACC_d = ACC_d + W_{GRP}$; if (DV_d AND BV_{GRP}) neq 0 and n=N

= 0 ; if (DV_d AND BV_{GRP}) eq 0 and n=N

= ACC_d ; if $n \neq N$

$Temp_d = ACC_d + W_{GRP}$; if EDV_d neq 0

= 0 ; otherwise

pattern address = baseaddress + offsetd +1 ; if EDV_d neq 0

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 offsetd 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 offsetd ('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.

2.1.3 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 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.

Prevention of false positives was guaranteed by the weight assignment process. It was found 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. Pattern length was used 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 appeared in a longer pattern as well as short patterns, then it was given a larger weight. Sufficient bit widths for sub-pattern and summation tuples were chosen to reduce the complexity.

Consider the example with tail "rds" from patterns 4 and 6 in Fig. 2.1 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. 2.5, 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. 2.3), 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. If so then, either the pattern splitting method(explained in the next section) or the weight tuple values of a sub-pattern are changed to avoid such a scenario.

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. 2.2.

Figure 2.5 All possible "fictitious" patterns producing non-zero EDVs for tail "rds".

2.1.4 Pattern Splitting

Consider an 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 at the correct tail offset. Also, these sub-patterns appear in identical-with-this-input offsets in other patterns. The 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. Such a possibility is eliminated by splitting pattern 1 into two smaller patterns "abc d" and "efl 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 hashing.

2.2 Hardware Implementation

N=3 (i.e., a window with up to three bytes or 24 bits) is assumed. The block diagram of the implementation is shown in Fig. 2.6.

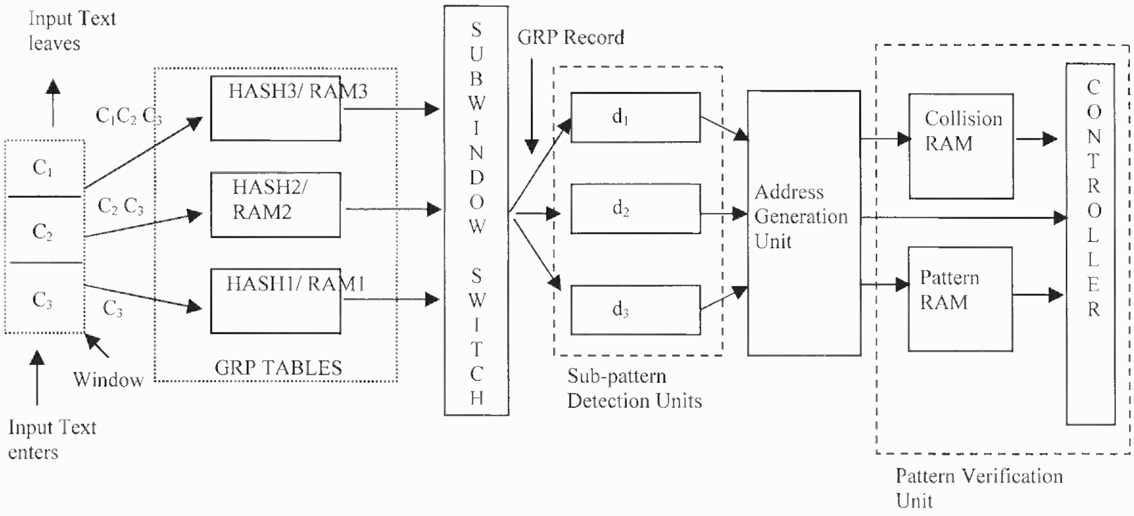


Figure 2.6 Hardware Architecture for N=3.

Hashing: Analysis of the current SNORT pattern set was performed and it was found that GRP(N) records are predominant and require a bigger RAM compared to the other GRP(i), for i=1, 2,…, N-1. Separate hash functions and RAMs are used 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. The hash functions apply simple XOR-ADD operations to the input to generate an address; they do not need separate key inputs. Three RAMs per GRP table are used 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 pattern splitting method as explained above is used.

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). The 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 the implementation, SNORT patterns are represented with a 41-bit vector (the longest pattern in the current SNORT contains 122 characters whereas 3x41=123). The 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 offsetd 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 offsetd counter is initially enabled after being reset to the default '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 offsetd being incremented to '2' and the next offsetd being enabled after being reset to its default value of '1'. Thus, the first offsetd counter keeps track of the '1' which is now in position two while the second offsetd 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";

It can be seen that pattern 1 has commonalities with patterns 2, 3 and 4. However, they are not the same. For the input text "abc123xyzklmppp", pattern 3 will be triggered. But it could also be induced 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. A script on SNORT pattern was ran and it was inferred that there could be at most two 1's in DV in a clock cycle. Thus, only two offsetd counters are needed to subvert such attacks. However, four counters were kept for future updates in the SNORT signatures. Also, since logic in the implementation is drastically reduced, adding more counters to the logic will not make a big difference to resource usage. In extreme cases pattern splitting approach can be used. 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 this pattern can then be detected 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.

2.3 Results and Comparison with Earlier Work

2.3.1 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 the proposed fixed-length sub-pattern 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 N is increased the logic consumption increases since more detection units are needed. Also bigger switching fabric is needed to forward the GRP vectors to the appropriate detection units in a cyclic manner. For a good choice of N, effects of N on the number of GRP records were studied for SNORT. Since the longest pattern has 122 characters, L could easily be obtained by dividing 122 by N. The results are shown in Table 2.1.

Table 2.1 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

The 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 the design for future pattern additions, experiments were carried out in two parts. In Part I, the sub-patterns were generated, 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, a modification of the already loaded set of patterns was enabled by adding the remaining set of 307 patterns. Information extracted for Parts I and II of the experiments is shown in Table 2.2.

Table 2.2 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

2.3.2 VHDL System Synthesis/Implementation

The synthesis and simulation of the design worked flawlessly for both parts of testing. For the bit vectors, L was set to 41 bits. Also, the off-line experiments for 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 12 bits (since 2^6 x 41 < 2^{12}). For the 10,135 GRP(3) records, it was deduced that there were only 971 distinct BVs. Hence, the BVs were moved into a separate smaller RAM with 1024 locations. Thus, instead of storing a 41-bit BV for every record, only a 10-bit pointer per record was stored, which resulted 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. VHDL was used 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. The 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. The 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 (the suggested implementation based on the results in Table III), 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 the 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.

2.3.3 Comparison with Earlier Approaches

Table 2.3 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. The fixed-length sub-pattern design employs 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. It can be concluded that the fixed-length sub-pattern 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 2.3 Comparison with other designs (N/A: not available or not applicable).

Design, Year	FPGA Device	Patterns	Chara-cters	MHz	Through-put (Gbps)	BRAM Mem (Kbits)	Logic cells/ character	BRAM bits/ character
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
Fixed-length sub-pattern method, 2009 (bw=6)	V2 Pro XC2VP70	6456	104,886	300.3	2.402	1818	0.050	17.74

CHAPTER 3

SIGNATURE DETECTION USING VARIABLE-LENGTH SUB-PATTERNS

3.1 Variable-Length Method

The sub-pattern extraction method proposed here represents a major improvement over earlier fixed-length method. The latter method assumed a fixed length for sub-patterns whereas this new method deals with variable-length sub-patterns. Also, the fixed-length method involved a slow offline process to guarantee avoidance of false positives in the decision process; the process creates all possible fictitious patterns that contain all possible combinations of known sub-patterns and tails, so its complexity grows exponentially with the population of the latter. In contrast, this variable-length method reduces drastically this offline processing time as it does not generate all possible combinations (more details follow in Section 3.2).

3.1.1 Pre-Processing

Initially, patterns of length greater than a preset *Max_Fragment_Length* number of characters are split into fragments (i.e., sequences of at most Max_Fragment_Length characters). Although the longest pattern in SNORT contains 213 characters (this is the newer SNORT version used for this method), 80% of them contain less than or equal to 24 characters. This fragmentation creates fragments of length less than or equal to this value. It is shown later in this chapter that this fragmentation reduces the size of the hardware 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 the example in this chapter, it is assumed that Max_Fragment_Length is 16. Fig. 3.1 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. Normally, end of a O_Pattern is fragmented into two equal halves. If O_Pattern is more than twice the Max_Fragment_Length, then the pattern is split in such a way that fragments of lengths Max_Fragment_Length characters each are produced (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 this fragmentation rule is not followed; these cases are presented in Section 3.2.

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

Figure 3.1 Pattern sets before and after fragmentation.

After fragmentation, next stage of pre-processing is carried out. 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 $W = \{weight_1, weight_2, ..., weight_m\}$; 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. 3.2).

Using these weight tuples, a *summation m-tuple* for each pattern is calculated in the new pattern set after fragmentation of the original set. Consider pattern "Badcommand" (pattern 3 in Fig. 3.1). The summation tuple for this pattern is calculated 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; the choice of this number is discussed 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");

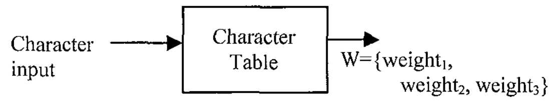
SUM("com") = W("c") + 2 * W("o") + 4 * W("m");

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. 3.1, 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").



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

Figure 3.2 Character table for m=3 and bw=3.

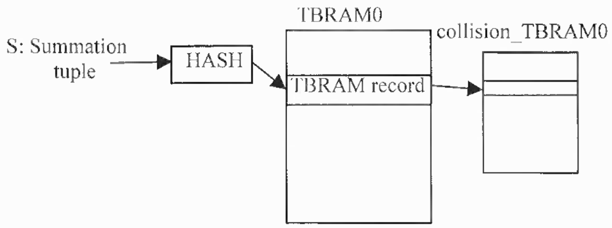
This summation method is carried out on all the patterns. Fig. 3.3 shows the tuples for a chosen character table. The position of individual characters in the group of three is taken into account to create different sums (i.e., weight tuples) for patterns like "Badcommand" and "daBcommand" that contain identical, but permuted characters. However, sometimes there may be 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 is identified in the pre-processing stage and is 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 in this chapter).

	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

Figure 3.3 Summation tuples for the pattern set in Fig. 3.1; SUM= (Sum$_1$, Sum$_2$, Sum$_3$).

Once the summation m-tuple of a pattern are pre-calculated, the m-tuple are stored in a table at a location which is produced by a hash function on this summation m-tuple. For the example, 16 distinct tables are created 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, their summation m-tuples are stored into one table instead of having separate tables for each of these lengths. In the example of Fig. 3.1, the summation m-tuples of 9-character patterns are placed into TBRAM0 and the rest of the patterns containing 10, 11 and 15 characters into the common table TBRAM1. The weights of characters are chosen in a way that ensures unique summation m-tuples for the patterns in a common TBRAM. In addition to the summation m-tuples, two bit values are stored, *start_fragbit* and *no_fragbit*, and a *collision_TBRAM_pointer* as shown in Fig. 3.4. 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 the chosen fragmentation is changed during pre-processing in order to remove this case.



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.

Figure 3.4 Summation m-tuples placed in TBRAM tables.

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. The maximum number of records per collision stored in collision_TBRAM varies with the implementation. According to the analysis performed for this implementation, 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 patterns are fragmented further to place them in exclusive locations in TBRAM (this process is explained later in Section 3.2).

STEP 2 (BIT/END VECTOR GENERATION): The static-time pre-processing divides each pattern into contiguous sequences of 1- to N-character sub-patterns. Two sets of sub-patterns are created 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. A point to note is 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 3.5.a and Fig. 3.5.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 the algorithm. Once all of the patterns have been separated into their sub-patterns, all distinct N-character sub-patterns are stored 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, tables GRP(i), for i = 1, ..., N-1, are created where GRP(i) stores all of the i-character sub-patterns that appear in the patterns. All of the GRP(i)'s, for i = 1, ..., N, are collectively denoted as GRP. A table may be empty if there is no sub-pattern of the corresponding length. Patterns could be divided into sub-patterns of any number of characters from 1 to N, however they are broken in such a way that the number of sub-patterns per pattern are minimized. This is not a rigid rule as there are exceptions when dealing with collisions. Such exceptions are discussed in Section 3.2.

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		

(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)

Figure 3.5 (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.

A *bit vector* (BV) and an *end vector* (EV) is created 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 seen later, using the two sub-pattern sets DSN3 and DSN4 to derive their BV and EV vectors, the possible presence of a pattern match can be detected. This along with a pattern summation tuple match is then used to confirm a pattern match. Although only three bits for BV and four bits for EV are needed in the case of DSN4, L=4 has been used 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. 3.5.a). This approach helps in placing 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. 3.6.a and Fig. 3.6.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)

Figure 3.6 GRP tables for the patterns in Fig. 1, assuming **(a)** N=3 and L=4; **(b)** N=4 and L=4.

3.1.2 Run-time Pattern Detection

The summation m-tuples in the TBRAMs are stored as explained above. The BV's and EV's are generated and the weight m-tuples for every character are stored in the character tables. Pattern detection unit is now explained in the following paragraphs.

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 the example with Max_Fragment_Length =16 there are sixteen accumulation units, ACC1 to ACC16, as shown in Fig. 3.7, 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.

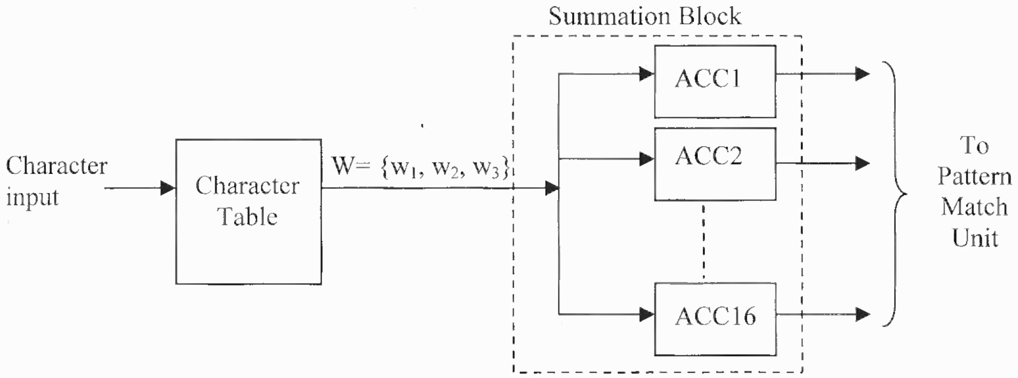


Figure 3.7 Summation Block.

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. There are two bit detection units for the DSN3 and DSN4 generated vectors and are denoted as BDN3 and BDN4, respectively. BDN3 has the GRP(1), GRP(2) and GRP(3) tables generated using N=3 (Fig. 3.6.a) whereas BDN4 has GRP(1), GRP(2), GRP(3) and GRP(4) tables generated using N=4 (Fig 3.6.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. 3.8. The hashing implementation is simple and made up of XOR and ADD operations on the incoming input. The method used to eliminate sub-patter collisions in hashing is explained in Section 3.2. 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.

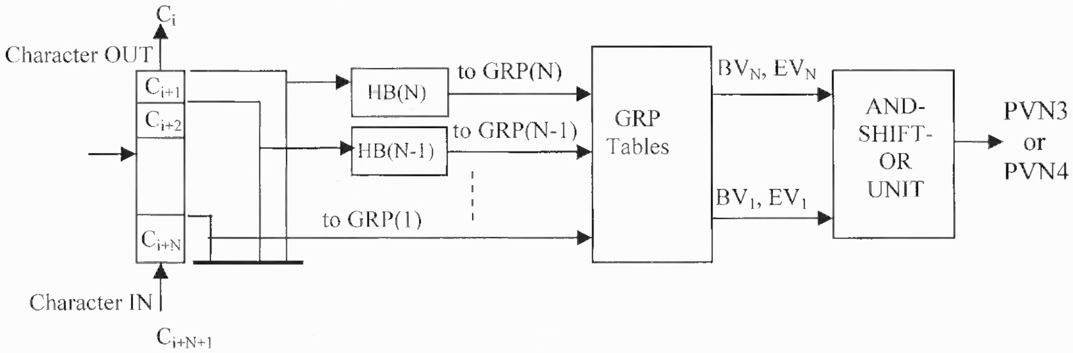


Figure 3.8 Bit Detection Unit for N=4, 3.

<u>Bit Detection Unit for N=4 (BDN4)</u>: 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 A). 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

EDV_1= ((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, the result of all the AND-OR operations in one DV-EDV-PV set, suppose DV_1-EDV_1-PV_1 are stored; 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 A 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.

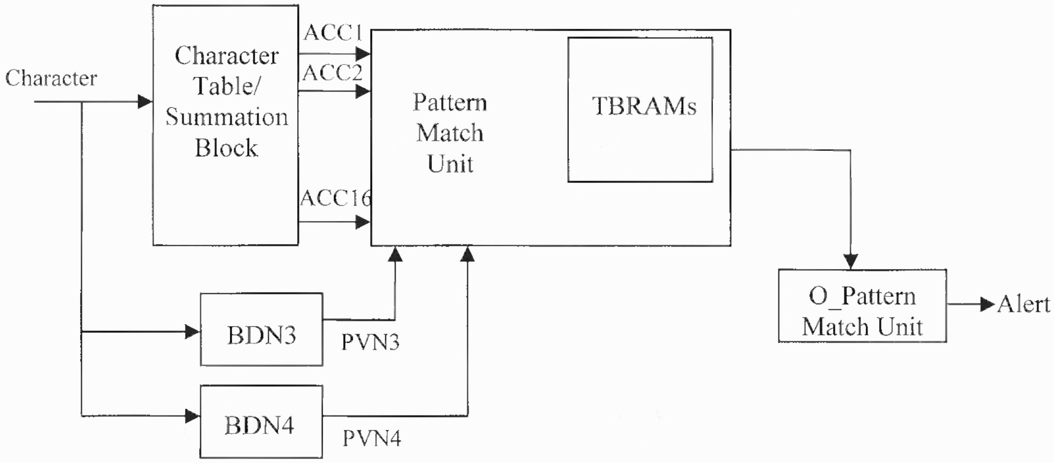
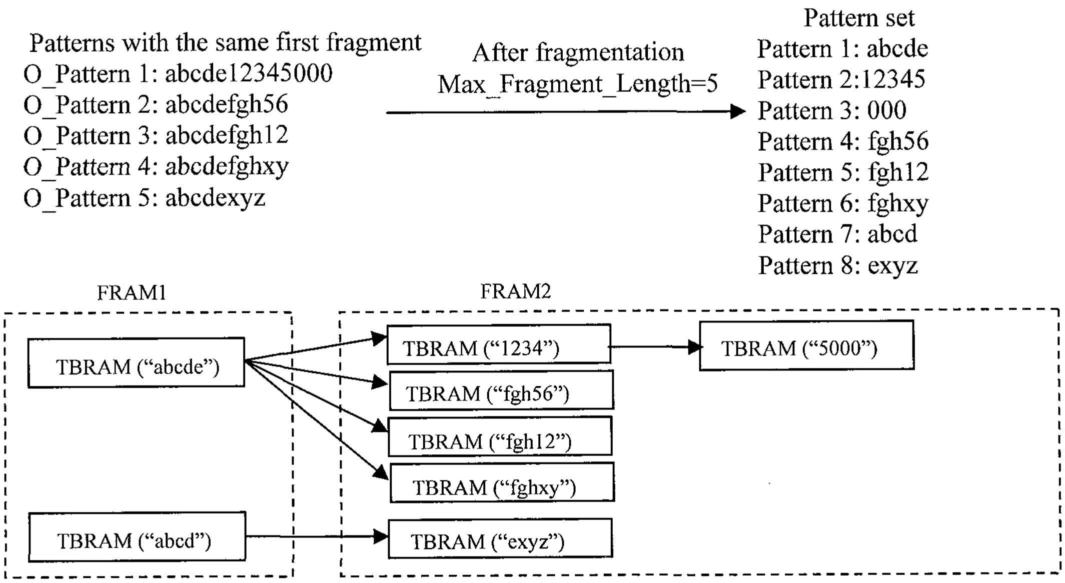


Figure 3.9 Block diagram of complete pattern detection system.

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. Collision TBRAMs are looked into 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 the complete system is shown in Fig. 3.9.

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 FRAM2 is accessed to fetch the subsequent fragments using the pointer to FRAM2. Using appropriate delay cycles the fragments are connected in the O_Pattern match unit to match the longer pattern. The data structure used for FRAMs is shown in Fig. 3.10.

The first node of a pattern is stored in FRAM1 and the others are stored in FRAM2. The maximum number of nodes which can have the same prefix fragment is set to four. If the number is more than four, then the final pattern concatenation process is moved to software. The link is disconnected and start_fragbit and no_fragbit are set to '1' ("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, in the experiments conducted for this method, there were no common prefix fragments because of the 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 the rule of dividing the O_Pattern into fragments of almost equal lengths causes the later to contain different patterns (O_Pattern 5 in Fig. 3.10). Thus normally there are no 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.



TBRAM ("5000"): Address of summation m-tuple of pattern "5000" in TBRAM

Figure 3.10 FRAM Block data structure.

3.2 Eliminating Pattern Collisions and False Positives

3.2.1 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, a collision_TBRAM is kept 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 the pattern is fragmented differently. Fig. 3.11 shows an example using pattern 3 and pattern 4 of Fig. 3.1. 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 the new pattern is broken up as per convenience into appropriate fragments (i.e., patterns) that remove this collision.

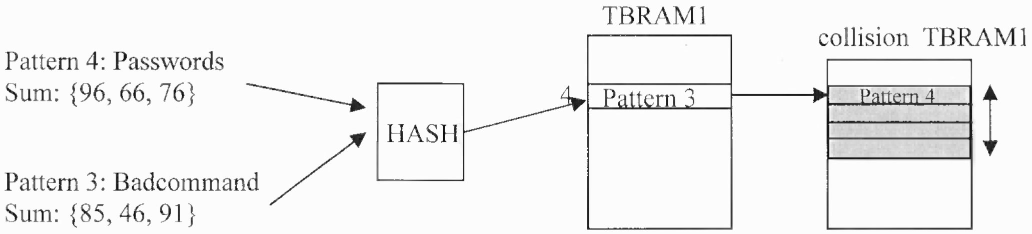


Figure 3.11 Pattern collisions in TBRAM.

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. 3.1, the criterion of fragmenting an O_Pattern into patterns was the length. But now, patterns of the new pattern set (i.e., Fig. 3.1) are fragmented to avoid collisions. Once this fragmentation is complete, new summation m-tuples for these newly formed patterns are stored appropriately. These cases are rare since an optimal placement of patterns can be achieved 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.

3.2.2 Hashing and Eliminating Collisions for Sub-patterns

Plain hash functions containing XOR and ADD operations are used to place the sub-patterns in the GRP RAMs. For the sake of efficiency, separate hash functions and RAMs are used 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. The hash functions apply simple operators to the input to generate an address; they do not need separate key inputs. The character window containing N characters are hashed separately using N characters for GRP(N), N-1 characters for GRP(N-1) and so on as shown before.

The important requirement for the design is that the GRP RAMs should output BV and EV ever clock cycle. Hence collisions in the GRP RAMs have to be avoided. A maximum of four hashing functions in a hash block HB(i), for i=2, …,N are used (this number varies depending on the size of the GRP RAM needed and also 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. 3.12 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.

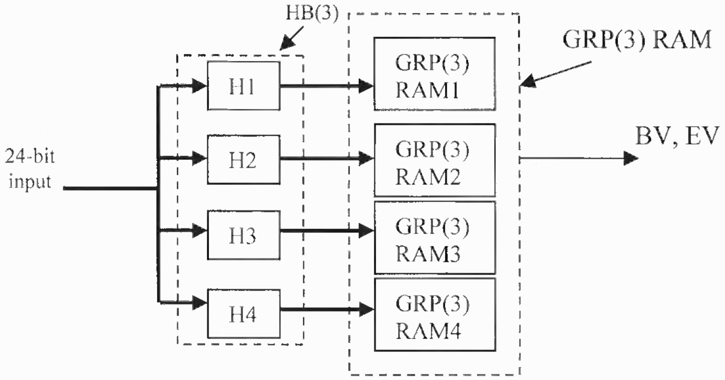


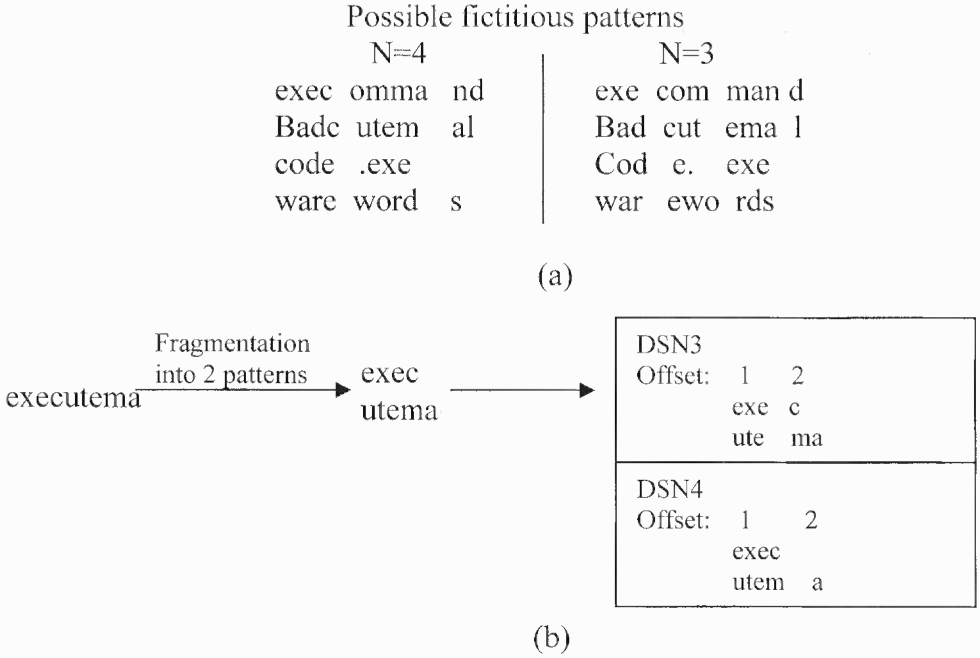
Figure 3.12 Hashing Block of the GRP(3) RAM in BDN3.

Since there are different length sub-patterns, there is an 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 the way the pattern is split, is changed. For example, it can be split as {“a”-“bc”-“def”} or {“ab”-“c”-“def”} or {“a”-“bcd”-“ef”} or in some other way such that the sub-pattern records can be placed in non-vacant locations in the GRP RAMs.

3.2.3 Eliminating False Positives for Patterns

Once the whole process of weight distribution and BV-EV generation for DSN3 and DSN4 is done, pattern-related false positives are checked with the following method and if found at static time, then appropriate action to eliminate them is taken. 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. 3.13.a, fictitious patterns which will generate non-zero EDVs are shown. It can be deduced from the Fig. 3.13.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. 3.1 for N=4 has "exec" at the first offset and the first pattern in BDN3 has "exe" at first offset. Now a pattern (other than first one) in BDN3 which has a sub-pattern that starts with character "c" at the second offset ahs to be searched. Pattern 3 has "com" at the second offset. Thus, if a string like "execommand" comes in, then a non-zero EDV in BDN3 and BDN4 will be generated, with identical PVs. It has to be ensured 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. This can also be countered by fragmenting the pattern or by breaking the pattern into sub-patterns differently.

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. Such a situation can be easily avoided 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 3.13. b. It can now be seen that the above given fictitious pattern cannot be generated with the new pattern set.



Possible fictitious patterns

N=4	N=3
exec omma nd	exe com man d
Badc utem al	Bad cut ema l
code .exe	Cod e. exe
ware word s	war ewo rds

(a)

(b)

Figure 3.13 (a) Fictitious patterns which generate non-zero EDV;
(b) Fictitious pattern prevention using appropriate fragmentation.

Fragmenting the patterns again looks similar to the approach in Section 3.2.2; however, here different problem of false positives is targeted. This approach is only for adding new patterns. For the SNORT database, the sub-pattern creation method for N=4 and N=3 along with the uniqueness of the summation m-tuples avoids false positives.

3.3 Results and Comparisons with Earlier Work

3.3.1 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 the 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. Some analysis to selecting Max_Fragment_Length was done. It can be easily inferred 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 Max_Fragment_Length is increased, the number of patterns that are fragmented will decrease since the design can now provide pattern matching for a longer fragment.

Fig. 3.14 shows pre-processing results for various fragment lengths. The LUT usage for an implementation performed will almost remain constant irrespective of the number of patterns that are added. Thus, the concern is for BRAM usage since this is where compression can be obtained. 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. A grouping of three consecutive characters for summation was used. The difference between using three character grouping and any other higher number of characters grouping is that the summation tuple value will be higher in the latter case thereby requiring more bits per summation tuple. However, three-characters grouping are sufficient to produce exclusive summation tuples in this case.

Max_Fragment_Length= 32	**Max_Fragment_Length= 24**	**Max_Fragment_Length= 16**
DSN4	**DSN4**	**DSN4**
GRP4 records: 10,439	GRP4 records: 10,419	GRP4 records: 10,311
distinct BV-EV pointers: 329	distinct BV-EV pointers: 225	distinct BV-EV pointers: 125
GRP3 records: 939	GRP3 records: 971	GRP3 records: 1091
distinct BV-EV pointers: 57	distinct BV-EV pointers: 38	distinct BV-EV pointers: 27
GRP2 records: 866	GRP2 records: 902	GRP2 records: 981
distinct BV-EV pointers: 65	distinct BV-EV pointers: 52	distinct BV-EV pointers: 57
DSN3	**DSN3**	**DSN3**
GRP3 records: 9176	GRP3 records: 9794	GRP3 records: 10077
distinct BV-EV pointers: 623	distinct BV-EV pointers: 501	distinct BV-EV pointers: 226
GRP2 records: 859	GRP2 records: 845	GRP2 records: 823
distinct BV-EV pointers: 129	distinct BV-EV pointers: 110	distinct BV-EV pointers: 45
Number of fragments: 6829	Number of fragments: 7155	Number of fragments: 8566
FRAM records	**FRAM records**	**FRAM records**
FRAM1: 946; FRAM2: 1206	FRAM1: 1275; FRAM2:1581	FRAM1: 2005; FRAM2: 3507

Figure 3.14 Pre-processing results for Max_Fragment_Length= 32, 24 and 16.

If two patterns belonging to the same TBRAM group have the same summation m-tuple then the weight m-tuple value of the character is changed and the summation tuples are re-calculated. If after a fixed number of attempts a solution is not obtained, the pattern is fragmented and the check is repeated. Once the check is done successfully, pattern addresses are generated such that the pattern can be placed 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 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 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.

To test the design for future pattern additions, experiments were carried out in two parts. In Part I, sub-patterns, their respective vectors and the pattern addresses for 5834 patterns from the SNORT rule set were generated. 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, a modification of the already loaded set of patterns was enabled by adding the remaining set of 621 patterns. Information extracted for Parts I and II of the experiments conducted is shown in Table 3.1.

Table 3.1 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

3.3.2 VHDL System Synthesis/Implementation

The synthesis and simulation of the design worked flawlessly. The design was implemented with Max_Fragment_Length of 24 characters. The parameters for the design with 24 characters for Max_Fragment_Length are discussed in this section. The length of BV for BDN3 and BDN4 was set to 8 bits and EV was set to 9 bits. Also, the 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 three characters are grouped 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, it was deduced that there are only 225 distinct BV-EV combinations. Hence, the BV-EV combination was moved into a separate smaller RAM with 256 locations. Thus, instead of storing an 8-bit BV and 9-bit EV for every record, only an 8-bit pointer per record is now stored 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. 3.15 shows the chosen parameter values for system synthesis. For GRP(1), a BV and EV pointer are not needed 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.

Patterns of varying lengths are grouped into a single TBRAM in such a way that they are equally distributed in the TBRAMs. Fig. 3.15 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, GRP tables are used. Since they are already stored in one of these tables using a single bit field in the GRP RAM can identify whether the GRP record is also a pattern of interest. Hence a separate TBRAMS for them is not needed. VHDL was used 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:
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)

DSN3
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

Figure 3.15 Parameter values for system synthesis.

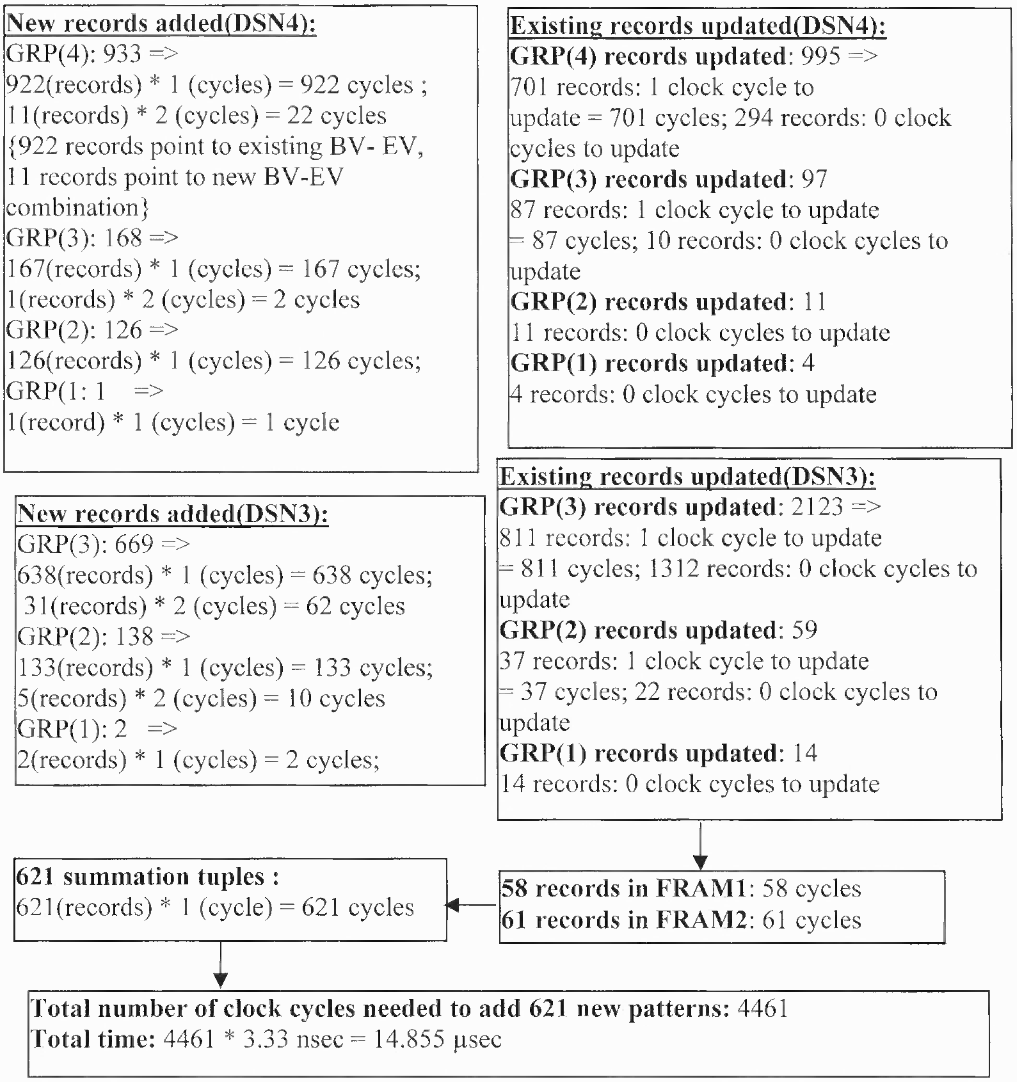
The hardware synthesis was done using Synplify Pro 9.1 as well as Xilinx ISE, with the parameters shown in Fig. 3.15. The design was implemented on a Virtex II Pro (XC2VP70) FPGA. For Max_Fragment_Length=24, it employs 102 18-Kbit BRAMs (Block RAMs), 5162 Flip Flops and 5569 LUTs, and operates at 300.1 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

Figure 3.16 Parameters needed to add a pattern.

The process to insert a new pattern is as follows. The AMIRIX PCI board used in the implementation has a 64-bit data bus. Due to the dual-ported BRAMs in the design, and the fact that reading and writing are independent of each other, BRAM updates can proceed while packets are being processed. Various record fields are grouped for the 64-bit bus. The different words needed to add a GRP record are shown in Fig. 3.16. 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. All the placement of the patterns and fragmentation was 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, the sub-patterns and BV-EV pointers are added first followed by the BV-EV values, FRAM records, if any, and then the summation tuples. When a new pattern is added, it is only needed 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.



New records added(DSN4):
GRP(4): 933 =>
922(records) * 1 (cycles) = 922 cycles ;
11(records) * 2 (cycles) = 22 cycles
{922 records point to existing BV- EV,
11 records point to new BV-EV
combination}
GRP(3): 168 =>
167(records) * 1 (cycles) = 167 cycles;
1(records) * 2 (cycles) = 2 cycles
GRP(2): 126 =>
126(records) * 1 (cycles) = 126 cycles;
GRP(1: 1 =>
1(record) * 1 (cycles) = 1 cycle

Existing records updated(DSN4):
GRP(4) records updated: 995 =>
701 records: 1 clock cycle to update = 701 cycles; 294 records: 0 clock cycles to update
GRP(3) records updated: 97
87 records: 1 clock cycle to update = 87 cycles; 10 records: 0 clock cycles to update
GRP(2) records updated: 11
11 records: 0 clock cycles to update
GRP(1) records updated: 4
4 records: 0 clock cycles to update

New records added(DSN3):
GRP(3): 669 =>
638(records) * 1 (cycles) = 638 cycles;
31(records) * 2 (cycles) = 62 cycles
GRP(2): 138 =>
133(records) * 1 (cycles) = 133 cycles;
5(records) * 2 (cycles) = 10 cycles
GRP(1): 2 =>
2(records) * 1 (cycles) = 2 cycles;

Existing records updated(DSN3):
GRP(3) records updated: 2123 =>
811 records: 1 clock cycle to update = 811 cycles; 1312 records: 0 clock cycles to update
GRP(2) records updated: 59
37 records: 1 clock cycle to update = 37 cycles; 22 records: 0 clock cycles to update
GRP(1) records updated: 14
14 records: 0 clock cycles to update

58 records in FRAM1: 58 cycles
61 records in FRAM2: 61 cycles

621 summation tuples :
621(records) * 1 (cycle) = 621 cycles

Total number of clock cycles needed to add 621 new patterns: 4461
Total time: 4461 * 3.33 nsec = 14.855 μsec

Figure 3.17 Total time to add the 621 new patterns.

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 the information of the pattern fragments (pattern addresses in the TBRAM as shown in Fig. 3.10) is stored 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. 3.17.

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. 3.18 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, the node for “abc” in position 1 is then deleted (as shown in Fig. 3.18. b). The other sub-patterns are removed in the same manner in subsequent clock cycles. To modify a pattern, the old pattern is deleted and then the modified pattern is added. The flexibility of updating or changing a pattern in the database without re-calculating hashing keys works to the design’s advantage as compared to the approach in [25].

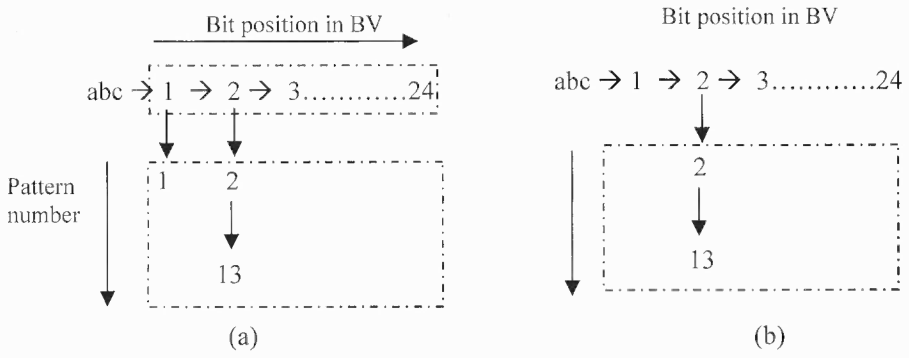


Figure 3.18 Linked list for updates with sub-pattern "abc".

3.3.3 Comparison with Earlier Approaches

Table 3.2 shows a comparison with the most prominent efforts in the area of pattern matching with FPGAs or ASICs. As discussed in the previous chapter, 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. The variable-length design is the most comprehensive so far as it employs the largest freely available SNORT database (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 [23] involves memory tiles where a 2-bit input selects one of four finite state machines. Although [23] does not list the number of patterns in the implementation, it contains a comparison with the design proposed in [14] that assumes 1466 rules with 18,031 characters. The work in [23] uses 3200 Kbits of memory, yielding a memory consumption ratio of 181.7 bits per character which is quite high compared to the variable-length design (see Table 3.2). Also, the updating process in the variable-length design is very simple as it does not require intricate knowledge of the design. An off-line script just simply creates sub-pattern records and pattern addresses. It can be easily concluded that the 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, the analysis is comprehensive as it involved a larger number of SNORT signatures than earlier approaches.

Table 3.2 Comparison with other designs (N/A: not available or not applicable).

Design, Year	FPGA Device	Patterns	Characters	MHz	Through-put (Gbps)	BRAM Memory (Kbits)	Logic cells/ Character	BRAM bits/ character
Baker [7], 2004 (no new rules)	Virtex-II Pro 100	361	8263	250	1.790	0	0.35	0
Sourdis [14], 2004 (no new rules)	Virtex-II 3000	1466	18,031	335	2.680	0	0.97	0
Clark [10], 2004 (no new rules)	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	Virtex-II Pro XC2VP30	2187	33,613	306	2.448	702	0.06	21.4
Variable-Length sub-pattern method, 2009 (Max_Fragment_Length=24)	Virtex-II Pro XC2VP70	6455	105,763	300.1	2.408	1836	0.052	17.77

CHAPTER 4

EFFICIENT PACKET CLASSIFICATION ON FPGAS TARGETING AT MANAGEABLE MEMORY CONSUMPTION

4.1 The Packet Classification Method

The packet classification problem is treated as a pattern matching problem involving two fields of the packet header at a time. Assume four consecutive fields F1, F2, F3 and F4 in a rule. The method compresses at static time the information that appears in every possible pair of fields in the rule-set by extracting fragments of 8 and 7 contiguous bits, and then encoding the position of 8-bit and 7-bit fragments in these pairs using position/bit vectors. An accumulated sum also is created for individual fields by adding up unique weight tuples previously assigned to 8-bit and smaller fragments in the fields of the rule-set. For example, for a source IP field which has 32 bits, 32 distinct accumulated sums are created. Using the position vectors, multiple matches of field pairs (i.e., F1-F2, F1-F3, F1-F4, F2-F3, F2-F4 or F3-F4) are identified simultaneously. Individual match results are then combined for field pairs by adding up individual field sums to produce a combined sum; the ultimate verification for a rule match is successful if the combined summation value is the same as the pre-stored value in the FPGA's BRAMs for this rule. The BRAM address for the rule is a function of the combined sum when considering all of its valid fields.

Packet classification method is explained with the help of the following brief example rule-set. Assume the packet classification rules of Table 4.1 involving five fields: source IP, destination IP, source port, destination port and protocol. The number following the '/' in the SIP and DIP fields is the mask which signifies the number of valid bits in the field. Since the protocol normally requires exact matching, pre-processing is not employed for the creation of position vectors.

Table 4.1 Sample rule-set

ID	Source IP (SIP)	Destination IP (DIP)	Source port (SP)	Destination port (DP)	Protocol	Group
1	90.24.13.4/32	51.63.17.19/32	1023 :1024	413 :413	TCP	15
2	89.24.13.4/32	5.6.7.9/32	1023 :1024	103 :109	TCP	15
3	11.71.19.14/23	23.98.128.80/21	0 :1023	0 :5	TCP	15
4	163.92.37.190/32	11.24.179.56/32	0 : 65535	8080:8080	TCP	13
5	97.166.41.112/31	182.125.194.192/23	0 : 65535	29301 : 29301	TCP	13
6	27.26.30.130/19	246.67.55.211/25	0 : 65535	3106 : 3108	TCP	13
7	19.14.103.41/32	1.6.0.0/0	1023 :1024	100 :101	TCP	11
8	1.6.0.0/0	5.6.7.9/32	1023 :1024	13 :13	TCP	7
9	101.121.77.33/25	23.98.128.80/21	0 : 65535	0 :65535	TCP	12
10	11.23.131.145/29	5.6.7.9/0	0 :1023	0 :65535	UDP	11
11	0.0.0.0/0	0.0.0.0/0	0:1023	13:13	TCP	3
12	0.0.0.0/0	0.0.0.0/0	0 :65535	0 :65535	Any	0

4.2 Pre-Processing Phase

4.2.1 Rule Grouping

The rules are initially separated into groups during pre-processing phase based on their number of valid fields; a field in a rule is considered valid if its content is not a don't care. For example, the first rule in Table 4.1 has all of the four fields valid and, hence, is placed in group G15 (i.e., "1111" in binary is "15" in decimal). Similarly rule 4 has the first, second and fourth fields valid and, hence, is placed in group "1101" or G13.

4.2.2 Fragmentation Schemes

The binary representation of the value in each field of each rule is then fragmented using two schemes that involve eight and seven contiguous bits, respectively. We denote these fragmentation schemes as FRAG8 and FRAG7, respectively. Using two fragmentation schemes aids the process of filtering out falsely generated bit vector matches. This process is explained in detail later.

FRAG8 and FRAG7 split the field pattern into 8-bit and 7-bit fragments respectively till the tail is reached. Fig. 4.1 shows the first field of rule 3 being fragmented according to both schemes. Since this field in rule 3 contains 23 valid bits, the tail will contain 7 bits. This fragmentation is then used to create a Bit Vector (BV) and an End Vector (EV) for every fragment value/pattern obtained. More specifically, BV shows the position of this fragment pattern in the field, actually for all the rules that eventually contain it, excluding their tail. That is, if a particular fragment pattern appears only in positions 1 and 3 of the same field in the same or different rules, then its BV will be "0110...0". The EV vectors store information about the tail fragments of patterns in fields. If a fragment appears as a tail, then it will contain '1' in the respective position of its EV vector. Multiple appearances of a fragment pattern in the same position (tail or non-tail) of multiple rules are registered only once in this pattern's BV or EV. The lengths of BV and EV depend on the fragmentation scheme, and also on the length of the field in bits. In Fig. 4.1, the field shown is the Source IP which is a 4-byte field and hence BV generated for it will be 3 bits long in FRAG8 and 4 bits long in FRAG7 scheme (since tail fragments are not included in it). Similarly length of EV will be 4 bits and 5 bits in FRAG8 and FRAG7 scheme respectively. In Fig. 4.1, the decimal value/pattern 11 appears in the first fragment and hence its BV will be "100" and the pattern 23 has a BV of "010" since it appears in the second 8-bit position (assuming just one rule in the set).

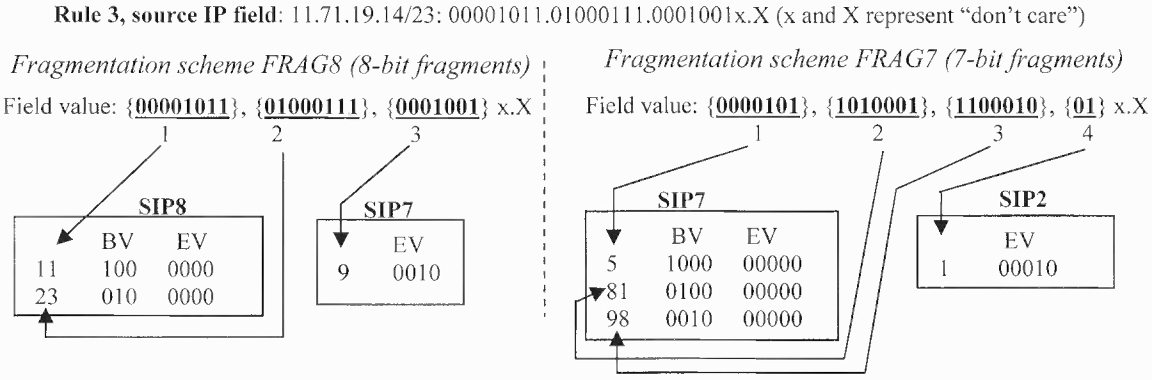

Rule 3, source IP field: 11.71.19.14/23: 00001011.01000111.0001001x.X (x and X represent "don't care")
Fragmentation scheme FRAG8 (8-bit fragments)
Field value: {00001011}, {01000111}, {0001001} x.X
1
2
3
SIP8
BV EV
11 100 0000
23 010 0000
SIP7
EV
9 0010
Fragmentation scheme FRAG7 (7-bit fragments)
Field value: {0000101}, {1010001}, {1100010}, {01} x.X
1
2
3
4
SIP7
BV EV
5 1000 00000
81 0100 00000
98 0010 00000
SIP2
EV
1 00010

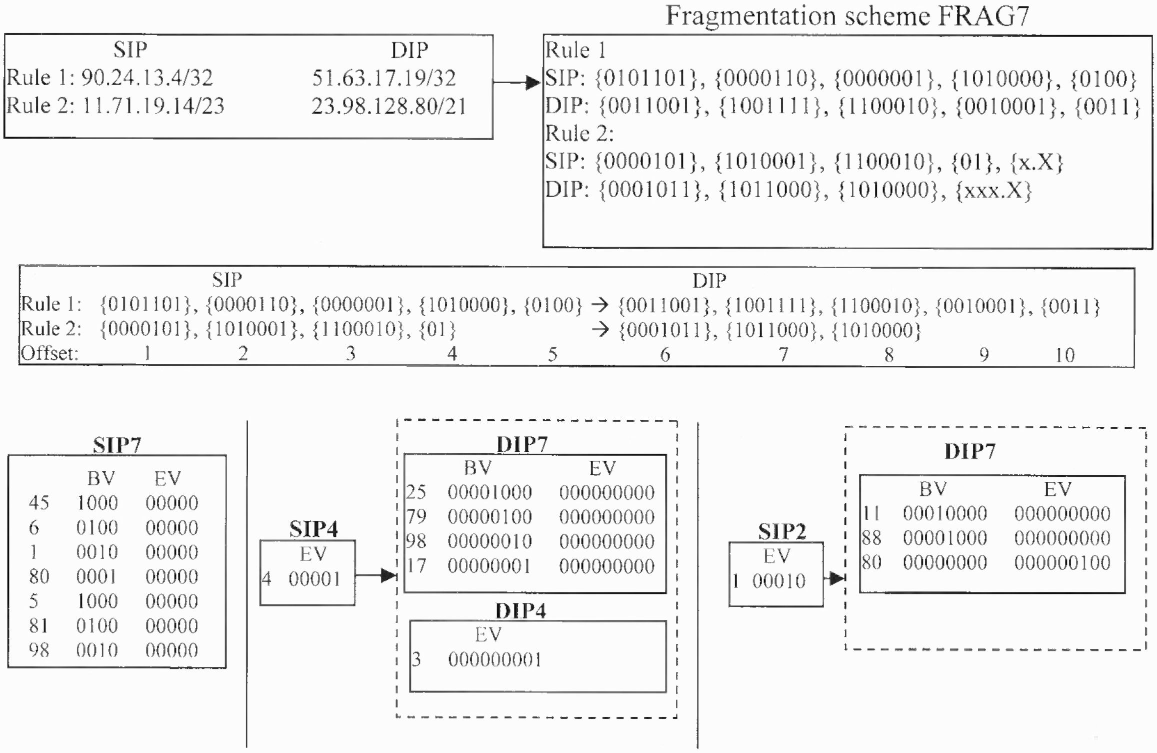
Figure 4.1 Applying the two fragmentation schemes FRGA8 and FRAG7.

For a given fragmentation scheme, say FRAG8, these BV and EV vectors are stored in separate tables corresponding to their lengths. The BV and EV vectors of SIP field for fragments of 8 bits are stored in table SIP8. Similarly, EVs of tail fragments having i bits are stored in table SIPi, where i=1, 2, …, 7. SIP7-SIP1 only contain EVs since these fragments will appear only as tails. This same procedure is carried out for the FRAG7 fragmentation scheme that splits fields into 7-bit fragments and creates its own SIP7-SIP1 tables. For FRAG7, SIP7 contains both BVs and EVs, and the rest of the tables contain only EVs. Similarly DIP, SP and DP fields are fragmented and the respective DIPi, SPi and DPi tables are created. As remaining rules are fragmented, the BV-EV vectors in the tables are appended and/or updated. Distinct tables are created for the distinct groups G15-G1.

4.2.3 Pairing of Fields

For every rule during pre-processing, operations are performed on all field pairs using both fragmentation schemes. The various pairings for the G15 group are: SIP-DIP, SIP-SP, SIP-DP, DIP-SP, DIP-DP, SP-DP. Similarly, for G3 there is only one pair SP-DP, and so on. Groups like G1, G2, G4 and G8 which has only one valid field and hence they are not paired and will be looked up directly using their BV-EV and summation tuples. The pairing process is now looked at with an example. For simplicity, the SIP and DIP fields are considered for rules 1 and 3 only. Both rules belong to G15, thus their BVs and EVs per pair are stored in common tables. The information regarding field pairing is stored in the SIP-DIP tables as follows. The BV and EV vectors for the first field, SIP, is generated as discussed before. The BV and EV vectors for the second field are calculated by combining the fields to get the correct offset values for the second one. For every table corresponding to the first field, there will be up to 8 tables under FRAG8 and 7 tables under FRAG7 that can contain information about the second field. This is because in a pair, the first field can contain 1 to 8 fragments (for FRAG8); thus, the information regarding the second field in the pair is stored in the respective table set (i.e., the bit length with which the first field had ended). For the pairing of SIP and DIP, the tables are denoted in the second field as SIP1-DIP to SIP8-DIP for the FRAG8 scheme and SIP1-DIP to SIP7-DIP for FRAG7, where SIP1 indicates information (BV, EV) about the first field that ended in a 1-bit fragment and the SIP1-DIP tables contain the information about the DIP field rules for which the first field (SIP) ended in a 1-bit fragment. In some cases the tables may be empty, thus not requiring any memory for storage. Fig. 4.2 shows the SIP and SIP-DIP tables for rules 1 and 3 with FRAG7. The first field of rule 1 ends in a

4-bit fragment and, hence, the DIP information of the second field in rule 1 is stored in the SIP4-DIP tables which are made up of DIP7 and DIP4 (since there are valid fragments of length 7 and 4 in DIP). Similarly the first field in rule 2 ends with a fragment of 2 bits and, hence, the DIP information for rule 2 is stored in the SIP2-DIP tables. SIP7 does not have a corresponding DIP table because none of the two rules considered has a SIP field ending with 7 bits. Since FRAG7 is illustrated in this figure, the BV vector of DIP in the second field can start at offset 2 and can go up to offset 9. Thus, the BV vector for the second field will contain 8 bits. Similarly the EV offsets vary from 2 to 10 and, hence, will have 9 bits. The tables are updated with new information accordingly as and when more rules are considered.



	SIP	DIP
Rule 1:	90.24.13.4/32	51.63.17.19/32
Rule 2:	11.71.19.14/23	23.98.128.80/21

Fragmentation scheme FRAG7

Rule 1
SIP: {0101101}, {0000110}, {0000001}, {1010000}, {0100}
DIP: {0011001}, {1001111}, {1100010}, {0010001}, {0011}
Rule 2:
SIP: {0000101}, {1010001}, {1100010}, {01}, {x.X}
DIP: {0001011}, {1011000}, {1010000}, {xxx.X}

	SIP						DIP				
Rule 1:	{0101101},	{0000110},	{0000001},	{1010000},	{0100}	→	{0011001},	{1001111},	{1100010},	{0010001},	{0011}
Rule 2:	{0000101},	{1010001},	{1100010},	{01}		→	{0001011},	{1011000},	{1010000}		
Offset:	1	2	3	4	5		6	7	8	9	10

SIP7

	BV	EV
45	1000	00000
6	0100	00000
1	0010	00000
80	0001	00000
5	1000	00000
81	0100	00000
98	0010	00000

SIP4

	EV
4	00001

DIP7

	BV	EV
25	00001000	000000000
79	00000100	000000000
98	00000010	000000000
17	00000001	000000000

DIP4

	EV
3	000000001

SIP2

	EV
1	00010

DIP7

	BV	EV
11	00010000	000000000
88	00001000	000000000
80	00000000	000000100

Figure 4.2 Pairings and BV-EV vector generation for the SIP and DIP fields using fragmentation scheme FRAG7.

4.2.4 Port Ranges

Since the fields are represented as patterns, the port ranges are converted into the prefix format. For example, if there is a rule with the SP specified as the range 0:1024 then this is represented as 0000000000000000:0000010000000000, which is transformed into the two prefix patterns {000000xxxxxxxxxx} and {0000010000000000}. The procedure of creating the BVs and EVs under the FRAG8 and FRAG7 fragmentation schemes is then carried out in the same way as previously shown. The pairing process on fields is also carried out as discussed above. This pairing process may result in the duplication of some rules which will force storage of multiple summation tuple lookups per rule (explained later) but as seen later in the results, the effect of such kind of duplication is not much on the memory consumption.

4.2.5 Choice of Fragmentation

Two fragmentation schemes are used because they collectively help in filtering out false detection alerts during the verification phase that employs a pre-calculated lookup table. Although fictitious matches may still be possible, most of them are normally filtered out.

The choice of 7 and 8 bits for simultaneous fragmentation was made based on experimental results. As shown in Fig. 4.3, any number of bits lesser than 7 for fragmentation increases the time in clock cycles for the next packet to be inputted into the system, which directly affects the throughput. On the other hand, any choice greater than 8 bits almost doubles the memory usage. Thus, the 8-bit and 7-bit fragmentations present a good tradeoff between throughput and memory usage.

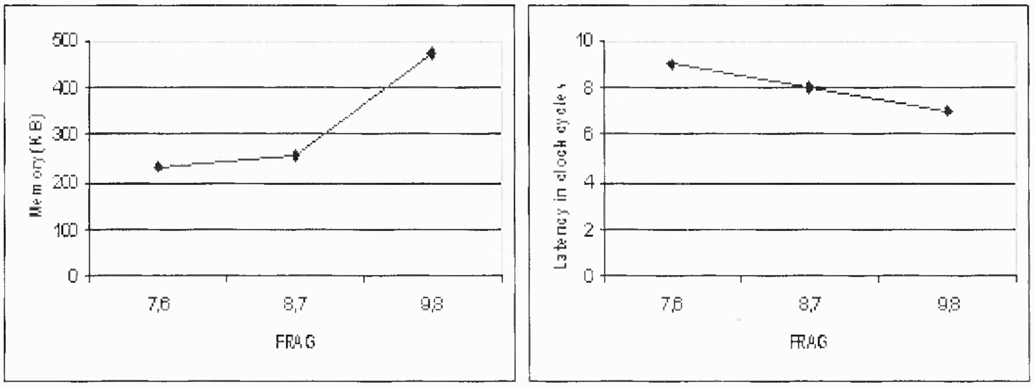
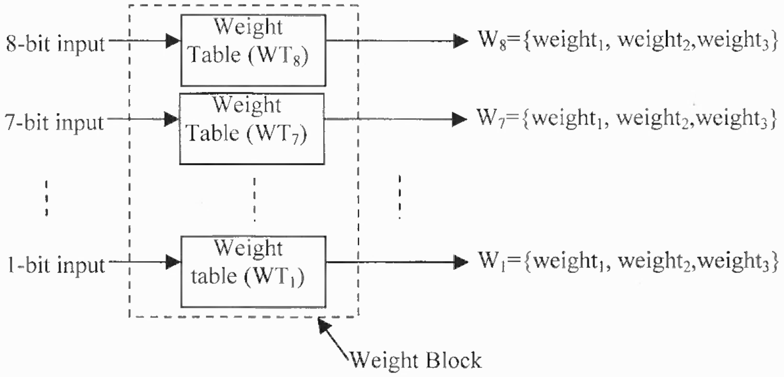


Figure 4.3 Comparison of various fragmentation schemes (pairs of bit choices are shown).

4.2.6 Summation Tuples

Once all of these operations have been performed on the rule-set, the tables containing the BVs and EVs will be available for every group. There is a separate block which contains unique weights for the field fragments ranging from 1 to 8 bits in length. There is one block per field and there are 8 different weight tables WT_1 to WT_8 in each block as shown in Fig. 4.4. The weight block takes one byte of input for addressing and every weight table picks up the appropriate number of most significant bits of the byte. Every value in the weight table is assigned a random m-tuple of weights represented by vector $W = \{weight_1, weight_2, \ldots, weight_m\}$; let $bw_1, bw_2, \ldots, bw_m$ be the respective number of bits in each weight element. The objective here is that summation of all the bws' should be such that we can create an exclusive weight tuple for every fragment in a table and also give some leeway for generating an exclusive summation value per rule. Since 8-bit per fragment is used, the summation of bws should be a minimum of 8 bits (for all the 256 possible fragments) but a leeway of 2 additional bits is kept in every table. This makes it 10 bits per element in the WT_8 table with a distribution of 4 bits, 3 bits and 3 bits for bw_1, bw_2 and bw_3 respectively. m=3 is chosen since this gives the pre-process phase a good control over weight tuples and also any number greater than 3 will then need more adders and comparators as will be inferred later during the description of the design.


8-bit input
Weight Table (WT_8)
W_8={$weight_1$, $weight_2$,$weight_3$}
7-bit input
Weight Table (WT_7)
W_7={$weight_1$, $weight_2$,$weight_3$}
1-bit input
Weight table (WT_1)
W_1={$weight_1$, $weight_2$,$weight_3$}
Weight Block

WT_8 {bw_1=4, bw_2=3, bw_3=3}

Value	$weight_1$	$weight_2$	$weight_3$
0	1	2	3
1	7	4	5
.	.	.	.
.	.	.	.
255	15	2	5

WT_7 {bw_1=3, bw_2=3, bw_3=3}

Value	$weight_1$	$weight_2$	$weight_3$
0	2	2	2
1	3	1	5
.	.	.	.
.	.	.	.
128	1	6	3

WT_1 {bw_1=3, bw_2=3, bw_3=3}

Value	$weight_1$	$weight_2$	$weight_3$
0	1	3	1
1	1	1	2

Figure 4.4 Weight tables.

Using these weight tuples a *summation m-tuple* is calculated for each field in the rule using 8 bits per fragment; tails can have less than or equal to 8 bits. Consider rule 10 in Table 4.1 with the two valid fields {11.23.131.145/29; 0:1023}. As seen earlier a port range of 0:1023 turns into a pattern "000000X" with the most significant 6 bits being common per range and the rest being don't cares. The summation m-tuple is calculated for this rule at static time in the following steps:

1) Split the fields into 8-bit fragments assuming FRAG8:

Field 1: "00001011" "00010111" "10000011" "10010";

Field 3: "000000"

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

SUM(Field1) = WT_8(“11”) + 2 * WT_8 (“23”) + 4 * WT_8 (“131”) + 8 * WT_5(“18”) ;

SUM(Field3) = WT_6 (“0”) ;

The weight-tuple tables for every field are different and can have different weight tuples for the same fragments since the exclusivity principle is on per table basis.

3) Once the individual summation m-tuples per field are calculated, they are summed up along with the protocol value (in binary) of the rule to obtain the final summation m-tuple for the rule.

SUM(rule 10) = SUM(Field1) + SUM(Field3) + Protocol

This summation method is applied to all the rules. The summation tuples act as addresses for table lookups in relation to the incoming packet. Fig. 4.5 shows the chosen summation tuples for the rules in Table 4.1.

	Sum_1	Sum_2	Sum_3
Rule 1:	108	88	129
Rule 2:	175	121	144
Rule 3:	85	46	91
.	.	.	.
.	.	.	.
.	.	.	.

Figure 4.5 Summation tuples for the rule-set in Table 4.1; SUM= (Sum_1, Sum_2, Sum_3).

Once the summation m-tuples of a rule are pre-calculated, they are stored along with the rule ID and the action to be taken for the rule in a table at a location which is produced by a hash function on this summation m-tuple. For the example rules set in

Table 4.1, up to 16 distinct tables are created to deal with 16 groups of rules. However, since some rules of certain groups are lesser in population than others, storage for their summation m-tuples can be combined into one table instead of having separate tables for each of these groups. Also, if the protocol field for the rule is 'any,' which means that the rule must be triggered independent of the protocol, then the summation m-tuples are stored without using the protocol field in a separate table that is accessed simultaneously. The weight m-tuples are assigned in such a way that the summation m-tuples do not produce false positives. The weight block is independent of the FRAG blocks and works independently of them. Also, there is no pairing or grouping of fields in relation to this block. Its purpose is to generate summation m-tuples for all possible bit lengths per field.

4.3 Runtime Rule Matching

The information about the position of the individual fragments in a rule pair is encoded in the vector tables, as shown in Fig. 4.2. The summation m-tuples for every rule are also stored in the tables separately. Now a way is devised find out if the incoming packet fields match the rules by using the BV and EV vectors in the SIP-DIP tables and simultaneously creating an address; the latter will depend on the incoming packet header and will employ the weight m-tuples in such a way that it can point to the summation m-tuples in the summation tables. The incoming fields are forwarded to the fragmentation-based tables of every group containing the BV and EV vectors and also to the weight tables containing the weight tuples. The BV-EV operations are explained first which involve OR, AND, and SHIFT operation on the BV-EV vectors.

4.3.1 Group Block

Every Group explained before have individual pairing blocks. These pairing blocks perform the function of detection of the individual fields. Every pairing block consists of BV-EV tables (as explained before) and EQi blocks in which pairing detection takes place and where i= 1, …, 8 for FRAG8 scheme and i=1, …, 7 for FRAG7 scheme. The operations in pairing block are explained using the SIP-DIP pair (pairing block shown in Fig. 4.6) and FRAG8 scheme and FRAG7 scheme. EQ8 contains a DV(8) vector which detects the non-tail fragments of the pair in the incoming packet header and a EDV(8) vector which detects the tail fragment of the pair. DV(8) is initialized to "100…0" for SIP side of the EQ8 block and for the DIP block, it is initialized to "01111000" since the source IP field in the rule can be of the form s1x.x.x.x , s1.s2x.x.x, s1.s2.s3x.x or s1.s2.s3.s4x where s1, s2, s3 and s4 represent the prefix part of a source IP with a maximum of 8 bits in them, and hence the second field in a SIP-DIP pair can start from any of the 2nd to 5th position in the offset vectors. DV(8) has the same length as EV. The following equations are performed in a EQ8 block("&" represents concatenation and ">>" represents a right shift operation):

$DV(8)_{n+1} = (DV(8)_n \text{ AND } (BV(8) \& \text{'0'})) >> 1$; DV(8) is present only for SIP8

$EDV(8)_{n+1} = DV(8)_n \text{ AND } EV(8)$;

where n is a particular clock cycle. At every clock cycle, using a fragment of the input packet field the tables are accessed, and the corresponding BV and EV vectors are outputted into EQ8. Since the other tables contain only EVs, blocks EQ1 to EQ7 perform the following equations:

$EDV(i)_{n+1} = (DV(8)_n \text{ AND } EV(i)) >> 1$; for i= 1, 2, …,7.

DV(8) from EQ8 is outputted to all the blocks EQ1 to EQ7. The outputs of the EQ blocks are the EDVs. A non-zero EDV signals a potential match in the field. For example, in FRAG8 scheme if the EDV(8) of SIP is non-zero in the third clock cycle then it implies that a SIP prefix of length 24 bits is matched.

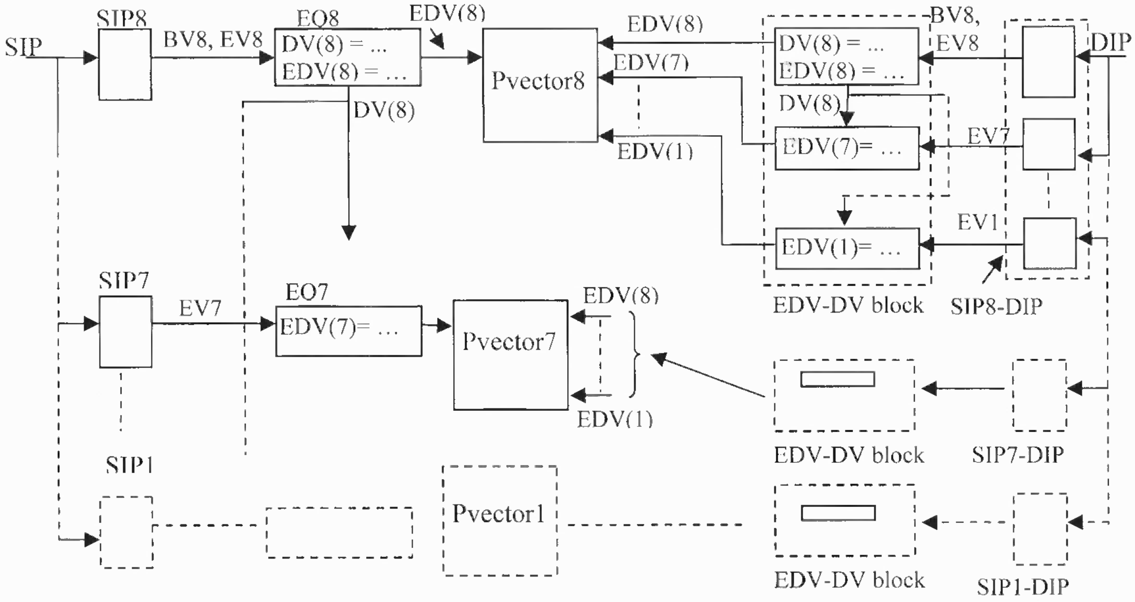
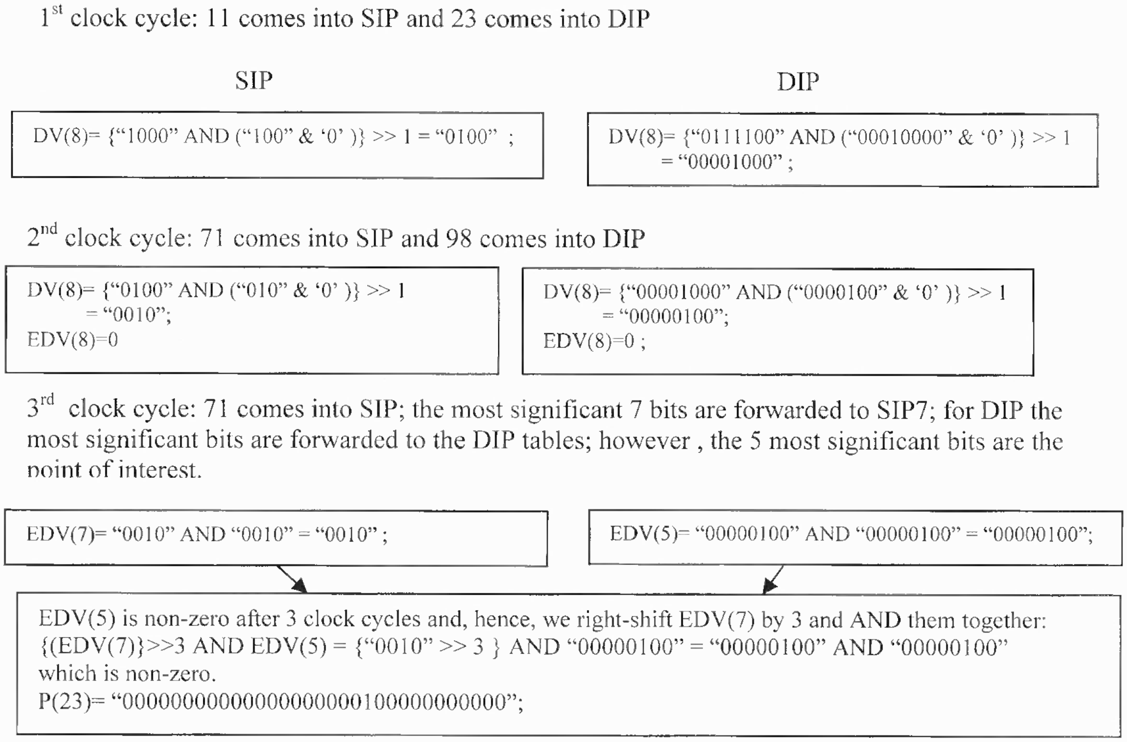


Figure 4.6 SIP-DIP block using FRAG8 (SIPDIP-FRAG8).

The Pvector blocks take in EDVs from EQ blocks, and generate Pvectors which are long bit vectors containing the position of the prefix matched. For example Pvector for SIPDIP blocks(for FRAG8 and FRAG7 schemes) will be 32 bits long. Likewise there will 32 different Pvectors P(1) to P(32) representing the 32 prefix bits of SIP. The prefix information of DIP part of the pair is represented in the 32 bits of the vector. For example if P(1) of SIPDIP has the 31st bit as '1' then it implies that there is a match in 1 bi of SIP and 31 bits of DIP.

The Pvector8 block shown in Fig. 4.6 generates four 32-bit vectors for the SIP prefixes of 8, 16, 24 and 32 bits in the first to fourth clock cycles; they are denoted by P(8), P(16), P(24) and P(32), respectively. Likewise Pvector7 block generates vectors P(7), P(15), P(23) and P(31) in the four clocks cycles. An example where a packet with source IP: 11.71.19.14 and destination IP: 23.98.128.80 arrives is now illustrated in the following description with the operations taking place in SIP-DIP-FRAG8 block. It can be seen from the rule-set that rule 3 has the first two fields of interest in the incoming packet. Fig. 4.7 shows the operations taking place in the SIP-DIP-FRAG8 block.



1st clock cycle: 11 comes into SIP and 23 comes into DIP

SIP

DV(8)= {“1000” AND (“100” & ‘0’)} >> 1 = “0100” ;

DIP

DV(8)= {“0111100” AND (“00010000” & ‘0’)} >> 1
= “00001000” ;

2nd clock cycle: 71 comes into SIP and 98 comes into DIP

DV(8)= {“0100” AND (“010” & ‘0’)} >> 1
= “0010”;
EDV(8)=0

DV(8)= {“00001000” AND (“0000100” & ‘0’)} >> 1
= “00000100”;
EDV(8)=0 ;

3rd clock cycle: 71 comes into SIP; the most significant 7 bits are forwarded to SIP7; for DIP the most significant bits are forwarded to the DIP tables; however , the 5 most significant bits are the point of interest.

EDV(7)= “0010” AND “0010” = “0010” ;

EDV(5)= “00000100” AND “00000100” = “00000100”;

EDV(5) is non-zero after 3 clock cycles and, hence, we right-shift EDV(7) by 3 and AND them together:
{(EDV(7)}>>3 AND EDV(5) = {“0010” >> 3 } AND “00000100” = “00000100” AND “00000100”
which is non-zero.
P(23)= “00000000000000000000100000000000”;

Figure 4.7 Detection example using FRAG8.

EDV(7) is non-zero in the 3rd clock cycle which means that the SIP value with prefix length 23 was found. To make sure that the second field was also found, since the information is stored in terms of pairs, the EDVs coming out of the DIP blocks are also looked into. It could be noticed that the EDV(5) value of DIP in the 3rd clock cycle is non-zero, implying that the non-zero EDV of SIP have to be right-shifted by 3 bits and then ANDed with EDV(5). This operation results in a non-zero vector. Thus, the match cane be represented by having the 21st bit of the P(23) vector as non-zero. The same set of operations are applied in the SIP-DIP-FRAG7 block; by ANDing the two vectors P(23) will be non-zero with the 21st bit set to '1'.

The pairing of blocks in individual groups is now looked into to show how it is arranged to make a decision about the complete rule. Let us take Group G13 where the valid fields are SIP, DIP and DP. Thus, the different table blocks will be SIP-DIP, SIP-DP and DIP-DP. The FRAG8 and FRAG7 fragmentation schemes are needed for the SIP-DIP tables whereas only one of the schemes for SIP-DP and DIP-DP is sufficient because SIP8-DIP8 (FRAG8) and SIP7-DIP7 (FRAG7) represent SIP and DIP using both schemes and hence the representation of DP7 and DP8 can be optimized using only SIP8-DP8 and DIP7-DP7, or SIP7-DP7 and DIP8-DP8. Fig. 4.8 shows the block diagram for the G13 pairings.

The intersection blocks for SIP-DIP blocks perform the AND operation of the vectors coming out of SIPDIP-FRAG8 and SIPDIP-FRAG7. The intersection block for SIP-DIP-DP finds the common bits in the 3 fields and forwards the values to the Address generation block (described later in this section) for selecting the summation tuples generated by the summation block. For example, if a particular incoming packet header matches 24 prefix bits of SIP, 23 prefix bits and 27 prefix bits of DIP simultaneously, and 16 prefix bits of DP in the FRAG blocks, then the intersection blocks will forward two vectors of 14 bits in length (since the SIP and DIP prefix length can be encoded in 5 bits each and the DP prefix length can be encoded in 4 bits) to the address generation block. The first vector will be the concatenation of ("10111", "10110", "1111") for SIP=24, DIP=23 and DP=16 and the second vector will be the concatenation of ("10111", "11010", "1111") for SIP=24, DIP=27 and DP=16.

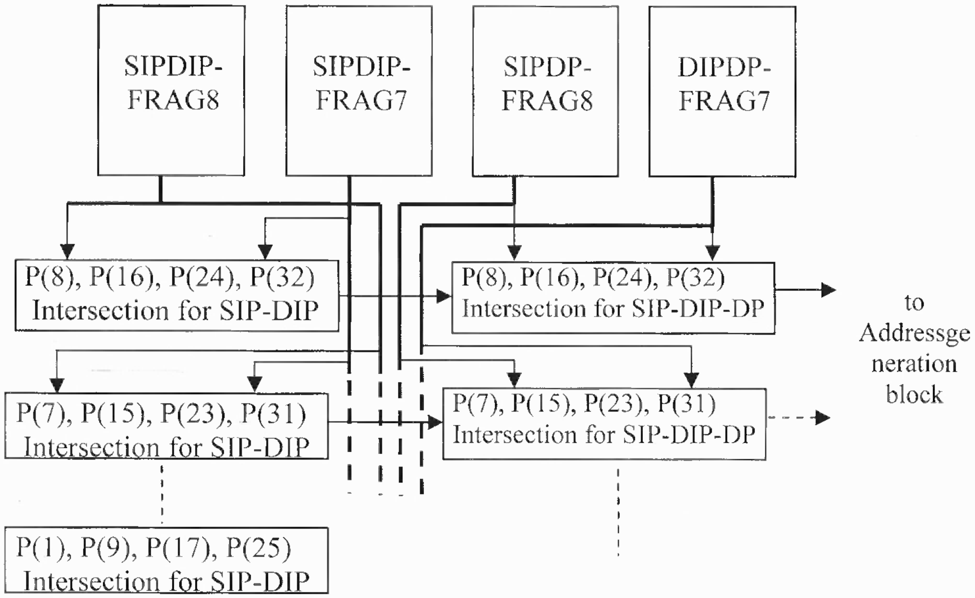

SIPDIP-FRAG8
SIPDIP-FRAG7
SIPDP-FRAG8
DIPDP-FRAG7
P(8), P(16), P(24), P(32) Intersection for SIP-DIP
P(8), P(16), P(24), P(32) Intersection for SIP-DIP-DP
to Addressge neration block
P(7), P(15), P(23), P(31) Intersection for SIP-DIP
P(7), P(15), P(23), P(31) Intersection for SIP-DIP-DP
P(1), P(9), P(17), P(25) Intersection for SIP-DIP

Figure 4.8 Pairings for Group G13.

4.3.2 Summation Block

The summation part of the detection takes place in the summation block. This block contains adders and shifters. The summation block pre-calculates the summation m-tuples in parallel while the BV and EV calculations are going on in the Group blocks, and stores them in temporary registers (SUM BANK block) in the various groups. Fig. 4.9 shows the summation block.

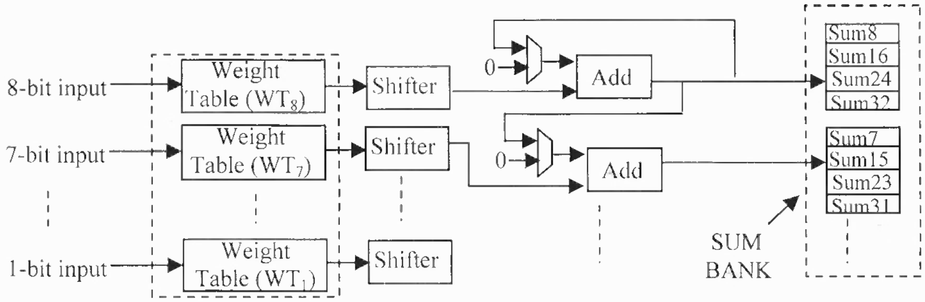


Figure 4.9 Summation block.

4.3.3 Detection Block

The detection of a rule in a packet header is performed by combining the Group block, summation block and the Address generation block.

Address Generation Block: It consists of the SUMF and SUMT blocks. Fig. 4.10 shows the address generation block for G13. The vector obtained from the intersection blocks is used to access the summation values from the SUM BANKs, which are added and hashed to access the summation m-tuples. The summation m-tuples table (SUMT) contains the summation m-tuples, the rule ID and the action to be taken on the packet. Every group forwards its best rule ID and eventually the best rule ID from all the groups is used to take action on the packet. Since the rules are ordered (i.e., the lowest ID represents the best matching), it is known which one to select. One of the summation m-tuples table is accessed directly without the protocol added to the summation, whereas the other table is accessed with the protocol field added.

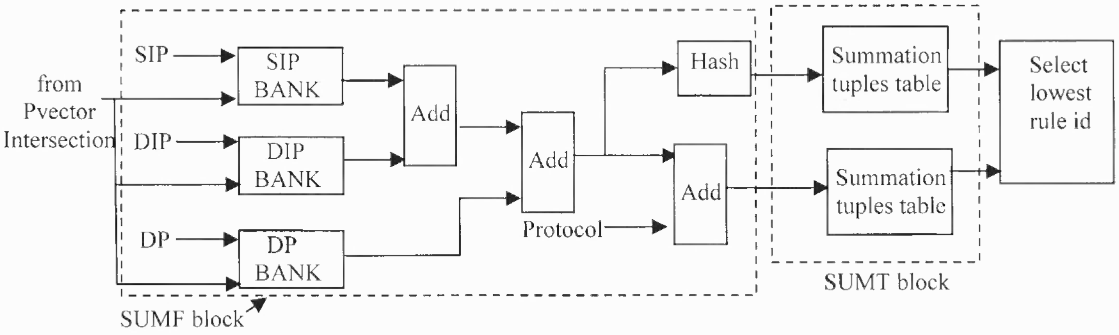


Figure 4.10 Address generation block for Group G13.

The concentration of rules is such that some groups will have more rules than others. Hence, some of the groups are combined and accommodated into common summation m-tuple tables. The final block diagram of the architecture for the packet classification is shown in Fig. 4.11. It can be seen in this diagram that G1-G12 have been clubbed together into one block. This is pertaining to the summation m-tuples (i.e., if the number of rules falling into these categories is very less then it does not warrant separate SUMF blocks). The architecture is flexible in terms of which groups need to implemented separately depending on the population of rules in the groups.

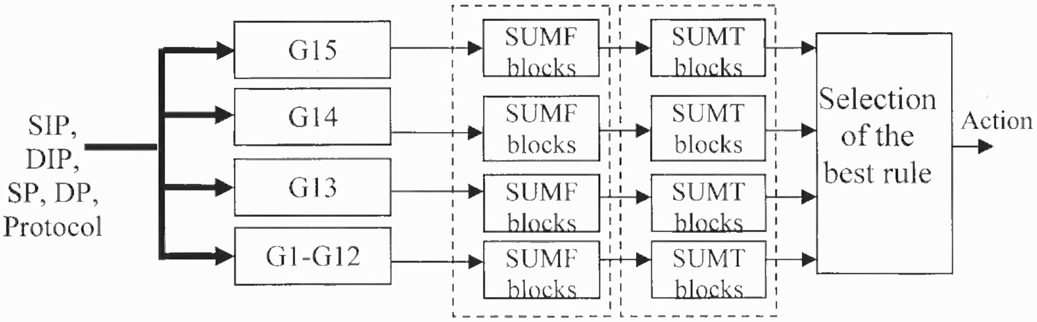


Figure 4.11 Block diagram.

4.4 Rule Splitting Method

The summation m-tuples are placed in the RAMs in such a way that there are no collisions. This is ensured by appropriately adjusting the weight tuples which in turn has an effect on the summation m-tuples. With appropriate allocation of bits per weight and the eventual summation m-tuples, the summation m-tuples can be placed in such a way that collisions are avoided. There are a maximum of four RAMs in a group that are accessed in parallel to check if there are matching m-tuples. The summation m-tuples are placed in these RAMs at locations which are a function of the summation m-tuples. The hash functions are simple XOR and modulo add functions which can be easily implemented in hardware. In cases where it is very difficult to place the m-tuples in non-vacant locations, rule splitting method is used. The data structure stored in this case is slightly different than the normal one. Consider the following rule (SIP: “123.4.5.89”; DIP: “135.6.7.0/23”; SP: 0:65535; DP: 1023:1023; Protocol: 6) with rule ID 1000. This rule falls in the G13 category. If this rule cannot be placed into the RAM, which means that the summation m-tuples for the rule do not point to a vacant location in the RAM, then the rule is split into at least two parts (can be more) and then linked together. For example, the aforementioned rule can be broken into (SIP: “123.4.5.89”; DIP: “135.6.7.0/23”) and stored into the G12 group, if possible without collision, and then DP:1023 is place separately into the G1 group. The information about the split is also stored. The summation m-tuples are placed in separate RAMs; if a match is found, then the other links are also looked into. For the above example, the summation m-tuples in the G12 RAM will be stored along with the rule ID and the group number of the second part of the rule (which is G1) and the new rule ID (which is assigned to the second part during the split). This rule splitting method could also be used to control the number of 1's in Pvector so that the pipelines in the design are not filled up, thereby having a positive effect on the throughput. The rules are easily spaced out into different groups by using a constraint on the number of 1's in a Pvector at any given time. This is done by generating a binary tree and checking the number of rules which can reflect a '1' in Pvector. There are different SUMT blocks for different sets of Pvectors, which makes it better since not more than five 1's in Pvectors were encountered for the rule set which was implemented for this dissertation.

4.5 Elimination of False Positives

The weight assignment process makes sure that any two genuine rules have different summation m-tuples. A false positive is then possible only if a fictitious combination of fragments (where at least one of the fragment is not part of the rule but is from a different rule) cause a non-zero EDV and at the same time generate a summation m-tuple which is same as one of the genuine rule. During pre-processing the population of rules in a group is further sub-divided and rules are clubbed together based on the length of the prefix in the field or fields to create an even distribution of summation m-tuples in the RAMs. An example is now looked at with a set of rules whose summation m-tuples are placed in the same set of RAMs. The possibility of false positive generation is investigated for two of the rules:

	SIP	DIP	SP	DP
1	97.166.41.112/32	182.125.194.192/32	0:65535	0:65535
2	27.26.30.130/32	127.206.10.2/32	0:65535	0:65535
3	……	…..	….	…

Assume an incoming packet with (SIP: 97.166.41.112) and (DIP: 127.206.10.2). This packet contains an SIP that matches the 1st rule and a DIP that matches the 2nd rule. The position-based encoding of the bit vectors for the two rules will produce a non-zero EDV because the header fragments will match partially more than one rule. If the summation m-tuple generated by this header is equal to some genuine summation m-tuple in the same set of RAMs, then the latter rule will be triggered. Thus, during pre-processing phase, all possibilities of such fictitious non-zero EDVs are checked and it is ensured that they do not generate a genuine summation m-tuple that will cause a false rule to be triggered. This is accomplished by either tweaking the weight m-tuples or applying the aforementioned rule splitting method. However, the majority of such cases are filtered out because of using simultaneously two fragmentation schemes (FRAG7 and FRAG8); a packet may match an incorrect rule under a scheme but it does not often match any rule under both the schemes simultaneously.

4.6 Experimental Results

Classbench [51] was used to generate rule-sets. Optimizations were performed in the implementation stage; for example, if a particular table had less than 8 members, then the use of RAM was avoided and only comparators were used instead to save the memory. If a particular group, say G15, had very few rules (less than 100), then the rule splitting method was used to separate the rules and place them in the corresponding groups instead of allocating a separate set of blocks for them (it would have consumed more resources). Fig. 4.12 shows the number of BRAMs needed for various numbers of rules. The BRAM consumption shown in the figure is only in GROUP blocks that store the bit vectors and not in the SUMT blocks. From the figure it can be deduced that the memory needed for storing bit vectors in the GROUPS flattens as the number of rules are increased, thus indicating that the memory needed for packet classification grows linearly with the number of rules.

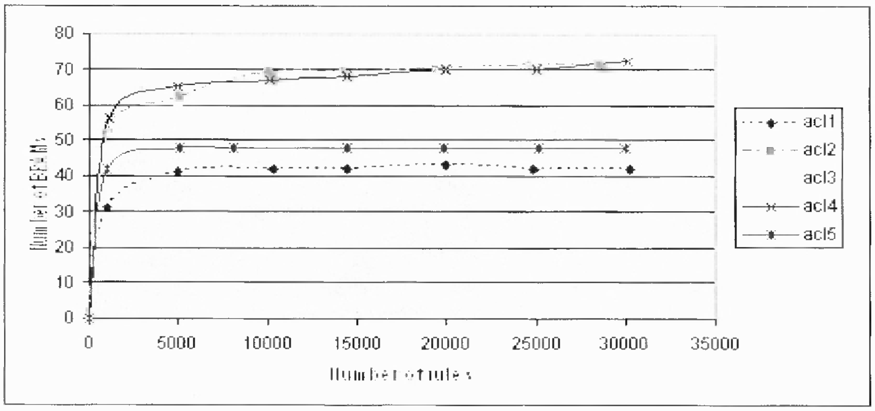


Figure 4.12 Number of BRAMs consumed by group blocks as a function of the rule population.

The design is a pipelined structure and has a worst case latency of 23 clock cycles. It takes 5 clock cycles to retrieve the EV-BV pair and 3 clock cycles for the DV-EDV calculations. A decision tree for the rules was created and it was found that there could be a maximum of five 1's in a Pvector; however, the real number is 3 since Pvectors are further separated into different SUMT blocks of summation tables which are independent of each other based on the prefix lengths. Still considering 5 as the worst case number, 7 clock cycles (5 + 2 FIFO latency) will be needed for the last non-zero Pvector to go into the summation selection block SUMF. This is then followed by add operations, hashing and RAM access, which take another 5 clock cycles. The final comparison of rule IDs take 3 clock cycles. Thus, the total latency for the worst case rule classification is 23 cycles. The Group block, SUMT block and the SUMF block work independently of each other and are interfaced with appropriate length FIFOs at their input. Hence when one of the block is working on a packet header the other blocks can work on a different packet header. Thus a new packet can be inputted into the design every 8 clock cycles.

Table 4.2 shows the memory consumption for different rule-sets created by Classbench [51]. The memory consumption is for the whole design and not for just some of the modules. Also, the number of rules due to port duplication is because as explained earlier if suppose there is a rule with port range 0:1024, then there will be two summations generated for the same rule, one with the port range 0:1023 easily represented with "000000x" and the second being 1024 represented with "0000010000000000". The hardware design was implemented using VHDL and was synthesized using Synplify Pro. The design was implemented for the acl3 file of Classbench[51]. It consumes 43,487 logic cells on a Virtex II Pro xc2VP70 and 46,450 flip-flops with 128 BRAMs. Only 151 Kbytes are used to store the rules and the bit vectors. 9, 7 and 7 bits were used for the three summation tuples, respectively, creating a total of 23 bits for summation. A total of 10 bits per three tuples of weight were used, using 4 bits for the 1st tuple and 3 bits each for the 2nd and 3rd tuple. The memory consumption is the least among all the earlier designs. The design can run at 242.1 MHz which is the highest operating frequency of all the designs that is known till now. The throughput for the design in the worst case (assuming 40 bytes per packet) is 9.68 Gbps.

Table 4.2 Results for various Classbench files

File	Number of rules	Number of rules due to port duplication	Number of BRAMs
acl1	10,246	13841	111
acl2	10,553	20144	141
acl3	10205	17106	133
acl4	10107	16184	127
acl5	8123	10433	108

Table 4.3 Comparisons with other works

Design	FPGA Device	Frequency	Number of rules	Memory (Kbytes)	Thoughput (Gbps)
[31]	Virtex-5	125.4 MHz	9603	612	80.23
[32]	Cyclone-3	128 MHz	10,000	286	3.41
[34]	Virtex-4	153 MHz	4000	178	1.88
[33]	Virtex II Pro	N.A.	128	221	16
The Packet Classification Method	Virtex II Pro	242.1MHz	10,205	151	9.68

Table 4.3 shows a comprehensive comparison of our design with others. It can be noticed that the design consumes the least memory even though it deals with more than 10,000 rules. In relation to [2], it can be noticed that although the design performs around 8 times slower, [2] uses dual-port memories with 407 Virtex-5 BRAMS which come in 36Kbit blocks. Since the Virtex-II Pro BRAMs come in 18Kbit blocks, [2] uses 814 18Kbit blocks whereas the packet classification design implemented in this dissertation use only 128 such BRAMs. The memory in the design is present in the Group blocks, and the SUMT blocks. As the number of rules grow more than 10,000, the memory curve for the Group block starts flattening out (Fig. 4.12) implying that the memory needed for storing BVs and EVs does not grow with the new rules. The only requirement of memory

for supporting new rules is for storing of new summation m-tuples. This also implies that the design memory increases only linearly with the growth in rule-set.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

Two novel designs for pattern matching with FPGAs have been proposed and implemented in this dissertation that can be utilized by NID systems. These approaches can also be exploited by other pattern-matching oriented applications, such as the detection of virus signatures. They are memory-oriented, high-throughput, compression-based designs that incorporate simple pattern detection techniques. Both approaches differ substantially from earlier approaches since they do not require long-distance routing of information inside the processing chip. Another major advantage of the two approaches is that they both support 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. The evaluation involved in both designs was comprehensive, involving a larger number of signatures than earlier approaches.

This dissertation has also proposed a novel method of packet classification which has the highest operating frequency among all known designs, and has a good balance between throughput and memory usage for more than 10,000 rules. It was shown that the memory consumption grows linearly with the number of rules which is a good indication of the design's scalability. Future work can focus on making this design capable of handling a new packet in every clock cycle which will further improve its throughput.

APPENDIX A

PSEUDOCODE FOR CALCULATION OF PVN4, PVN3, PV, DV AND 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 $\geq$ i)
//this if loop selects the appropriate BV and EV to be ANDed  with $DV_k$
    X= N-k+i;
```

```
// X variable stores the appropriate subscript of BV, EV to be ANDed with $DV_k$
    else
     X= i-k;
    end if;
    tempDV = tempDV OR ($DV_k$ AND $BV_X$);
 //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 $BV_X(0)$) $\neq 0$
// 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 $BV_X(v)$) $\neq 0$
    {
// 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 $EV_X(v)$) $\neq$ 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
```

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] C. Wu, S. Wen, N. Huang, and C. Kao, "A Pattern Matching Coprocessor for Deep and Large Signature Set in Network Security System," *IEEE GlobeComm,* 2005.

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] Y.H. Cho, S. Navab, and W.H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," *12th International Conference on Field Programmable Logic and Applications*, Montpellier, France, Sept. 2002, pp. 452–461.

[12] 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.

[13] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks, "Internet Worm and Virus protection in Dynamically Reconfigurable Hardware," *Military and Aerospace Programmable Logic Devices Conference*, 2003, p. E10.M.

[14] 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.

[15] 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.

[16] 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.

[17] SNORT® Open Source Network Intrusion Prevention and Detection System, http://www.snort.org

[18] H. Roan, W. Hwang, and C. Dan Lo, "Shift-Or Circuit for Efficient Network Intrusion Detection Pattern Matching," *International Conference Field Programmable Logic and Applications*, 2006.

[19] 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.

[20] 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.

[21] 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.

[22] G. Papadopoulos and D. Pnevmatikatos, "Hashing + Memory = Low Cost, Exact Pattern Matching," *Intern. Conf. on Field Programmable Logic and Application,* Aug. 2005, pp. 39-44.

[23] 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.

[24] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," *12th USENIX Security Symposium,* 2003, Vol. 12.

[25] 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.

[26] R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, 20(10), 1977, pp. 761–772.

[27] 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.

[28] 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.

[29] 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.

[30] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in Proc. Hot Interconnects VII, Aug. 1999.

[31] W. Jiang and V. Prasanna, "Large-scale wire-speed packet classification on FPGAs," Proc. of the ACM/SIGDA international symposium on Field Programmable Gate arrays, 2009

[32] A. Kennedy, X. Wang, Z. Liu, and B. Liu, "Low power architecture for high speed packet classification," in Proc. Architectures for Network and Communications Systems, 2008.

[33] G.S. Jedhe, A. Ramamoorthy, and K. Varghese, “A scalable high throughput firewall in FPGA” in Proc. Field-Programmable Custom Computing Machines(FCCM), 2008.

[34] A. Nikitakis and I. Papaefstathiou, “A memory-efficient FPGA-based classification engine” in Proc. Field-Programmable Custom Computing Machines(FCCM), 2008.

[35] D.E. Taylor and J.S. Turner, “Scalable packet classification using distributed crossproducing of field labels,” in Proc. INFOCOM, 2005.

[36] S. Singh, F. Baboescu, G. Varghese, and J. Wang, “Packet classification using multidimensional cutting,” in Proc. SIGCOMM, pages 213–224, 2003.

[37] S. Dharmapurikar, H. Song, J.S. Turner, and J. W. Lockwood, “Fast packet classification using bloom filters,” in Architectures for Network and Communications Systems, pages 61–70, 2006.

[38] I. Papaefstathiou and V. Papaefstathiou, “Memory-efficient 5D packet classification at 40 Gbps,” in Proc. INFOCOM, pages 1370–1378, 2007.

[39] I. Sourdis, “Designs & Algorithms for Packet and Content Inspection,” PhD thesis, Delft University of Technology, 2007.

[40] H. Song and J.W. Lockwood, “Efficient packet classification for network intrusion detection using FPGA,” In Proc. FPGA, pages 238–245, 2005.

[41] W. Eatherton, G. Varghese, and Z. Dittia, “Tree bitmap: hardware/software IP lookups with incremental updates,” In SIGCOMM Comput. Commun. Rev., 34(2):97–122, 2004.

[42] F. Baboescu and G. Varghese, “Scalable packet classification,” in Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pages 199–210, 2001.

[43] T.V. Lakshman and D. Stiliadis, “High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching,” in Proceedings of ACM SIGCOMM, pages 191-202, September 1998.

[44] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, “Algorithms for advanced packet classification with ternary CAMs,” in Proc. SIGCOMM, pages 193–204, 2005.

[45] F. Yu, R.H. Katz, and T.V. Lakshman, “Efficient multimatch packet classification and lookup with TCAM,” in IEEE Micro, 25(1):50–59, 2005.

[46] M.H. Overmars and A.F. Van der Stappen, “Range searching and point location among fat objects,” in Journal of Algorithms, 21(3), pp. 629-656, November 1996.

[47] P. Gupta and N. McKeown, “Algorithms for packet classification,” in IEEE Network, vol. 15, pp. 24–32, Mar/Apr 2001.

[48] D.E. Taylor, “Survey and taxonomy of packet classification techniques,” in ACM Comput. Surv., vol. 37, no. 3, pp. 238–275, 2005.

[49] P. Tsuchiya. “A search algorithm for table entries with non-contiguous wildcarding,” unpublished report, Bellcore.

[50] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. “Fast and Scalable Layer Four Switching,” in Proc. of ACM Sigcomm, pages 203-14, September 1998.

[51] D. Taylor and J. Turner, “ClassBench: A Packet Classification Benchmark”, in IEEE Infocom’05, March 2005.