Efficient Packet Classification for Network Intrusion Detection using FPGA *

Haoyu Song
Department of CSE
Washington University
St. Louis, USA
hs1@arl.wustl.edu

John W. Lockwood
Department of CSE
Washington University
St. Louis, USA
lockwood@arl.wustl.edu

ABSTRACT

FPGA technology has become widely used for real-time network intrusion detection. In this paper, a novel packet classification architecture called BV-TCAM is presented, which is implemented for an FPGA-based *Network Intrusion Detection System* (NIDS). The classifier can report multiple matches at gigabit per second network link rates. The BV-TCAM architecture combines the *Ternary Content Addressable Memory* (TCAM) and the *Bit Vector* (BV) algorithm to effectively compress the data representations and boost throughput. A tree-bitmap implementation of the BV algorithm is used for source and destination port lookup while a TCAM performs the lookup of the other header fields, which can be represented as a prefix or exact value. The architecture eliminates the requirement for prefix expansion of port ranges. With the aid of a small embedded TCAM, packet classification can be implemented in a relatively small part of the available logic of an FPGA. The design is prototyped and evaluated in a Xilinx FPGA XCV2000E on the FPX platform. Even with the most difficult set of rules and packet inputs, the circuit is fast enough to sustain OC48 traffic throughput. Using larger and faster FPGAs, the system can work at speeds greater than OC192.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special Purpose and Application-based Systems

General Terms

Algorithms, Design, Security

Keywords

Reconfigurable Hardware, FPGA, Packet Classification, NIDS, TCAM, BV, Tree Bitmap

*This work was funded by a grant from Global Velocity. John Lockwood serves as a consultant to the company.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'05, February 20–22, 2005, Monterey, California, USA.
Copyright 2005 ACM 1-59593-029-9/05/0002 ...$5.00.

1. INTRODUCTION

Network intrusion detection systems that protect high-speed computer networks demand both high throughput and flexibility to handle new threats. Such systems classify Internet packets based on both the header fields and the strings in the packet content or traffic flow. FPGA technology is desirable since it offers both high performance and the ability to reconfigure.

Packet header classification is an integrated part of a full-featured NIDS. Rules in an intrusion detection database usually contain 5-tuple header filters (i.e. Source IP Address, Destination IP Address, Protocol, Source Port and Destination Port) plus some strings (also known as "*signature*").

In a full-featured NIDS rule database, signatures can have variable lengths and be located at any offset in the packet. Packet header fields, however, are constant in length and appear at fixed location in the packet. Though in high level we can see content string matching is another dimension of classification problem, existing packet classification algorithms are not easy to be extended to handle the string matching. Due to the different nature of strings and packet header fields, it is desirable to separate the header classification process from the string matching process. A cross-product of the two results can be used to determine a complete rule match.

A signature, however, might indicate a potential attack only in a specific context defined by the header fields. Matching the signatures independently of the header can greatly reduce the system's performance. In fact, if the performance of the cross-product is poor, attackers may overload the system by injecting worst-case traffic. The system can be vulnerable to a *Denial of Service* (DoS) attack or fails to catch the sneak attacks. For this reason, some software based NIDS match the header filters first then scan the content in the context. But the performance of these systems is generally poor due to the lack of the parallelism and inefficient data structure. *Snort* is a popular open source NIDS which uses signatures to detect malicious activities over the Internet [12, 1]. Unfortunately, this software based system cannot keep up with high speed networks. The system drops packets when the input traffic load exceeds the processing power of the CPU, on which the software runs. Within Snort, the incoming packet header compares against the header filters sequentially and then the packet payload compares against the signatures in the context sequentially. On the

other hand, in hardware, packet header classification and content scanning can be performed in parallel, which improves the overall system throughput. More efficient and hardware-oriented data structures can be used to accelerate the processing of each part.

For packet classification, some algorithms achieve high performance at the cost of high system complexity, high resource usage, or high power consumption. Some algorithms are very efficient in terms of resource usage but with poor processing throughput. Some algorithms are only suitable for the software implementations. Analysis of the characteristics of the NIDS database reveals ways to best exploit hardware parallelism and efficiently utilize the FPGA core components.

Snort is a popular open source NIDS which uses signatures to detect malicious activities over the Internet [12, 1]. Unfortunately, this software based system cannot keep up with high speed networks. The system drops packets when the input traffic load exceeds the processing power of the CPU, on which the software runs. One of the well-known signature-based rule sets is provided with Snort. Snort rules are contributed by the network security community. Most of the recently found network exploits can be extracted by experts as new signatures and be added to the Snort rule set promptly. The database of signatures has become very large and keeps growing. However, among the thousands of the Snort rules, there are only about 200 distinct header rules[1]. This reveals that the number of distinct header rules in NIDS is typically small comparing to the number of rules in a core router database. This occurs since NIDS are usually deployed at the edge of an enterprise network and used to protect the internal network form outside world. Though the size of header rule set is moderate, it is still prohibitive for a linear search. One more subtle point is that header classification for a NIDS needs to provide all the matches rather than just the highest priority one, because any header match may lead to a complete rule match. So we cannot apply priority-based algorithms that terminate without giving a thorough list of matching rules.

In this work, we detail the design of a packet classification architecture named BV-TCAM for network security applications in FPGA hardware. BV-TCAM uses a small embedded TCAM with programmable logic and block RAMs in a Xilinx FPGA. We show that it should be possible to port this design into a newer and larger FPGA for a full functional NIDS and work at over OC192 throughput.

The rest of the paper is organized as follows. Section 2 states the problem we intend to solve and set it in context. Section 3 reviews some related works that motivated our design. Section 4 describes our design in detail and Section 5 prototypes the design and evaluate its performance. Lastly Section 6 summarizes our contributions and concludes the work.

2. PROBLEM STATEMENT

The formal statement of the general packet classification problem is: There are k relevant packet header fields H_1, H_2, ..., H_k, where each field is a bit string and allows one of three kinds of matches: exact match, prefix match or range match. The header rule set contains a sequence of N rules R_1, R_2, ... R_N. Each rule is a combination of k header fields. A rule is said to match a packet if each field in the rule matches the corresponding field in the packet header in the specified way.

In many of the packet classification applications, an action occurs on the matched packet. For example, in a network router, the matching guides the forwarding decision. In a network firewall, packets are filtered or logged when a match is detected. But in network intrusion detection, a header match is not enough to identify a malicious packets. Further inspection needs to be conducted.

An example rule from Snort database is shown below:

> *alert tcp $EXT_NET any → $HOME_NET 53*
> *(msg:"DNS named version attempt";*
> *flow:to_server,established;*
> *content:"|07|version"; offset:12; nocase;)*

If the content signatures appears at specified location in any established TCP flow which is from outside network to any host's port 53 in local network, an alert message will be sent to the administrator. We can separate this rule to a header part and a content part. We represent this header rule as a 5-tuple string {*Source IP Address = any, Destination IP Address = internal network prefix, Protocol = TCP, Source Port = any, Destination Port = 53*}. It turns out that many rules may share a common Snort header rule. We classify the packet based on the header first, then use this information to guide further inspections of the packet content.

3. RELATED WORK

Using FPGA for network intrusion detection has become a hot topic in recent FPGA research [7, 5, 13, 6, 4, 3]. One compute-intensive task in NIDS is pattern matching. Most of the related work focused on the efficient pattern matching problem. However, packet header classification is another integral part of a full-featured NIDS.

Many algorithmic and architectural approaches have been proposed to classify packets. The software solutions are weak in terms of performance while the hardware solutions are overly complicated or costly to implement. It is still an open and challenging problem to find practical solutions. A good review of packet classification algorithms and architectures can be found in [16]. Here we focus on a particular method well suited for an FPGA implementation.

Ternary Content Addressable Memory (TCAM) is currently the most popular method for packet classification in practice. TCAM has the ability to store a "don't care" state in addition to a binary bit value. Input keys are compared with every TCAM entry in parallel. Given N distinct rules in a rule set, it only needs the $O(N)$ storage and performs in $O(1)$ lookup time. But there are issues related to the TCAM solution. TCAMs have low density and high power consumption. TCAMs also do not support direct range representation. In the header rules, the source port and destination port fields are usually defined as ranges. Although a range can be converted into a series of prefixes, this processing can greatly expand the rule set size. For example, in the worst case, a sub-range of a k bit field can be converted into $2(k-1)$ prefixes. The number of expansions is multiplied

[1]The Snort rules are represented in an abstract and compact way, so one such header rule may represent several rules in an implementation. But the total number of distinct header rules is still small and much less than the number of content rules

to be up to $4(k-1)^2$ when two port ranges are defined in the header fields. This means in the worst case a single rule may expand to 900 TCAM entries. Analysis on Washington University's CTS firewall rule set shows even a small set of 150 header rules explodes to 17000 different rules after the ports are transformed from range to prefixes. So direct use of a TCAM in this way is very inefficient.

There has been research on methods to match ranges more efficiently with TCAMs. Spitznagel *et al.* proposed a novel TCAM architecture called Extended TCAM (ETCAM) which supports direct range lookup [14]. A special logic circuit that performs the range check is appended after the normal TCAM cells. By only doubling the overall number of CMOS logic cells, it maintains the rule database in its original size.

A scheme proposed by Liu introduced a range mapping method for TCAM without expanding the rule set size [10]. Given the fact that the number of distinct port ranges is limited even in a large database, a bit vector is created for each range field. Each distinct range is assigned a bit position in the bit vector. So in a TCAM entry, besides the normal {value, mask} pair for IP addresses and protocol fields, two bit vectors present with the corresponding range bit set to 1 and all other bits set to "don't care". A lookup key translation table is also created for each port field. The table index is the port value; each entry in this table is a bit vector. The bit is set to 1 only if the port value is within the corresponding range. To perform a header classification, the packet ports values are used to lookup the key translation tables, then the outputs are attached to the other 3 fields to form the lookup key to TCAM. Though this scheme doesn't expand the number of TCAM entries, it does have cost: the number of bits for each entry is expanded proportionally to the number of distinct port ranges, which is not scalable. Secondly, this scheme needs two large translation tables with a size of $64K \times (\#of distinct ranges)$. This is so large that it will not fit in the embedded block RAMs available in most FPGA devices and it could not even be able to be implemented in off-chip SRAMs.

Yu *et al.* proposed another TCAM-based solution for intrusion detection [20]. They address the multi-match packet classification by preprocessing the header rule set to efficiently use the TCAM capacity: Firstly, they extend the rule set and add a corresponding memory. The TCAM uses the first match entry to retrieve all matches. Secondly, a method is shown to remove the negations without significantly expanding the rule set. Both steps only moderately expand the size of the rule set in practice but it cannot guarantee the worst case scalability.

The Parallel Packet Classification (P^2C) by Lunteren *et al.* [11] is also a TCAM-based algorithm. Each field of the header is encoded to less bits through preprocessing. For each header rule, code words of all fields are concatenated to form a TCAM entry. Though the total number of TCAM entries still equals the number of header rules, each entry needs much less bits than original header rule. This scheme is impressive for a large scale rule set, however, to assemble the TCAM lookup key, each header field must perform a single field search first to retrieve a code. The proposed solution also needs the port range expansion and performs tree based lookups. This tends to lower the system throughput.

Another practical packet classification algorithm often referred as Lucent Bit Vector (BV) was initially proposed by Lakshman *et al.* [8]. The BV scheme is targeted for hardware implementation. It decomposes the multiple header fields matching problem into several instances of single field matching problem. The idea is to search for rules that match each field of the packet header and represent the results as a set of bit vectors. Each rule is represented as one bit in every bit vector. If a header field matches the same field of a rule, the corresponding vector bit is set to 1 otherwise it remains 0. After all bit vectors are acquired, the rules that match the header can be obtained by intersecting the bit vectors. This scheme is simple in that it only use memory access and logic AND operation. If a binary search is used for each field, this scheme has an $O(logN)$ search time where N is the number of rules in the rule set but it needs $O(N^2)$ memory, which is large in practice. The authors implemented BV in an FPGA operating at 33 MHz and five 128 Kbyte Synchronous SRAM chips. The configuration supports up to 512 rules and processing 1 million packets per second in the worst case.

In the original BV algorithm, in the case where the number of rules is large, the bit vector is wider than the memory data bus causing a bit vector retrieval to require several sequential memory accesses. Baboescu *et al.* enhanced the BV idea and proposed an improved algorithm called Aggregated Bit Vector (ABV) [2]. They built a bit vector using the longest prefix search tree. Each bit vector is partitioned into k blocks. The natural block size is the largest number of bits that one memory access can fetch. An aggregate bit vector summarizes the bit vector which is stored along with the normal bit vector. If there is any 1 in a block, the corresponding bit in aggregate bit vector is set to 1; otherwise it remains 0. Upon lookups, the aggregate bit vectors for all the checked fields are ANDed first to get the intersection. Based on this smaller bit vector, only those blocks that contain any potential match are checked further. Preprocessing is needed to reorder the rules so that the 1s are denser in the bit vector, therefore the aggregate bit vector is more useful to reduce the number of memory access.

The BV and ABV algorithms are hardware-based. The bit vector lookups for different header fields can execute in parallel. For the 5-tuple header search, 5 groups of independent accessible memories are used. However, the searching time over each tuple is unbalanced: there are 32 bits in an IP address but only 16 bits in a protocol port. The prefix lookup for IP is 2 times slower in average than the lookup for port and thus affects the overall performance negatively. The design described in this paper handles this problem by using a hybrid architecture.

4. BV-TCAM ARCHITECTURE

Our design combines and optimizes the TCAM and Bit Vector algorithms for packet header classification in NIDS. As mentioned earlier, network intrusion detection systems require header classification to report all matches, not just one. In usual applications, TCAM is associated with a priority encoder than only reports the ID of the matched entry with the highest priority. In this application, we prefer an un-encoded TCAM. That is, the number of output bits equals the number of TCAM entries and each bit indicates the matching status of the corresponding TCAM entry. Just like the BV output, the Un-encoded TCAM output forms another bit vector and each bit in the vector indicates the match to the corresponding rule field(s) or not. So the idea

Table 1: Example Header Rule Set

ID	Source IP	Destination IP	Protocol	Source Port	Destination Port
1	any	192.168.0.0/16	tcp	$\geq$ 1024	2589
2	any	192.158.0.0/16	tcp	10101	any
3	any	192.168.50.2	tcp	any	443
4	192.168.0.0/16	any	udp	49230	60000
5	any	any	tcp	any	110
6	any	any	tcp	146	1000:1300

is that the header fields are partitioned in a way that some of them are classified using TCAM while the others are classified using Bit Vector algorithms. Particularly, we exclude source and destination port fields from TCAM while keeping IP address and protocol fields in TCAM. We order the rules in same sequence, hence we can intersect all the output bit vectors to get the set of matches. This method optimizes the size of the TCAM, as it does not expand the number of TCAM entries.

TCAM could be programmed using the three fields of an IP packet directly, but we do better by further savings of the expensive TCAM entries. We observe that several different header rules usually share the same address and protocol fields, so we can compress these rules into a single TCAM entry. A similar idea is used in [9] to optimize the hardware implementation of an irregular TCAM architecture. Extra logic is needed to decompress or map the TCAM output bit vector to a full size bit vector. In order to realize this idea, we sort the header rule set to group those rules together that share the identical address and protocol fields, and then label each rule with a global identifier in order. We only program the distinct first three fields into the TCAM. Each output bit of the TCAM is used to set or reset a group of D type registers. In this way, a full size bit vector is formed in which each bit corresponds to a header rule.

To better illustrate our design, a small set of example header rules is shown in Table 1. Note that the rules are already sorted as described above. The corresponding circuit that performs TCAM related partial classification is shown in Figure 1. Though this scheme is the fastest, it does not support incremental updates. Because of the fact that the header rule database update is much more infrequent than the actual lookup, the field-reprogrammable capability of the FPGA is utilized. Whenever a new update is needed, a new bit file can be generated then reconfigured into the FPGA. Faster incremental updates can be supported by attaching a memory entry to each TCAM output which encodes the partial decompressed vector. Since typically a packet may match a few TCAM entries, several memory accesses are needed to retrieve and decode the memory word to build a full bit vector. In our design we use the first method for simplicity and fast prototype.

We adopt the bit vector algorithms as described in ABV algorithm for port classification. Specifically, we build the port prefix lookup tree for bit vector searching. Since we can build a very wide memory data bus using on-chip Block RAM, we do not need to use the aggregated bit vector for this scale of problem[2]. To build the binary port prefix

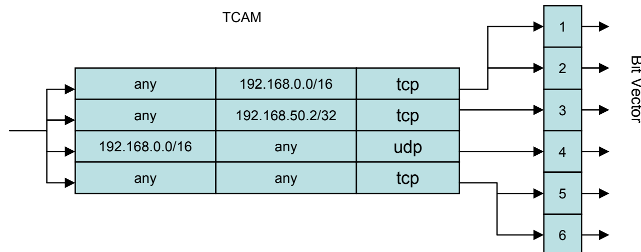

Figure 1: TCAM for Compressed Header Rules

lookup tree, each port's ranges are first transformed into a series of prefixes. All prefixes are then inserted into a binary decision tree. The branch decision at each level is decided by the bit pattern in the prefix. Each valid prefix node has a bit vector created. The bit vector indicates all the rules with its port definition matching to this prefix. The lookup procedure becomes a longest prefix matching problem for the packet port. Upon receiving a port value from a packet header for classification, The search is conducted by traversing the tree using the bits of the address, starting with the most significant bit. The search terminates when the bits are exhausted or a leaf of the tree is reached. The bit vector stored at the matched longest prefix node is retrieved. After we get all three bit vectors from the TCAM and the two longest prefix lookup trees, the set of matches can be determined by simply intersecting these bit vectors.

Here we still transform ranges to prefixes, however, we use them to build decision trees rather than a brute force TCAM programming. A decision tree uses cheap memories to store the data structure instead of expensive TCAM entries. The size of the decision tree is not as sensitive to the number of prefixes as the TCAM, because the size of TCAM increases linearly with the number of prefixes. In decision tree, additional prefixes may not change the size of the data structure at all if the prefixes can be mapped on existing tree nodes. Another important feature of decision tree is that the worst-case search time does not depend on the number of prefixes in set, but only depends on the depth of the tree.

Optimizations are possible to search for the longest prefix match. In order to speed up the lookup process, multi-bit trie schemes were developed which perform a search using multiple bits of the lookup bit string at a time. Controlled Prefix Expansion and Leaf Pushing are two important techniques for fast multi-bit trie lookup introduced by Srinivasan and Varghese [15]. These and other similar techniques are optimized for performance but have high memory consump-

[2]There are only a few hundred distinct header rules in Snort database. If the entire bit vector can be read in one memory access, there is no need to aggregate the bit vector. This can both improve the performance and eliminate the preprocessing effort. We will show this is the case in our design

tion. Unlike the IP lookup problem, where each valid trie node only stores the next hop IP information, our scheme stores a much wider bit vector. With limited resources, we desire a scheme which consumes less memory and could fit in an FPGA. The *Tree Bitmap* introduced by Eatherton and Dittia [18] works well for this problem. This technique avoids prefix expansion and leaf pushing while using a clever indexing scheme to dramatically reduce the memory penalty associated with a naive implementation. For each node in a multi-bit trie, Tree Bitmap algorithm uses an *Extending Paths Bitmap* to represent the subset of the potential children that are actually present, and an *Internal Prefix Bitmap* to represent the prefixes associated with the given node. Children of a node are stored in consecutive memory locations. Similarly, the next hop information associated with a node is stored in a group of consecutive memory locations. By counting the number of 1's in the bitmaps, the scheme allows the use of a single pointer to reference the children and next hop information. In our application, a bit vector is stored instead of the next hop information. The bit vector is much wider hence more resource consuming than the next hop information, but the total number of bit vectors is small, so this not a serious concern for our design.

Table 2: Example Source Port Prefixes Expansion

Prefix	Rule ID
*	3,5
0000 01** **** ****	1
0000 1*** **** ****	1
0001 **** **** ****	1
001* **** **** ****	1
01** **** **** ****	1
1*** **** **** ****	1
0010 0111 0111 0101	2
1100 0000 0100 1110	4
0000 0000 1001 0010	6

Using the example shown in Table 1, we transform the source port range to a series of prefixes as shown in Table 2. The corresponding multi-bit trie using Tree Bitmap is shown in Figure 2. The trie stride is 4. Each black dot represents a valid prefix and is virtually associated with a bit vector. The four valid prefixes contained in the root trie node are labeled in the *Internal Prefix Bitmap* and the four possible child branches are labeled in the *Extending Paths Bitmap*. Besides these two Bitmap vectors, 2 pointers are maintained pointing to the first prefix's bit vector and the first child node's address. All other trie node are structured in similar way. There is a slight difference in the deepest level trie node where there are no more extending paths present. Instead, each so called "extending path" is already an exact match. So the pointer for "extending path" here actually points to a base address where the bit vectors are stored. While splitting tree technique is proposed in [17] to deal with this issue, we apply this alternative to keep the design simpler and lower the resource consumption.

For example, if a packet with source port 2559 (09FF in Hexadecimal) needs to be classified, The root node bitmaps are checked which correspond to the first nibble of the port, "0000". So far the best match is "*" and meanwhile the extending paths bitmap indicates a possible child. This best match is latched and then the pointer is followed to the child

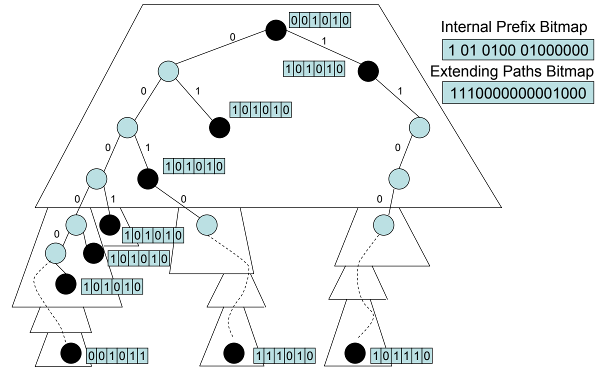

Figure 2: Example Tree Bitmap for Source Port

node. In the child node, there is no further possible child node for the second nibble "1001" and the best match prefix is "0000 1*** **** ****". The old best match is flushed and the new best match is the longest prefix match. The bit vector's address is calculated by the base pointer and the bit offset in the internal prefix bitmap. Finally, the bit vector "101010" is retrieved, which means the header rule 1, 3 and 5 are all matched. Whenever it is impossible to advance along the trie paths to find a new prefix match, the stored best match pointer is used to retrieved the bit vector.

5. IMPLEMENTATION AND EVALUATION

A full FPGA-based NIDS is under development which will implement a full-featured network intrusion detection system. While our solution is general enough to perform any kind of packet classification, we optimize our design to incorporate the Snort rule set. We show that only a small amount of memory and logic is needed to implement a circuit that achieves a fast header classification rate.

Specifically, we prototype the design in a Xilinx XCV-2000E FPGA. The block diagram of the circuit is illustrated in Figure 3. Our packet header rule set is extracted form Snort database V1.9.0[3]. There are a total of 222 unique header rules. Since the NIDS is usually deployed on the edge of the protected network and only monitors the pass-through traffic, we can logically change the source IP address from the external network to a wildcard "any" wherever the peer IP address is in the internal network. After this translation, the {*Source IP, Destination IP, Protocol*} combinations are successfully compressed to have only 33 distinct values. That means in the most compact way, we just need a $33 \times 72bits$ TCAM. This size of TCAM can be implemented using an embedded core on the FPGA without consuming too many resources.

The brute force implementation of TCAM using logic gates includes a set of registers and logic to perform parallel bitwise XOR and NAND operations. A 2-entry TCAM example is shown in Figure 4. In our design, we use Xilinx Core Generator to generate the TCAM component [19]. This TCAM core is comprised of multiple blocks of SRL16Es linked by carry-chains. The TCAM has a single-clock la-

[3]The Snort database is updated often and the latest version, V2.2.0, has been released. But we find that the size of the header rule set barely change, especially for the 3 fields that affect the TCAM. We believe our implementation of the BV algorithm can scale well to foreseeable Snort updates.

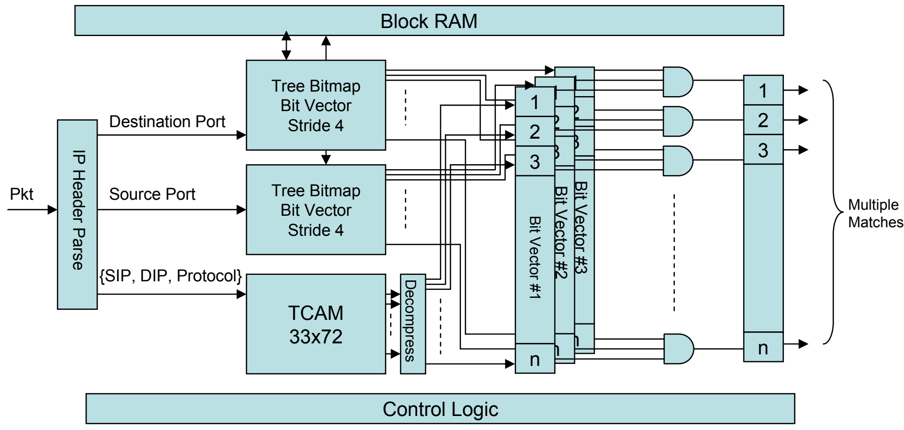

Figure 3: BV-TCAM Circuit Block Diagram in XCV2000E

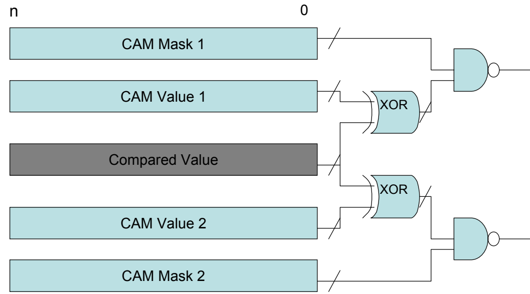

Figure 4: TCAM Logic in Hardware

tency on its read operation which is desirable for our high speed processing. We set the match address options to be "multiple-match unencoded" so that the output is the bit vector we need. The core uses 1188 SRL16Es which counts only 3% of the available SRL16Es in XCV2000E. With a larger FPGA like XC2V10000, the resource consumption for TCAM is as little as only 1%. This also implies that the TCAM scales to a reasonable larger size of header rule database.

For the multi-bit trie, the worst case memory consumption and lookup efficiency can be determined, given the stride of S and the trie degree of k. The worst case memory consumption happens when we have a complete trie with depth l. We define N as the total number of trie nodes and b as the total number of bits to store the node data structure, except the memory for bit vectors. Since we store the trie data structure in limited on-chip Block RAM, it is critical to keep the size of the trie data-structure as small as possible under the throughput constraint. Each trie node maintains an "Internal Prefix Bitmap" with length $(k-1)$ and an "Extending Paths Bitmap" with length k. There are also 2 pointers and we assign 16 bits for each[4]. So the size of one node is

$$b = 31 + 2 \times k$$

[4]This means that we can support up to 64K prefixes and 64K trie nodes. This number of prefixes is more than enough for practical network intrusion detection systems. For example, in Snort's source port prefix trie, there are only 477 binary trie nodes in total and less than 100 distinct prefixes.

Since the length of port field is 16 bits, we know

$$k = 2^S \quad and \quad l = 16/S$$

thus

$$N = \frac{k^l - 1}{k - 1} = \frac{2^{16} - 1}{2^S - 1}$$

and

$$m = N \times b$$

In Table 3 we summarize the memory usage for different cases.

Table 3: Multi-bit Trie Data Structure

S	k	l	N	m
1	2	16	65,535	2,293,725
2	4	8	21,845	851,955
4	16	4	4,369	275,247
8	256	2	257	139,551

The larger the stride is, the less the memories are needed and at same time less memory accesses are needed to get the final bit vector. On the FPGA XCV2000E, there are 655,360 bits of Block RAM available. The multi-bit tries with stride 4 and 8 both satisfy the constraints. But the bitmap's length increases exponentially as the stride becomes larger, this in turn lowers the speed to calculate the addresses of bit vector or next trie node. For a real database with a smaller stride, the bitmaps is significantly sparser, so the real number of trie nodes is actually far less than the worst case estimation. In terms of the memory efficiency, a smaller stride is more favorable. As a tradeoff, we choose stride size 4 in our design. So in worst case, at most 4 steps walking in the trie are needed to obtain the bit vector.

Parsing the Snort header rules, we get 87 distinct prefixes for source port and 177 for destination port. We have 222 rules in total that means each bit vector is 222 bits long. So the total memories required to store the bit vectors are only 58,608 bits (i.e. 9% of the total available bits in XCV2000E). This is small enough to be hold in on-chip block RAM.

In Table 4, we give the total memory usage for the source port prefix trie under different strides, including the bit vectors.

Table 4: Source Port Prefix Trie Data Structure Size

Stride	*# of Nodes*	*Memory Usage(bits)*
1	477	35,055
2	256	29,298
4	143	28,323
8	85	65,469

Clearly, at stride 4, we achieve the optimal memory efficiency as well as a relatively fast lookup speed. In the worst case, at most 5 memory accesses are needed to retrieve a bit vector (i.e. 4 accesses to traverse the trie and 1 access to retrieve the bit vector). Destination port prefix trie hold the similar results. We assumes that in the implementation, each trie node uses a 64-bits word, each bit vector uses a 256-bits word and both can be read in one clock cycle by using fast on-chip Block RAMs.

An FPGA-based Tree Bitmap algorithm has been implemented at Washington University [17]. This circuit is called the Fast Internet Protocol Lookup (FIPL) search engine. Multiple FIPL engines can work together to improve the system throughput. We directly borrow this implementation and modify it to fit the BV algorithm. The major difference is that we use on-chip Block RAM exclusively to store all data structures and bit vectors. The resulting circuit consumes less than 1% of the available logic resources.

In an OC48 network, the worst case traffic pattern for packet classification occurs when the link is saturated with packets having the smallest length of 40 bytes. The packet arrival rate reaches $2.4G/(40 \times 8) = 7.5M/s$. For an FPGA that runs on a synchronous 100MHz clock; this gives 13 cycles to classify one packet. FIPL was designed for larger scale IP prefix lookup so the data structure was stored in off-chip memory. In our design, the data structure is small enough to be hold in on-chip Block RAM. This greatly improve the memory access efficiency since only a single clock cycle is needed to retrieve a memory word. A single FIPL engine for each port can satisfy our worst case throughput requirement. The whole circuit consumes less than 10% of the available logic and less than 20% of the available block RAMs.

With more advanced FPGA parts such as the V2Pro and the Virtex-4, we can achieve several times higher clock frequency and more use of additional memory and logic resources. By deploying more lookup engines and pipelining the design as described in [17], 10Gbps throughput can be achieved.

6. CONCLUSIONS

The BV-TCAM architecture efficiently classifies header rules for NIDS in an FPGA. The multi-match requirement sets this work apart from other general packet classification systems. Our major contributions have two aspects. First, while using the TCAM as a component, we avoid the need to expand the size of the rule set by only using TCAM to classify the fields that is represented as prefix or exact value. We further compress the number of entries needed in TCAM due to the fact that the number of distinct combined values of these fields is much less than the total number of rules. Second, after the port ranges are transformed to prefixes, we use a Tree Bitmap approach to implement the multi-bit trie Bit Vector algorithm. To the best of our knowledge, this is the first attempt to use it in Bit Vector algorithm implementation for packet classification.

Through the parallel operation and data structure size compression, the architecture is optimized for both throughput and storage efficiency. It is fit for straightforward FPGA implementation with fairly low system complexity. The circuit is sufficiently general to handle large scale packet classification problems. In this paper, we focused on the intrusion detection application. With the aid of the fast header classification, other deep packet inspection functions, such as the multi-pattern string matching, can benefit in terms of both lower false positive rate and lower processing overhead.

7. ACKNOWLEDGMENTS

The parsing and extraction of the Snort database performed by FPX group members allowed us to analyze the header rules. The implementation of the tree bitmap algorithms by Dave Taylor enables us to quickly prototype the BV-TCAM architecture. We also thank the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] Snort - The Open Source Network Intrusion Detection System. In *http://www.snort.org*.

[2] F. Baboescu and G. Varghese. Scalable Packet Classification. In *ACM Sigcomm*, San Diego, CA, Aug. 2001.

[3] Z. Baker and V. Prasanna. Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proceedings of FPL'04*, 2004.

[4] Z. Baker and V. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of FPGA'04*, 2004.

[5] Y. Cho and W. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read Only Memories. In *Proceedings of IEEE FCCM'04*, 2004.

[6] C. Clark and D. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *Proceedings of FPL'03*, 2003.

[7] B. L. Hutchings, R. Franklin, and D. Carver. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proceedings of IEEE FCCM'02*, 2002.

[8] T. V. Lakshman and D. Stiliadis. High-Speed Policy-based Packet Forwarding using Efficient Multi-dimensional Range Matching. In *ACM Sigcomm*, Sept. 1998.

[9] T. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu, and N. Dulay. Irregular Reconfiguration CAM Structures for Firewall Application. In *Proceedings of FPL'03*, 2003.

[10] H. Liu. Efficient Mapping of Range Classifier into Ternary-CAM. In *IEEE Symposium on High Performance Interconnects (HotI)*, Stanford, CA, Aug. 2002.

[11] J. V. Lunteren and T. Engbersen. Fast and Scalable Packet Classification. *IEEE Journal on Selected Areas in Communications*, 21:560–570, May 2003.

[12] M. Roesch. SNORT - lightweight intrusion detection for networks. In *13th Systems Administration Conference*, 1999.

[13] I. Sourdis and D. Pnevmatikatos. A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proceedings of FCCM'04*, 2004.

[14] E. Spitznagel, D. Taylor, and J. Turner. Packet Classification using Extended TCAMs. In *IEEE International Conference on Network Protocols (ICNP)*, 2003.

[15] V. Srinivasan and G. Varghese. Faster IP Lookups using Controlled Prefix Expansion. In *SIGMETRICS*, 1998.

[16] D. Taylor. Survey and Taxonomy of Packet Classification Techniques. *Tech. Report WUCSE-2004-24, Department of CSE, Washington University in St. Louis*, 2004.

[17] D. Taylor, J. Turner, J. Lockwood, T. Sproull, and D. Parlour. Scalable IP Lookup for Internet Routers. *IEEE Journal on Selected Areas in Communications*, 21:522–534, May 2003.

[18] W.N.Eatherton. Hardware-Based Internet Protocol Prefix Lookups. *Master Thesis, Washington University in St. Louis, http://www.arl.wustl.edu/*, 1999.

[19] Xilinx. Contend-Addressable Memory v4.0. *Xilinx Product Specification DS253 (v1.0)*, March 2003.

[20] F. Yu and R. Katz. Efficient Multi-Match Packet Classification and Lookup with TCAM. In *IEEE Symposium on High Performance Interconnects (HotI)*, Stanford, CA, Aug. 2004.