Computer Communications 109 (2017) 38–52

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

BitCuts: A fast packet classification algorithm using bit-level cutting

Zhi Liu[a,b,*], Shijie Sun[a], Hang Zhu[a], Jiaqi Gao[a], Jun Li[b,c]

[a] Department of Automation, Tsinghua University, Beijing, China
[b] Research Institute of Information Technology (RIIT), Tsinghua University, Beijing, China
[c] Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing, China

ARTICLE INFO

Article history:
Received 24 November 2016
Revised 21 March 2017
Accepted 1 May 2017
Available online 8 May 2017

Keywords:
Packet classification
High-performance
Decision-tree algorithm

ABSTRACT

Packet classification is one of the fundamental techniques required by various network management functionalities. As the state-of-the-art of software packet classification, decision-tree algorithms employ various geometrical cutting schemes to build optimized decision trees. However, existing cutting schemes cannot meet the desired performance on large rulesets because they sacrifice either classification speed or memory size by design. In this paper, we reveal the inefficiencies of current cutting schemes — equi-sized cutting and equi-dense cutting — and propose a new cutting scheme and its corresponding decision-tree algorithm, named "BitCuts". BitCuts achieves only 42%–59% the memory accesses of HyperSplit, HyperCuts and EffiCuts, the typical decision-tree algorithms. In addition, BitCuts accelerates child-node indexing with bit-manipulation instructions, enabling fast tree traversal. A DPDK-based evaluation on the ACL10K ruleset shows that BitCuts achieves 2.0x – 2.2x the throughput of HyperSplit, HyperCuts and EffiCuts. Furthermore, BitCuts is the only algorithm that achieves 10 Gbps throughput with 3 cores. The memory consumption of BitCuts is only 12% of HyperCuts, 19% of EffiCuts, and is comparable to that of HyperSplit, which proves that BitCuts outperforms existing algorithms in achieving a good trade-off between speed and space.

1. Introduction

With the rapid growth of Internet applications, both the volume and the variety of Internet traffic have increased dramatically. Therefore, traffic management has become challenging, since it is required to conduct fine-grained management for high-speed networks. In typical functionalities like access control list (ACL), quality of service (QoS) [1], firewall (FW), and intrusion detection and prevention (IDS/IPS), packet classification on multiple fields is the main technique.

Both software and hardware packet classification solutions have been intensively studied. TCAM (Ternary Content Addressable Memory) is the popular hardware solution since it compares the input header value with all TCAM entries simultaneously, and thus has the ideal O(1) search complexity. However, since ternary bit representation is less general than arbitrary range, packet classification rules with multiple range fields always result in substantial number of TCAM entries, and thus limit the rule number TCAM supports. Previous works address this problem by proposing various port range encoding schemes [2–5], rule compressing and rewriting techniques [6–8]. However, such techniques still lack the flexibility to deal with large number of rules or to support more packet fields.

On the other hand, software based packet classification solutions [9–19] have been proposed and have become a competitive alternative. As industry and academia explore the deployment of Network Function Virtualization (NFV), for both cloud [20] and carrier [21] scenarios, there is a high demand for network functionalities implemented by software. In such scenarios, packet classification is the key module that determines the throughput and latency [22].

To achieve fast classification speed, RFC[9] and HSM[10] adopt the idea of decomposition. These algorithms first independently search on each field, and carry out crossproducting through multi-phase lookup to generate the final match. Table search is implemented through indexing, which simplifies the complexity of parallel lookup into O(1) for single phase. As a result, the total lookup complexity is bounded by the number of phases, making such algorithms favourable in terms of lookup speed. However, such algorithms are still inefficiency in terms of memory size in that the size of the lookup tables at each phase could be very large.

* Corresponding author at: Department of Automation, Tsinghua University, Beijing, China.
E-mail addresses: zhi-liu12@mails.tsinghua.edu.cn, zhiliu.thu@gmail.com (Z. Liu), ssj13@mails.tsinghua.edu.cn (S. Sun), zhu-h13@mails.tsinghua.edu.cn (H. Zhu), gaojq12@mails.tsinghua.edu.cn (J. Gao), junl@tsinghua.edu.cn (J. Li).

http://dx.doi.org/10.1016/j.comcom.2017.05.001
0140-3664/

As the state-of-the-art of software packet classification, decision-tree algorithms solve the problem geometrically, by decomposing rule space recursively and building a decision tree. Typical decision-tree methods include HiCuts [11], HyperCuts [12], HyperSplit [13], ABC [14], Boundary Cutting [15], EffiCuts [16], etc. It is proved in [23] that constructing the optimal decision tree is NP-complete. Therefore, these algorithms employ various heuristic to choose a good cutting and trade-off between speed and memory. To further optimize the space consumption for decision trees, grouping [16,24,25] and data structure compressing [26,27] techniques are also studied. However, existing decision-tree algorithms still have inefficiency in their space decomposition schemes, resulting in redundant cutting or degraded classification speed.

This paper proposes BitCuts, a decision-tree algorithm that performs bit-level cutting. The contributions of this paper are summarized as follows: We study the cutting schemes of existing decision-tree algorithms, and reveal their inefficiencies in terms of speed and space. A new cutting scheme named BitCuts and its corresponding decision-tree construction algorithm are proposed. "Bit-level cut" is able to zoom into densely clustered rule space and cut at the right granularity, which avoids unnecessary partitions and excessive memory consumption. BitCuts uses parallel bit-indexing to support fast child-node traversal and enable large node fanout. For 5-tuple rules, the child-node indexing can be implemented by two bit-manipulation instructions, achieving ultra-fast decision-tree traversal. In order to build an efficient decision tree, a bit-selection algorithm is proposed to determine the cutting bits at each tree node, targeted at finding the most effective bits for separating the rules.

BitCuts achieves an optimized trade-off that obtains high classification speed and reasonable memory consumption. The evaluation shows that BitCuts outperforms HyperCuts and EffiCuts in both speed and space, with the same parameter configurations. A DPDK-based evaluation shows that BitCuts achieves about 2.0x – 2.2x the throughput of existing algorithms on large rulesets, and is the only algorithm that achieves 10Gbps throughput with 3 cores. The memory consumption of BitCuts is only 12% of HyperCuts, 19% of EffiCuts, and comparable to that of HyperSplit. Meanwhile, its memory consumption is less than 1MB for the ACL10K ruleset.

The remainder of the paper is organized as follows. Section 2 introduces related decision-tree algorithms. Section 3 reveals the inefficiency of the existing cutting schemes, based on a rule distribution study, and proposes the new cutting scheme "BitCuts". Section 4 gives the high-level overview of BitCuts and introduces the procedures of Bitcuts: offline preprocessing and online classification. Section 5 elaborates upon how the cutting bits are selected at each node during the tree construction, and the algorithm's trade-off between speed and space. The implementation details of BitCuts are illustrated in Section 6, and the evaluation results are shown in Section 7. Section 8 contains some discussions, and we draw conclusions and describe future work in Section 9.

2. Background

Given a group of pre-defined rules, the task of packet classification is to identify the matching rule for each input packet. Each rule R contains d header field specifications, $R[1], R[2] \ldots R[d]$, each written in prefix or range representation. Typical header fields include source IP (SIP), destination IP (DIP), transport layer protocol (PROTO), source port (SP) and destination port (DP), etc. An incoming packet matches the rule only if each corresponding packet header field $H[1], H[2] \ldots H[d]$ matches the rule specification $R[1], R[2] \ldots R[d]$. Each R also contains a *Priority* and an *Action*. If multiple rules match the input packet, the one with the highest *Priority* is returned and the associated *Action* is applied.

Table 1
Example ruleset.

Rule	Field X(Bit 0–2)	Field Y(Bit 3–5)
R_0	100	010
R_1	100	011
R_2	11*	011
R_3	111	001
R_4	10*	***
R_5	***	00*

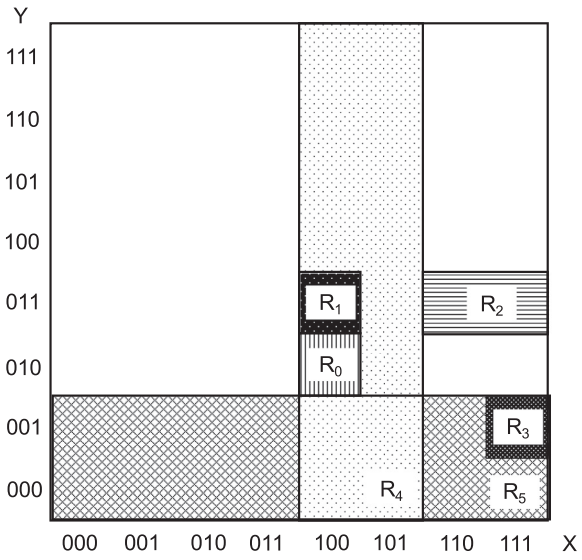


Fig. 1. Geometrical layout of rules in Table 1.

Most decision-tree algorithms solve the problem by conducting geometric "cuts" in the d-dimensional rule space. For those algorithms, each rule is taken as a d-dimensional hyper-rectangle, and each input packet represents a point in such a space. Table 1 gives an illustrative example ruleset with two fields, each represented by a 3-bit prefix. The corresponding geometric layout is shown in Fig. 1. Given the above problem formulation, existing decision-tree algorithms employ various preprocessing techniques to optimize the cuts and build the corresponding decision tree. Each "cut" decomposes the space into multiple partitions, and each of the partitions corresponds to a child node as well as the rules colliding with it. Two key metrics are widely used to measure the decision trees – memory access number and memory size. Specifically, the number of memory accesses required in the tree traversal is closely related to the online classification speed. Mostly, both number of average and worst memory accesses are used for the measurement. Meanwhile, the size of the decision tree is also an important consideration, which determines the possibility to fit into main memory and influences the cache miss rate. Therefore, the decision tree should require a limited number of memory accesses, as well as moderate memory size, so that it might fit into the cache.

The following subsections give an overview of typical decision-tree algorithms, including HiCuts [11], HyperSplit [13], HyperCuts [12] and EffiCuts [16].

2.1. HiCuts

HiCuts (Hierarchical Intelligent Cuttings) [11] is a seminal decision-tree algorithm for packet classification. To illustrate how HiCuts works on the example ruleset, Fig. 2a shows how the cuttings are conducted in the geometrical view, and the corresponding decision-tree is given in Fig. 2b. At each node, HiCuts cuts

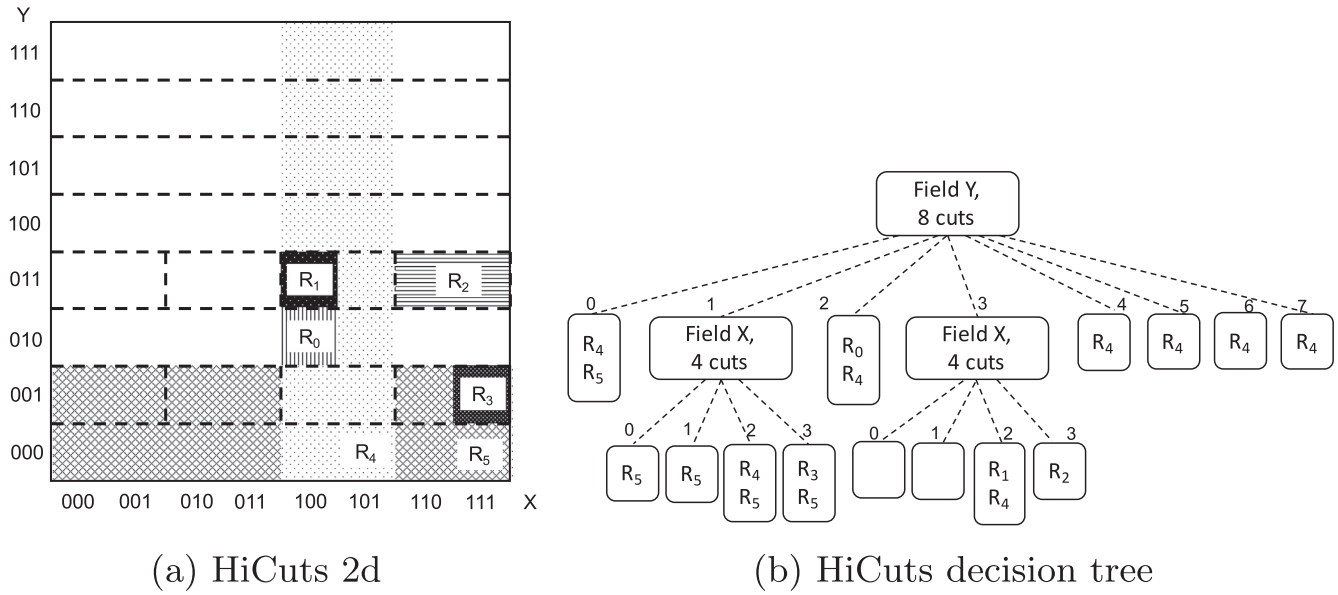


Fig. 2. Example of HiCuts.

the corresponding rule space into 2^k equal-sized partitions along a single dimension. As shown in Fig. 2b, the root node represents the whole space, and cuts the space into 8 partitions along Field Y. Each of the 8 child nodes represents one of the partitions as well as the rules colliding with it. The cutting stops when the rule number of the node is below a pre-defined parameter *BINTH* (Bin Threshold), configured as 2 in our example. From Fig. 2b, it is observed that Child 0, 2, 4 – 7 of root node do not need further cuttings. In this case, a leaf node will be initialized, containing a block of the pointers referencing the corresponding rules. In contrast, each of the non-leaf nodes (root, Child 1 and 3) holds a d-tuple specification of its covering space, the cutting dimension, partition number, as well as a child pointer array that contains the addresses of child nodes. During the classification, the incoming packet traverses from the root node and calculates the child index for the next traverse. Since HiCuts cuts the space equally, the index calculation simply requires dividing the width of cutting dimension by the partition size. The node traversal recurs until the a leaf node is encountered. Then a linear search among the leaf node rules will be conducted to determine the highest-priority match.

Since HiCuts cuts the space by powers of two, fine-grained cuttings always lead to duplications of large rules and result in exponential memory increases. HiCuts constrains the memory explosion by limiting the total number of rules in all its children within a pre-defined factor, named *space factor* (or *spfac*), of the number of rules contained in this node. For the cutting dimension, HiCuts tries to cut each dimension based on the *spfac*, and finally chooses the dimension to minimize the maximum number of rules in each child.

2.2. HyperCuts

HyperCuts [12] improves upon HiCuts in that it is able to cut multiple dimensions simultaneously, instead of cutting one single dimension at a node. Similar to HiCuts, HyperCuts cuts the space of each non-leaf node evenly and uses a pointer array to simplify child indexing.

Due to the design of multidimensional cutting, the heuristic of choosing the cutting dimensions and partition number is more complex. For the total partition number of a non-leaf node, HyperCuts borrows an idea from HiCuts. It bounds the maximum partition number by a function of N: $f(N) = spfac * \sqrt{N}$, where N is the number of rules corresponding to the current node and *spfac* (space factor) is defined beforehand by the user.

Given the bound of the maximum partition number, the complexity of the heuristic falls in how to determine the optimized combination of cutting dimensions and the partition number of each dimension. For the cutting dimensions, HyperCuts projects the rules on each dimension and counts the number of unique segments. The algorithm then chooses the dimensions with numbers above the average. To determine the partition number of each dimension, the heuristic first separately derives the local optimum partition number for each dimension. It then tries to generate a multidimensional cutting by combining the cuts of multiple dimensions and slightly adjusting the partition number around a local optimum.

2.3. HyperSplit

For HiCuts and HyperCuts, the cutting algorithm can generate a large number of identical nodes due to the fine-grained cuttings [17,28]. Such duplications leads to a considerable memory footprint, and might even exceed the memory limit. To tackle this problem, HyperSplit [13] is proposed, to improve memory efficiency as well as to provide moderate classification speed. The idea of HyperSplit is to build a balanced binary tree so that the rules are distributed evenly among its children, and therefore prevent excessive rule duplication. Therefore, HyperSplit conducts binary cuttings along the boundaries of the rules. Each non-leaf node of a HyperSplit decision tree chooses a single dimension and an end-point value, to cut the search space into two halves. A heuristic is introduced to choose the best dimension and threshold for the binary cutting. It first projects the rules along each of the d dimensions, resulting in a number of end-points in each dimension, and then chooses the dimension with the most end-points. Along the selected dimension, it chooses the end-point so that rules are distributed evenly among the two resulting partitions. A child node and its colliding rules are generated for each partition. Fig. 3 shows the geometrical cuttings and the resulting decision tree of HyperSplit on the example ruleset. The space decomposition stops when the number of rules colliding with the node is below *BINTH*, con-

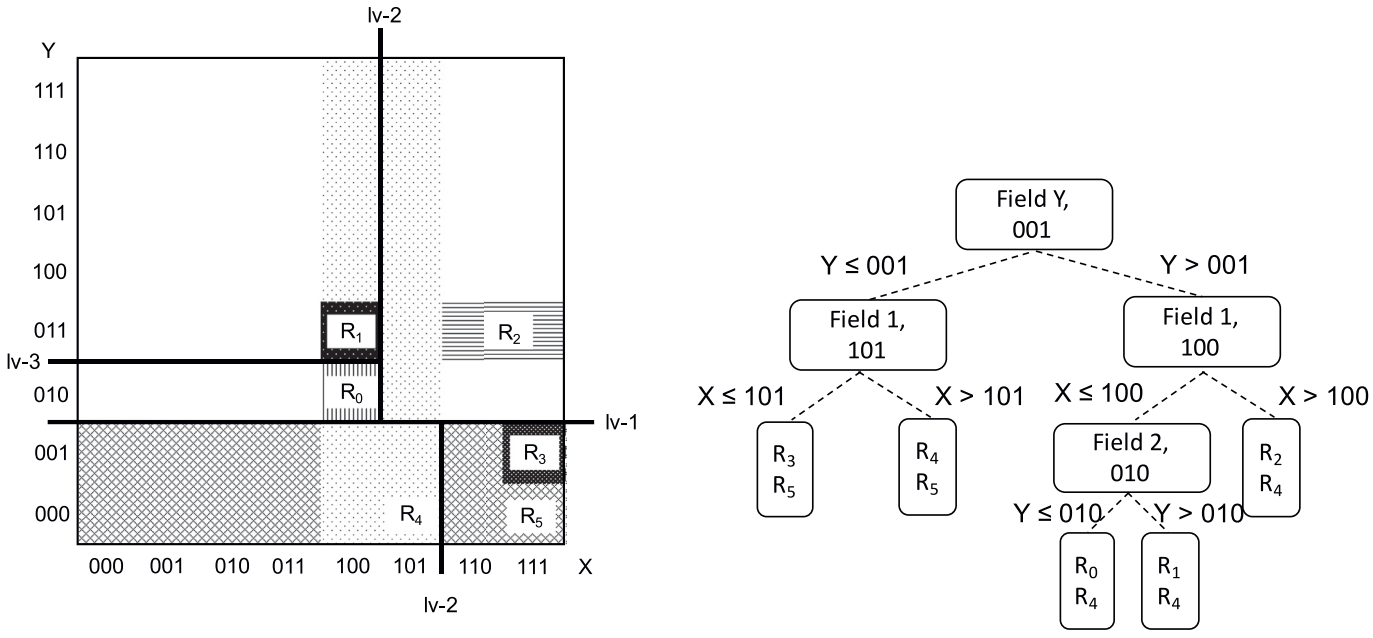


Fig. 3. Example of HyperSplit.

figured as 2 in this example, or the search space is fully covered by its highest priority rule.

2.4. EffiCuts

EffiCuts [16] is one of the latest major works for packet classification. As an improvement of HyperCuts, EffiCuts has two major contributions — grouping and equi-dense cutting. To achieve lower memory size, EffiCuts divides rules into subsets so that rules in the same subset are easier to separate and cause moderate rule replications. To be more specific, it identifies a subset of rules to be "separable" if all the rules in this subset are "either large or small in each field". The intuition is that, fields with smaller rules are cut since there are more unique projections. And the replication introduced by cutting can be dramatically reduced if there is no large rule on such fields. However, the above grouping strategy may result in more than 20 subsets. To tackle this problem, EffiCuts conducts further merging to reduce the total subset number. The merging policy guarantees that, at most one field in the resulting subset contains both large and small rules. The resulting number of subset is 5–6 for the evaluated rulesets.

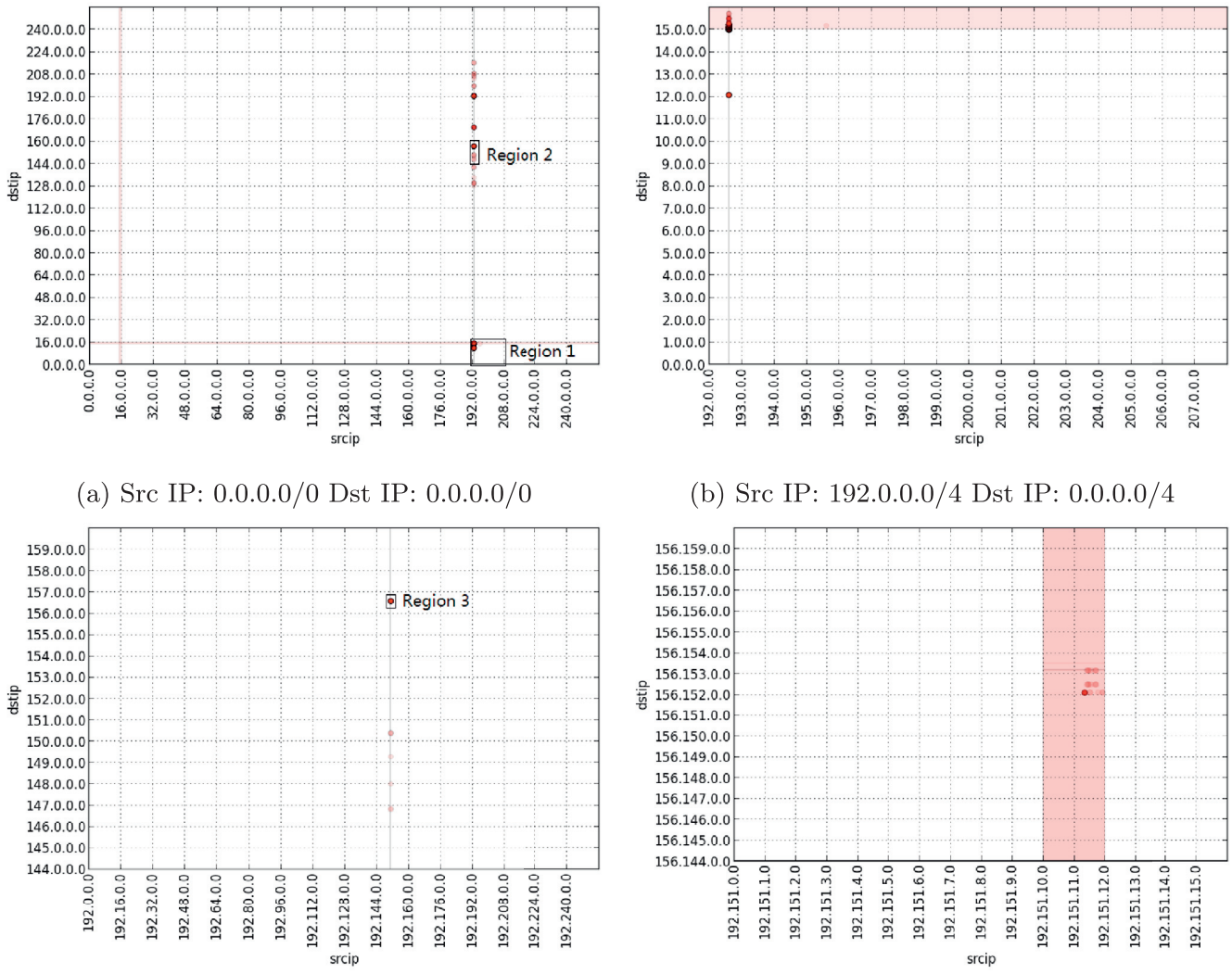
On the other hand, EffiCuts improves on HyperCuts in that it proposes Equi-dense Cutting. The idea is to conduct fine cuttings at regions where rules are densely-clustered, and conduct coarse cuttings elsewhere. Such unequal cutting is achieved by fusing the original equi-sized sibling of HyperCuts. The fusing heuristic iteratively merges continuous siblings with some redundant rules, and ensures that the rule number of resulting nodes do not exceed the maximum of original siblings. Due to such design of fusing, the child node indexing requires binary search among fused nodes. Therefore, the fusing is only conducted if the number of resulting nodes is no larger than 8. Compared with HyperCuts, such improvement merges a large number of child nodes, but may increase the depth of the tree. Overall, the primary benefit of Equi-dense Cutting is to further reduce the memory size.

3. Motivations

3.1. Rule distribution of real-life rulesets

The distribution of rules greatly impacts the decision-tree metrics and is an important consideration in the algorithm design. Therefore, an in-depth study of the ruleset distribution is made. The real-world rulesets from [29] are used, which contain rules for ACL (Access Control), FW (Firewall), and IPC (IP Chain). As an example, the distribution of the ACL ruleset is depicted in Fig. 4a by two dimensions — SIP and DIP. This ruleset contains 753 rules, with 97 distinct SIP ranges and 205 distinct DIP ranges. The red dots represent rules with small IP ranges or exact IPs. The lines (or rectangles) stand for rules with large ranges, such as the vertical lines (e.g., SIP 15.0.0.0/8, DIP 0.0.0.0/0; SIP 192.151.10.0/23, DIP 0.0.0.0/0), and horizontal line (e.g., SIP 0.0.0.0/0, DIP 15.0.0.0/8). Taking a closer look, Fig. 4b and Fig. 4c zoom into the highlighted region 1 (SIP 192.0.0.0/4, DIP 0.0.0.0/4), and region 2 (SIP 192.0.0.0/8, DIP 144.0.0.0/4) respectively. Fig. 4d zooms into the highlighted region 3 (SIP 192.151.0.0/20, DIP 156.144.0.0/12). In all the figures, the default rule (SIP 0.0.0.0/0, DIP 0.0.0.0/0) is omitted for brevity. According to our studies, two fundamental patterns are revealed here. It is worth noting that, the patterns discovered are also consistent with the findings in [14].

Small ranges are co-located. As shown in Fig. 4b, 582 rules are clustered at the top-left region (SIP 192.151.10.0/23, DIP 15.0.0.0/8), accounting for 77% the rules of the ACL ruleset. In addition, these rules co-located in a tiny region, which only covers $1/2^{19}$ the SIP range (192.0.0.0/4) and $1/2^4$ the DIP range (0.0.0.0/4) of the Fig. 4b region. Such an obvious clustering pattern originates from the fact that these rules are constructed to regulate the outbound traffic of hosts in particular subnets. Specifically, nearly all of the source hosts belong to the subnet of 192.151.10.0/23, and a large portion of involved DIPs are within 15.0.0.0/8. Such clustering also applies to regions containing a small number of rules. For instance, in Fig. 4c where 48 rules are located, 40 of 48 rules (83.3%) are clustered



(a) Src IP: 0.0.0.0/0 Dst IP: 0.0.0.0/0

(b) Src IP: 192.0.0.0/4 Dst IP: 0.0.0.0/4

(c) Src IP: 144.0.0.0/4 Dst IP: 192.0.0.0/8

(d) Src IP: 192.151.0.0/20 Dst IP: 156.144.0.0/12

Fig. 4. Distribution of ACL1 ruleset on IP fields.

within region 3. The co-locating pattern is also observable in port fields.

Fig. 5a and b depict the distribution of destination port of FW (271 rules) and IPC (1551 rules) rulesets, shown by CDF[1]. The x-axis represents the value range of destination port field, i.e. [0, 65535]. Since port ranges specified by rules are of various length, every integer value within each range is taken as a data point. The distribution is shown by plotting the CDF for all the resulting data points. It is discovered that the port field also manifests a skewed distribution across the value range.

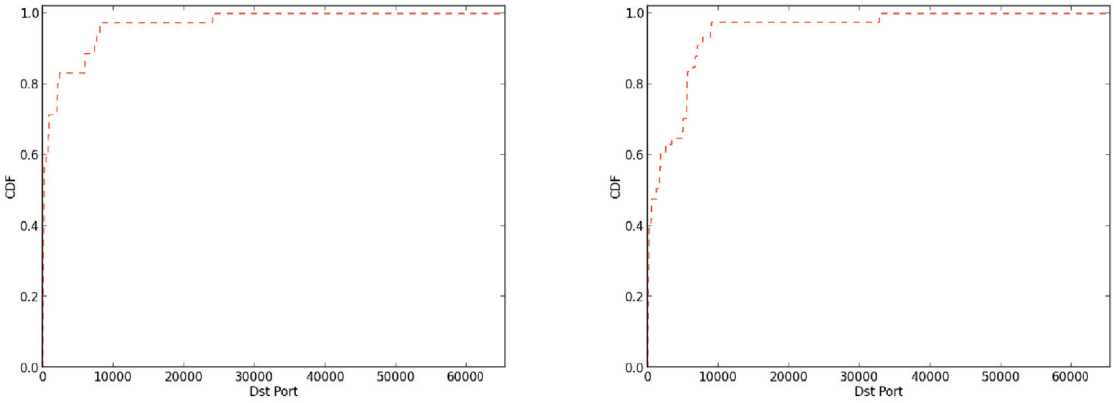
Wildcard rules overlap with small ranges. In Fig. 4 it is found that rules with small ip ranges, which account for the majority of rules, are commonly covered by large ranges. The most typical example is that the default rule covers all the other rules. As shown in Fig. 4d, the vertical rectangle (SIP 192.151.10.0/23, DIP 0.0.0.0/0) covers nearly all the small range rules. Moreover, in the upper part of Fig. 4b, 582 rules are covered by the rule (SIP 0.0.0.0/0, DIP 15.0.0.0/8). For port fields, there are also a couple of large ranges (e.g. [0, 65535], [0, 1023], [1024, 65535]) that overlap with exact port values. This pattern results from common practices of rule construction. Network administrators commonly use wildcard rules to cover a large portion of hosts and transport layer ports to enforce the default action, and write rules with small ranges (or exact values) to specify the actions for particular hosts and applications.

3.2. Inefficiency of existing cutting schemes

According to the algorithm review above, decision-tree algorithms can be categorized based on their approach to “cutting”. The design of cutting implies a tradeoff between search time and memory size, resulting in different pros and cons.

Equi-sized cutting. Equi-sized cutting is employed by HiCuts and HyperCuts. This scheme cuts the rule space equally at each node and simplifies the indexing to child nodes. The child index is generated by dividing the header value by the partition width on each dimension and combining the quotients together. Therefore, such algorithms are able to build up broader and flatter decision trees, achieving fast lookup. When rules are distributed uniformly, equi-sized cutting is able to distribute them evenly among child nodes.

[1] The large port ranges, such as [0, 65535], [0, 1023], [1024, 65535], are removed before plotting the figures



(a) CDF of FW1 destination port

(b) CDF of IPC1 destination port

Fig. 5. Distribution of port fields.

However, as we discovered in our ruleset study, a number of wildcard rules overlap with large portions of co-located small rules. Therefore, fine cuts are required to separate the small rules, resulting in excessive child nodes and wildcard rule duplication. This leads to substantial memory consumption for decision trees, especially on large and highly overlapped rulesets.

Equi-dense cutting. HyperSplit and EffiCuts belong to this category. Instead of cutting with fixed width, this scheme cuts along appropriate rule boundaries so that the rules are distributed evenly among the resulting child nodes. Since the intervals between the cutting boundaries are not uniform, during packet lookup, the array of cutting boundaries must be searched through to find the matching child. The complexity is at least $O(log(n))$ with n partitions, higher than the $O(1)$ complexity of equi-sized cutting. Therefore, the fanout of such decision trees is alway limited, which also leads to the increase of tree depth. On the other hand, these algorithms are able to achieve lower memory overhead due to their narrowness and careful selection of the splitting boundaries. Therefore, algorithms with equi-dense cutting essentially trade search speed for memory size, providing moderate search speed with lower memory consumption.

This paper proposes an approach that improves on existing cutting schemes by introducing "BitCuts", where each cut consists of the operations on a number of discrete bit positions. Fig. 6b shows how BitCuts classifies the rules in Table 1. The cut in this example consists of three bits — B_1, B_4, and B_5 — resulting in eight child nodes. Each child node represents a group of subregions, and each group is identified by the same unique bit value.

Since most rules in Table 1 contain the same value — $B_0 = 1$, $B_3 = 0$ — they are geometrically co-located in one-fourth of the whole space (highlighted area at bottom right), as shown in Fig. 6a. We observe that the three cutting bits effectively discriminate the rules, without generating too many child nodes. Although each child node also incorporates subregions in the other three-fourths of the space, they do not contribute additional duplicated rules into the child nodes. The reason is that these subregions only contain rules with wildcard fields, which cover the same child nodes as in the bottom-right region. As shown in Fig. 6a, R_5 covers the same child nodes in the bottom-left and bottom-right areas, and it is the same for R_4 in the top-right and bottom-right areas.

Compared with existing cutting schemes, BitCuts has the following advantages:

- **Efficient cutting.** By choosing the appropriate cutting bits at each node, BitCuts is able to separate the rules efficiently without excessive partitions, and therefore achieves fewer memory accesses and reasonable memory consumption.
- **Fast indexing.** BitCuts uses parallel bit indexing to support fast child-node traversal and large node fanout. For 5-tuple rules, the child node indexing can be implemented by two bit-manipulation instructions, enabling ultra-fast decision-tree traversal.

  Compared with typical packet classification algorithms, BitCuts achieves faster tree traversal than both equi-sized and equi-dense cutting algorithms. Also, the memory size of BitCuts is comparable to those of equi-dense algorithms. Therefore, BitCuts is able to improve upon the current trade-off, achieving faster classification speed while retaining reasonable memory size.

4. Bitcuts overview

Before diving into the detailed design of BitCuts, we provide an overview of BitCuts by introducing its two stages: offline preprocessing and online classification.

4.1. Offline preprocessing

The offline processing takes the ruleset as input, and constructs the BitCuts decision tree for the online classification to traverse during packet lookups. Similar to other decision-tree based algorithms, BitCuts constructs a multi-layer decision tree by recursively cutting the rule space to reduce the search scope. Specifically, each node in the decision tree contains a bitmask, where the "ones" indicate the positions of "cutting bits", which are used to index the child nodes. The bit positions are discrete, and are selected carefully during the preprocessing. As shown in the example of Fig. 6a, BitCuts generates a group of child nodes according to the bitmask, and each child node corresponds to a set of subregions with the same bit values. The goal of the bit selection heuristic, which will be explained later, is to achieve better rule separation among the children, and therefore lower the tree depth with a moderate number of cutting bits.

The recursive building algorithm of the BitCuts decision tree is shown in Algorithm 1. The function firstly checks whether the rule number in *ruleset* is below a threshold *BINTH*. If so, a leaf node containing these rules is initialized. In cases where the rule number is above *BINTH*, the function first calls the bit-selection heuris-

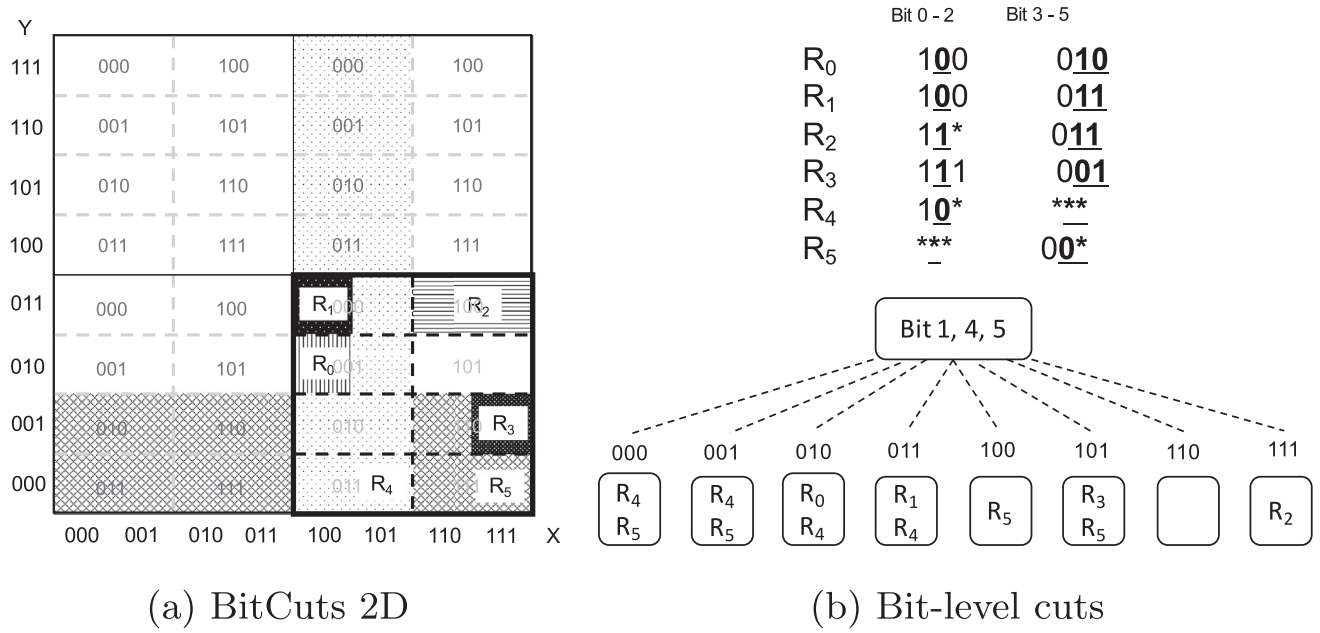


Fig. 6. Example of bit-level cuts.

Algorithm 1 Tree-building algorithm.

```
1: function BuildNode(ruleset, node)
2:     if ruleset.rule_num ≤ BINTH then
3:         InitLeaf(ruleset, node)
4:     else
5:         bitmask = BitSelection(ruleset)
6:         bitnumber = bit_num(bitmask)
7:         node → bitmask = bitmask
8:         node → child_base = malloc(node_size ∗ (1 ≪ bitnumber))
9:         node → type = INTER
10:        buckets = SplitRules(ruleset, bitmask)
11:        for i = 0 to (1 ≪ bitnumber) − 1 do
12:            BuildNode(buckets[i].ruleset, node → child_base + i)
13:        end for
14:    end if
15: end function
```

tic *BitSelection*() to acquire *bitnumber* bits. With the selected bits, *SplitRules*() divides the rules into $2^{bitnumber}$ buckets. Then the algorithm iterates through each of the buckets, and calls itself to recursively build the child node for the corresponding subset.

4.2. Online classification

After the offline preprocessing stage, the Bitcuts decision tree is constructed as shown in Fig. 7, with non-leaf nodes and leaf nodes annotated with different colors. In a tree node structure, a flag is included to indicate leaf or non-leaf. Besides that, each leaf node contains the rule number to search linearly, and a pointer that references the block containing these rule addresses. Each non-leaf node contains a bitmask, and a base pointer to the array of child nodes on the next level. Since the children of the same node are stored linearly, BitCuts aligns the size of the nodes when there are both leaf and non-leaf child nodes. The bold arrows in Fig. 7 show an example of the packet lookup. The tree is traversed recursively from *root*. The classification procedure is detailed in Algorithm 2. In *BitCutSearch*(), BitCuts first accesses the root node, and calls *BitIndexing*() to calculate the index to the child node based on *bitmask* and *header_tuples*. *BitIndexing*() extracts the bits indicated by

Algorithm 2 BitCuts searching algorithm.

```
1: function BitcutsSearch(header_tuples)
2:     node = root
3:     while node → type ! = LEAF do
4:         index = BitIndexing(header_tuples, node → bitmask)
5:         node = node → child_base + index
6:     end while
7:     return LinearSearch(node)
8: end function
```

bitmask from *header_tuples*, and concatenates the bits to generate the child index, indicating the next node to traverse. This recursion continues until it reaches a leaf node and gets a list of rule pointers, and then the packet is compared with each of the rules referenced by the pointers to get a final match.

5. Bit-selection algorithm

This section introduces the algorithm designed to build the bitmask during the tree-building procedure. The goals of the bit-selection algorithm are twofold: First, rule separation, with a view to eliminating a large portion of rules from further consideration as the decision tree is traversed; second, the bit-level cutting at each node should not introduce too much memory overhead.

For algorithms like HiCuts, HyperSplit, HyperCuts and EffiCuts, the preprocessing algorithms cut the spaces into partitions, and keep track of the number of rules in each child. The tree-building recursion stops when the number of rules in the child node is below *BINTH*.

5.1. Bit separability

BitCuts takes another perspective for this procedure. That is, each cut "separates" a number of rules. As the decision tree is built and more cuts are involved, more rules are separated among the child nodes. The formal definition of rule separation is as follows:

Definition 1. Given the cutting C that divides the ruleset F = {R_0, R_1, ... R_{N-1}} into m buckets K = {K_0, K_1, ... K_{M-1}}, the rule pair (R_s,

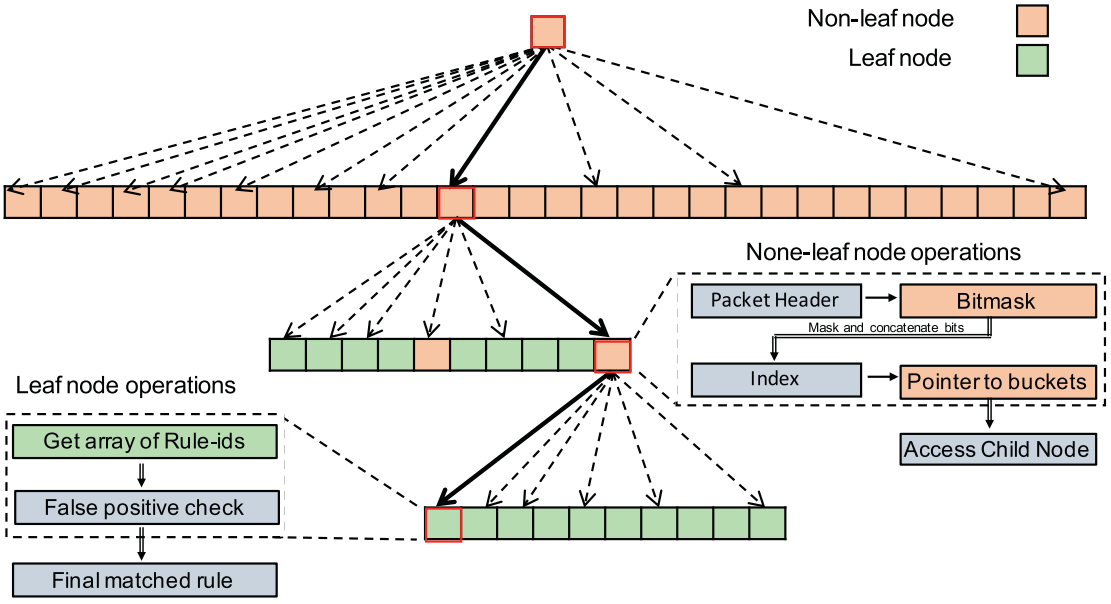


Fig. 7. Overall BitCuts lookup procedure.

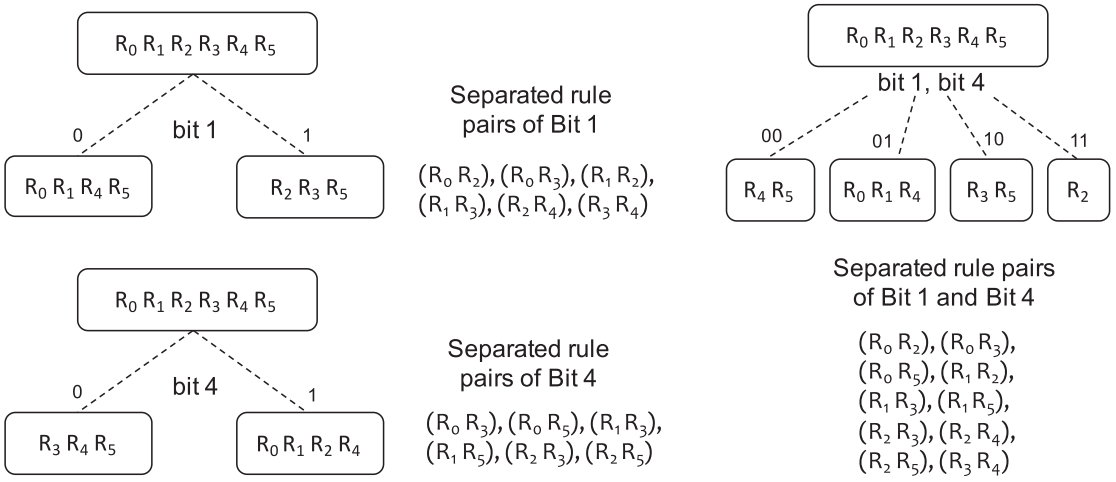


Fig. 8. Rule pairs separated by bits.

R_d) is defined as separated iff R_s and R_d do not co-locate in any of the buckets in K.

Fig. 8b illustrates how the rules are separated by each bit based on the example in Fig. 6. B_1 divides the original ruleset into two subsets: {R_0, R_1, R_4, R_5} and {R_2, R_3, R_5}. After examining the rule pairs that do not co-located in the same bucket, it is found that the separated rule pairs are S_1 ={(R_0, R_2), (R_0, R_3), (R_1, R_2), (R_1, R_3), (R_2, R_4), (R_3, R_4)} are separated. Likewise, the separated rule pairs of Bit 4 are S_4 ={(R_0, R_3), (R_0, R_5), (R_1, R_3), (R_1, R_5), (R_2, R_3), (R_2, R_5)}.

As is shown in the above example, each bit corresponds to a set of rule pairs that it separates, which is an important measurement of the effectiveness of the bit. This is called a "Bit Separability Set"(BSS) in BitCuts. For example, B_0 and B_3 do not separate any rules in Table 1, so the size of their BSSes are zero, and the cut should not incorporate such bits.

5.2. Cut separability

In BitCuts, the cut of each node may incorporate multiple bits for better rule separation. The separability of a single cut is derived based on the above definition of bit separability. Considering the cut consisting of B_1 and B_4, as shown in Fig. 8, the examination of the resulting buckets reveals that the separated rule pairs are {(R_0, R_2), (R_0, R_3), (R_0, R_5), (R_1, R_2), (R_1, R_3), (R_1, R_5), (R_2, R_3), (R_2, R_4), (R_2, R_5), (R_3, R_4)}, which happens to be the union of S_1 and S_4.

Theorem 1 (Composition of Bit separability). *Let the separated rules corresponding to bit B_i be the set S_i ={($R^i_{s_1}$, $R^i_{d_1}$), ($R^i_{s_2}$, $R^i_{d_2}$), ... ($R^i_{s_l}$, $R^i_{d_l}$), ... }, where $s_l < d_l$. For a "cut" C consisting of bits $B_{i_1}, B_{i_2} \ldots B_{i_w}$, the separated rule pairs of the cut obey $S_C \supseteq S_{i_1} \cup S_{i_2} \ldots \cup S_{i_w}$.*

Proof. For any rule pair $(R^{i_k}_{s_l}, R^{i_k}_{d_l}) \in S_{i_k}, k = 1 \ldots w$, the separated two rules must have different values at bit B_{i_k} – i.e. $R^{i_k}_{s_l}[B_{i_k}] = v_0$, $R^{i_k}_{d_l}[B_{i_k}] = v_1$, $v_0 \neq v_1$, $v_0 \neq *$, $v_1 \neq *$. For the "cut" consisting of $B_{i_1}, B_{i_2} \ldots B_{i_m}$, $R^{i_k}_{s_l}$ must be located within the buckets whose index value on B_{i_k} is v_0, and $R^{i_k}_{d_l}$ must be located within the buckets whose index value on B_{i_k} is v_1, so that $R^{i_k}_{s_l}$ and $R^{i_k}_{d_l}$ must be located at different child nodes and be separated. That

is, for any $(R_{s_l}^{i_k}, R_{d_l}^{i_k}) \in S_{i_k}, k = 1 \ldots w$, $(R_{s_l}^{i_k}, R_{d_l}^{i_k}) \in S_c$. Therefore $S_C \supseteq S_{i_1} \cup S_{i_2} \ldots \cup S_{i_w}$. □

By Theorem 1, the separability of one cut could be generated by the union of the BSSes. Note that this is an approximation of the cut separability, since the union may not cover all the pairs separated by the cut. For example, ranges [0, 2] and [3, 3] can only be separated when the least two bits are involved at the same time, and neither of the bits is able to separate the ranges individually. Therefore, BitCuts takes these rules as inseparable and excludes such rule pairs in the following bit-selection algorithm. Although this design choice sacrifices optimum bit selection, we still find the evaluation result superior to those of competing algorithms.

5.3. Bit-selection algorithm design

According to the "rule separation" perspective, the bit selection procedure could be analogized to a Set-Cover Problem. Consider the ruleset R = {R_i, i = 1, ..., N}. A group of subsets $S = \{S_1, S_2, \ldots, S_w\}$ has w elements, where S_i is a set of rule pairs separated by B_i, representing the bit separability. The universe of the rule pairs is $U = \{S_1 \cup S_2 \ldots \cup S_k \ldots \cup S_w\}, k = 1, \ldots, w$. Therefore, the bit selection procedure could be formulated as finding a minimum set of S so that all the elements of U are covered.[2] According to the SCP formulation, ideally the selected bits should cover all of the rule pairs in U. However in the construction of a multi-layer decision tree, the bit selection should stop as the number of children grows excessively. Therefore, the actual bit selection algorithm is more complex and uses the SCP formulation as a heuristic for bit selection. The bit-selection algorithm is shown in Algorithm 3.

Algorithm 3 Bit-selection algorithm.

```
1:  function BitSelection(ruleset, prev_bitmask)
2:      candidate_bits ← all_bits.Exclude(prev_bitmask)
3:      bitmask ← {}
4:      bs_matrix ← CalculateBitSeparability(ruleset, candidate_bits)
5:      while True do
6:          newbit ← SelectBestBit(bs_matrix, candidate_bits)
7:          candidate_bits ← candidate_bits − newbit
8:          bitmask ← bitmask ∪ newbit
9:          buckets ← BitSplitRules(bitmask, ruleset)
10:         if MeetStopCriteria(buckets) then
11:             break;
12:         end if
13:         UpdateBitSeparability(bs_matrix, newbit)
14:     end while
15:     return bitmask, buckets
16: end function
```

The algorithm initializes *candidate_bits* and *bitmask*, so that the bits included in the ancestor nodes are excluded from the current selection (Line 1 - 2). Then it calls *CalculateBitSeparability()*, to calculate the BSS for each bit (Line 3). Afterwards, the algorithm enters the iteration to choose a number of bits in the bitmask, where each iteration picks one bit according to a greedy strategy. Inside the iteration, the algorithm examines the separability of each candidate bit, and adds the bit with the largest BSS to the *bitmask* (Line 6 - 8). The new *bitmask* is then used to split the ruleset into buckets (LIne 9). With the buckets generated, the algorithm decides whether bit selection should stop by examining the stop criteria(Line 10). If any of the criteria is met, the algorithm returns *bitmask* together with *buckets*(Line 15); otherwise, it continues with the iteration. Since the added bit might separate some rules included in other separability sets, the algorithm updates the BSS of the other candidate bits before the next iteration. The update operation simply requires subtracting the pairs in the BSS of added bits from the BSSes of other candidate bits (Line 13).

5.4. Bit separability calculation

In order to implement the heuristic, *CalculateBitSeparability()* is called to generate the BSS for each bit. A brute-force calculation proceeds as follows: For each pair of rules, test each bit to determine if the two rules can be separated. Each bit testing of a single rule requires shifting and bitwise AND. This implementation requires $N^2 \times W$ bit testings, where N is the number of rules and W is the rule width in terms of bit. In our implementation, *CalculateBitSeparability()* instead conducts bit testing for all the rules, to split them into two groups (1 or 0), which only requires $N \times W$ bit testings. Afterwards it iterates through the two groups and generates the "Separated Rule Pairs". For example, consider the BSS of bit 1 in Fig. 8. Given the $N = 6$ rules, 6 bit testings will generate two groups based on the bit value (0, 1, or *) of the rule. When the wildcard bit of a certain rule is encountered, it will result in duplications (e.g., R_5) and does not contribute to any "Separated Rule Pairs". Such rules are removed from both groups during the calculation. Then an iteration through the two groups will generate all the rule pairs separated by one bit.

5.5. Bit-selection stop criteria

As shown in Algorithm 3 (line 10), in each iteration the algorithm checks whether to continue the bit selection, and returns the current bitmask if any of the stop criteria is met. A "speed-space" tradeoff is involved in the criteria. On one hand, incorporating more bits in one "cut" will contribute to better rule separation and lower tree depth. However, for a tree node with m activated bits in its bitmask, the number of child nodes is 2^m. The memory consumption incurred by the cut increases exponentially as more bits are selected. Therefore, the stop criteria are essential to prevent memory explosion. In BitCuts, the bit selection stops if any of the following criteria is met:

- ***The space factor (Spfac) reaches a pre-defined threshold.*** Assume that m bits were selected and the original N rules were divided into 2^m buckets, where bucket i has N_i rules. Define the space factor:

$$Spfac = \left(\sum_{i=0}^{2^m - 1} N_i + 2^m \right) / N \tag{1}$$

  As shown in Eq. (1), the space factor is defined as the ratio between the total number of children plus all child rules (antecedent) and the number of parent rules (consequent). Therefore, the space factor implies the space expansion ratio caused by the current cut.
- ***The maximum rule number of all the buckets is below BINTH (Bin Threshold).*** Similar to the definition in other decision-tree algorithms, *BINTH* is an important parameter in balancing speed and space. A higher *BINTH* will make a shorter decision tree, and thus save memory size. Generally, the *BINTH* check is conducted before constructing a node to decide leaf or non-leaf, and is included in Algorithm 1 (Line 2). Since BitCuts actually merges multiple cuts into one node, the bit-selection algorithm need to keep track of the maximum number of bucket rules, and stop the bit selection when it drops below *BINTH*.

[2] Note that here U is not all the rule pairs. Here we only consider the rule pairs that can be separated by individual bits.

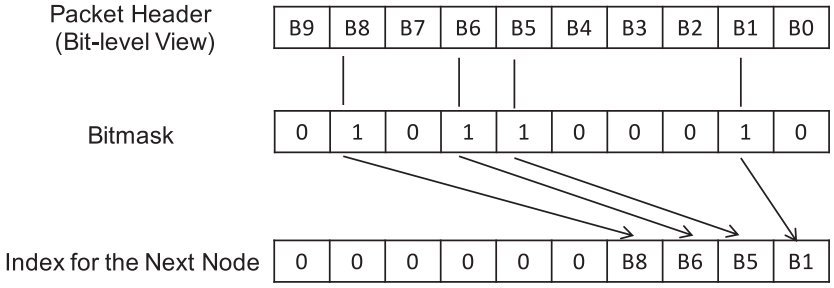


Fig. 9. Bit Indexing using PEXT instruction.

6. Implementation of bit-level operations

This section introduces the implementation of the important bit-level operations involved in both the offline and online procedures of BitCuts, namely *BitIndexing*() and *BitSplitRules*().

6.1. Bit indexing

Bit indexing implements the bit-level cutting of the online classification stage. One brute-force approach to bit indexing is to "shift and compare" to extract each individual bits and concatenate them to the child index. Considering a packet header with W bits, the complexity of such operations is $O(W)$. In BitCuts, bit indexing is implemented by PEXT(Parallel Bits Extract) [30] instruction. PEXT is included in the BMI2 instruction set, which was introduced with the Intel Haswell processor, and currently is supported by a wide range of processors [31]. The operations of PEXT are illustrated in Fig. 9. It extracts arbitrary bit positions, as specified in *Bitmask*, from *PacketHeader*. The instruction takes only 3 cycles and supports data length of 64 bits on Intel 64 architecture [32]. For 5-tuple header (104 bits), the bit indexing takes up to 2 PEXT operations, which is far more efficient than "shift and compare" and enables fast lookup for BitCuts decision trees.

6.2. Bit split rules

Function *BitSplitRules* is called in the preprocessing stage (shown in Algorithm 3, Line 5). Given the bitmask and a rule, the function *BitSplitRule* determines which of the child nodes the rule falls in. Since fields like IP addresses are represented by binary prefixes, we can easily get the corresponding bit value (0, 1 or *) in such fields. However, for range-based fields, it is nontrivial to derive the value for a certain bit position. Therefore, the algorithm *BitSplitRule* is designed as Algorithm 4 to tackle this problem.

The *BitSplitRule* algorithm first converts the input rule into a set of prefix-represented rules (PRR). Although the IP and Protocol fields are inherently prefixes, the port fields are specified by ranges, and can be expressed by multiple prefixes. Therefore, *convert_rule_to_bitstrings* first converts each rule into multiple PRR representations. Although this conversion increases the number of entries to split, these PRRs are shown as the original rule in the resulting buckets and do not introduce additional rule duplication.

With the bitmask and converted PRRs for the input rule, *BitSplitRule* then iterates through each PRR and determines all the buckets that the rule falls into. Note that the input rule falls in one bucket if any of its converted PRRs falls into it. Given m selected bit positions, the number of buckets is 2^m. A brute-force solution is to check each of the 2^m buckets and see if the selected bit values of a prefix cover the index value, which has the complexity of $O(2^m)$. An optimization is made to cut down the overall complexity. The algorithm checks the prefix at the selected bit positions and extracts the exact-value part, as well as the wildcard part. Then it

Algorithm 4 Rule splitting algorithm.

```
1: function BitSplitRules(rule, bitmask, buckets)
2:     rule_bitstrings ← convert_rule_to_bitstrings(rule)
3:     for bitstring in rule_bitstrings do
4:         wildcard_pos_encode = 0
5:         wildcard_number = 0
6:         exact_bitvalue = 0
7:         ▷ Extract the exact-value bits and wildcard positions of the prefix
8:         for i = 0 to bitmask.bit_number − 1 do
9:             pos = find_ith_one(bitmask, i)
10:            if bitstring[pos] is "*" then
11:                wildcard_pos_encode |= (1 ≪ i)
12:                wildcard_number++
13:            else
14:                exact_bitvalue |= (bitstring[pos] ≪ i)
15:            end if
16:        end for
17:        ▷ Enumerate through each possible index that the prefix falls into
18:        for v = 0 to (1 ≪ wildcard_number) − 1 do
19:            wildcard_bitvalue = PDEP(v, wildcard_pos_encode)
20:            value = wildcard_bitvalue | exact_bitvalue
21:            bucket[value].add(rule)
22:        end for
23:    end for
24: end function
```

enumerates all the possible values of the wildcard part and combines the result with the exact-value part. In this way, the complexity is determined by the wildcard length of most rules, which is generally low, since the majority of rules are exact rules or rules with small ranges. To enumerate the values of the wildcard positions, another bit-level instruction — *PDEP* — is used to generate different values. *PDEP* is the reverse of *PEXT*. *PDEP* scatters the lower-order bits into positions specified in *wildcard_pos_encode*, and has the same cost (3 cycles) as *PEXT*, therefore dramatically accelerates the procedure.

7. Evaluation

7.1. Evaluation methodology

In this section, we compare the performance of BitCuts with HyperSplit, HyperCuts and EffiCuts. For BitCuts, HyperCuts and EffiCuts, the parameters are set as $BINTH = 8$ and $spfac = 4$. For EffiCuts, since its grouping technique is orthogonal to BitCuts, the EffiCuts implementation in the evaluation only incorporates equi-dense cutting. Since HyperSplit uses binary cutting without *spfac* configuration, we set its $BINTH = 8$. The rulesets generated by ClassBench [29], and all types of rules (i.e., ACL, FW, IPC) are eval-

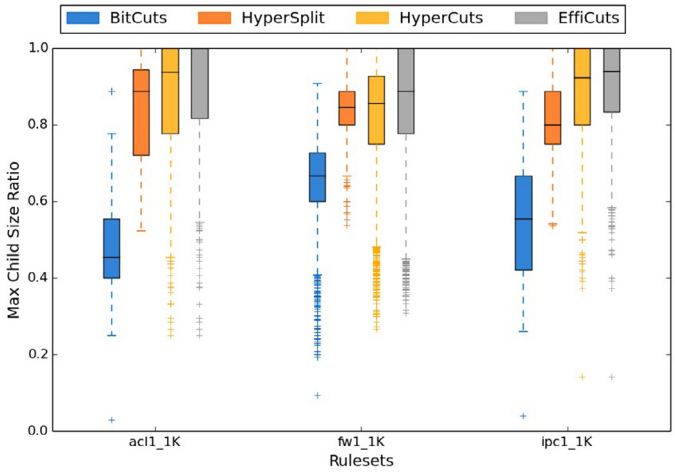


Fig. 10. Decision-tree characteristic comparison on 1K rulesets.

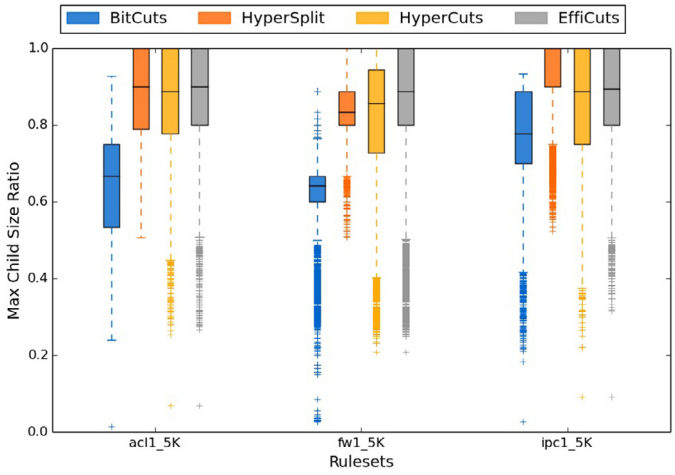


Fig. 11. Decision-tree characteristic comparison on 5K rulesets.

uated. To compare the performance on rulesets of different sizes, rulesets of 1K, 5K and 10K rules are used for all three types. In the following subsections, the characteristic of decision trees, the preprocessing metrics and the online classification throughput are evaluated and compared.

7.2. Decision-tree characteristics

We first compare the characteristics of the BitCuts decision tree with other algorithms to verify whether the "bit-level cutting" achieves better rule spread-out — i.e., whether the rules are well-scattered among the child nodes. For all the decision trees, we measure this at each non-leaf node using the ratio of the maximum child rule number to the parent rule number. A lower ratio indicates that the node has a better rule spread-out. Finally, the distribution of all the ratios is drawn as a box figure. Fig. 10 shows the distributions for four algorithms on the ACL1K, FW1K and IPC1K rulesets. Figs. 11 and 12 show the results on 5K and 10K rulesets. The lower, middle and upper bound of the box represent the first quartile, median and third quartile of the values, respectively. The figures depict the data points beyond dashed whiskers as outliers, which are 1.5 × *IQR* (interquartile range) beyond the quartiles. It can be observed that BitCuts achieves the best rule spread-out among all the algorithms. In all cases, it achieves its lowest value of 25th, median, and 75th value. Taking a look at the bottom values, we see that HyperSplit, whose value is around

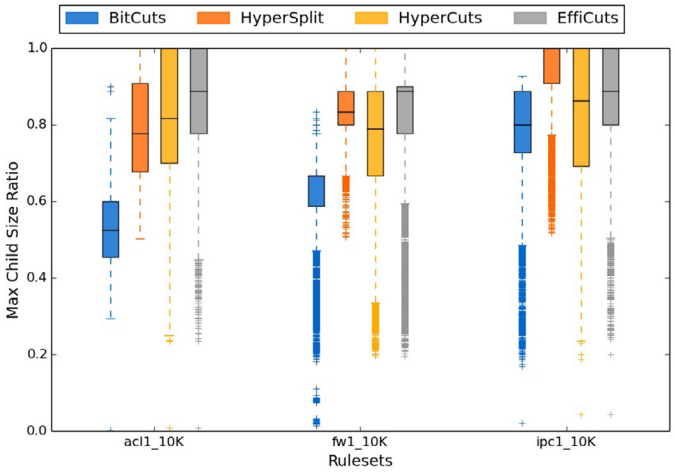


Fig. 12. Decision-tree characteristic comparison on 10K rulesets.

0.5 for all the rulesets, does not achieves outstanding rule spread-out due to its use of binary cuts. For BitCuts, HyperCuts and EffiCuts, we note that in all cases, BitCuts has the lowest bottom values. Since BitCuts, HyperCuts and EffiCuts are configured with the same *spfac*, the results indicate that our proposed bit-level cutting achieves the best rule spread-out given the same memory expansion.

7.3. Memory accesses and memory space

In order to reveal the speed and space of various algorithms, some metrics about the decision trees, including the number of memory accesses required for tree traversing and memory size, are evaluated. Particularly, the number of memory accesses along a search path is the depth of the leaf node plus the number of linear accesses corresponding to that leaf.

Average-case memory accesses. As shown in Fig. 13, BitCuts has the smallest average memory accesses among all the algorithms. Overall, the average memory accesses of BitCuts is about 42% that of HyperSplit, 59% that of HyperCuts, and 53% of EffiCuts. In most cases, HyperSplit has the greatest average memory accesses, due to its conservative choosing of cutting numbers. HyperCuts and EffiCuts outperform HyperSplit due to their larger fanout in a single node, but their average memory accesses are still larger than that of BitCuts, because the cutting of BitCuts is more efficient in terms of rule separation.

Worst-case memory accesses. For worst case memory accesses, BitCuts is about 51% of HyperSplit, 54% of HyperCuts and 51% of EffiCuts, as shown in Fig. 14. For all the rulesets, the worst-case memory accesses of BitCuts is below 21, indicating that BitCuts effectively reduces the worst memory accesses for various rulesets. For all the other three algorithms, on the contrary, there are cases where more than 35 accesses are required on large rulesets.

Memory size. From Fig. 15 we observe that BitCuts has achieved significant improvement on memory size than HyperCuts and EffiCuts. On average it consumes only 12% the memory of HyperCuts and 19% of EffiCuts. Recalling that BitCuts on average requires 59% the memory accesses of HyperCuts and 52% that of EiffCuts, BitCuts outperforms HyperCuts and EffiCuts in both classification speed and memory consumption. Although BitCuts consumes nearly 2.8x the memory size of HyperSplit, BitCuts only requires less than half of the memory accesses of HyperSplit. In particular, the size is still smaller than 1MB for ACL10K, which is

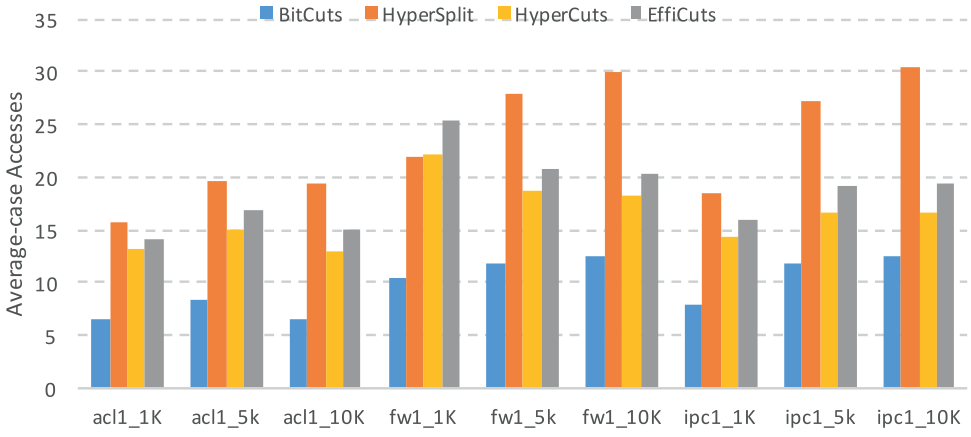


Fig. 13. Average memory accesses.

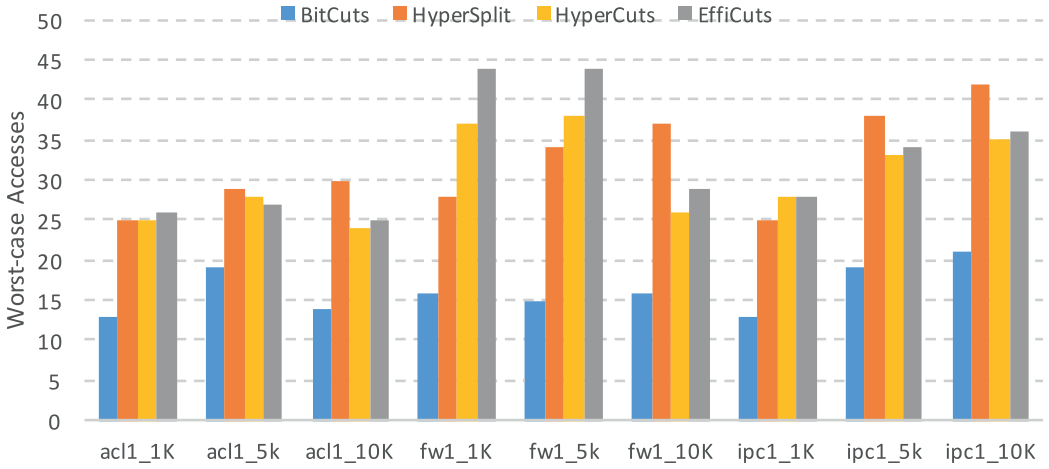


Fig. 14. Worst memory accesses.

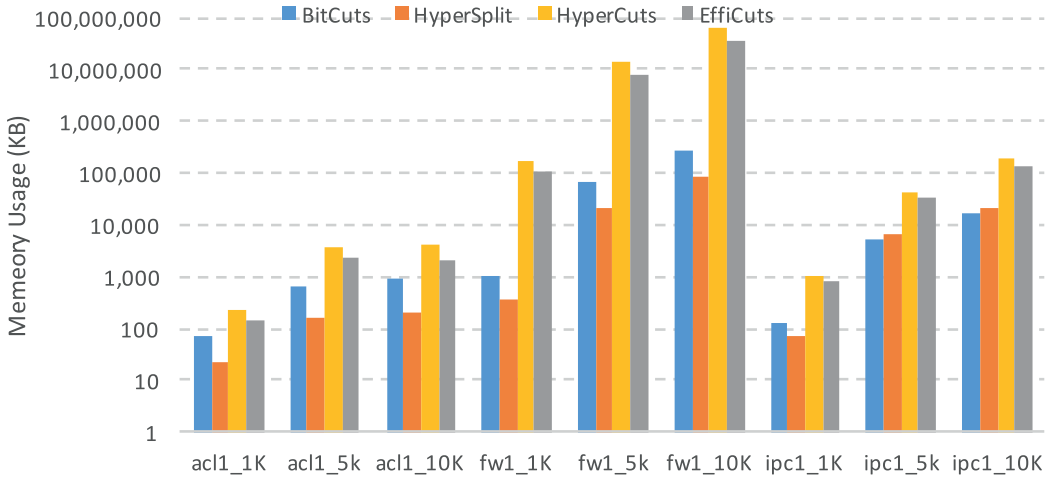
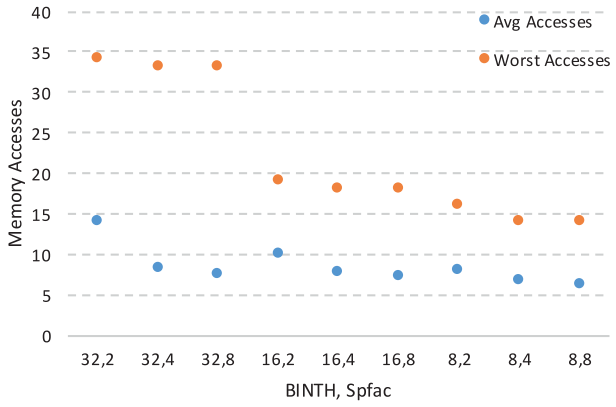


Fig. 15. Memory usage.

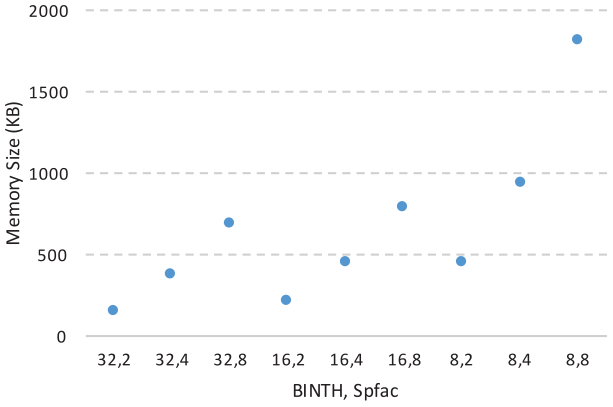
small enough to be accommodated in cache. For rulesets like IPC5K and IPC10K, BitCuts achieves the lowest memory consumption, due to its efficient cutting method and the saving of non-leaf nodes. BitCuts thus achieves a reasonable trade-off between speed and space.

7.4. Sensitivity study

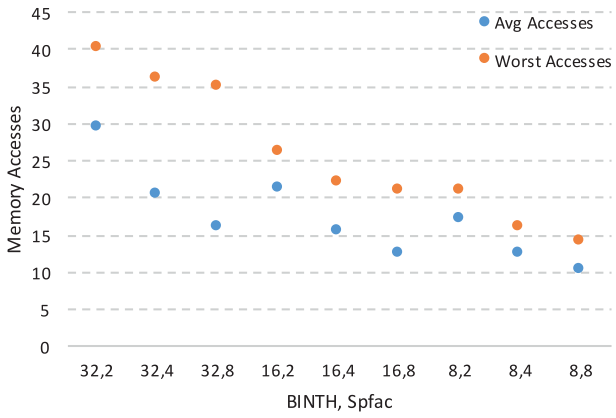
Given a ruleset, the performance of BitCuts algorithm are determined by BitCuts parameters — *BINTH* and *Spfac*. Empirically, the parameters are chosen through experiments, based on the resulting memory access number and memory size. For instance, the worst-case memory access number could be used to estimate the



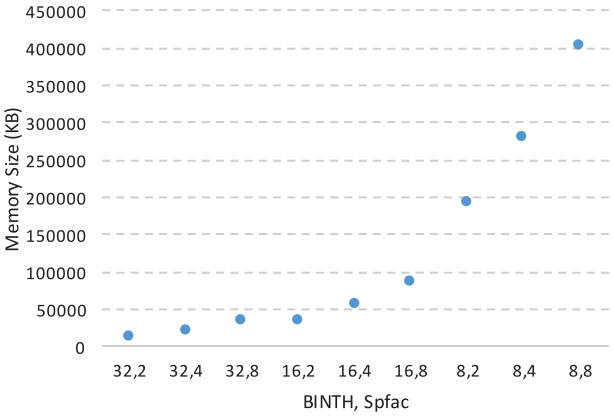
(a) Memory Accesses ACL10K



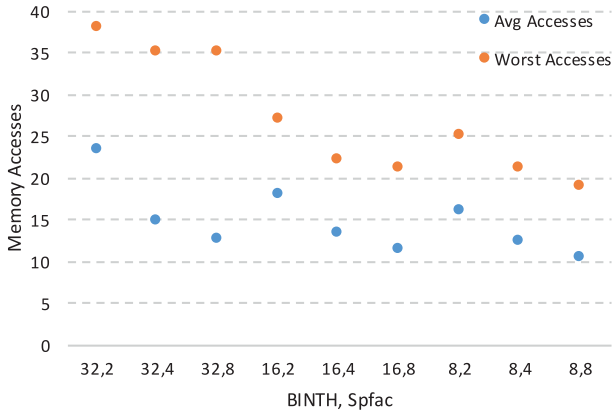
(b) Memory Size ACL10K



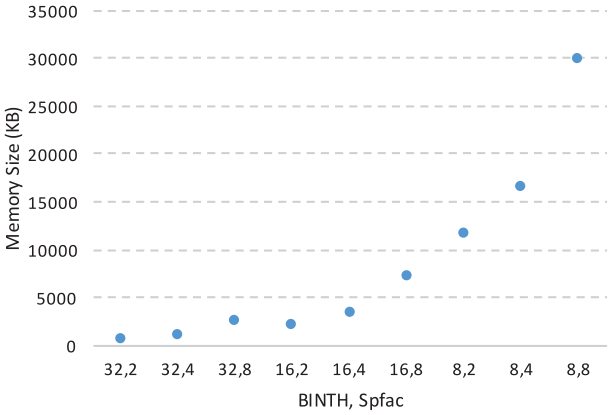
(c) Memory Accesses FW10K



(d) Memory Size FW10K



(e) Memory Accesses IPC10K



(f) Memory Size IPC10K

Fig. 16. Sensitivity Study of memory accesses and size.

lower bound of throughput. In addition, if the data structure is small enough to load into faster memory hierarchy, such as L3/L2 cache, the decision-tree traversal could benefit from faster data access in cache and accelerate the online classification. To provide the insight about the effect of parameters. A sensitivity study based on ACL10K, FW10K and IPC10K rulesets are conducted. For these experiments, BitCuts is ran on all three rulesets with BINTH of 8, 16, 32 and Spfac of 2, 4 and 8, respectively. Results about memory size and memory accesses (both average and worst accesses) are presented.

Fig. 16a, c, and e show the average and worst memory accesses on ACL10K, FW10K and IPC10K rulesets. The x-axis represents the parameters of different runs, and y-axis shows the number of memory accesses. It is observed that, as BINTH gets smaller and Spfac gets larger, both average and worst memory access number becomes lower. Particularly, reducing BINTH is effective to reduce

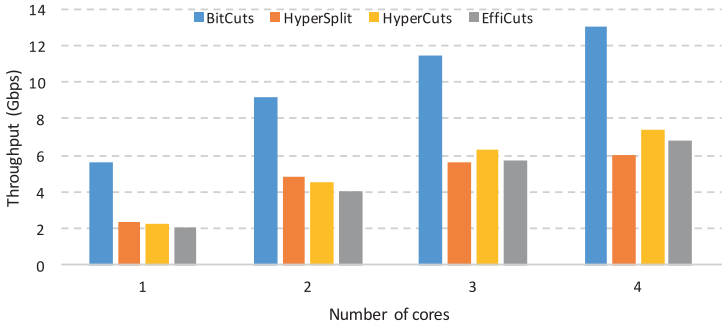


Fig. 17. Average throughput.

the worst memory accesses, and therefore is helpful to provide determined worst-case classification speed. On the other hand, increasing Spfac is able to lower the average memory accesses constantly.

Fig. 16b, d and f show the memory size results on ACL10K FW10K and IPC 10K rulesets. The results show that both BINTH and Spfac have obvious impact the memory size. The memory grows as BINTH decreases or Spfac increases. For BINTH, its impact on memory size is greater on FW and IPC rulesets than that on ACL ruleset. Spfac, which controls the cutting number on all non-leaf nodes, manifests the same trend of memory impact across all rulesets. Given the same BINTH, the memory size grows sub-linearly as the Spfac increases.

7.5. Online classification throughput

The ruleset ACL10K is used to test the throughput of the decision-tree lookup. The testbed is set up on two HP Z228 workstations, each with a 4-core Intel Xeon Processor E3-1225 CPU, 20 GB memory, and an X710 NIC with two 10 Gbps ports. The packet size of the testing traffic is 64 bytes.

A traffic generator is developed to continuously generate and receives packets, and calculate the throughput accordingly. The generator runs in a mode wherein each flow only contains one packet. Packets containing different headers are processed by the algorithms, and there is no continuous flow in the traffic. The distribution of the header values, which directly influences the memory access pattern, is the key factor in the testing. Some may suggested that random headers could be an option. However, actual testing on random traffic shows that all the algorithms perform similarly, and the deeper tree nodes are never touched, which is the "favorable case" for all algorithms. Therefore, we decided to test the "bad case" for all algorithms. The "bad case" testing traffic is composed of the headers matching each of the decision-tree leaves. An extra tree-building is run to output the information of all the leaf nodes into a file. Then the traffic generator loads the file to synthesize the testing traffic. As shown in Fig. 17, DPDK-based evaluation shows that BitCuts achieves about 2.1× the throughput of HyperSplit, 2.0× that of HyperCuts and 2.2× that of EffiCuts. BitCuts is the only algorithm that achieves over 10 Gbps throughput with 3 cores, indicating that BitCuts is superior to the other algorithms for high-speed packet classification.

8. Discussion

8.1. Other algorithms using bit-level classification

There are also other algorithms that use bit-level classification. The Modular Packet Classification (MPC) [18] is the first algorithm to incorporate bit-level packet classification. A three-step search framework is proposed for online classification. The first step checks partial prefixes on multiple fields to index into a jump table, where each table entry corresponds to a subset of rules and points to a decision tree. The decision tree is traversed by examining the bit position stored in each tree node. The classification resorts to linear search when the number of matching rules falls below a threshold. However, the first step of prefix-based indexing does not adapt well to skewed rule distribution, and may introduce considerable memory overhead. In addition, the decision trees of MPC only choose a single bit at one node, which does not effectively reduce the length of the search path.

D2BS [19] also selects discrete bit positions to divide the rules and construct a jump table, where each table entry points to a block of rules that correspond to the concatenated bit index. The rules in the blocks are searched linearly. However, D2BS only incorporates single-step bit indexing, and does not build a decision tree to recursively reduce the search scope. Typically, D2BS selects 16 – 18 bits at the first stage. Since all the bits selected are used to divide the rules in one step, the bit number is limited due to its large memory overhead, and is still not sufficient to achieve reasonable rule separation. By contrast, BitCuts builds a multi-level decision tree; for each child node and its corresponding rules, the algorithm selects the most effective bits for segregating the rules. Therefore, BitCuts incorporates far more bits during classification and avoids selecting a large number of bits at a single node.

8.2. Memory consumption

Although the memory consumption of BitCuts is higher than that of HyperSplit, the reduction of memory access makes BitCuts faster than HyperSplit. And it achieves a great improvement upon equi-sized cutting algorithms like HyperCuts. For scenarios where memory consumption is critical, the rules can be separated into several subgroups, and use BitCuts to build decision trees for each. Prior works like EffiCuts [16], SAX-PAC [25] and RFG [24] proposed several grouping techniques, and were able to reduce the memory consumption by a huge factor.

9. Conclusion and future work

This paper proposes BitCuts, a decision-tree algorithm that performs bit-level cutting. BitCuts achieves an optimized speed and space trade-off that, it obtains high classification speed and maintains reasonable memory consumption. The evaluations based on rules from ClassBench show that BitCuts outperforms HyperCuts and EffiCuts in both speed and space. DPDK-based evaluations demonstrate that BitCuts achieves about 2.0x – 2.2x the throughput of existing algorithms on large rulesets, and is the only algorithm that achieves over 10Gbps throughput with 3 cores. For memory consumption, BitCuts has only 12% the memory consumption of HyperCuts, 19% that of EffiCuts, and is comparable to HyperSplit. Its memory size is less than 1 MB on the ACL10K ruleset.

BitCuts is a promising scheme to meet the increasing performance requirements of packet classification. Future work includes developing more advanced bit-selection algorithms to further reduce memory consumption, and developing heuristics that meet hard memory limitations. Meanwhile, the bit-level separability defined in this paper is capable of identifying effective bits and redundant bits for classification. Adopting such bit-level information for ruleset compressing is also an interesting research direction, which may also benefit hardware solutions such as TCAM.

Acknowledgement

This work was supported by the National Key Research and Development Program of China (2016YFB1000102), the National Key Technology R&D Program of China under Grant No. 2015BAK34B00, and the Science and Technology Project of State Grid No. KY-SG-2016-031-JLDKY.

References

[1] H.J. Chao, Next generation routers, Proc. IEEE 90 (9) (2002) 1518–1558.
[2] A. Bremler-Barr, D. Hendler, Space-efficient tcam-based classification using gray coding, IEEE Trans. Comput. 61 (1) (2012) 18–30.
[3] K. Lakshminarayanan, A. Rangarajan, S. Venkatachary, Algorithms for advanced packet classification with ternary cams, in: ACM SIGCOMM Computer Communication Review, vol. 35, ACM, 2005, pp. 193–204.
[4] Y.-K. Chang, C.-I. Lee, C.-C. Su, Multi-field range encoding for packet classification in tcam, in: INFOCOM, 2011 Proceedings IEEE, IEEE, 2011, pp. 196–200.
[5] K. Zheng, H. Che, Z. Wang, B. Liu, X. Zhang, Dppc-re: tcam-based distributed parallel packet classification with range encoding, IEEE Trans. Comput. 55 (8) (2006) 947–961.
[6] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, A. Shukla, Packet classifiers in ternary cams can be smaller, in: ACM SIGMETRICS Performance Evaluation Review, vol. 34, ACM, 2006, pp. 311–322.
[7] A.X. Liu, C.R. Meiners, E. Torng, Tcam razor: a systematic approach towards minimizing packet classifiers in tcams, IEEE ACM Trans. Netw. (TON) 18 (2) (2010) 490–500.
[8] E. Norige, A.X. Liu, E. Torng, A ternary unification framework for optimizing tcam-based packet classification systems, in: Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, IEEE Press, 2013, pp. 95–104.
[9] P. Gupta, N. McKeown, Packet classification on multiple fields, ACM SIGCOMM Comput. Commun. Rev. 29 (4) (1999) 147–160.
[10] B. Xu, D. Jiang, J. Li, Hsm: A fast packet classification algorithm, in: Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on, vol. 1, IEEE, 2005, pp. 987–992.
[11] P. Gupta, N. McKeown, Packet classification using hierarchical intelligent cuttings, in: Hot Interconnects VII, 1999, pp. 34–41.
[12] S. Singh, F. Baboescu, G. Varghese, J. Wang, Packet classification using multidimensional cutting, in: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM, 2003, pp. 213–224.
[13] Y. Qi, L. Xu, B. Yang, Y. Xue, J. Li, Packet classification algorithms: from theory to practice, in: INFOCOM 2009, IEEE, 2009, pp. 648–656.
[14] H. Song, J.S. Turner, Abc: adaptive binary cuttings for multidimensional packet classification, IEEE ACM Trans. Netw. (TON) 21 (1) (2013) 98–109.
[15] H. Lim, N. Lee, G. Jin, J. Lee, Y. Choi, C. Yim, Boundary cutting for packet classification, IEEE ACM Trans. Netw. (TON) 22 (2) (2014) 443–456.
[16] B. Vamanan, G. Voskuilen, T. Vijaykumar, Efficuts: optimizing packet classification for memory and throughput, in: ACM SIGCOMM Computer Communication Review, vol. 40, ACM, 2010, pp. 207–218.
[17] P. He, G. Xie, K. Salamatian, L. Mathy, Meta-algorithms for software-based packet classification, in: 2014 IEEE 22nd International Conference on Network Protocols, IEEE, 2014, pp. 308–319.
[18] T.Y. Woo, A modular approach to packet classification: algorithms and results, in: INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 3, IEEE, 2000, pp. 1213–1222.
[19] B. Yang, J. Fong, W. Jiang, Y. Xue, J. Li, Practical multituple packet classification using dynamic discrete bit selection, IEEE Trans. Comput. 63 (2) (2014) 424–434.
[20] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, V. Sekar, Making middleboxes someone else's problem: network processing as a cloud service, ACM SIGCOMM Comput. Commun. Rev. 42 (4) (2012) 13–24.
[21] H. Hawilo, A. Shami, M. Mirahmadi, R. Asal, Nfv: state of the art, challenges, and implementation in next generation mobile networks (vepc), IEEE Netw. 28 (6) (2014) 18–26.
[22] C. Lan, J. Sherry, R.A. Popa, S. Ratnasamy, Z. Liu, Embark: securely outsourcing middleboxes to the cloud, in: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), 2016, pp. 255–273.
[23] H. Laurent, R.L. RIVEST, Constructing optimal binary decision trees is np-complete, Inf. Process. Lett. (1976).
[24] X. Wang, C. Chen, J. Li, Replication free rule grouping for packet classification, in: ACM SIGCOMM Computer Communication Review, vol. 43, ACM, 2013, pp. 539–540.
[25] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster, Sax-pac (scalable and expressive packet classification), in: ACM SIGCOMM Computer Communication Review, vol. 44, ACM, 2014, pp. 15–26.
[26] W. Xiang, Q. Yaxuan, W. Kai, X. Yibo, L. Jun, Towards efficient security policy lookup on many-core network processing platforms, China Commun. 12 (8) (2015) 146–160.
[27] H. Asai, Y. Ohara, Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup, in: ACM SIGCOMM Computer Communication Review, vol. 45, ACM, 2015, pp. 57–70.
[28] H. Song, Design and evaluation of packet classification systems, Dept. of Computer Science and Engineering, Washington University (2006).
[29] D.E. Taylor, J.S. Turner, Classbench: a packet classification benchmark, in: Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies., vol. 3, IEEE, 2005, pp. 2068–2079.
[30] I. Coorporation, Intel 64 and ia-32 architectures software developer instruction set reference manual, 2016, URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.
[31] Bit manipulation instruction sets. URL https://en.wikipedia.org/wiki/Bit_Manipulation_Instruction_Sets.
[32] I. Coorporation, Intel 64 and ia-32 architectures optimization reference manual, 2016, URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.