Prefix Cuttings for Packet Classification with Fast Updates

Weitao Han, Peng Yi and Le Tian
National Digital Switching System Engineering & Technological R&D Center
Zhengzhou, China
[e-mail: hanweitao.cn@gmail.com]
*Corresponding author: Weitao Han

Received November 12, 2013; revised February 12, 2014; revised March 18, 2014; accepted April 10, 2014; published April 29, 2014

Abstract

Packet classification is a key technology of the Internet for routers to classify the arriving packets into different flows according to the predefined rulesets. Previous packet classification algorithms have mainly focused on search speed and memory usage, while overlooking update performance. In this paper, we propose PreCuts, which can drastically improve the update speed. According to the characteristics of IP field, we implement three heuristics to build a 3-layer decision tree. In the first layer, we group the rules with the same highest byte of source and destination IP addresses. For the second layer, we cluster the rules which share the same IP prefix length. Finally, we use the heuristic of information entropy-based bit partition to choose some specific bits of IP prefix to split the ruleset into subsets. The heuristics of PreCuts will not introduce rule duplication and incremental update will not reduce the time and space performance. Using ClassBench, it is shown that compared with BRPS and EffiCuts, the proposed algorithm not only improves the time and space performance, but also greatly increases the update speed.

***Keywords*:** Network security, packet classification, decision tree, incremental update

This research was supported by the Major State Basic Research Development Program of China (973 Program) under Grant No. 2012CB315906 and 2013CB329104; the National Natural Science Foundation of China under Grant No. 61309019 and 61372121; the National High Technology Research and Development Program of China (863 Program) under Grant No. 2013AA013505

http://dx.doi.org/10.3837/tiis.2014.04.016

1. Introduction

With the improvement of the Internet, people have higher requirements for network bandwidth, network security, and network latency. Therefore, Internet devices must be able to classify packets quickly so as to guarantee Quality of Service. Since packet classification is the key technology of Routers, Fire Walls and Network Intrusion Detection Systems (NIDS), its speed directly affects the overall performance of the Internet.

Given a predefined ruleset, packet classification is to find out the best rule that matches the incoming packet. An excellent packet classification algorithm should have the following three features: fast search speed, small memory usage and easy to update. Because of the importance of packet classification, many researchers have carried out detailed studies. However, the early research mainly focuses on the search speed and memory usage, while ignoring the update performance. As a matter of fact, after a ruleset is defined, as the network environment changes, its rules change as well. A ruleset will increase or reduce its rules according to the real time network environment. It is vital for a packet classification algorithm to be able to adapt to the incremental update of a ruleset. Therefore, the current problem that packet classification algorithm needs to address is to meet the time, space and update performance at the same time. However, in the real life network, with the size and the complexity of the ruleset growing, the difficulty in designing a global optimal packet classification algorithm increases.

In this paper, we propose PreCuts—a packet classification algorithm with excellent scalability and update speed. It achieves fast updates without reducing the time and space performance. Its update time is almost the same as a search time. After scrutinizing the characteristics of IP field, we put forward three heuristics to build a 3-layer decision tree, with each layer being constructed by one heuristic. The main contribution of this paper is as follows.

- Prefix Index Cutting: we propose a ruleset partition method to group rules with the same highest byte of IP address. This method could partition the ruleset into 216 subsets at most, and with no overlap between every two subsets.
- Prefix Length Cutting: after prefix index cutting, we partition rules with the same IP prefix length. This further reduces the space complexity of the subsets.
- Bit Partition: we propose an information entropy-based bit partition method. First we choose several specific bits of IP prefix and then group the rules with the same bits. Bit Partition is the last step in processing the ruleset. It realizes the fine-grained partition of the ruleset.

The rest of this paper is organized as follows. Section 2 briefly describes the background of packet classification. Section 3 shows the motivation of PreCuts. Section 4 illustrates the proposed algorithm. Section 5 summarizes the performance results, and the last section concludes this paper.

2. Background

2.1 Problem Statement

Packet Classification is a process that classifies packets into flows according to the predefined ruleset. **Table 1** illustrates a simplified example of a ruleset. Each rule *r* consists of

D fields and an action. Ruleset R is a set of rules. Let $r=\{F_1, F_2, \cdots, F_D, act\}$, and packet $p=\{f_1, f_2, \cdots f_D\}$, then we call p matches r if $f_i \in F_i$ for all $i=1,2,\cdots,D$. From a geometric point of view, the whole ruleset can be seen as a multidimensional space, each rule of the ruleset can be seen as a hypercube, and a packet is a point in the multidimensional space. If a packet matches a rule, the corresponding point will fall within the hypercube of the correct rule. Therefore, packet classification can be treated as a point location problem in computational geometry [1].

Table 1. Simplified Example of Ruleset

Rule	*Field*1	*Field*2	*Field*3	*Field*4	*Field*5	*Act*
*R*1	0010	1010	[4,4]	[3,3]	TCP	*act*0
*R*2	0001	1000	[2,2]	[2,3]	UDP	*act*1
*R*3	0101	1100	[1,2]	[3,3]	TCP	*act*2
*R*4	0001	1010	[2,4]	[2,3]	UDP	*act*1
*R*5	1101	110*	[1,2]	[3,4]	TCP	*act*0
*R*6	01**	111*	[4,4]	[1,2]	TCP	*act*1
*R*7	100*	10**	[3,3]	[2,2]	TCP	*act*2
*R*8	11**	0***	[2,3]	[1,1]	UDP	*act*0
*R*9	1***	1***	[1,4]	[1,2]	UDP	*act*0
*R*10	****	****	[1,4]	[1,4]	ALL	*act*2

Previous packet classification research centers on how to use heuristics to find the optimal way to locate the point [2]. Nevertheless, the rulesets being studied are static; thus the algorithms are designed based on such static rulesets. As a result, when confronted with a ruleset which needs incremental update, their update speed is unsatisfactory. However, what the present-day internet needs is an algorithm which can not only meet the internet demands, but also achieve incremental update with few overhead.

2.2 Packet Classification Algorithm

The current packet classification algorithms can be divided into two categories: RAM-based algorithmic solutions and hardware-based TCAM solutions. RAM-based algorithmic solutions focus on building the best search structure, including the following algorithms. a) Dimensional decomposition-based algorithms [3-5]. They are fast in searching but inadequate in space performance. These algorithms build their search structure by encoding the entire ruleset. This renders them incompetent in incremental update. b) Decision tree-based algorithms [6-10]. They build the decision tree by recursively partitioning the rule space. The heuristics of cutting space inevitably introduce rule redundancy. As a result, when a new rule is being inserted into the ruleset, it may simultaneously fall into various child nodes. The more the rules being inserted, the worse the time and space performance. In the worst case, the search structure has to be rebuilt.

Hardware-based TCAM solutions [11-15] are well known for their parallel search capability and constant processing speed. However, due to the limitation of TCAM's native circuit structure, these solutions face two major problems, power consumption and range encoding. Current researches mainly focus on addressing the above two problems, while the problem of rule update is almost left unstudied. Since the circuit structure of TCAM renders inserting and deleting rules even more complex, incremental update can only be realized at the expense of extra resources.

2.3 Related work

Previous researchers proposed many packet classification algorithms, but early algorithms [3], [7] only apply to small ruleset. HyperSplit proposed by Qi et al. [2] uses non-equal sized cuts for more efficient memory usage, but it still has rule duplication, and the depth of the decision tree is not satisfactory. BRPS proposed by Chang [18] uses a hierarchical list of sorted ranges and prefixes that allow the binary search to be performed on the list at each level to find the best matched rule, but there are still redundant rules in the search structure. Besides, when BRPS is faced with a huge amount of update tasks, unwanted data structure will reduce the time and storage performance. EffiCuts [8] is an improved algorithm over HyperCuts. It proposes four novel ideas: separable trees, selective tree merging, equi-dense cuts, and node co-location. Employing the same space-decomposition as HyperCuts, EffiCuts still does not fully solve the rule redundancy problem. Although EffiCuts can reduce the memory usage, its heuristics decide that it cannot achieve fast updates. Since incremental update causes more rule duplication, as the number of updated rules is increasing, the performance of EffiCuts will reduce.

3. Motivation

A packet classification algorithm which is easy to update should possess the following features:

(1) Rule Completeness

The number of different rules in the search structure should be equal to the number of the rules in the ruleset. The completeness of rules ensures that the newly inserted rule can easily find its position in the search structure according to the rule priority.

(2) Rule Uniqueness

Each different rule in the search structure should be unique. No rule redundancy is allowed. Take decision tree-based algorithm as an example, let each leaf node ruleset be $R_1, R_2, \cdots, R_n$, it is common that

$$R_1 \bigcup R_2 \bigcup \cdots \bigcup R_n = R \tag{1}$$

The uniqueness of rules requires that there is no overlap between any two leaf nodes, i.e. $\forall i, j$, $R_i \bigcap R_j = \varnothing$. Consequently, the uniqueness of rules ensures that the newly inserted rule will not introduce other rule duplication, thus avoiding the performance decrease caused by incremental update.

The heuristics of previous algorithms all fail to achieve these two features. Their heuristics use some certain points in a dimension to cut the rule space, and take these points as the basis to build the decision tree. When a rule needs to be updated, if one of these points is in the hypercube of this rule, this rule will be updated in more than two nodes. This will result in rule duplication. Accordingly, the more rules need to be updated the more redundant rules will be produced. This in turn will result in poorer performance.

In order to guarantee the incremental update performance, our heuristics must possess the above two features. Based on a thorough analysis of the characteristics of the IP domains, we propose three heuristics, *Prefix Index Cutting*, *Prefix Length Cutting*, and *Bit Partition*. All of them promise Rule Uniqueness. Moreover, PreCuts does not remove any rule in the ruleset, and it keeps every rule in the search structure. Therefore, PreCuts also has the feature of Rule Completeness.

We use the three heuristics to cut rulesets step by step. Considering the fact that the highest byte of the low boundary of the IP address is unique, our first heuristic puts the rules with the same highest byte together. This ensures that there is no rule redundancy. Besides, it is common that the bits in the IP prefix are definite, i.e., each bit in a given IP prefix has only one value. When cutting the rulesets based on these definite bits, we can avoid the rule redundancy caused by the wildcards. However, if the rules have different IP prefix lengths, we cannot ensure that all the bits that we choose to cut the ruleset are not in the wildcards. So we introduce our second heuristic to group the rules with the same IP prefix length. In such groups, we employ the third heuristic which uses definite bits to partition the rules to a finer degree.

To linearly search the least rules like other decision tree-based algorithms, PreCuts uses three heuristics to cut the rulesets layer by layer. In this way, PreCuts gradually narrows the scale of the ruleset. When the rules in a leaf node are no more than a given amount (e.g. no more than 8), we linearly search the rules to find the best matched rule. Our simulation profiles show that after the 3 layer cuttings, more than 98% of the leaf nodes have no more than 8 rules. In fact, although our first two heuristics can cut the ruleset into relatively small sub rulesets, some of them still have a large number of rules. To overcome this, we add the third heuristic to further reduce the number of the rules in the leaf nodes. However, if we introduce a fourth layer cutting, the pre-process will be more complex. This will reduce the space performance.

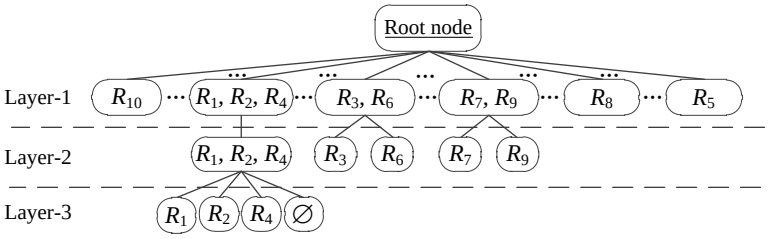


Fig. 1. Example of PreCuts

According to **Table 1**, **Fig. 1** depicts a toy example to show the flavor of our algorithm. In Layer-1, we group the rules with the same first two bits in the source and destination IP addresses. Since we use 4 bits to split the ruleset, the total number of the child nodes are 16 (Not all child nodes are shown.). In Layer-2, we put the rules with the different IP prefix length in different groups. For example, the source IP prefix length of R3 is 4 and the destination IP prefix length of R3 is 4, while the source IP prefix length of R6 is 2 and the destination IP prefix length of R6 is 3. Therefore, R3 and R6 belong to different child nodes. In Layer-3, we choose certain bits in IP prefix to cut rulesets. For example, in {R1, R2, R4}, we choose two bits (i.e., the fourth bit of source IP address and the third bit of destination IP address) to cut the rules into three different leaf nodes.

Combining the aforementioned elements, PreCuts is not only easy to understand, but also easy to evaluate. This simplicity makes it readily employable. Next, we will go into the details of PreCuts.

4. The Proposed Algorithm

PreCuts employs a 3-layer search structure. First, we use prefix index cutting to split the ruleset into subsets. Second, we use prefix length cutting to cluster rules with the same IP

prefix length. Last, we use bit partition to build the decision trees in each subset. Firstly, we introduce some relevant definitions.

4.1 Relevant Definitions

Table 2. Symbol Description

Symbol	Description
$BI(r)$	Byte index of rule r
$BI(R)$	Byte index of ruleset R
AS	Accordance set
$PL(r)$	Prefix length of rule r
$PL(R)$	Prefix length of ruleset R
Es	Split bit
Vs	Split vector
$SV(r,Vs)$	Split bit value of Vs on rule r
$E(R,Vs)$	Split entropy of Vs on ruleset R

Definition I Byte Index

Let the source and destination IP addresses of a 5-tuple rule r be $S_1.S_2.S_3.S_4 / M_A$ and $D_1.D_2.D_3.D_4 / M_D$, then the 2-dimension array (S_1, D_1) constructed by the highest bytes of the IP field is the byte index of rule *r*, recorded as $BI(r)$. If all the rules in ruleset *R* have the same byte index, then the byte index of *R* exists, recorded as $BI(R)$.

Definition II Accordance Set

The set formed by the rulesets which have byte index is named accordance set, recorded as *AS*.

Definition III Prefix Length Coordinate

Let the length of source and destination IP addresses of *r* be M_A and M_D, then the prefix length coordinate of *r* is $PL(r) = (M_A, M_D)$. If all the rules in *R* have the same prefix length coordinate, the prefix length coordinate of *R* exists, recorded as $PL(R)$.

Definition IV Split Bit

In the IP field, the bit which can divide the ruleset into 2 subsets with different bit values is named split bit, recorded as Es.

Definition V Split Vector

The bit vector which contains *n Es* and can divide the ruleset into 2^n subsets is named split vector Vs. Its total length is 64 bits.

Definition VI Split Value

Let the 64 bit vector formed by the source and destination IP addresses of rule *r* be *Vr*, then the bit string formed through merging the split bit value with *Vs* is named the split value of *r*, recorded as $SV(r,Vs) = bitMerge(Vr \wedge Vs)$, and the number of the possible values is 2^n. If all the rules in ruleset *R* have the same split value, then, the split value of *R* exists, recorded as $SV(r,Vs) = bitMerge(Vr \wedge Vs)$.

Definition VII Split Proportion

$$P(R_i, Vs) = |R_i| / |R_{Vs}| \tag{2}$$

where, $R_i = \{r \mid S(r, Vs) = V_i, r \in R_{Vs}\}$, V_i is the constant vector, $SV(R_i, Vs) = V_i$, $|R_i|$ is the number of the rules whose split value is V_i, and $|R_{Vs}|$ represents the total number of the rules.

Definition VIII Split Entropy

$$E(R, Vs) = -\sum_{i=0}^{2^n-1} P(R_i, Vs) \times \log P(R_i, Vs) \tag{3}$$

Split entropy reflects the balance degree of subsets split by *Vs*.

4.2 Prefix Index Cutting

The basic idea of prefix index cutting is to cluster rules with the same byte index, i.e.

$$R' = \{r \mid BI(r) = (S, D), r \in R\} \tag{4}$$

where, R' is the subset after clustering, *S* and *D* are two constants, and *R* is the original ruleset.

When making prefix index cutting, we first extract the byte index of the rules, and then put the rules with the same prefix index into one group. **Fig. 2** shows the pseudo code of prefix index cutting. In the pseudo code, the input is *R*, and the output is the prefix index set *BIset* and the subsets *RA*.

Algorithm 1 *Prefix Index Cutting*

Input: R
Output: $BIset, RA$

```
1: //init
2: BIset = {∅}
3: for r ∈ R do
4:   if BI(r) ∉ BIset then
5:     add BI(r) to BIset
6:   end if
7: end for
8: for b_i ∈ BIset do
9:   for r ∈ R do
10:    if BI(r) = b_i then
11:      add r to RA[i]
12:    end if
13:  end for
14: end for
15: return BIset, RA
```

Fig. 2. Pseudo code of Prefix Index Cutting

According to Theory I, prefix index cutting will not introduce rule duplication. Therefore, when updating rules, it is impossible for one rule to fall into more than two nodes, so as to avoid new rule redundancy.

Byte index is decided by the highest byte of IP address. The max number of the possible values of a given $BI(r)$ is 2^{16}. As a result, in the worst case, we will obtain 2^{16} byte index groups. However, through simulation, it is found that, in the actual ruleset, the number of different byte indexes is far less than the theoretical boundary. But in order to guarantee the update and time performance, we retain every possible byte index pointer in the search structure.

Consequently, even in the worst case, PreCuts still have enough byte index pointers to ensure that each byte index group has a corresponding node. Moreover, no matter how many byte index groups there are, PreCuts can still find the byte index group that the packet belongs to in one memory read.

There are not 2^{16} byte index groups in one ruleset. If the corresponding byte index group is not empty, then the pointer will directly point to the group. In contrast, if the group is empty, the pointer is set to 0. Saving every possible pointer will slightly increase the space complexity, but the algorithm can locate the corresponding pointer straight away according to the byte index of the packet. In this way, only one memory read is required to find the byte index group to which the packet belongs.

The search algorithm of byte index is as follows: let the pointers be numbered in the ascending order and let the head of the incoming packet be *head*, then we get $BI(head) = (S, D)$, and the number of the pointer which the packet corresponds to is $BIn(head) = S \times 2^n + D$ (n is the width of S). According to $BIn(head)$, the search algorithm spots the pointer, and jumps to the next node to continue the search. **Fig. 3** is the prefix index cutting tree of **Table 1**. It is built by the first two bits of IP addresses. Let the IP address of *head* be 100*,10** , $BI(head) = (10,10)$, then the corresponding number of the node is $BIn(head) = 2 \times 2^2 + 2 = 10$.

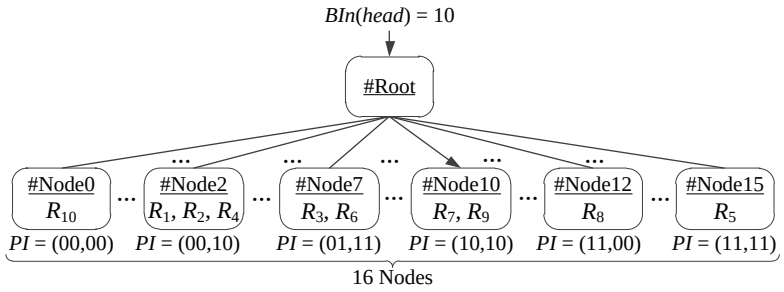


Fig. 3. Example of Prefix Index Cutting

Theory I There is no rule overlap in any two rulesets which have different byte indexes.

Proof: $\forall X, Y \in AS$ and $BI(X) \neq BI(Y)$, in X and Y, $\forall r_X \in X$, $\forall r_Y \in Y$, from $BI(X) \neq BI(Y)$, we get $BI(r_X) \neq BI(r_Y)$, thus r_X and r_Y have different IP addresses, then $r_X \neq r_Y$. This means there is no rule overlap in X and Y.

4.3 Prefix Length Cutting

The basic idea of prefix length cutting is to put the rules which have the same prefix length coordinate into one group, i.e.

$$R' = \{r \mid PL(r) = (M, N), r \in R\} \tag{5}$$

where R' represents the subset, M and N are two constants.

Similar to prefix index cutting, when performing prefix length cutting, we first calculate the prefix length coordinate of the rules, and then put the rules which have the same coordinate into one group (called PL Group). **Fig. 4** shows the pseudo code of prefix length cutting. In the pseudo code, the input is R, and the output is the prefix length set *PLset* and the subsets *RL*.

According to Theory II, prefix index cutting will not bring rule duplication.

There are two conditions in searching the prefix length nodes, namely, packet lookup and rule lookup.

(1) Packet Prefix Length Lookup

Since there is no mask information in the packet head, we cannot find the match rule using the mask information. To solve this problem, we find the child node that the packet corresponds to through judging whether the packet IP address is within the IP range of the node. In this way, a packet may belong to more than one child node, but when performing prefix length cutting, the rules are put into the subsets according to their priority. The first match rule is the optimal rule. Let the incoming packet head be *head* . Taking Node10 in **Fig. 3** as an example, the packet prefix length lookup is shown in **Fig. 5.** Let the IP address of the packet be $IP(head)=(8,9)$, and we obtain $IP(head)\in IPRange(Child2)$, then the packet corresponds to #Child2.

Algorithm 2 *Prefix Length Cutting*

Input: R
Output: $PLset, RL$
1: //**init**
2: $PLset=\{\varnothing\}$
3: **for** $r\in R$ **do**
4: **if** $PL(r)\notin PLset$ **then**
5: **add** $PL(r)$ **to** $PLset$
6: **end if**
7: **end for**
8: **for** $l_i\in PLset$ **do**
9: **for** $r\in R$ **do**
10: **if** $PL(r)=l_i$ **then**
11: **add** r **to** $RL[i]$
12: **end if**
13: **end for**
14: **end for**
15: **return** $PLset, RL$

Fig. 4. Pseudo code of Prefix Length Cutting

To get the longest prefix match, we only need to search the corresponding prefix length nodes in a reversed order. For example, in **Fig. 5**, firstly, we determine whether *IP*(*head*) belongs to *IPRange*(*Child*2), if yes, then we continue searching Child2; if no, then we will turn to search Child1. For PreCuts, the rules in the left node have shorter prefix than those in the right. Therefore, the rule found through the above method is the longest match rule.

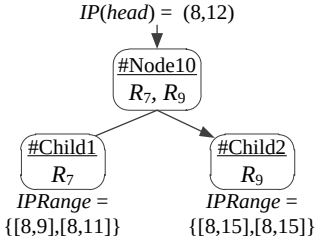


Fig. 5. Packet Prefix Length Lookup

(2) Rule Prefix Length Lookup

As is shown in **Fig. 6**, let the update rule be *r*, and compare $PL(r)$ with the prefix length coordinate in $PLset$, then we get that *r* belongs to Child2.

Theory II There is no rule overlap in any two rulesets which have different prefix length coordinates.

Proof: Take two arbitrary ruleset X and Y, and $PL(X) \neq PL(Y)$. In X and Y, $\forall r_X \in X$, $\forall r_Y \in Y$. From $PL(X) \neq PL(Y)$, we get $PL(r_X) \neq PL(r_Y)$, thus $r_X \neq r_Y$. This means there is no rule overlap in X and Y.

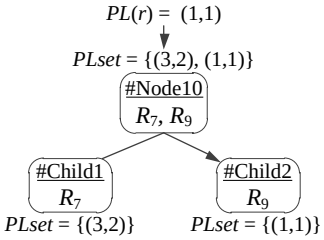


Fig. 6. Rule Prefix Length Lookup

4.4 Bit Partition

The basic idea of bit partition is to choose the split vector with the largest split entropy as the partition standard, and then divide the ruleset into several subsets with no overlaps. The process is as follows.

We first calculate the partition entropy of every possible partition vector, and choose the partition vector Vs_M which has the largest entropy as the final partition vector.

$$Vs_M = \max_i E(R, Vs_i) \tag{6}$$

According to Vs_M, we obtain the partition value of each rule in the ruleset, and then put the rules with the same partition value into one group. The pseudo code of prefix bit partition is shown in **Fig. 7**. In the pseudo code, the input is the ruleset *R*, and the output is the final partition vector Vs_M and the subsets *RB*.

Algorithm 3 *Bit Partition*

Input: R

Output: Vs_M, RB

```
1: //init
2: V0 = {Any Possible Vector}, c' = 0
3: for v ∈ V0 do
4:    if c' < C(R,v) then
5:       c' ← C(R,v)
6:       Vs_M ← v
7:    end if
8: end for
9: for r ∈ R do
10:   add r to RB[SV(r,Vs_M)]
11: end for
12: return Vs_M, RB
```

Fig. 7. Pseudo code of Bit Partition

According to Theory III, prefix bit partition will not bring rule duplication.

In PreCuts, all the possible split value pointers are retained in the search structure. If the next node has no rules, the value of the pointer is 0. There are two reasons to retain all the nodes pointers.

(1) To improve the search speed.

We do not have to compare every split value of the current node to determine the child node that corresponds to the packet. We can locate the corresponding child node right from the split value of the packet head. This reduces the comparisons that the lookup needs.

(2) To improve the update performance.

Since the pointers of all the possible child nodes exist, in update, we do not need to add node pointers for new rules, but only need to modify the existing pointers. This reduces the update complexity.

Fig. 8 shows the bit tree built according to #Node2 in **Fig. 3,** which selects two split bits and has four leaf nodes.

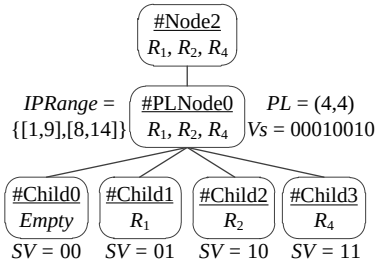


Fig. 8. Example of Bit Tree

From Definition V, it is shown that the more split bits that are chosen, the better the partition result. This is because the increase of the number of the split bits can reduce the depth of the decision tree and promote the time performance. Choosing too many partition bits will greatly increase the pre-processing complexity. For example, to choose 8 split bits from IP addresses, the possible cases are 4.43×10^9. Traversing all the values will bring a great deal of calculation. Therefore, we make some optimizations to prefix bit partition, as is shown in the following.

(1) Restrict the max number of the split bits.

[16] shows that, the prefix length of IP addresses is mainly less than 9 or larger than 12. According to this, we set the max number of split bits as 4, so as to enable the partition bit to achieve satisfactory partition in all cases.

(2) Restrict the select range of the split bits.

In a byte index group, the byte indexes of all the rules are the same. Therefore, it is useless to choose the split bit in the highest byte of the IP address. Additionally, if the split bit is chosen in the wildcards, rule duplication will be introduced. Given these two aspects, we select the split bits within the IP prefix except for the highest byte. The maximum number of candidate values is 48.

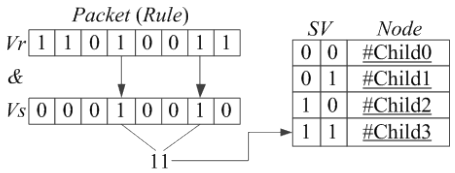


Fig. 9. Bit Lookup

We use the method in [17] to complete the lookup in bit tree, as is shown in **Fig. 9**. Let the IP address of *head* be 1101, 0010, then we get $S(head, Vs) = bitMerge(Vr \wedge Vs) = 11$, the node that packet corresponds to is Child3. The way of update rule lookup is the same as the packet lookup.

Theory III Bit partition will not bring rule duplication.

Proof: Given that we choose the split bit within the IP prefix, since there is no wildcard in IP prefix, the bit of any rules in IP prefix has set values: '0' and '1'. Therefore, each rule has a set split value and one rule can only belong to one child node. In this way, rule duplication can be avoided.

4.5 Lookup and Update

4.5.1 Lookup Algorithm

According to the search structure of PreCuts, let the packet header be *head*, the lookup algorithm is as follows.

Step1 Jump to the byte index node that $A(head)$ corresponds to;

Step2 Jump to the prefix length node that $IP(head)$ corresponds to;

Step3 If the current node is leaf node, perform Step4; if the node is not a leaf node, according to *Vs* in the current node, jump to the child node that $S(head, Vs)$ corresponds to, then perform Step3.

Step4 Perform a linear search to find the match rule.

Fig. 10 illustrates the pseudo code of lookup algorithm.

Algorithm 4 *Search Algorithm*

Input: *head*, *rootNode*

Output: *matchRule*

```
1: currNode ← rootNode.childNode(An(head))
2: if currNode ≠ 0 then
3:   for childNode ∈ currNode do
4:     if IP(head) ∈ IPRange(childNode) then
5:       currNode ← childNode
6:       break
7:     end if
8:   end for
9:   return FAILURE
10: else
11:   return FAILURE
12: end if
13: while currNode ≠ 0 do
14:   if currNode = LeafNode then
15:     matchRule ← linerSearch(currNode.ruleList)
16:     return matchRule
17:   end if
18:   currNode ← currNode.childNode(s(head, Vs))
19: end while
20: return FAILURE
```

Fig. 10. Pseudo code of Packet Lookup

According to the data structure of PreCuts, we analyze the search complexity of each layer. In Layer-1, PreCuts retains all the possible *BI* pointers. So, we only need one search to pinpoint the next node. Thus, the search complexity of this layer is O(1). In Layer-2, we need to linearly search every *IPRange*. Let the number of the *IPRange* be *P*, then the search complexity is O(*P*). In Layer-3, for each bit tree node, PreCuts can also fix the next node through only one search, and finally perform a linear search in the leaf nodes. Let the depth of the decision tree be *d*, the number of rules in the leaf node is *N*, then the search complexity is O(*d*+*N*). In all, the search complexity of PreCuts is O(*d*+*P*+*N*). However, since d is relatively small, the search complexity is O(*P*+*N*). **Table 3** compares the time complexity of PreCuts with BRPS and EffiCuts.

Table 3. Complexity of BRPS, EffiCuts and PreCuts. *M* is the total number of rules, *d* is the number of dimension, *N* is the number of rules in leaf node

Scheme	Time Complexity
BRPS	O(*d*log*M*)
EffiCuts	O(*H*+*N*)
PreCuts	O(*P*+*N*)

The analysis of the time complexity of PreCuts shows that its time performance mainly depends on the number of PL Groups (in a node) and the number of rules in the leaf node because PreCuts has to perform a linear search in these two parts, so their size will affect the

time performance. In the worst case (Max Number), the numbers of these two parameters are relatively high, but the average number is far smaller than that in the worst case. Therefore, the average time performance is still good. When performing the search algorithms, if a node needs a large amount of linear search, a separate thread can be added to finish it. In this way, the impact of the worst case on the time performance of PreCuts can be avoided.

4.5.2 Update

Algorithm 5 *Find Rule Node Algorithm*

Input: $r, rootNode$
Output: $updateNode$

```
1: Node lastNode
2: currNode ← rootNode.childNode(An(r))
3: if currNode = 0 then
4:    updateNode ← rootNode
5:    return updateNode
6: end if
7: lastNode ← currNode
8: for l ∈ currNode.Lset do
9:    if L(r) = l then
10:      currNode ← currNode.child[l]
11:      break
12:   end if
13: end for
14: if currNode = 0 or currNode = lastNode then
15:    updateNode ← lastNode
16:    return updateNode
17: end if
18: while currNode ≠ 0 do
19:    if currNode = LeafNode then
20:       updateNode ← currNode
21:       return updateNode
22:    end if
23:    lastNode ← currNode
24:    currNode ← currNode.childNode(S(r,Vs))
25: end while
26: return lastNode
```

Fig. 11. Pseudo code of Finding Rule Node

There are two kinds of rule update, namely, reconstruction and incrementation. Reconstruction update means reconstructing the search structure based on a complete ruleset. When the current search structure cannot meet the actual demand of the Internet, reconstruction update will be used. While incrementation update means adding or deleting some rules in the current search structure. When a ruleset needs to add or delete some rules, incrementation update will be used. Incrementation update will not significantly change the current search structure, but just add or delete a certain amount of rules in the search structure.

We have already discussed the reconstruction update of PreCuts. Here, we only focus on the incremental update. Let the rule waiting to be inserted (or deleted) be r, then we find out the node to which r belongs. Next, we perform the function of $addRule(r,updateNode)$ and $deleteRule(r,updateNode)$ to insert and delete rules, respectively. **Fig. 11** illustrates the pseudo code of finding rule node.

The search method of update algorithm in Layer-1 and Layer-3 is the same as the search algorithm. So is their complexity. In Layer-2, through linearly searching the *PL* values within the current node, the update algorithm spots the next node corresponding to the new rule. Let the number of *PL* values be L, then the update complexity is $O(L)$. Therefore, the update complexity of PreCuts is $O(L+N)$. As can be seen in **Table 4**, the update complexity of BRPS and EffiCuts is the same, and is higher than PreCuts because with similar update method and lookup method, PreCuts only needs relatively low overhead to complete the update.

Table 4. Update Complexity of BRPS, EffiCuts and PreCuts

Scheme	Update Complexity
BRPS	$O(dM)$
EffiCuts	$O(dM)$
PreCuts	$O(L+N)$

PreCuts has a very simple search structure, in which packet lookup and rule update are implemented using similar ways. That is, the overhead of one packet lookup and one rule update is almost the same, which ensures a fairly fast update speed of PreCuts. It is worth noting that the search structure of PreCuts does not introduce redundant rules in updating, and consequently PreCuts achieves fast rule update.

5. Performance Evaluation

In this section, we evaluate the performance of PreCuts and compare it with other packet classification schemes in terms of memory usage, search speed and update speed. In this paper, we use ClassBench [16] to generate classifiers representative of the real ones, with the scales of ruleset ranging from 1,000 to 1,000,000. Being widely used in the research of packet classification, ClassBench mainly includes three types of rules: ACL (Access Control List), FW (Fire Wall) and IPC (Linux IP Chains). For example, FW1-1K is 1,000 Fire Wall security scheme rules. All the rules are 5-dimensional. They contain 32-bit source/destination IP addresses, 16-bit source/destination port numbers and 8-bit transport layer protocol. The implementation platform is an Ubuntu 10.04 system running on an Intel Core-i3 2350m and 2 GB of main memory.

We implemented BRPS and EffiCuts with all optimizations used in [18] and [8], respectively. The bucketsize of EffiCuts is 8 while other parameters use their default values in [8].

5.1 Details of PreCuts

Table 5. Details of PreCuts

Ruleset	ACL					FW					IPC				
	1k	5k	10k	50k	100k	1k	5k	10k	50k	100k	1k	5k	10k	50k	100k
BI Group Number	56	192	703	1287	1300	62	989	1229	1131	864	132	175	431	2633	2871
PL Group Number	191	660	2073	9427	11611	153	1566	2411	8635	9551	592	1886	4037	31310	51945
Leaf Node Number	493	2135	6838	49882	99744	317	3974	9237	47939	93970	843	3290	6992	49876	99684
Tree Depth (wst)	5	5	4	5	5	4	4	4	4	5	4	4	4	4	4
Tree Depth (avg)	2.97	3.12	3.17	3.42	3.76	2.90	2.78	3.21	3.49	3.68	2.47	2.64	2.63	2.56	2.74

Table 5 illustrates the details of PreCuts, displaying 5 characteristics of the PreCuts decision tree.

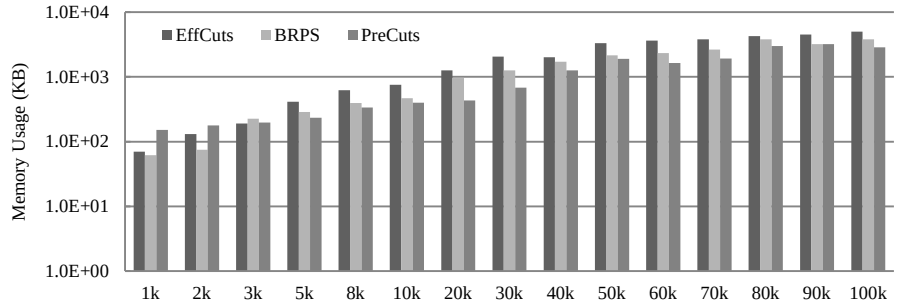
- BI Group Number: BI Group Number counts the child nodes in the first of the 3 layers of the PreCuts search structure. The larger the scale of the ruleset, the more child nodes the first level contains. The biggest BI Group Number (ipc100k) is 2871, which is far less than the theoretical boundary.
- PL Group Number: We cluster the rules with the same prefix length coordinate in every BI group. PL Group Number counts the child nodes in the second level of the PreCuts search structure. As can be seen from **Table 5**, after Byte Index Grouping, to some extent, the PL Group Number increases.
- Leaf Node Number: This number counts the total leaf nodes of PreCuts. The number of the leaf nodes increases as the size of the ruleset expands. The larger the ruleset, the closer the leaf node number gets to the number of the rules in the ruleset. This shows that PreCuts has better adaptability for large sets; the larger the ruleset, the better the decision method that PreCuts uses to partition the ruleset.
- Tree Depth: Tree depth is an important index to evaluate the performance of the decision tree-based algorithms, and it directly relates to the time performance. As can be seen from **Table 5**, including the first two levels of the search structure, the total tree depth of the decision tree is less than 5, and the worst average tree depth is just 3.76 (acl100k). Since the first two levels invariably employ two layers to describe, the final decision tree depth depends on the bit tree depth. According to the description in Section IV, a PreCuts decision tree with 5 tree depth can partition the ruleset into subsets, and this number is far greater than the number of rules in the largest ruleset.

5.2 Memory Usage

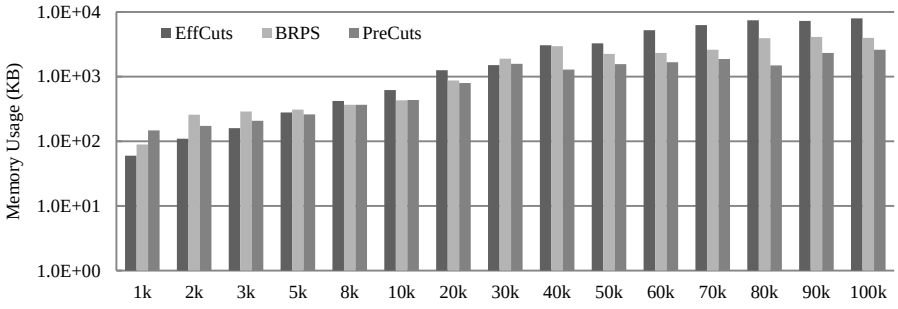
In this part we compare the memory usage of BRPS, EffiCuts and PreCuts. In order to ensure good time performance, PreCuts retains all the prefix index pointers and the child node pointers of the bit tree. This inevitably brings some redundant pointers to the decision tree

structure. These redundancies will reduce as the size of the ruleset grows, for the redundant pointers will be filled up by more rules.

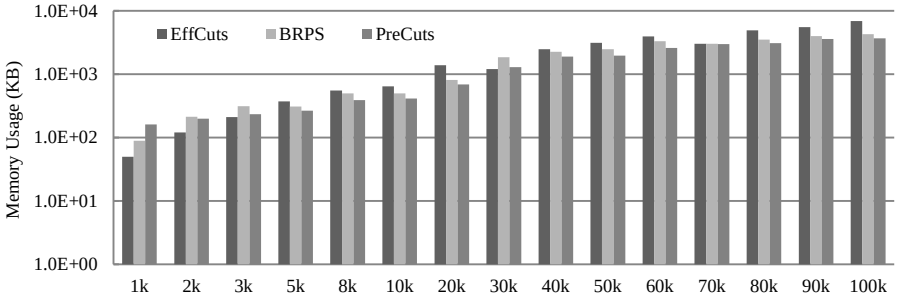
Fig. 12 compares the memory usage of BRPS, EffiCuts and PreCuts. From the figure we can see that, for relatively small ruleset (less than 10k), PreCuts has similar memory usage as BRPS and EffiCuts, while for a larger ruleset (100k), the average memory usage of PreCuts is 27.0% less than BRPS and 33.5% less than EffiCuts. Moreover, as a whole, the average memory usage of PreCuts is 24.1% less than BRPS and 38.5% less than EffiCuts.



(a) ACL



(b) FW



(c) IPC

Fig. 12. Memory Usage

5.3 Search Speed

In this part, we compare the time performance of BRPS, EffiCuts and PreCuts. We realize the three algorithms in simulation, and use the clock counter of CPU to measure the clocks that the three algorithms need in searching. **Fig. 13** compares the search cycles of the three algorithms when dealing with different rulesets. The trace file used in the test is randomly generated by ClassBench [16]. From the simulation results, compared to EffiCuts, it is obvious

that PreCuts makes a great improvement, its search cycles have reduced by 84.9% on average. Compared to BRPS, its search cycles have reduced by 21.9% on average. When processing ACL and IPC rulesets, BRPS and PreCuts have similar search performance, whereas, when processing FW rulesets, PreCuts is better.

The performance of PreCuts is much superior than EffiCuts in that the two algorithms adopt different data structures. The multi-decision tree search structure of EffiCuts makes it mandatory for the search algorithm to traverse each and every decision tree before obtaining the best match rule. This greatly raises the search cycles. For PreCuts, it is a different story. It only has to traverse one decision tree to get the best match rule. Consequently, the search cycles of EffiCuts are several times as many as those of PreCuts.

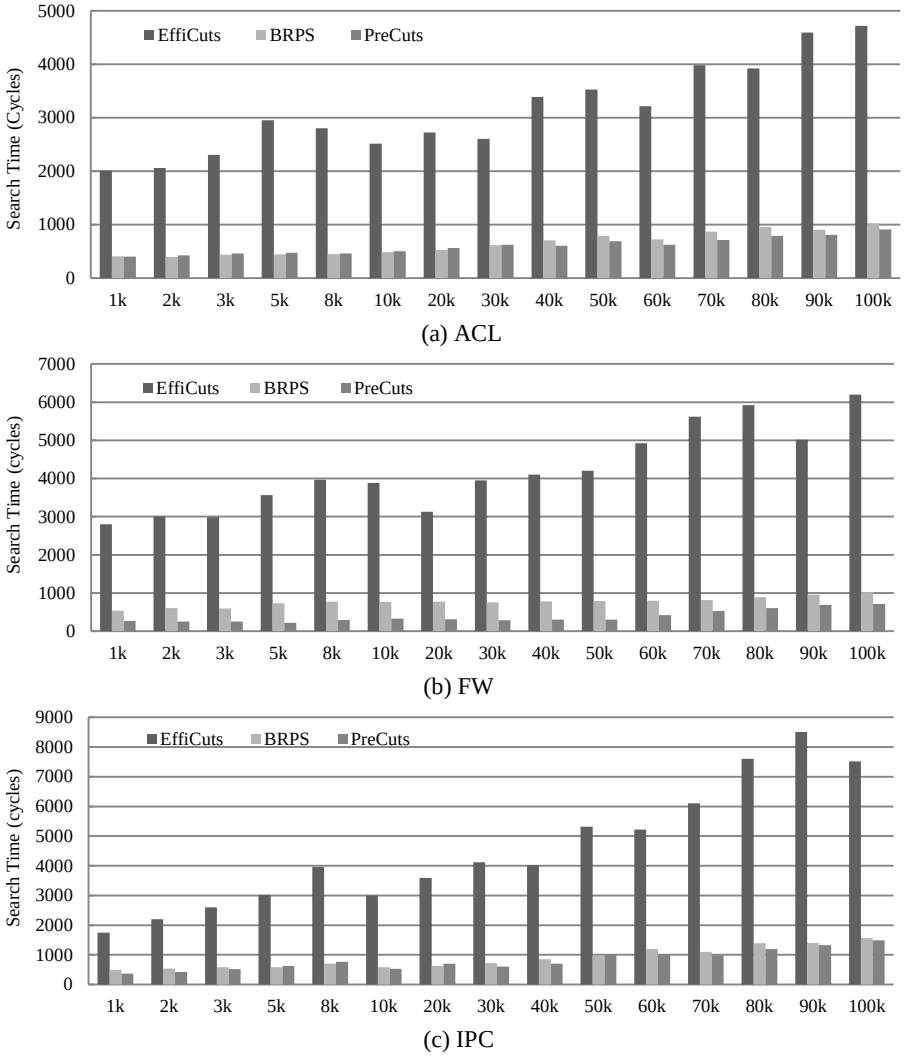


Fig. 13. Search Cycles

BRPS has the similar performance with PreCuts thanks to its huge amount of completed optimized heuristics. However, in practice, these optimized heuristics are hardly realizable. On the contrary, PreCuts is comparable with or even surpasses BRPS in time performance

without any extra optimization. This simplicity of PreCuts ensures its easy realization. With the same time performance but easier realization, PreCuts is superior to BRPS.

5.4 Update Speed

We adapt the method in [18] to test the update performance. First, we randomly extract 10% of the rules from the ruleset and build the search structure according to the remaining 90% of the rules. Then we randomly insert the extracted rules into the search structure to obtain the insertion cycles. After that, we randomly select another 10% of rules from the classifier and delete them from the search structure to obtain the deletion cycles. The average of the insertion and deletion cycles is obtained to judge the update performance. Since EffiCuts has no specific algorithms for incremental update, we only compare the update performance of BRPS and PreCuts, as is shown in **Fig. 14**. The average number of cycles that PreCuts needs in an update is 95% less than BRPS.

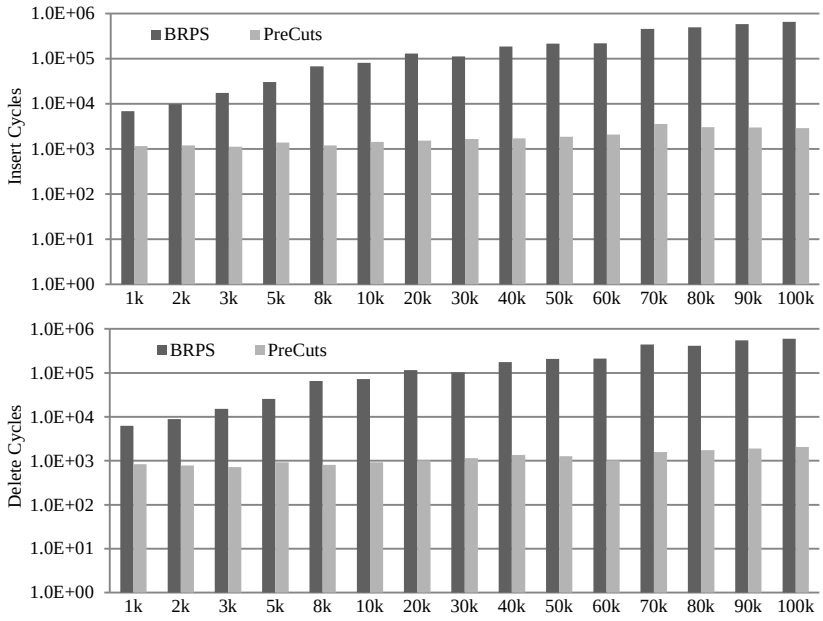


Fig. 14. Update Cycles

It can also be seen from **Fig. 14** that, compared to BRPS, the number of the cycles that the incremental update of PreCuts needs will not increase significantly as the ruleset gets larger. This is mainly because the average depth of the decision tree of PreCuts does not grow as the ruleset gets larger. This almost keeps the number of the nodes traversed unchanged when PreCuts makes incremental updates to rulesets of various sizes. Consequently, PreCuts can sustain an excellent performance in making incremental updates to larger rulesets.

6. Conclusion

Previous packet classification algorithms mainly focus on time and space performance; however, the update performance is often overlooked. In this paper, we propose PreCuts

which achieves fast updates. To improve the update speed without reducing the time and space performance, we design a novel 3-layer search structure. Considering the characteristics of the IP field of the rules, the ruleset partition on the three levels is carried out using three different ways, namely, *Byte Index Cutting*, *Prefix Length Cutting* and *Bit Partition*. The experimental results show that our scheme outperforms other schemes not only in terms of classification speed and memory usage but also in terms of update speed.

References

[1] Fong J, Wang X, Qi Y, et al., "ParaSplit: a scalable architecture on FPGA for terabit packet classification," in *Proc. of IEEE 20th Annual Symposium on High-Performance Interconnects*, pp. 1-8, August, 2012. Article (CrossRef Link)

[2] Yaxuan Q, Lianghong X, Baohua Y, et al. "Packet classification algorithms: from theory to practice," in *Proc. of IEEE INFOCOM'09*, pp. 648-656, April, 2009. Article (CrossRef Link)

[3] Gupta P and Mckeown N, "Packet classification using hierarchical intelligent cuttings," in *Proc. of Hot Interconnects VII*, pp. 34-41, August, 1999. Article (CrossRef Link)

[4] B. Xu, D. Jiang and J. Li, "HSM: a fast packet classification algorithm," in *Proc. of IEEE 19th International Conference on Advanced Information Networking and Applications*, pp. 987-992, March, 2005. Article (CrossRef Link)

[5] F. Baboescu and G. Varghese, "Scalable packet classification," *ACM SIGCOMM Computer Communication Review*, vol.31, no. 4, pp. 199-210, August, 2001. Article (CrossRef Link)

[6] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34-41, 2000. Article (CrossRef Link)

[7] S. Singh, F. Baboescu, G. Varghese and J. Wang, "Packet classification using multidimensional cutting," in *Proc. of ACM SIGCOMM'03*, pp. 213-224, August, 2003. Article (CrossRef Link)

[8] B. Vamanan, G. Voskuilen and T. N. Vijaykumar, "Efficuts: optimizing packet classification for memory and throughput," in *Proc. of ACM SIGCOMM'10*, pp. 207-218, August, 2010. Article (CrossRef Link)

[9] C. Yeim-Kuan and W. Yu-Hsiang, "CubeCuts: a novel cutting scheme for packet classification," in *Proc. of IEEE 26th International Conference on Advanced Information Networking and Applications Workshops*, pp. 274-279, March, 2012. Article (CrossRef Link)

[10] C. Yeim-Kuan and C. Han-Chen, "Layered cutting scheme for packet classification," in *Proc. of IEEE International Conference on Advanced Information Networking and Applications*, pp. 675-681, March, 2011. Article (CrossRef Link)

[11] Y. Ma and S. Banerjee, "A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification," in *Proc. of ACM SIGCOMM'12*, pp. 335-346, August, 2012. Article (CrossRef Link)

[12] C. R. Meiners, A. X. Liu and E. Torng, "TCAM Razor: a systematic approach towards minimizing packet classifiers in TCAMs," in *Proc. of IEEE International Conference on Network Protocols*, pp. 266-275, October, 2007. Article (CrossRef Link)

[13] C. R. Meiners, A. X. Liu and E. Torng, "Topological transformation approaches to optimizing TCAM-based packet classification systems," *IEEE/ACM Transactions on Networking*, vol. 19, no. 1, pp. 237-250, June, 2011. Article (CrossRef Link).

[14] B. Vamanan and T. N. Vijaykumar, "TreeCAM: decoupling updates and lookups in packet classification," in *Proc. of 7th Conference on Emerging Networking Experiments and Technologies*, pp. 27, December, 2011. Article (CrossRef Link)

[15] C. R. Meiners, A. X. Liu and E. Torng, "Bit weaving: a non-prefix approach to compressing packet classifiers in TCAMs," *IEEE/ACM Transactions on Networking*, vol. 20, no. 2, pp. 488-500, April, 2012. Article (CrossRef Link)

[16] D. E. Taylor and J. S. Turner, "ClassBench: a packet classification benchmark," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 499-511, June, 2007. Article (CrossRef Link)

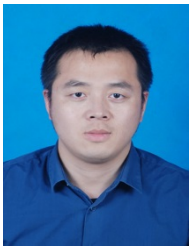
[17] B. Yang, J. Fong, W. Jiang, Y. Xue and J. Li, "Practical multi-tuple packet classification using

dynamic discrete bit selection," *IEEE Transactions on Computers*, vol. 63, no. 2, pp. 424-434, January, 2014. Article (CrossRef Link)

[18] C. Yeim-Kuan, "Efficient multidimensional packet classification with fast updates," *IEEE Transactions on Computers*, vol. 58, no. 4, pp. 463-479, February, 2009. Article (CrossRef Link)

Weitao Han received the B.E. degree in Communication Engineering from Hebei University of Technology, Tianjin, China, in 2011. He is currently a M.S. student in National Digital Switching System Engineering & Technological R&D Center (NDSC), Zhengzhou, China. His research interests include high-performance networks, packet classification algorithms, Internet router design and deep packet inspection architectures.

Peng Yi received the B.E. degree in Communication Engineering from Zhengzhou University, Zhengzhou, China, in 2000, and the M.S. and Ph.D. degrees in Computer Science from National Digital Switching System Engineering & Technological R&D Center (NDSC), Zhengzhou, China, in 2003 and 2006, respectively. He is currently an associate professor in NDSC. His research interests include network virtualization, computer networking, computer architecture and deep packet inspection.



Le Tian received the B.E. degree in Communication Engineering from Zhengzhou University, Zhengzhou, China, in 2010, and the M.S. degrees in Computer Science from National Digital Switching System Engineering & Technological R&D Center (NDSC), Zhengzhou, China, in 2013. He is currently a Ph.D. student in NDSC, his research interests include high-performance networks, packet classification algorithms, network virtualization and heterogeneous access networks.