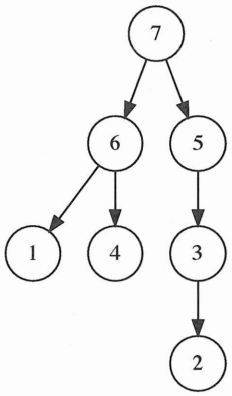

Figure 3.2: The graph $G_{\mathcal{R}}$ for rule list $\mathcal{R}$ in Table 3.2.

Table 3.3: Rewriting a non–included rule r_3 to $r_3^{'}$ and $r_3^{''}$.

Filter $\mathcal{R}$	
r_3	$*\,0\,0\,0\,1\,*$
r_4	$0\,*\,*\,0\,1\,*$

Filter $\mathcal{R}^{'}$	
$r_3^{'}$	$0\,0\,0\,0\,1\,*$
$r_3^{''}$	$1\,0\,0\,0\,1\,*$
r_4	$0\,*\,*\,0\,1\,*$

The mean values of 10 trials are shown in Fig. 3.3. Clearly, the proposed method decreased the latency compared with the SGM [77] and SWBP [61] in all cases. In particular, the difference in latency between SGM and the proposed algorithm with 4000 and 5000 rule is greater than that with 1000 and 2000. Thus, the greater the number of rules, the more efficient the proposed algorithm becomes.

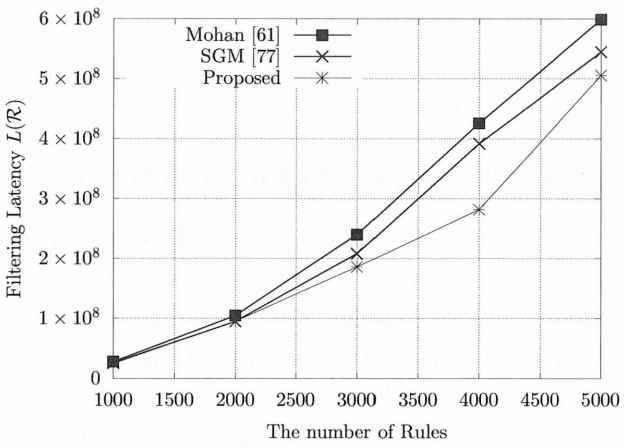

Figure 3.3: Latency of the proposed method, SGM [77], and the SWBP [61].

Table 3.4: Example of a rule list.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{U}}$
$r_1^P = * \; 0 \; * \; 1$	4
$r_2^P = 0 \; 0 \; 0 \; 0$	1
$r_3^P = 0 \; * \; 0 \; 0$	1
$r_4^D = 0 \; * \; 1 \; *$	3
$r_5^P = * \; 1 \; * \; 1$	3
$r_6^P = * \; * \; * \; 1$	0
$r_7^D = * \; * \; * \; *$	4
$L(\mathcal{R},\mathcal{U}) = 60$	

3.2.2 Difference between Single–Machine Job Sequencing Problem and RORO

Most of the time, the dependency relation determines the precedence relation, i.e., if $r_i^{e_i}$ and $r_j^{e_j}$ are dependent under $i < j$, $r_j^{e_j}$ generally cannot be placed ahead of $r_i^{e_i}$. For r_3^P and r_7^D in Table 3.4, if r_7^D is placed ahead of r_3^P, then packet 0100 is evaluated as D, not P. Thus, r_7^D depending on r_3^P means that r_7^D is placed behind r_3^P. However, although r_2^P and r_7^D are dependent, r_7^D can be placed ahead of r_2^P when r_3^P is placed ahead of r_7^D. Only packet 0000 may cause a policy violation with respect to placing r_7^P ahead of r_2^P. However, if r_3^P is placed ahead of r_7^D, because 0000 is evaluated as P by r_3^P, placing r_7^D ahead of r_2^P does not cause policy violation. Thus, $r_j^{e_j}$ depending on $r_i^{e_i}$ does not always mean that $r_j^{e_j}$ can not be placed ahead of $r_i^{e_i}$.

Avoiding this characteristic of RORO, we can obtain the optimal rule order, if the graph of dependency relations is a forest of oriented trees [34]. That is, by using the algorithm for single–machine job sequencing [34], we can achieve a better ORO solution, if the graph of dependency relations is a forest of oriented trees.

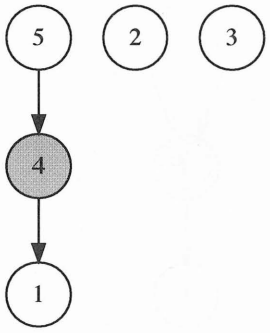

Figure 3.4: Forest of oriented trees.

3.3 Rewriting Rules to Oriented Trees

3.3.1 Rewriting Rules

The proposed rule rewriting algorithm begins with an empty rule list $\mathcal{R}'$, obtains a rule from the input rule list, and inserts it into $\mathcal{R}'$ from r_1 to r_n. If inserting rule r_i into $\mathcal{R}'$ does not make the dependency graph of $\mathcal{R}'$ into a forest of oriented trees, the algorithm expands r_i into $r_{i_1}, r_{i_2}, \ldots r_{i_k}$ so that the graph of $\mathcal{R}' \cup \{r_{i_1}, r_{i_2}, \ldots, r_{i_k}\}$ becomes a forest of oriented trees. Note that the algorithm expands r_i so that at least one rule in $r_{i_1}, r_{i_2}, \ldots, r_{i_k}$ is included in some rule in $\mathcal{R}'$. For example, rules r_1^P to r_5^P in Table 3.4 form a forest of oriented trees as shown in Fig. 3.4. Adding r_6^P to the rule list consisting of r_1^P to r_5^P violates the property that the dependency graph is a forest of oriented trees as illustrated in Fig. 3.5. The rewriting algorithm expands r_6^P to $r_{6_1}^P$, $r_{6_2}^P$, and $r_{6_3}^P$, as shown in Table 3.5 and inserts $r_{6_1}^P$ and $r_{6_2}^P$ into the rule list. Because the set of packets matching $r_{6_3}^P$ is included in that of r_4^P, i.e., $r_{6_3}^P$ is redundant, $r_{6_3}^P$ is not inserted into the rule list. The resulting graph of Table 3.5 is shown in Fig. 3.6 as a forest of oriented trees.

3.3.2 Merging Rules

Because expanding a rule consisting of many '$*$' generates a lot of rules, the algorithm attempts to remove such rules. If rules r_i^e and r_j^f satisfy all of the following conditions, they can be merged:

1. the evaluation types of r_i^e and r_j^f are the same,
2. r_j^f can be placed just behind r_i^e,
3. (a) r_i^e is included in r_j^f or r_j^f is included r_i^e, or
   (b) the difference in the conditions of r_i^e and r_j^f is just one bit.

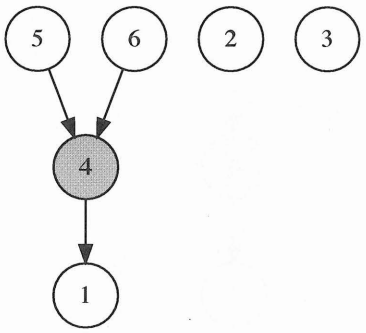

Figure 3.5: Component of $1, 4, 5$, and 6 is not an oriented tree.

Table 3.5: Expand r_6^P to $r_{6_1}^P$, $r_{6_2}^P$, and $r_{6_3}^P$.

Filter $\mathcal{R}$
$r_4^D = 0 * 1 *$
$r_{6_1}^P = 1 * * 1$
$r_{6_2}^P = 0 * 0\ 1$
$r_{6_3}^P = 0 * 1\ 1$

As an example, even if r_1^P and r_2^P in Table 3.4 have the same evaluation types, because r_1^P is not included in r_2^P and r_2^P is not included in r_1^P, and the number of different bits in those rules is 3, they can not be merged. Because the evaluation types of r_4^D and r_6^P are different, those rules cannot be merged. Even though r_3^P is included in r_6^P and their evaluation types are the same, because r_6^P can not be placed just behind r_3^P, those rules cannot be merged. In contrast, r_5^P and r_6^P can be merged.

The proposed rule reconstruction algorithm merges redundant rules before rewriting the rules.

3.3.3 Experiments

We demonstrate the efficiency of the proposed algorithms by presenting the results of experiments conducted on an Intel Core i7–7820X 3.60 GHz CPU with 131 GB main memory under Cent OS 7.3. We implemented the proposed algorithm, the rule reordering algorithm SGM [77],

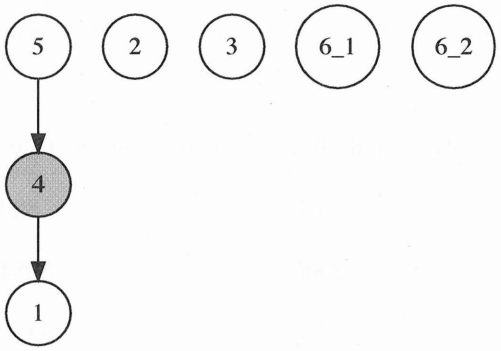

Figure 3.6: Add r_{6_1} and r_{6_2}.

the reordering algorithms of Tanaka et al. [75,76] and the latest reordering algorithm of Mohan et al. [61] in C. The rules and headers used in the experiments were generated by the standard packet classification benchmark ClassBench [80]. Because ClassBench generates rules without an action, we added an evaluation type P or D to each rule generated by ClassBench with a probability of 1/2. Using ClassBench, we generated standard 5–tuple rules and headers consisting of source/destination addresses, source/destination port numbers and protocol numbers. Because the lengths of these components are 32, 32, 16, 16, and 8 bits respectively, the length of the condition of the rule and header was 104 bits. The number of headers was about 100 times greater than the number of rules.

We measured the latency and the reordering time of a rule list for every algorithm. The medians and the averages of 10 trials for reordering and reconstruction are shown in Figs. 3.7, 3.8, 3.10, and 3.11. Note that we plot the reordering and reconstructing times on a logarithmic scale in Figs. 3.10 and 3.11.

Figure 3.9 shows that the proposed algorithm can increase the latency compared with the given rule list with 1000 and 2000 rules. As shown in Figs. 3.10 and 3.11, the proposed rule reconstruction algorithm is slower than the other rule reordering algorithms. However, Figs. 3.7 and 3.8 show that the proposed rule reconstruction algorithm decreases the median and average latency.

Comparing Figs. 3.7 and 3.8, the median is clearly a better indicator of the proposed algorithm than the average. Table 3.6 suggests that the proposed reconstruction algorithm increases the average number of rules. Indeed, Table 3.7 and Fig. 3.9 show that the proposed algorithm inefficiently increases the number of rules and can, in the worst case, increase the latency. These results indicate that the proposed algorithm is influenced by the characteristics of the given rules. It is important to select the appropriate algorithm for the characteristics of the rule list.

Table 3.6: The average # of rules generated by the proposed algorithm.

given rule list	reconstructed rule list
1000	8236.8
2000	39849.6
3000	53125.4

Table 3.7: The maximum # of rules generated by the proposed algorithm.

given rule list	reconstructed rule list
1000	20811
2000	86548
3000	177906

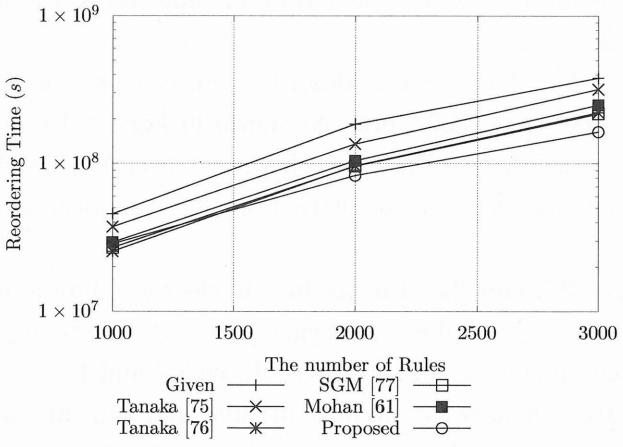

Figure 3.7: Median latency.

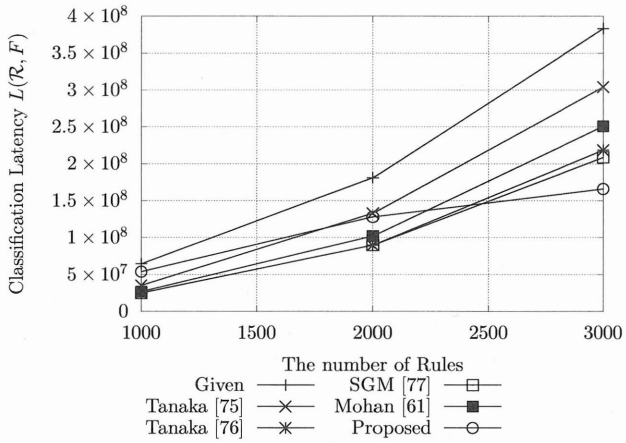

Figure 3.8: Average latency.

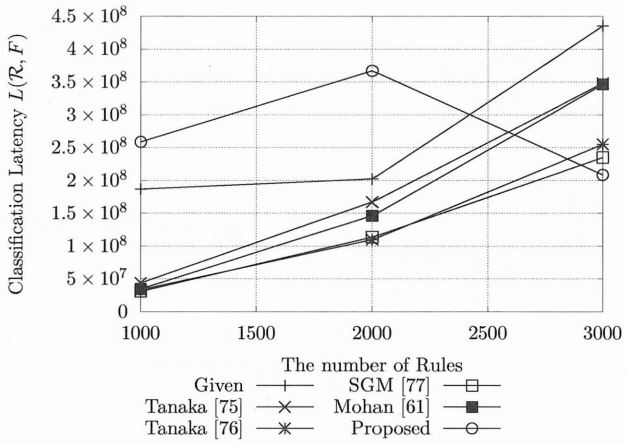

Figure 3.9: Maximum latency.

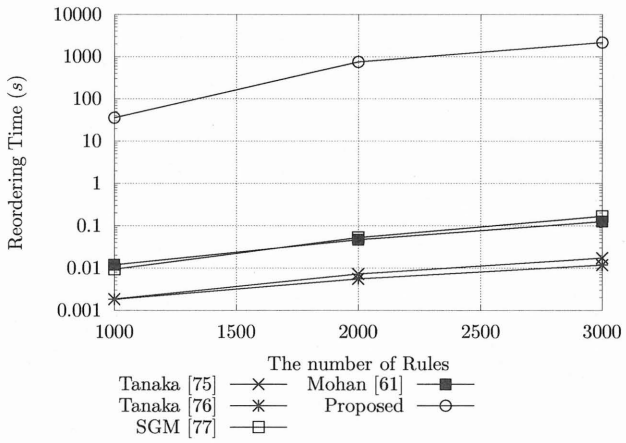

Figure 3.10: Median reordering/reconstructing time.

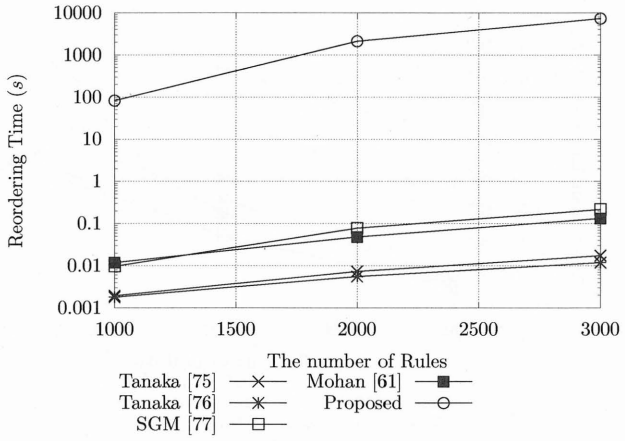

Figure 3.11: Averages of Reordering/Reconstructing Times.

Chapter 4

Determining Equivalence of the Rule List Policies

Any algorithm for reordering or reconstructing the rule list must retain its original classification policy. In this chapter, we present an algorithm that determines the equivalence of two rule lists by constructing the ZDD representing these policies. The effectiveness of this algorithm is confirmed through a series of experiments.

Given two decision lists L_1 and L_2, deciding whether they represent the same Boolean function is **coNP**–complete [23]. Because deciding equivalence of two decision lists can be reduced to deciding equivalence of two rule lists, deciding equivalence of two rule list is also **coNP**–complete.

Therefore, determining the equivalence of reordered or reconstructed rule lists is generally difficult. However, for rule lists of practical size, we can identify policy violations by constructing appropriate ZDDs.

4.1 Determining Equivalence of Rule List Policies via ZDD

A procedure for constructing a ZDD according to $M(r_i)$ is shown in Algorithm 8. Further, a ZDD construction method for $M(r_i)$ using a BDD/ZDD library such as CUDD [68] is given in Algorithm 9.

Algorithm 8 first constructs ZDD Z according to the set of the only null combination $00\cdots0$ and sets *ptr* to point Z. By scanning the condition $r_i^e = b_1 b_2 \cdots b_w$, it then forms a non–terminal node with numeral k whose 1 edge points to *ptr* and 0 edge points to the 0–terminal node if $b_k = 1$, and a non–terminal node with numeral k in which both edges point to *ptr* if $b_k = *$, and then *ptr* points to a currently generated node. An example of constructing a ZDD using Algorithm 8 is shown in Fig. 4.1.

First, Algorithm 9 constructs ZDD Z according to the set of the only null combination. Then, by scanning the condition $r_i^e = b_1 b_2 \cdots b_w$, it applies the change($k$) operation to Z if $b_k = 1$, and takes the union of Z and Y if $b_k = *$, where Y is Z.change(k). An example of how

Algorithm 8: make ZDD for Rule r

input : rule $r_i^e = b_1 b_2 \cdots b_w$
output: ZDD for rule r_i^e

```
1  i ← w ;
2  pointer ptr to 1–terminal node;
3  while i > 0 do
4      if b_i = '*' then
5          make a non-terminal node v whose variable is i and its left and right edges
           point to ptr ;
6          ptr points to v;
       end
7      if b_i = '1' then
8          make a non-terminal node v whose variable is i and its left edge point to 0 and
           right edges point to ptr ;
9          ptr points to v;
       end
10     i ← i − 1 ;
   end
11 return ZDD pointed by ptr ;
```

to construct the ZDD using Algorithm 9 is shown in Fig. 4.2.

Algorithm 10 describes the method of constructing the ZDD according to $\mathcal{R} = \langle r_1^{e_1}, r_2^{e_2}, \ldots, r_n^{e_n} \rangle$. First, Algorithm 10 constructs the null combination Z and then scans rules in descending order from $r_n^{e_n}$. The method constructs ZDD Z_i for r_i^e. It sets Z to $Z \cup Z_i$ if $e = D$, or to $Z \setminus Z_i$ otherwise. An example of constructing the ZDD using Algorithm 10 is shown in Fig. 4.3.

The equivalence of rule lists $\mathcal{R}_1$ and $\mathcal{R}_2$ can be confirmed by constructing ZDD_1 and ZDD_2 and checking whether the two ZDDs are the same. The CUDD package and the algorithms presented here implement the ZDD as a hash table and use shared ZDD techniques [59]. Whether two ZDDs, A and B are the same depends on whether the addresses of the two ZDDs are the same. Thus, the equivalence of ZDDs can be checked in one step. Thus, we can determine the equivalence of rule lists if the ZDDs for these rule lists can be constructed.

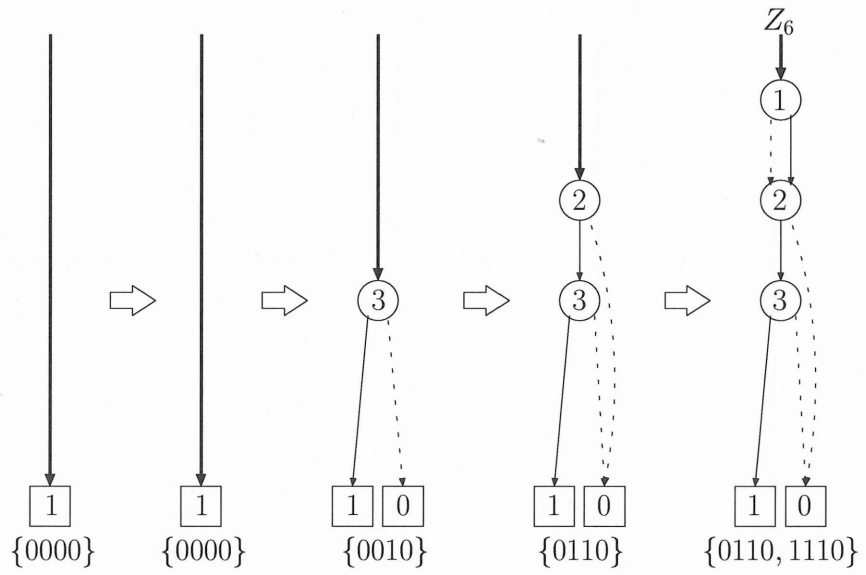

Figure 4.1: Construction process of ZDD according to the set of packets matching r_6^D by Algorithm 8.

Algorithm 9: make ZDD for Rule r using BDD/ZDD Library like CUDD [68]

input : rule $r_i^e = b_1 b_2 \cdots b_w$
output: ZDD for rule r_i^e

```
1 make ZDD Z for the set of null combination {00...0};
2 i ← w ;
3 while i > 0 do
4 |   if b_i = '1' then Z.change(i);
5 |   else if b_i = '*' then
6 |   |   Z ← Z ∪ Z.change(i);
  |   end
7 |   i ← i − 1 ;
  end
8 return Z;
```

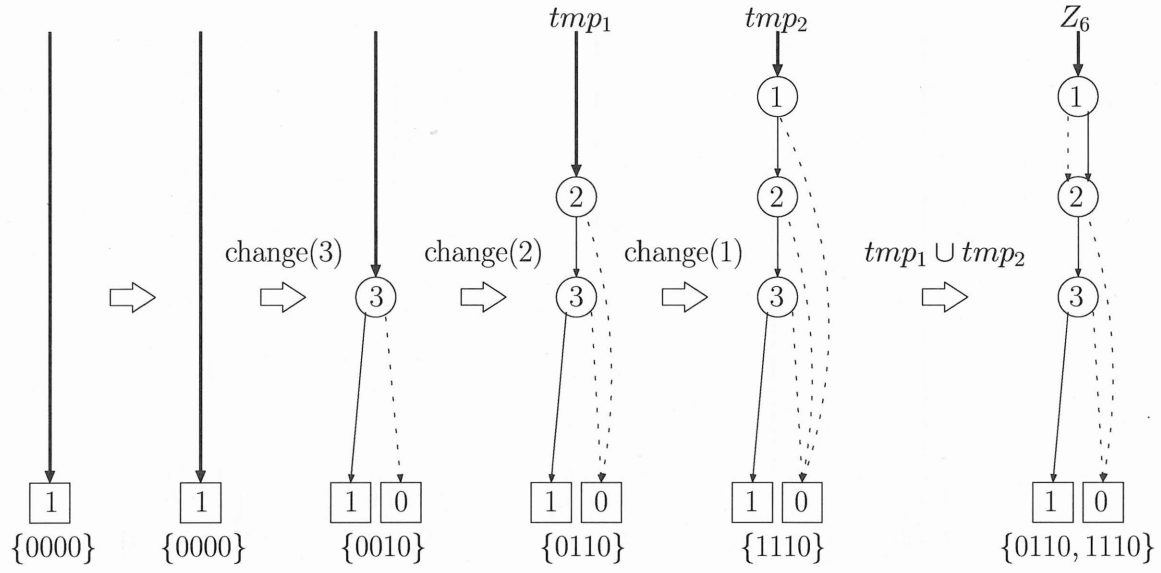

Figure 4.2: Construction process of ZDD according to the set of packets matching r_6^D by Algorithm 9.

Algorithm 10: make ZDD for Rule List $\mathcal{R}$

input : rule list $\mathcal{R} = \langle r_1^{e_1}, r_2^{e_2}, \ldots, r_n^{e_n} \rangle$
output: ZDD for rule list $\mathcal{R}$

```
1 make zdd Z for the empty set ;
2 i ← n ;
3 while i > 0 do
4 |   make zdd Z_i for r_i^{e_i} by Algorithm 9 ;
5 |   if e^i = D then Z ← Z ∪ Z_i ;
6 |   else Z ← Z \ Z_i ;
7 |   i ← i − 1 ;
  end
8 return Z ;
```

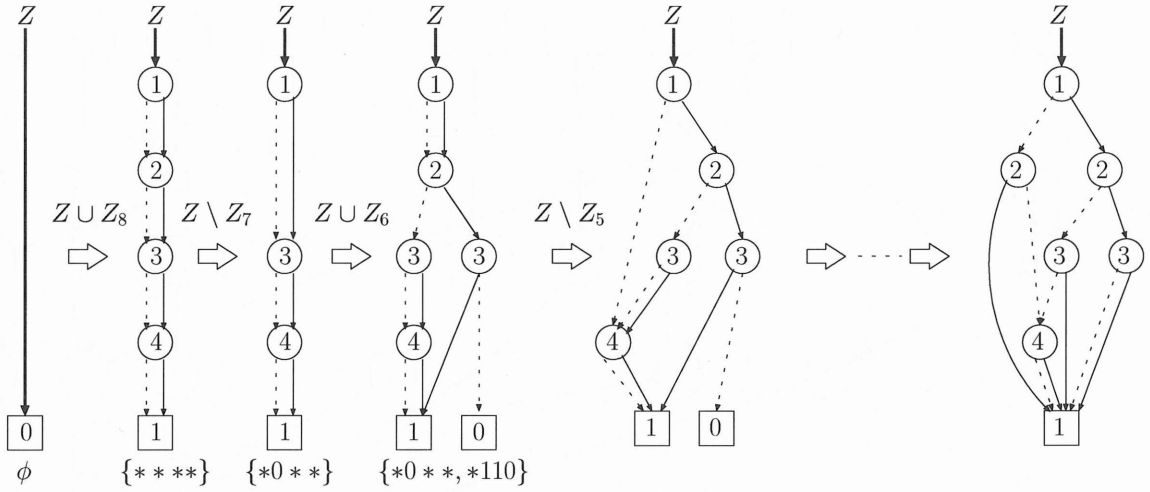

Figure 4.3: Construction process for ZDD via Algorithm 10

Table 4.1: List 1.

Filter $\mathcal{R}_1$
$r_1^B = * \; 1 \; 1 \; 0$
$r_2^A = 0 \; * \; 1 \; *$
$r_3^B = 1 \; 0 \; * \; 0$
$r_4^C = 1 \; 1 \; 0 \; 1$
$r_5^C = 1 \; 1 \; 1 \; 1$
$r_6^D = 1 \; * \; * \; *$
$r_7^A = * \; 1 \; * \; 1$
$r_8^D = * \; 0 \; * \; 0$
$r_9^C = * \; * \; * \; *$

Table 4.2: List 2.

Filter $\mathcal{R}_2$
$r_1^A = 0 \; 0 \; 1 \; 0$
$r_2^B = * \; * \; 1 \; 0$
$r_3^C = 0 \; 0 \; 0 \; 1$
$r_4^A = 0 \; * \; * \; 1$
$r_5^D = 1 \; 1 \; 0 \; 0$
$r_6^C = * \; 1 \; * \; *$
$r_7^B = 1 \; 0 \; 0 \; 0$
$r_8^D = 1 \; 0 \; * \; 1$
$r_9^D = * \; * \; * \; *$

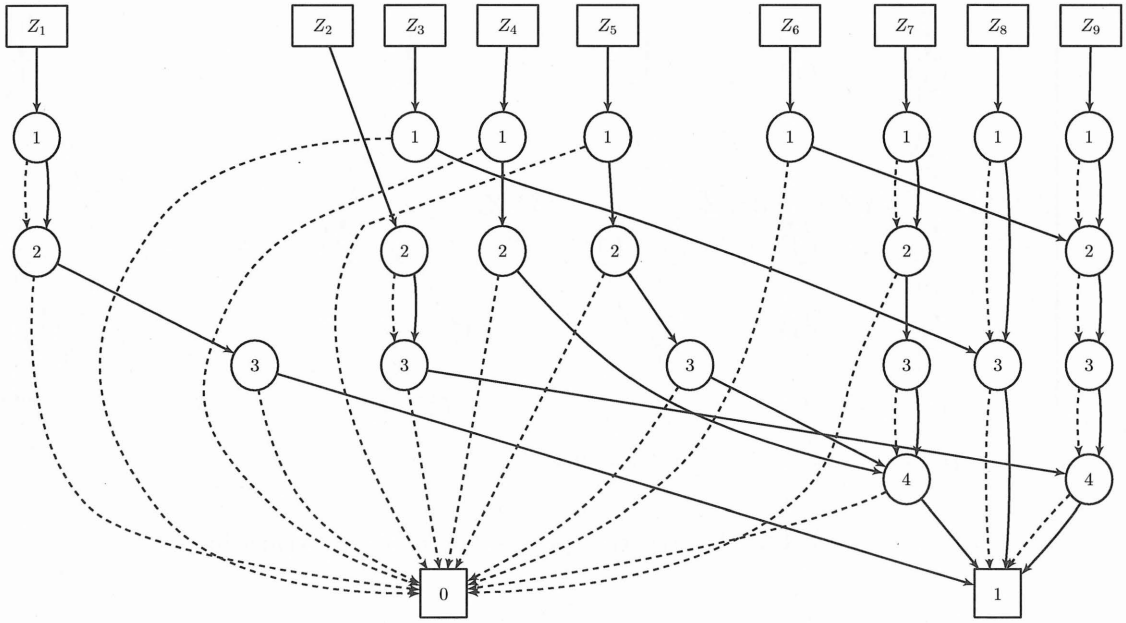

Figure 4.4: ZDDs for rule list in Table 4.1.

4.2 Determining Equivalence of Rule Lists with Multiple Actions

In the previous subsection, we presented the algorithms for determining the equivalence of rule lists with only two actions P and D. In this section, using Tables 4.1 and 4.2, we introduce an algorithm for determining the equivalence of rule lists that are not limited to two actions. In the following, such a rule list is called a rule list with multiple actions.

To determine the equivalence of rule lists with multiple actions $A_1, A_2, \ldots, A_m$, the determi-

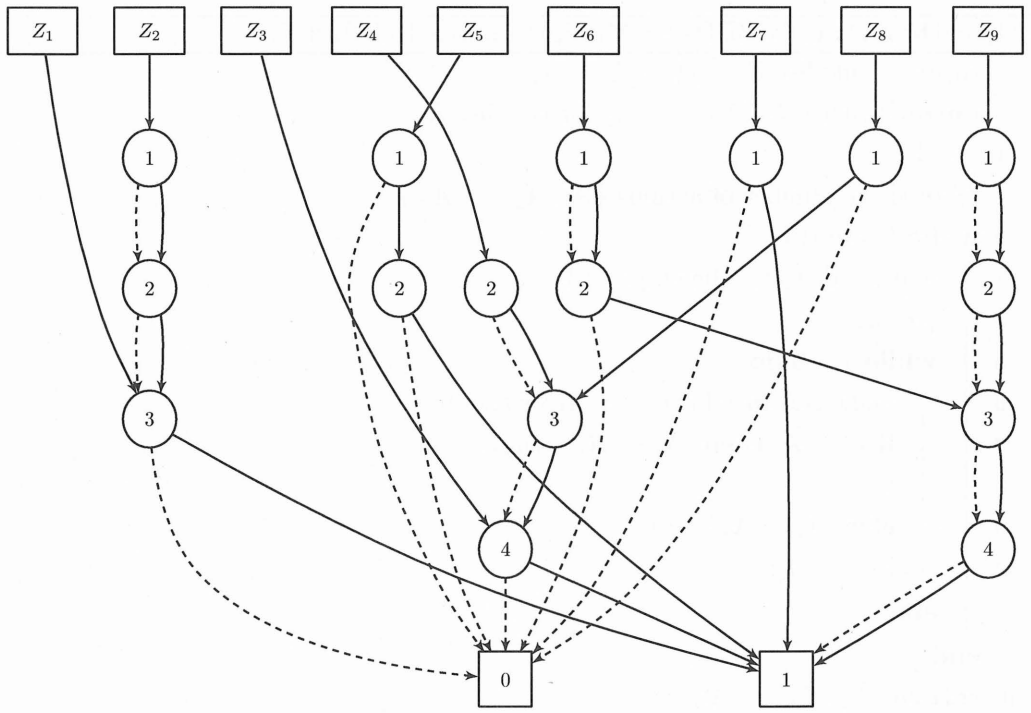

Figure 4.5: ZDDs for rule list in Table 4.2.

Table 4.3: A function $f : \mathcal{P} \to \{A, B, C, D\}$ represented by Tables 4.1 and 4.2.

$0000 \mapsto D$	$0100 \mapsto C$	$1000 \mapsto B$	$1100 \mapsto D$
$0001 \mapsto C$	$0101 \mapsto A$	$1001 \mapsto D$	$1101 \mapsto C$
$0010 \mapsto A$	$0110 \mapsto B$	$1010 \mapsto B$	$1110 \mapsto B$
$0011 \mapsto A$	$0111 \mapsto A$	$1011 \mapsto D$	$1111 \mapsto C$

nation method constructs ZDDs as described in the previous section for m actions.

The procedure for constructing ZDDs $X_1, X_2, \ldots, X_m$ for a rule list with multiple actions is described in Algorithm 11.

Algorithm 11 constructs ZDD X_i for an action A_i, repeating the process from line 2 to line 9. The difference between Algorithms 10 and 11 is whether A_i is specified on line 7.

For rule lists $\mathcal{R}_1$ and $\mathcal{R}_2$, equivalence can be determined by constructing $2m$ ZDDs $X_{11}, X_{12}, \ldots, X_{1m}$ and $X_{21}, X_{22}, \ldots, X_{2m}$ and checking the equality of ZDDs X_{1k} and X_{2k}.

4.3 Experiments

To confirm the efficiency of the proposed algorithm for a rule list with only two actions P and D, we implemented it in C under the Cent OS Release 6.10 (Final) on an Intel Core i5-3470 3.20 GHz CPU with 2 GB main memory. We generated the rule lists using ClassBench [80] and

Algorithm 11: make ZDDs for Multiple Actions Rule List $\mathcal{R}$

input : rule list $\mathcal{R} = \langle r_1^{e_1}, r_2^{e_2}, \ldots, r_n^{e_n} \rangle$

output: ZDDs $X_1, X_2, \ldots, X_m$ for rule list $\mathcal{R}$

```
1  i ← 1 ;
   // m is the number of actions {A_1, A_2, ..., A_m};
2  while i ≤ m do
3  |  make zdd X_i for the empty set ;
4  |  j ← n;
5  |  while j > 0 do
6  |  |  make zdd tmp for r_j^{e_j} by Algorithm 9 ;
7  |  |  if e^j = A_i then X_i ← X_i ∪ tmp ;
   |  |  ;
8  |  |  else X_i ← X_i \ tmp ;
9  |  |  i ← i − 1 ;
   |  end
   end
10 return X_1, X_2, ..., X_m ;
```

measured the time for required to determine the equivalence of the rule lists.

Figure 4.7 shows the time required to construct the ZDD by proposed method, where the units of measurement are seconds. Figure 4.7 shows that the time required to determine the equivalence of the rule lists is less than 0.2 s. This shows the remarkable effectiveness of the proposed algorithm. Thus, the proposed method is effective for rule lists of practical size.

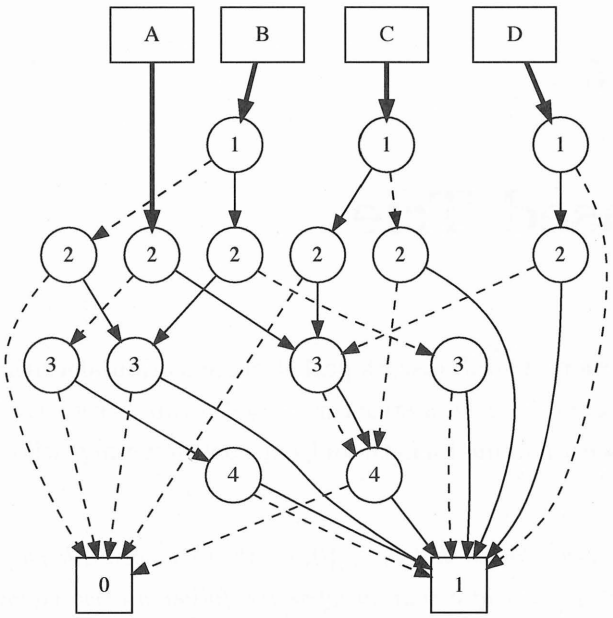

Figure 4.6: ZDDs for policies according to Tables 4.1 and 4.2.

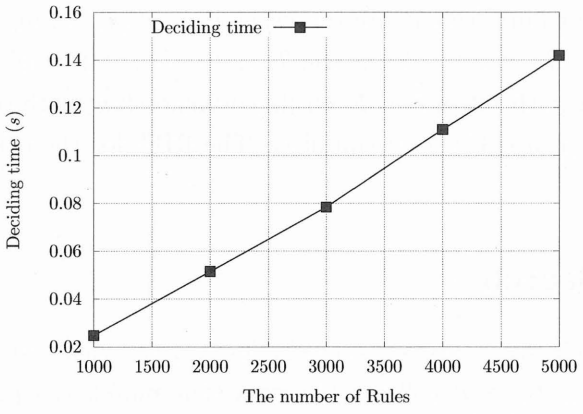

Figure 4.7: Deciding time (s).

Chapter 5

Run-Based Trie

In the following section, we will discuss packet classification algorithms.

Mikawa et al. proposed a data structure called a run–based trie (RBT) [57]. They define a run as a bitstring with maximal length and not containing any wild–cards. A run is defined as follows:

Definition 5.0.1. *(run form) Let $r_i \in \{0,1,*\}^w$ be a bitmask rule of length w. A substring $b_i b_{i+1} \cdots b_j$ $(1 \leq i \leq j \leq w)$ of r that satisfies the following two conditions is called run;*

i) $b_k = 0 \vee b_k = 1 \quad (i \leq k \leq j)$

ii) $(i \geq 2 \Rightarrow b_{i-1} = *) \ \wedge \ (j \leq w-1 \Rightarrow b_{j+1} = *)$.

For instance, a bitmask rule of length 16

$$* \ * \ 0 \ 1 \ * \ * \ 0 \ 0 \ 1 \ * \ * \ * \ 1 \ 0 \ 1 \ 0$$

consists of 3 runs, 01, 001, and 1010. These runs begin at the third, 7th, and 13th bits in the rule, respectively. Runs in a rule r_i are represented as $\rho_i^1, \rho_i^2, \ldots, \rho_i^k$ $(0 \leq k \leq \lceil w/2 \rceil)$. An RBT consists of w tries $T_1, T_2, \ldots, T_w$. Each trie T_k is constructed by placing the bit pattern of the run beginning at the k–th bit of $r_i \in \mathcal{R}$ on its corresponding path of T_k. In addition, we mark ρ_i^j on the path if the run is the j–th run of r_i. The RBT for the rule list in Table 5.1 is shown in Fig. 5.1.

5.1 Simple Search

A simple RBT search [57] traverses tries $T_1, T_2, \ldots, T_w$ with the bit patterns of the packet beginning at the k–th bit and collects the runs that match the pattern. The matched rules are then calculated from the collected runs and the highest priority rule in the match rules is returned. If there are no matching rules, the default rule r_n is returned. For example, packet 01010 traverses the heavy lines in Fig. 5.1 and collects runs ρ_1^1, ρ_5^1, ρ_4^1 and ρ_5^2. Because 01010 only matches rule r_5, the highest priority rule for 01010 is r_5.

Table 5.1: Bitmask rules.

	Filter $\mathcal{R}$
r_1	0 * * 0 1
r_2	1 * 1 1 *
r_3	1 0 0 * *
r_4	* 1 0 * 1
r_5	0 * * 1 *
r_6	* * * * *

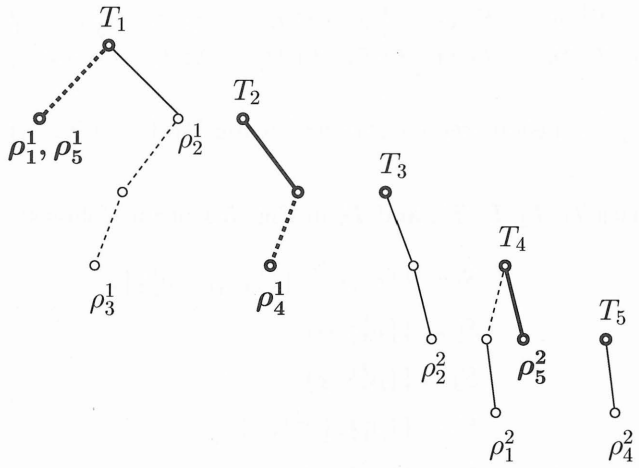

Figure 5.1: Run–based trie for rule list in Table 5.1.

Let us compute the time complexity of this simple search. First, to traverse tries $T_1, T_2, \ldots, T_k$, the simple search requires $w + (w-1) + \cdots + 1$ steps. The time complexity of traversing all tries is $O(w^2)$. Then, because the number of runs on the tries is at most $n \times \lceil w \rceil$, the time complexity required for comparing the runs is $O(nw)$. Thus, the time complexity of the simple search is $O(nw + w^2)$.

5.2 Decision Tree constructed from RBT

Mikawa et al. also proposed a search algorithm using a decision tree constructed from an RBT [57]. Because there is a limited number of patterns used to collect runs for each trie T_i in an RBT search, the patterns are enumerated as $S_1, S_2, \ldots, S_w$. For instance, the patterns used

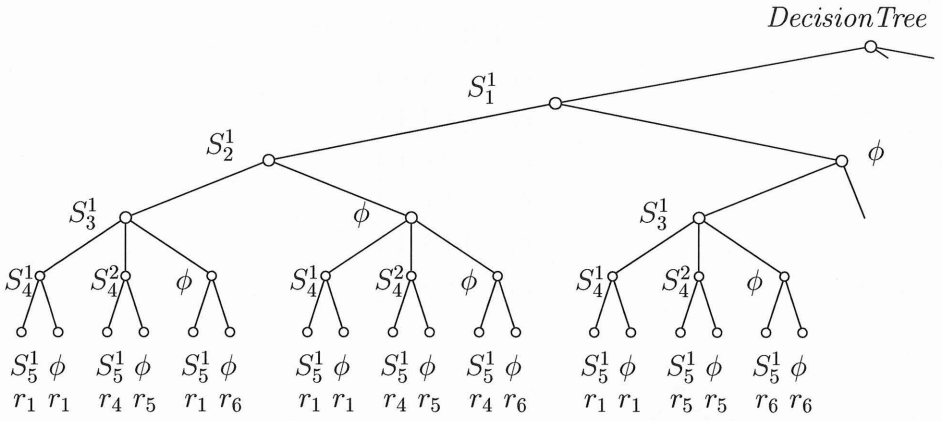

Figure 5.2: Decision tree constructed using RBT in Fig. 5.1 (rewritten).

to collect runs for tries T_1, T_2, T_3, T_4, and T_5 in Fig. 5.1 are as follows:

$$\begin{aligned} S_1 &= \{\{\rho_1^1, \rho_5^1\}, \{\rho_2^1\}, \{\rho_2^1, \rho_3^1\}\}, \\ S_2 &= \{\{\rho_4^1\}, \phi\}, \\ S_3 &= \{\{\rho_2^2\}, \phi\}, \\ S_4 &= \{\{\rho_1^2\}, \{\rho_5^2\}, \phi\}, \\ S_5 &= \{\{\rho_4^2\}, \phi\}, \end{aligned} \tag{5.1}$$

where ϕ denotes no match with a run.

Based on these patterns, the decision tree is constructed by taking the Cartesian product of $S_1 \times S_2 \times \cdots \times S_w$. The decision tree constructed from the patterns in Eq. 5.1 is shown in Fig. 5.2. Each path from the root to a leaf of the decision tree is equivalent to a search path obtained by traversing the RBT from T_1 to T_w. Computing the highest–priority rule for each path on the decision tree in advance, we obtain the highest priority rule by only traversing the decision tree using the RBT. The time complexity of this decision tree search is $O(w^2)$, because we can reach a leaf by traversing the RBT.

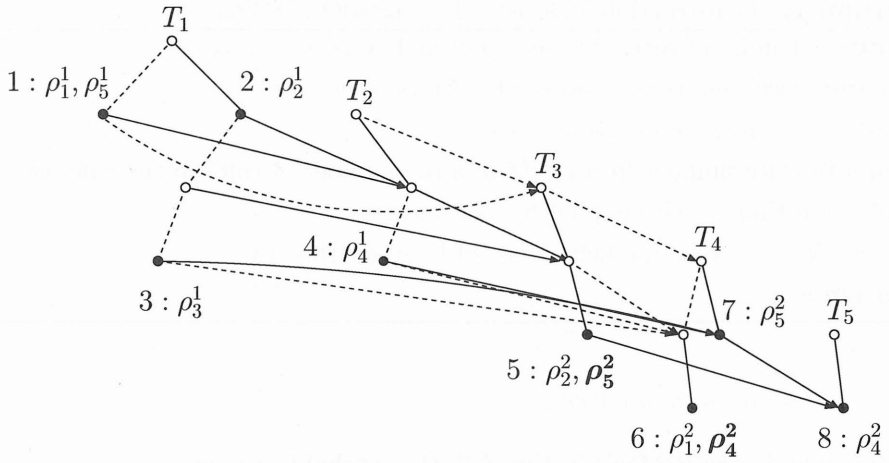

Figure 5.3: RBTwP for rule list in Table 5.1.

5.3 Decision Tree based on RBT with Pointers

5.3.1 RBT with Pointers

In the worst case, the RBT search accesses the bitstring of a packet w^2 times. Even if it reaches down to a node at depth k on T_k, the next starting point is the root of T_{k+1}. Because searching for runs at nodes of depth k means that bits $p[1]$ to $p[k+d-1]$ have already been referenced, traversing from the root of T_{k+1} to a node at depth k is redundant. Focusing on this point, we propose a search method that refers to the bits of a packet at most w times. This method adds new arcs to a non–terminal node that has no 0–arc or 1–arc. In addition, a run from a node on T_k is copied to nodes on T_1, ..., T_{k-1}. For the rule list in Table 5.2, an RBT with added arcs and copied runs is shown in Fig. 5.3. We call this an RBT with pointers (RBTwP). Traversing the RBTwP, we access the bitstring of a packet at most w times, in searching for the highest–priority rule for the packet. Thus, the time complexity and space complexity of this method are $O(w)$ and $O(nw)$, respectively.

5.3.2 Decision Tree based on RBTwP

As well as the decision tree constructed from naive RBT [57], we can construct a decision tree based on RBTwP.

First, we number the nodes that have run (denoted by the numeral on the left side of the colon in Fig. 5.3). We call this number the *node number*. The set of reachable node numbers from node i is limited according to Eq. 5.1. This reachable set is defined as follows:

Definition 5.3.1. *(Reachable set in RBTwP)*

$$N_i = \{\ k \mid \textit{there exists a path } p \textit{ from } i \textit{ to } k,\ \neg\exists j \in p\}, \tag{5.2}$$

Algorithm 12: ConstructionOfDecisionTreeBaesdOnRBTwP

input : Family of reachable sets *root* and $N_1, N_2, \ldots, N_k$

output: Decision Tree based RBT with pointers

1 make a root node of decision tree d ;

2 add a priority number $[n]$ to d // n is the number of rules in the rule list;

3 call Algorithm 13 with explicitly *root* and d;

// $N_1, N_2, \ldots, N_k$ is implicitly passed to Algorithm 13 ;

4 **return** d ;

where i, j, and k are node numbers.

For example, in the RBTwP in Fig. 5.3, the reachable sets are

$$\begin{aligned} root &= \{1, 2\}, \\ N_1 &= \{4, 5, 6, 7\}, \\ N_2 &= \{3, 4, 5, 6\}, \\ N_3 &= \{6, 7\}, \\ N_4 &= \{6, 7\}, \\ N_5 &= \{8\}, \\ N_6 &= \phi, \\ N_7 &= \{8\}, \\ N_8 &= \phi, \end{aligned} \tag{5.3}$$

where *root* is the root node of T_1.

Based on these sets, we construct the decision tree using the Cartesian product–like operation shown in Algorithm 12. Algorithm 12 forms a decision tree by calling Algorithm 13, in which lines 1–5 recursively make node i based on N_i and calculate the highest–priority rule at node i.

Because this decision tree can be searched by the RBTwP at most w steps, the time complexity of this decision tree search is $O(w)$.

Although this decision tree is very efficient in terms of search time complexity, it may have redundant sub–trees. As shown in Fig. 5.5, the sub–trees in the shaded area are redundant, because a node on the sub–tree has no priority rule $[i]$ greater than the highest rule on preceding nodes. Thus, such a sub–tree can be pruned.

Algorithm 13: SpanEdges

input : Reachable Set N and a node of decision tree v

1 **foreach** $i \in N$ **do**

2 make a decision tree node i and an edge from v to i ;

3 calculate the candidate highest priority rule number c ;

4 **if** *c is less than the priority rule of ancient node* **then**

5 add $[c]$ to i;

end

6 call Algorithm 13 with N_i and i ;

end

Decision Tree constructed on RBT with pointers

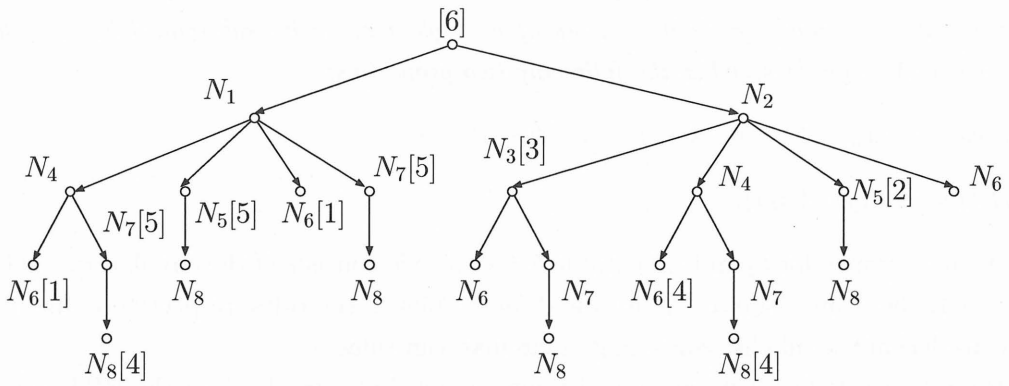

Figure 5.4: Decision tree constructed on RBTwP in Fig. 5.3.

Decision Tree constructed on RBT with pointers

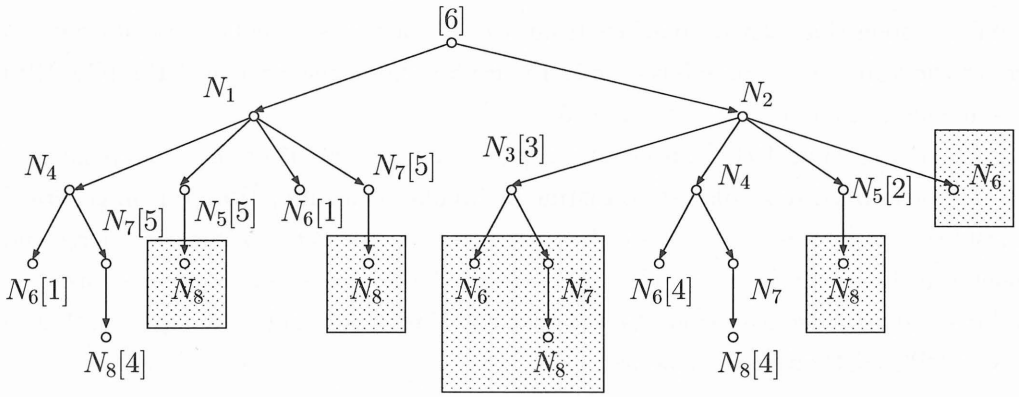

Figure 5.5: Redundant sub–tree in decision tree constructed on RBTwP

5.4 Cascaded Circular–RBT

The time complexity of an RBT search depends on n as $O(nw + w^2)$, because there are at most $n\lceil w/2 \rceil$ runs on the RBT, and a comparison occurs when there is a run on a node when traversing the RBT. If each rule in a rule list has just one run, we can classify a packet in constant time, i.e., independent of the number of rules, via a modified RBT. In this section, we first propose a data structure for the rule list consisting of circular–run rules. Secondly, we provide definitions for the consecutive ones property (C1P) [9] and the circular ones property (Circ1P) [17,81]. We then propose a packet classification algorithm called cascaded circular–RBT.

5.4.1 Circular–RBT

We define a circular–run as follows:

Definition 5.4.1. *(circular–run) Let $r_i \in \{0,1,*\}^w$ be a bitmask rule of length w. A substring of r is called circular–run if it consists of a single run or its substring $b_i b_{(i+1)} \cdots b_w b_1 \cdots b_j$ $(1 \leq i \leq w, 1 \leq j < i)$ satisfies the following two properties:*

i) $b_k = 0 \vee b_k = 1 \quad (i \leq k \leq w \vee 1 \leq k \leq j)$

ii) $b_l = * \quad (j < l < i)$.

As an example, for r_2 and r_3 in Table 5.2, each rule consists of the circular–run 111 and 001. These circular–runs begin at the fifth and fourth bits in the rules, respectively. In this section, we consider only a rule list consisting of circular–run rules.

Because a rule has just one run, the superscript j of a mark ρ_i^j on the RBT is redundant. Thus, we denote a run in a circular–run rule by ρ_i instead of ρ_i^j. When a node on the RBT has more than one run, because all except the run of the highest priority rule are redundant, those redundant runs are removed. Because every rule has just one run, if a packet matches run ρ_i, then the packet also matches rule r_i.

For an RBT constructed from a rule list consisting of circular–run rules, if node v has run ρ_i and no node that can be reached from a b–arc of v has a higher–priority run than ρ_i, we remove the b–arc of v, where b is 0 or 1. Figure 5.8 shows the circular–RBT (CircRBT) formed by removing redundant arcs from Fig. 5.7.

Although circular–RBT can classify a packet in constant time, i.e., independent of n, they can only be applied to a rule list consisting of circular–run rules. However, in general, a rule list consists of non–circular–run rules. Thus, we regard a rule list as a binary matrix and permute it such that the matrix has Circ1P [17,81]. In the following, we say that a rule list has Circ1P if the binary matrix reduced from the rule list has Circ1P. If a rule list has Circ1P, we can apply the CircRBT method to the rule list.

Table 5.2: Circular–run rules.

	Filter $\mathcal{R}$
r_1	0 0 1 * *
r_2	1 1 * * 1
r_3	1 * * 0 0
r_4	* * 1 1 0
r_5	0 1 * * *
r_6	* * * * *

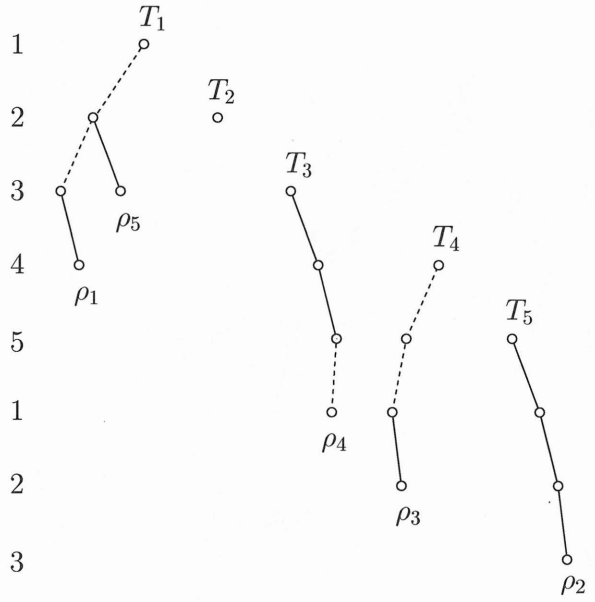

Figure 5.6: RBT for rule list in Table 5.2.

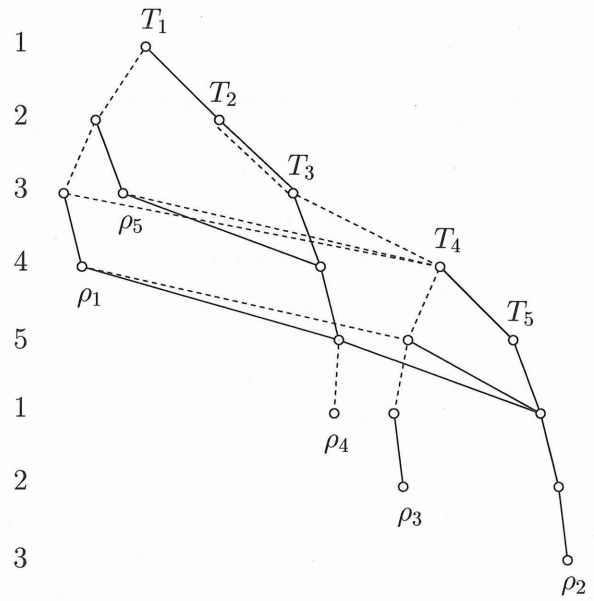

Figure 5.7: RBT added arcs and copied runs for rule list in Table 5.2.

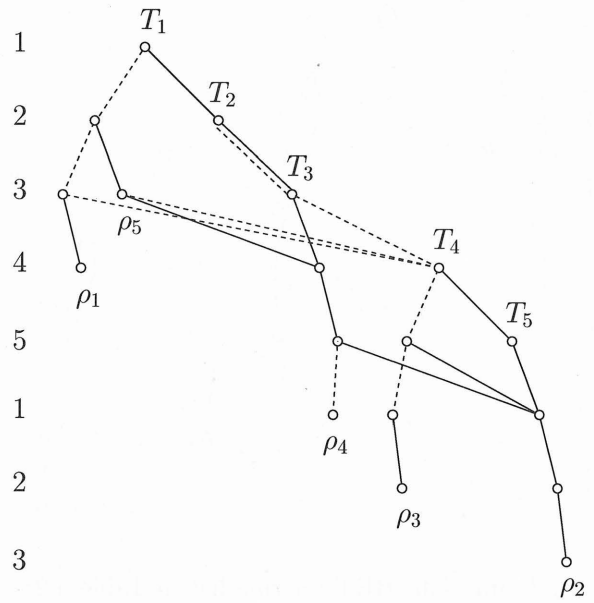

Figure 5.8: Circular–RBT for rule list in Table 5.2.

Table 5.3: Bitmask rules (rewritten).

Filter $\mathcal{R}$	
r_1	$0 * * 0\ 1$
r_2	$1 * 1\ 1 *$
r_3	$1\ 0\ 0 * *$
r_4	$* 1\ 0 * 1$
r_5	$0 * * 1 *$
r_6	$* * * * *$

Table 5.4: Replacing 0, 1 by 1 and $*$ by 0.

c_1	c_2	c_3	c_4	c_5
1	0	0	1	1
1	0	1	1	0
1	1	1	0	0
0	1	1	0	1
1	0	0	1	0
0	0	0	0	0

5.4.2 Consecutive Ones Property and Circular Ones Property

Replacing the 1s and 0s in a rule by 1 and the $*$ in the rule by 0, we reduce the problem of whether a rule list has Circ1P to whether a binary matrix has Circ1P. For example, whether the rule list in Table 5.3 has Circ1P can be determined by reducing it to the binary matrix in Table 5.4 and checking whether the matrix has C1P. To explain the Circ1P, we first define C1P.

Definition 5.4.2. *(Consecutive ones property (C1P)) A binary matrix has C1P if there is a permutation of its columns such that the* 1*s are consecutive in each row. A binary matrix that has C1P is called a C1P matrix and a binary matrix that does not have C1P is called a non–C1P matrix.*

For instance, the binary matrix in Table 5.5 has C1P, because there is a permutation $1 \rightarrow 1$, $2 \rightarrow 3$, $3 \rightarrow 2$ and $4 \rightarrow 4$ such that the 1s in each row in the permuted matrix are consecutive (see Table 5.6). In contrast to the binary matrix in Table 5.5, that in Table 5.7 is non–C1P.

Using partition refinement [27,90], the problem of whether an $n \times r$ binary matrix has C1P is solvable in $O(n + r + c)$ steps, where c is the number of 1s in M.

Definition 5.4.3. *(Circular ones property (Circ1P)) A binary matrix has Circ1P if there is a permutation of its columns such that the* 1*s or* 0*s are consecutive in each row. A binary matrix that has Circ1P is called a Circ1P matrix, and a binary matrix that does not have Circ1P is called a non–Circ1P matrix.*

Suppose a binary matrix is wrapped around a vertical cylinder. If the columns can be permuted such that the 1s in each row are consecutive on the cylinder, then the matrix has Circ1P. The binary matrix in Table 5.8 has Circ1P, because it can be permuted so that the 0s are consecutive for each row (see Table 5.9). In contrast, the matrix in Table 5.10 is non-Circ1P.

Whether a binary matrix has Circ1P can be determined by complementing those rows whose first column is 1 (i.e., row $M_1, M_2, \ldots, M_r$ is complemented if $M_{i1} =$ '1') and checking whether the matrix has C1P [81]. As we can determine whether a matrix has C1P in $O(n + r + c)$ steps, Circ1P can be checked in $O(nr)$ steps.

Table 5.5: C1P–matrix.

c_1	c_2	c_3	c_4
1	0	1	0
1	1	1	1
0	1	1	0
0	1	0	1
1	0	1	0
0	0	1	0

Table 5.6: Rearranging columns.

c_1	c_3	c_2	c_4
1	**1**	0	0
1	1	1	1
0	1	1	0
0	0	**1**	**1**
1	**1**	0	0
0	1	0	0

Table 5.7: Non–C1P matrix.

c_1	c_2	c_3	c_4
1	1	0	0
1	1	1	1
0	1	1	1
0	0	1	1
0	1	1	0
0	1	0	1

Table 5.8: Circ1P–matrix.

c_1	c_2	c_3	c_4	c_5
1	0	1	0	1
1	1	0	1	0
0	0	0	1	1
1	1	1	0	1
0	0	0	1	0
0	0	1	0	1

Table 5.9: Rearranging columns.

c_5	c_3	c_1	c_2	c_4
1	1	1	0	0
0	0	1	1	1
1	0	0	0	1
1	1	1	1	0
0	0	0	0	1
1	1	0	0	0

Table 5.10: Non–Circ1P matrix.

c_1	c_2	c_3	c_4	c_5
1	0	1	0	1
1	1	0	0	0
0	1	1	0	0
0	1	1	1	0
1	1	1	1	0
0	1	0	0	1

5.4.3 Cascaded Circular–RBT

Although the CircRBT method is efficient with respect to both time and space, it can only be applied to a rule list that has Circ1P. In general, because a rule list is non–Circ1P, CircRBT cannot be applied. In this section, we propose an algorithm that divides a rule list into multiple rule lists that have Circ1P, then constructs CircRBTs for these rule lists. Finally, a list of CircRBTs is constructed and used to classify packets.

We present a method of partitioning a rule list in Algorithm 14. First, this algorithm makes a rule list $\mathcal{R}_1$ consisting of r_1 and r_2. Then, it repeats the process of taking one of the remaining rules and inserting it into the existing rule list or creating new rule list in order from r_3 to r_n.

When rule list $\mathcal{R}$ has fewer than 3 rules, the algorithm returns the only $\mathcal{R}$, because the rule list must have Circ1P. If rule list $\mathcal{R}$ has more than 2 rules, rule list S (consisting of the head and second element of $\mathcal{R}$) is inserted into $L_{\mathcal{R}}$ on lines 4–7. Lines 8–19 search for a rule list $\mathcal{R}$ such that $\{r\} \cup \mathcal{R}_i$ has Circ1P, and insert r into $\mathcal{R}_i$ if there is such a rule list; otherwise, a new rule list S consisting of only r is created and inserted into $\mathcal{R}$. For instance, given the rule list in Table 5.11, Algorithm 14 divides it into the three rule lists in Tables 5.12, 5.13, and 5.14.

For a series of rule lists having Circ1P, $L_{\mathcal{R}} = [\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k]$, we construct a list of CircRBTs $L_{\mathcal{F}} = [\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k]$. We call this a cascaded CircRBT. A method of packet classification using cascaded CircRBT is shown in Algorithm 15. This algorithm applies a circular–run–

Algorithm 14: RuleListPartition

```
   input  : Rule list R
   output: List of rule list L_R = [R_1, R_2, ..., R_k]
 1 make an empty list of rule list L_R ;
   if |R| < 3 then
 2 |  add R to L_R ;
 3 |  return L_R ;
   end
 4 make a new rule list S ;
 5 add head(R) to S and remove head(R) ;
 6 add head(R) to S and remove head(R) ;
 7 add S to L_R ;
 8 while R ≠ ∅ do
 9 |  r ← head(R) ;
10 |  it ← head(L_R) ;
11 |  flag = false ;
12 |  while it ≠ L_R.end do
   |  |  if *it ∪ {r} has Circ1P then
13 |  |  |  add r to *it ;
14 |  |  |  flag = true ;
15 |  |  |  break ;
   |  |  end
16 |  |  it ← it.next ;
   |  end
   |  if flag ≠ true then
17 |  |  make a new rule list S ;
18 |  |  add r to S ;
19 |  |  add S to L_R ;
   |  end
20 |  remove r from R ;
   end
```

based search to a packet in the order $\mathcal{F}_1$ to $\mathcal{F}_k$, and returns the highest–priority rule number from those obtained from circular–run–based searches. On line 3, $\mathcal{F}_i(p)$ returns a rule number obtained by searching CircRBT $\mathcal{F}_i$ with p. The time and space complexity of this packet classification algorithm are $O(kw)$ and $O(nw)$, respectively, where k is the size of the list of CircRBT. If k is independent of n, we have a constant time packet classification that is independent of the number of rules.

Algorithm 15: ListOfCircRBTSearch

input : List of CircRBT $L_{\mathcal{F}}$ and packet p
output: Highest priority rule number for p

```
1 cand ← n + 1 // n is the number of rules ;
2 i ← 0 ;
  while i < |L_F| do
3 |   c ← F_i(p) ;
  |   if c < cand then
4 |   |   cand ← c ;
  |   end
5 |   i ← i + 1 ;
  end
6 return cand ;
```

Table 5.11: Example of non–Circ1P rule list.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}
r_1	1	*	0	*	1	1	0	*	*	*	0	1
r_2	1	0	*	0	*	1	0	0	0	*	1	*
r_3	0	1	0	1	0	1	*	*	0	0	*	*
r_4	*	1	0	*	1	0	*	1	0	1	*	1
r_5	*	0	1	1	*	*	*	*	*	1	*	*
r_6	1	0	0	0	*	*	*	*	1	*	*	0
r_7	*	1	*	*	1	0	0	*	*	*	0	0
r_8	0	*	1	0	0	0	*	0	*	0	*	*
r_9	*	*	*	*	*	*	*	*	*	*	*	*

5.4.4 Experiments

We demonstrate the efficiency of the proposed algorithm through a series of experiments and comparisons with a linear search, Grouper [50], and MDD [71]. An open–source implementation is available for Grouper, whereas the linear search, a classification algorithm using MDD, and the proposed algorithm were implemented in C. Experiments were conducted under the Cent

Table 5.12: Circ1P rule list 1.

	c_{12}	c_5	c_3	c_{10}	c_2	c_8	c_9	c_4	c_1	c_7	c_{11}	c_6
r_1	1	1	0	*	*	*	*	*	1	0	0	1
r_2	*	*	*	*	0	0	0	0	1	0	1	1
r_4	1	1	0	1	1	1	0	*	*	*	*	0
r_9	*	*	*	*	*	*	*	*	*	*	*	*

Table 5.13: Circ1P rule list 2.

	c_{12}	c_9	c_1	c_4	c_3	c_2	c_{10}	c_5	c_6	c_7	c_8	c_{11}
r_3	*	0	0	1	0	1	0	0	1	*	*	*
r_5	*	*	*	1	1	0	1	*	*	*	*	*
r_6	0	1	1	0	0	0	*	*	*	*	*	*

Table 5.14: Circ1P rule list 3.

	c_{10}	c_8	c_4	c_3	c_1	c_6	c_5	c_2	c_7	c_{11}	c_{12}	c_9
r_7	*	*	*	*	*	0	1	1	0	0	0	*
r_8	0	0	0	1	0	0	0	*	*	*	*	*

OS Release 6.8 (Final) on an Intel Core i5–3470 3.20 GHz CPU with 2 GB main memory. We generated the rules and the headers based on the standard benchmark for packet classification algorithms, ClassBench [80]. ClassBench provides acl, fw, and ipc seed files. To generate the rule for these experiments, the ipc seed file was used. The original rules generated by ClassBench were converted to arbitrary bitmask rules. The original rules consisted of five fields, namely the source/destination address, source/destination port number, and protocol number. Because these fields have lengths of 32, 32, 16, 16, and 8 bits, respectively, the length of the rule and header was 104 bits. The number of headers was about $1M$.

Using the generated rules and headers, we measured the time and memory requirements of constructing the data structures and the time required to determine the highest–priority rule for every algorithm. The construction and search processes were measured in seconds.

Table 5.15 presents the number of instances for which MDD and the proposed data structure could not be constructed within 4 h. As can be seen from Table 5.15 and Figs 5.9 and 5.11, MDD and the proposed algorithm are relatively slow and consume significant amounts of memory.

The means of 10 trials (except instances exceeding 4 h) are shown in Figs. 5.9, 5.10, and 5.11. Note that the time and memory are plotted on a logarithmic scale in Figs. 5.9 and 5.11. As MDD cannot be constructed within 4 h when there are 5000 rules, we do not show these results in Figs. 5.9, 5.10, and 5.11.

The proposed partitioning algorithm makes multiple calls to a routine that determines whether the rule list has Circ1P, and this routine requires the construction of a new graph for the rule list, wasting information that might have been collected from previous calls. Thus, as shown in Fig. 5.9, the proposed algorithm might take a significant amount of time to construct cascaded CircRBT.

In the experiments, Grouper required the least memory to make the lookup table. Thus, although the space complexity of the proposed algorithm is theoretically better than that of Grouper $2^w/t \cdot tn$ and $O(nw)$, the memory requirements of Grouper are lower than those of the proposed algorithm.

Figure 5.10 shows that the search times for the linear search and Grouper [50] are dependent on the number of rules. Although MDD can classify packets in constant time up to 3000 rules, it

Table 5.15: # of instances exceeding 4 h construction time in ipc.

	1000	2000	3000	4000	5000
Grouper	0	0	0	0	0
MDD	0	1	4	2	10
Proposed Algorithm	0	1	0	1	2

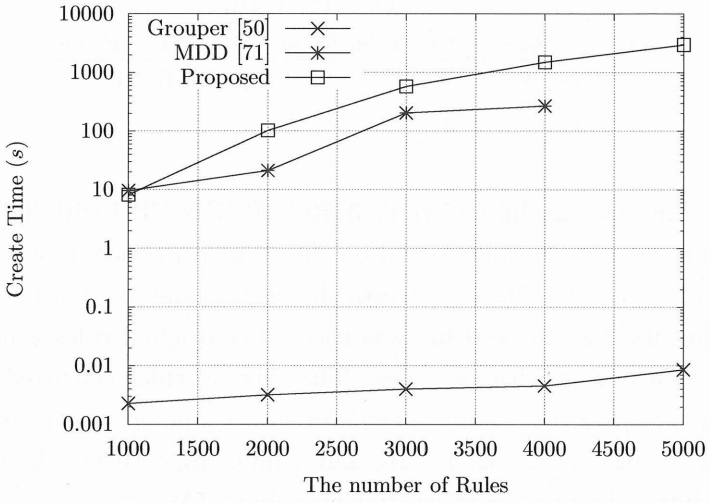

Figure 5.9: Construction times (s).

takes significantly more time for more than 4000 rules. As MDD exceeds the 2GB main memory with 4000 rules, it takes much more time to classify packets In contrast, the proposed algorithm classifies packets in constant time, independent of the number of rules. This agrees with the theoretical performance in Table 1.8. Because the rules generated by ClassBench are easy to partition, the size of the list of CircRBT list is small, allowing the proposed algorithm to classify packets in a constant time that is independent of the number of rules.

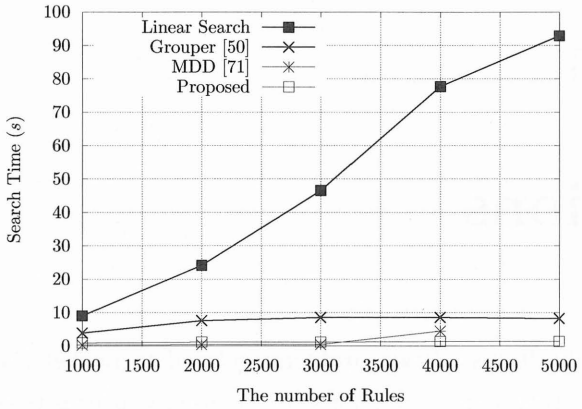

Figure 5.10: Classification times (s).

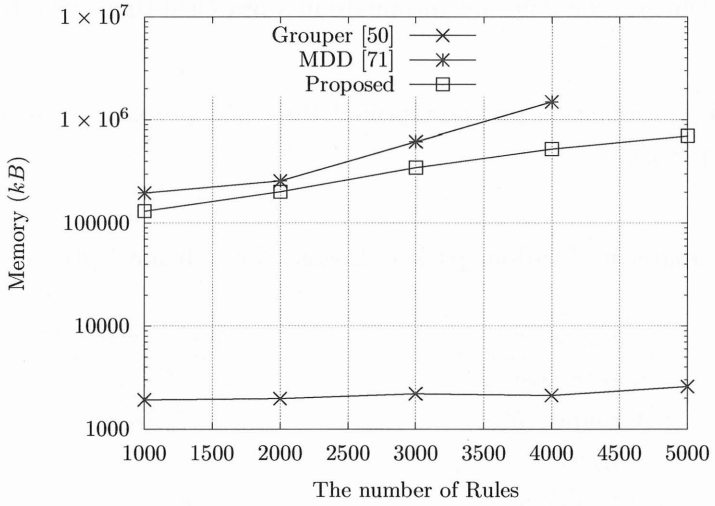

Figure 5.11: Required memory (kB).

Chapter 6

Conclusions

This thesis has mainly discussed two problems related to packet classification. One is an optimization problem for static early decisions and the other concerns packet classification algorithms based on software.

For the early–decision optimization problem, we identified the following issues:

1. It is necessary to develop a policy checker to ensure that the reconstructed rule list maintains the original policy.

2. ORO should take the packet arrival distribution as input and consider variations in the rule weights. This was formulated as RORO.

3. There is a problem with the pseudo–code for the state–of–the–art SGM [77] whereby its naive implementation does not terminate in a practical time when the precedence relation is dense.

4. There is no formulation for the Optimal Rule *List* problem instead of the Optimal Rule *Ordering* problem.

Table 6.1: Comparison of various packet classification schemes with arbitrary bitmask rules.

Algorithm	Worst–case Time	Worst–case Space
Linear Search	$O(nw)$	$O(nw)$
Grouper [50]	$O(tn/w)$	$O(2^{w/t} \cdot tn)$
MDD [71]	$O(w)$	$O(2^w)$
RBT Search [57]	$O(nw + w^2)$	$O(nw)$
RBT Decision Tree [57]	$O(w^2)$	$O(n^w)$
MOB [44]	$O(nw)$	$O(nw^2)$
Decision Tree constructed on RBT with pointers	$O(w)$	$O(2^w)$
Cascaded Circular–RBT	$O(kw)$	$O(nw)$

We obtain the following results for these problems:

1. We developed a policy checker using ZDDs and confirmed its effectiveness with 5000 rules.
2. - We showed that the problem of computing the weight for the default rule r_n with order σ under an uniform distribution U is #**P**–complete and developed an algorithm based on ZDDs for this problem.
   - We showed that RORO is **NP**–hard.
   - We proposed a simulated annealing algorithm for RORO and confirmed its effectiveness with 500 rules.
   - We developed a pairing algorithm that pairs rules causing policy violations until there are no such rules, allowing the rules to be simply sorted according to their weights. This algorithm decreased the classification latency and reordering time compared with SGM.
   - We fixed the bug in the pseudo–code of SGM, and accelerated this algorithm by by changing the representation of the precedence relation from an adjacent matrix to an adjacent list, and confirmed its effectiveness in experiments.
   - We formulated the ORL problem and developed a corresponding algorithm. This algorithm rewrites the rule list such that its graph of precedence relations becomes a forest of oriented trees, from which the order of rules can be optimized. The effectiveness of this algorithm was demonstrated through a series of experiments.

We now consider tasks for future work with regard to ORO. Although we showed that RORO is #**P**–complete, the computational complexity of ORL is still unclear. Because the effectiveness of the proposed algorithms was only demonstrated in experiments using the packet classification benchmark ClassBench [80], it is necessary to extend these results to the NFV environment (e.g., Open vSwitch [64]).

For the problem of packet classification based on software, we showed that it is necessary to develop a classification algorithm according to arbitrary bitmask rules, whereby the packet classification time is independent of the number of rules. In this study, we proposed an algorithm that first constructs a decision tree from RBT [57] with pointers, and then classifies packets via the decision tree. The time complexity of this algorithm is $O(w)$. We also developed a classification algorithm that divides a rule list into some rule lists $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k$ so that every rule in $\mathcal{R}_i$ has only circular–runs, constructs a list of CircRBTs $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}$, and classifies packets via this list. The time and space complexities of this algorithm are $O(kw)$ and $O(nw)$.

There are several tasks for future work with regard to packet classification based on software. To construct the decision tree based on RBTwP, pruning the decision tree by deleting all nodes whose children point to the same node and sharing all equivalent sub–trees would be a useful development. In terms of dividing a rule list into several Circ1P rule lists, there are three tasks. First, the time complexity of dividing a rule list is $O(n^2 r)$, which is somewhat long. Reducing this time is an important task. Second, because the effectiveness of the proposed algorithms has

only been demonstrated using ClassBench [80], it would be interesting to determine whether the number of rule lists k is independent of the number of rules n for any rule list. Finally, the effectiveness of our algorithms should be studied in a network virtualization environment, as should our algorithms for static early decisions.

Acknowledgment

I would like to express my sincere gratitude to my supervisor, Professor Ken TANAKA. I am grateful to Dr. Kenji Mikawa from Center for Academic Information Service, Niigata University for his assistance. My sincere thanks to Professors Leo NAGAMATSU and Kazuto MATSUO from Field of Information Sciences, Graduate School of Science, Kanagawa University for their helpful comments. Finally, I would like to thank the members of TANAKA laboratory for their support.

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[2] E. Al-Shaer, A. El-Atawy, and T. Samak. Automated pseudo-live testing of firewall configuration enforcement. *IEEE Journal on Selected Areas in Communications*, 27(3):302–314, April 2009.

[3] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi. Network configuration in a box: towards end-to-end verification of network reachability and security. In *2009 17th IEEE International Conference on Network Protocols*, pages 123–132, Oct 2009.

[4] E. S. Al-Shaer and H. H. Hamed. Discovery of policy anomalies in distributed firewalls. In *IEEE INFOCOM 2004*, volume 4, pages 2605–2616 vol.4, March 2004.

[5] F. Baboescu, , and G. Varghese. Packet classification for core routers: is there an alternative to cams? In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, volume 1, pages 53–63 vol.1, March 2003.

[6] Florin Baboescu and George Varghese. Scalable packet classification. *SIGCOMM Comput. Commun. Rev.*, 31(4):199–210, August 2001.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 155–168, New York, NY, USA, 2017. ACM.

[8] X. Bi, Y. Zhou, and J. Yu. Clustering boundary cutting for packet classification based on distribution density. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 661–666, Dec 2017.

[9] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335 – 379, 1976.

[10] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA, 1990. ACM.

[11] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

[12] Milind M Buddhikot, Subhash Suri, and Marcel Waldvogel. Space decomposition techniques for fast layer-4 switching. In *International Workshop on Protocols for High Speed Networks*, pages 25–41. Springer, 1999.

[13] Francis Chang, Kang Li, and Wu chang Feng. Approximate caches for packet classification. *IEEE INFOCOM 2004*, 4:2196–2207 vol.4, 2004.

[14] H. J. Chao. Next generation routers. *Proceedings of the IEEE*, 90(9):1518–1558, Sep. 2002.

[15] I. L. Chvets and M. H. MacGregor. Multi-zone caches for accelerating ip routing table lookups. In *Workshop on High Performance Switching and Routing, Merging Optical and IP Technologie*, pages 121–126, May 2002.

[16] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. Fast packet classification using bloom filters. In *2006 Symposium on Architecture For Networking And Communications Systems*, pages 61–70, Dec 2006.

[17] Michael Dom, Jiong Guo, and Rolf Niedermeier. Approximation and fixed-parameter algorithms for consecutive ones submatrix problems. *Journal of Computer and System Sciences*, 76(3):204 – 221, 2010.

[18] Q. Duan and E. Al-Shaer. Traffic-aware dynamic firewall policy management: techniques and applications. *IEEE Communications Magazine*, 51(7):73–79, July 2013.

[19] E. S. M. El-Alfy and S. Z. Selim. On optimal firewall rule ordering. In *2007 IEEE/ACS International Conference on Computer Systems and Applications*, pages 819–824, May 2007.

[20] A. Feldman and S. Muthukrishnan. Tradeoffs for packet classification. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 3, pages 1193–1202 vol.3, March 2000.

[21] T. Fuchino, T. Harada, K. Tanaka, and K. Mikawa. A reordering method via rules pairing based on average weights. *IEICE Technical Report, Circuits and Systems*, 118(295):31–36, nov 2018.

[22] Errin W. Fulp. Optimization of network firewall policies using directed acyclic graphs. In *In Proc. IEEE Internet Management Conf, extended abstract*, 2005.

[23] David Guijarro, Vctor Lavn, and Vijay Raghavan. Monotone term decision lists. *Theoretical Computer Science*, 259(1):549 – 575, 2001.

[24] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. *Micro, IEEE*, 20(1):34–41, Jan 2000.

[25] P. Gupta and N. McKeown. Algorithms for packet classification. *Netwrk. Mag. of Global Internetwkg.*, 15(2):24–32, March 2001.

[26] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. *SIGCOMM Comput. Commun. Rev.*, 29(4):147–160, August 1999.

[27] Michel Habib, Christophe Paul, and Laurent Viennot. Partition refinement techniques: An interesting algorithmic tool kit. *International Journal of Foundations of Computer Science*, 10(02):147–170, 1999.

[28] H. Hamed, E. Al-Shaer, and W. Marrero. Modeling and verification of ipsec and vpn security policies. In *13TH IEEE International Conference on Network Protocols (ICNP'05)*, pages 10 pp.–278, Nov 2005.

[29] Hazem Hamed and Ehab Al-Shaer. Dynamic rule-ordering optimization for high-speed firewall filtering. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 332–342, New York, NY, USA, 2006. ACM.

[30] Hazem Hamed, Adel El-Atawy, and Ehab Al-Shaer. Adaptive statistical optimization techniques for firewall packet filtering. In *INFOCOM*. IEEE, 2006.

[31] P. He, G. Xie, K. Salamatian, and L. Mathy. Meta-algorithms for software-based packet classification. In *2014 IEEE 22nd International Conference on Network Protocols*, pages 308–319, Oct 2014.

[32] K. Hikage and T. Yamada. Algorithm for minimizing overhead of firewall with maintenance of rule dependencies. *Proc. IEICE General Conference 2018*, 2016(1):6, mar 2016 (in Japanese).

[33] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, Boston, MA, 2017. USENIX Association.

[34] W. A. Horn. Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAM Journal on Applied Mathematics*, 23(2):189–202, 1972.

[35] Hongxin Hu, Gail-Joon Ahn, and Ketan Kulkarni. Fame: A firewall anomaly management environment. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 17–26, 2010.

[36] T. Ikemoto, K. Tanaka, and K. Mikawa. A dividing method of rule set for packet filtering optimization. *Proc. 13th Forum on Information Technology*, 2015:181–182, sep 2014 (in Japanese).

[37] T. Inoue, R. Chen, T. Mano, K. Mizutani, H. Nagata, and O. Akashi. An efficient framework for data-plane verification with geometric windowing queries. *IEEE Transactions on Network and Service Management*, 14(4):1113–1127, Dec 2017.

[38] T. Inoue, T. Mano, K. Mizutani, S. I. Minato, and O. Akashi. Rethinking packet classification for global network view of software-defined networking. In *2014 IEEE 22nd International Conference on Network Protocols*, pages 296–307, Oct 2014.

[39] W. Jiang. Scalable ternary content addressable memory implementation using fpgas. In *Architectures for Networking and Communications Systems*, pages 71–82, Oct 2013.

[40] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892, October 1989.

[41] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning. *Oper. Res.*, 39(3):378–406, May 1991.

[42] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, Lombard, IL, 2013. USENIX.

[43] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX.

[44] Y. Kobayashi, T. Takahashi, K. Mikawa, and K. Tanaka. A packet classification algorithm based on trie considering matching orders of bits. *IEICE Technical Report, Circuits and Systems*, 2015(315):65–70, nov 2015 (in Japanese).

[45] Y. Kobayashi, T. Takahashi, K. Mikawa, and K. Tanaka. Fast packet classification using trie. In *Proc. IEICE General Conference*, page 160, March 2016 (in Japanese).

[46] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.*, 28(4):203–214, October 1998.

[47] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary cams. *SIGCOMM Comput. Commun. Rev.*, 35(4):193–204, August 2005.

[48] E.L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*, 2:75 – 90, 1978. Algorithmic Aspects of Combinatorics.

[49] W. Li and X. Li. Hybridcuts: A scheme combining decomposition and cutting for packet classification. In *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pages 41–48, Aug 2013.

[50] Jay Ligatti, Josh Kuhn, and Chris Gage. A packet-classification algorithm for arbitrary bitmask rules, with automatic time-space tradeoffs. In *Proceedings of the International Conference on Computer Communication Networks (ICCCN)*, pages 145–150, August 2010.

[51] Hyesook Lim, Youngju Choe, Miran Shim, and Jungwon Lee. A quad-trie conditionally merged with a decision tree for packet classification. *Communications Letters, IEEE*, 18(4):676–679, April 2014.

[52] Hyesook Lim, Nara Lee, Geumdan Jin, Jungwon Lee, Youngju Choi, and Changhoon Yim. Boundary cutting for packet classification. *IEEE/ACM Trans. Netw.*, 22(2):443–456, April 2014.

[53] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *SIGCOMM Comput. Commun. Rev.*, 41(4):290–301, August 2011.

[54] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[55] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, Firstquarter 2016.

[56] K. Mikawa, K. Tanaka, and J. Koide. A solution for packet filter optimization problem using block segmentation. *IEICE Transactions on Communications*, J94-B(10):1408–1417, Oct 2011 (in Japanese).

[57] Kenji Mikawa and Ken Tanaka. Run-based trie involving the structure of arbitrary bitmask rules. *IEICE Transactions on Information and Systems*, E98.D(6):1206–1212, 2015.

[58] S. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Design Automation, 1993. 30th Conference on*, pages 272–277, June 1993.

[59] Shin-ichi Minato. Zero-suppressed bdds and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2):156–170, 2001.

[60] G. Misherghi, L. Yuan, Z. Su, C. N. Chuah, and H. Chen. A general framework for benchmarking firewall optimization techniques. *IEEE Transactions on Network and Service Management*, 5(4):227–238, December 2008.

[61] Ratish Mohan, Anis Yazidi, Boning Feng, and B. John Oommen. Dynamic ordering of firewall rules using a novel swapping window-based paradigm. In *Proceedings of the 6th International Conference on Communication and Network Security*, ICCNS '16, pages 11–20, New York, NY, USA, 2016. ACM.

[62] B. Nagpal, N. Singh, N. Chauhan, and R. Murari. A survey and taxonomy of various packet classification algorithms. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 8–13, March 2015.

[63] Christos M. Papadimitriou. *Computational complexity.* Addison-Wesley, Reading, Massachusetts, 1994.

[64] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association.

[65] X. Shao, K. Tanaka, and K. Mikawa. Rule list optimization method via dag. In *Proc. IEICE General Conference 2018*, page 334, March 2018 (in Japanese).

[66] R. Shima, K. Tanaka, and K. Mikawa. A solution for optimum filtering rules allocation. *Proc. 10th Forum on Information Technology*, 10(4):175–176, sep 2011 (in Japanese).

[67] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 213–224, New York, NY, USA, 2003. ACM.

[68] F. Somenzi. Cudd package. `http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf`.

[69] Haoyu Song and John W. Lockwood. Efficient packet classification for network intrusion detection using fpga. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 238–245, New York, NY, USA, 2005. ACM.

[70] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended tcams. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings.*, pages 120–131, Nov 2003.

[71] A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 92–95, Nov 1990.

[72] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. *SIGCOMM Comput. Commun. Rev.*, 29(4):135–146, August 1999.

[73] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191–202, October 1998.

[74] Thibaut Stimpfling, Normand Blanger, Omar Cherkaoui, Andr Bliveau, Ludovic Bliveau, and Yvon Savaria. Extensions to decision-tree based packet classification algorithms to address new classification paradigms. *Computer Networks*, 122:83 – 95, 2017.

[75] Ken Tanaka, Kenji Mikawa, and Manabu Hikin. A heuristic algorithm for reconstructing a packet filter with dependent rules. *IEICE Trans. Commun.*, 96(1):155–162, Jan 2013.

[76] Ken Tanaka, Kenji Mikawa, and Kouhei Takeyama. Optimization of packet filter with maintenance of rule dependencies. *IEICE Communications Express*, 2(2):80–85, Feb 2013.

[77] A. Tapdiya and E.W. Fulp. Towards optimal firewall rule ordering utilizing directed acyclical graphs. In *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, pages 1–6, Aug 2009.

[78] D. E. Taylor and J. S. Turner. Scalable packet classification using distributed crossproducing of field labels. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 1, pages 269–280 vol. 1, March 2005.

[79] David E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, September 2005.

[80] David E. Taylor and Jonathan S. Turner. Classbench: A packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, June 2007.

[81] Alan Tucker. Matrix characterizations of circular-arc graphs. *Pacific J. Math.*, 39(2):535–545, 1971.

[82] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. Efficuts: Optimizing packet classification for memory and throughput. *SIGCOMM Comput. Commun. Rev.*, 40(4):207–218, August 2010.

[83] J. van Lunteren and T. Engbersen. Fast and scalable packet classification. *IEEE Journal on Selected Areas in Communications*, 21(4):560–571, May 2003.

[84] P. Warkhede, S. Suri, and G. Varghese. Fast packet classification for two-dimensional conflict-free filters. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1434–1443 vol.3, April 2001.

[85] T. Y. C. Woo. A modular approach to packet classification: algorithms and results. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 3, pages 1213–1222 vol.3, March 2000.

[86] B. Yan, Y. Xu, and H. J. Chao. Adaptive wildcard rule cache management for software-defined networks. *IEEE/ACM Transactions on Networking*, 26(2):962–975, April 2018.

[87] Bo Yan, Yang Xu, Hongya Xing, Kang Xi, and H. Jonathan Chao. Cab: A reactive wildcard rule caching system for software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 163–168, New York, NY, USA, 2014. ACM.

[88] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, April 2016.

[89] S. Yingchareonthawornchai, J. Daly, A. X. Liu, and E. Torng. A sorted-partitioning approach to fast and scalable dynamic packet classification. *IEEE/ACM Transactions on Networking*, 26(4):1907–1920, Aug 2018.

[90] V.P. You. On matrices that do not have the consecutive ones property. Master's thesis, Department of Mathematics, Simon Fraser University, 2009.

[91] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and P. Mohapatra. Fireman: a toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 15 pp.–213, May 2006.

[92] P. Zhang, C. Zhang, and C. Hu. Fast testing network data plane with rulechecker. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, Oct 2017.

Appendix A

Simulated Annealing for RORO

In this appendix, we present a multi–start local optimization method (MSL) and a simulated annealing method for RORO [40,41].

A.1 Multi–Start Local Optimization (MSL)

To implement MSL, we first define a neighbor $N(\sigma)$ for solution σ. In the proposed method, Eq. (A.1) is used to determine a neighbor to solution σ.

$$N(\sigma) = \left\{ \sigma' \;\middle|\; \left| \{k \mid \sigma(k) \neq \sigma'(k)\} \right| = 2 \right\} \tag{A.1}$$

$N(\sigma)$ is obtained by choosing $i, j \in [n]$ $(i \neq j)$ and swapping $\sigma(i)$ and $\sigma(j)$. For example, $N(\sigma)$ for $\sigma = (2\ 3\ 1\ 4)$ with Equation (A.1) is

$$(3\ 2\ 1\ 4), (1\ 3\ 2\ 4), (4\ 3\ 1\ 2), (2\ 1\ 3\ 4), (2\ 4\ 1\ 3), (2\ 3\ 4\ 1).$$

The proposed method randomly generates an order $o : [|N(\sigma)|] \to [|N(\sigma)|]$ and searches for solutions in $N(\sigma)$ in order o.

To implement MSL, we define an evaluation function h for σ and σ' as well as a neighbor. Swapping the ith rule jth rules, the set of packets defined between the ith and jth rules may vary.

After interchanging the ith and jth rules, the sets of packets decided by the ith rule $r_{\sigma^{-1}(i)}$, jth rule $r_{\sigma^{-1}(j)}$, and kth rule $r_{\sigma^{-1}(k)}$ $(i+1 \leq k \leq j-1)$ are given by Eqs. (A.3), (A.4), and (A.5), respectively, where $\tau_{(i,j)}$ is the order that only interchanges the ith and jth elements. From the above, let the evaluation function h for solutions σ and σ' be

$$h(\sigma, \sigma') = \sum_{k=i}^{j} k \cdot \left(|E(\mathcal{R}_\sigma, k)|_F - |E(\mathcal{R}_{\sigma'}, k)|_F \right), \tag{A.2}$$

where $\sigma' = \tau_{(\sigma^{-1}(i), \sigma^{-1}(j))} \circ \sigma$.

MSL with the neighbor given by Eq. (A.1) and the evaluation function in (A.2) is described in Algorithm 16. This algorithm takes rule list $\mathcal{R}$ and a parameter *loop*, repeats *loop* to randomly generate feasible solution τ, and returns the optimal solution.

Algorithm 16: Multi-Start Local Optimization

input : Rule list $\mathcal{R}$, Loop parameter *loop*
output: Rule Order σ

```
σ ← initial random feasible order ;
o ← searching order for |N(σ)| ;
i ← 0 ;
while i < loop do
    τ ← random feasible order ;
    do local optimization for τ according to o ;
    if h(σ, τ) < 0 then σ ← τ;
    i ← i + 1 ;
end
```

A.2 Simulated Annealing

Simulated annealing takes a temperature and allows a solution to transit to a worse solution with probability p defined by the temperature and an evaluation function.

A simulated annealing algorithm for RORO is given in Algorithm 17. The algorithm utilizes neighbor $N'(\sigma)$ when temperature t is high compared with the initial temperature T as $T \geq \frac{100}{t}$. $N'(\sigma)$ is a set of solutions σ' whose weights do not vary in $N(\sigma)$. Because $E(\mathcal{R}, i)$ is computed at every evaluation of the solution, the algorithm uses $N'(\sigma)$ when t is high.

$$\begin{aligned} &E\big(\mathcal{R}_{\tau(\sigma^{-1}(i),\sigma^{-1}(j))\circ\sigma}, \sigma^{-1}(i)\big) \\ &= E\big(\mathcal{R}_\sigma, \sigma^{-1}(j)\big) \cup \big(E\big(\mathcal{R}_\sigma, \sigma^{-1}(i)\big) \cap M\big(r_{\sigma^{-1}(j)}\big)\big) \cup \cdots \cup \big(E\big(\mathcal{R}_\sigma, \sigma^{-1}(j-1)\big) \cap M\big(r_{\sigma^{-1}(j)}\big)\big) \end{aligned} \tag{A.3}$$

$$E\big(\mathcal{R}_{\tau(\sigma^{-1}(i),\sigma^{-1}(j))\circ\sigma}, \sigma^{-1}(j)\big) = E\big(\mathcal{R}_\sigma, \sigma^{-1}(i)\big) \setminus M\big(r_{\sigma^{-1}(i+1)}\big) \setminus \cdots \setminus M\big(r_{\sigma^{-1}(j)}\big) \tag{A.4}$$

$$\begin{aligned} &E\big(\mathcal{R}_{\tau(\sigma^{-1}(i),\sigma^{-1}(j))\circ\sigma}, \sigma^{-1}(k)\big) \\ &= E\big(\mathcal{R}_\sigma, \sigma^{-1}(k)\big) \setminus M\big(r_{\sigma^{-1}(j)}\big) \cup \big(E\big(\mathcal{R}_\sigma, \sigma^{-1}(i)\big) \setminus M\big(r_{\sigma^{-1}(i+1)}\big) \setminus \cdots \setminus M\big(r_{\sigma^{-1}(k-1)}\big)\big) \cap M\big(r_{\sigma^{-1}(k)}\big) \end{aligned} \tag{A.5}$$

A.3 Experiments

To confirm the efficiency of the proposed algorithms, we implemented them in C++ under Mac OSX 10.9.5 on an Intel Core i5 1.4GHz CPU with 4 GB main memory. We generated rule lists and header lists based on the standard benchmark for packet classification Class Bench [80], and used a ZDD library [68]. To evaluate th proposed algorithms MSL and SA algorithms, we implemented the state–of–the–art rule reordering algorithm SGM [77] in C++. The latency of a given rule list for MSL, SA, and SGM is shown in Fig. A.1. Although MSL decreases the

Algorithm 17: Simulated Annealing

input : Rule list $\mathcal{R}$,
Initial accepted probability *init*,
Inner Loop parameter *loop*,
Temperature factor *temp*,
Freezing Parameter $freeze$

output: Rule Order σ

$\sigma \leftarrow$ initial random feasible order ;
$o \leftarrow$ searching order for $|N(\sigma)|$;
choose initial temperature T so that $init = e^{-\Delta/t}$;
$t \leftarrow T$;
$count\_out \leftarrow 0$;
while $count\_out < freeze \cdot |N(\sigma)|$ **do**
  $count\_in \leftarrow 0$;
  while $count\_in < loop \cdot |N(\sigma)|$ **do**
    if $t < \frac{T}{100}$ **then** get $\sigma^{'} \in N^{'}(\sigma)$ according to o;
    else get $\sigma^{'} \in N(\sigma)$ according to o;
    $\Delta = h(\sigma, \sigma^{'})$;
    if $\Delta \leq 0$ **then** $\sigma \leftarrow \sigma^{'}$, $out\_count \leftarrow 0$;
    else
      choose a random value $q \in [0, 1]$;
      if $q < e^{-\Delta/t}$ **then** $\sigma \leftarrow \sigma^{'}$;
      $out\_count \leftarrow out\_count + 1$;
    end
    $in\_count \leftarrow in\_count + 1$;
  end
  $t \leftarrow t \cdot temp$
end

latency for the given rule list, its latency is greater than that of SA and SGM. However, in all cases, SA has the lowest latency Thus, SA is efficient for RORO.

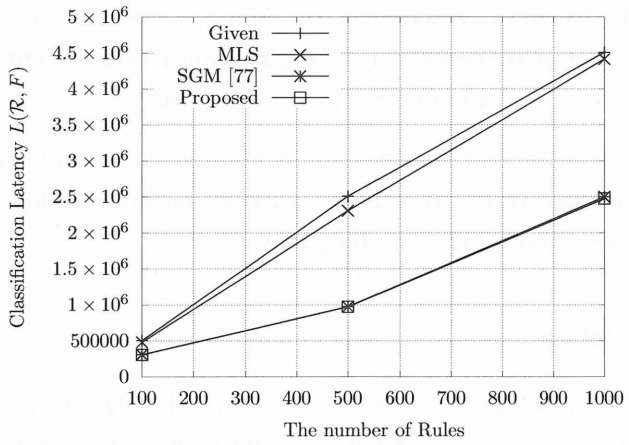

Figure A.1: Latency for rule lists with 100, 500, and 1000 rules.

Appendix B

Research Achievement

B.1 Journals

1. T. Harada, Y. Ishikawa, K. Tanaka, K. Mikawa, "A Packet Classification Method via Cascaded Circular–Run–Based Trie," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, (to appear)
2. T. Harada, K. Tanaka and K. Mikawa, "A Method for Computing the Number of Packets Matching a Filtering Rule," IEICE Transactions on Information & Systems, Vol. J101–D, No.3, pp.522-529, Mar., 2018, (in Japanese)

B.2 Conferences

1. T. Harada, K. Tanaka and K. Mikawa, "A Heuristic Algorithm for Relaxed Optimal Rule Ordering Problem," 2nd Cyber Security in Networking Conference (CSNet'18), Oct., 2018
2. T. Harada, K. Tanaka and K. Mikawa, "Acceleration of Packet Classification via Inclusive Rules," IEEE Conference on Communications and Network Security (CNS), pp.598–599, May, 2018

B.3 Technical Reports

1. T.Harada, K. Tanaka, K. Mikawa, "Deciding Equivalence of The Rule List Policies via ZDD," IPSJ SIG Technical Reports, Vol.2019-AL-171, No.8, pp.1–8, Jan., 2019 (in Japanese)
2. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "A Reordering Method via Rules Pairing based on Average Weights," IEICE Technical Report, Circuits and Systems, Vol.118, No.295, pp.31-36, Nov., 2018 (in Japanese)
3. T. Harada, K. Tanaka, K. Mikawa, "Packet Filter Reconstruction by Rules Encapsulation," IEICE Technical Report, Circuits and Systems, Vol. 118, No.82, pp.93-98, Jun., 2018, (in Japanese)

4. T. Harada, K. Tanaka, K. Mikawa, "A Packet Classification Method for a List of Run-Based Tries Consisting of a Single Run,", IPSJ SIG Technical Reports, Vol.2018-AL-167, No.3, pp.1-8, Mar. , 2018 (in Japanese)

5. T. Harada, K. Tanaka, K. Mikawa, "Heuristic Algorithms for Optimal Rule Ordering with Varying Rule Weights," IPSJ SIG Technical Reports, Vol.2018-AL-166, No.10, pp.1-8, Jan., 2018 (in Japanese)

6. T. Harada, K. Tanaka, K. Mikawa, "Computing the Number of Packets that match A Filtering Rule via MTZDDs," IEICE Technical Report, Circuits and Systems, Vol.117, No.96, pp.45-50, Jun., 2017 (in Japanese)

7. T. Harada, K. Tanaka, K. Mikawa, "An RBT Decision Tree Construction for Sparse Rules," IEICE Technical Report, Computation, Vol.117, No.28, pp.9-15, May, 2017 (in Japanese)

8. T. Harada, K. Tanaka, K. Mikawa, "A Fast Search Method for Run-Based Trie Consisting of A Single Run," IPSJ SIG Technical Reports, Vol.2017-AL-162, No.2, pp.1-7, Mar., 2017 (in Japanese)

9. T. Harada, K. Tanaka, K. Mikawa, "A Fast Search Method for Run-Based Tries via Pointers," IEICE Technical Report, Circuits and Systems, Vol. 116, No.315, pp.13-18, Nov., 2016 (in Japanese)

B.4 Programs

`https://github.com/tanakalab`