2024 Doctoral Thesis

Rule List Optimization Problem and its Solutions

Supervisor Prof. Ken TANAKA

Field of Information Sciences, Graduate School of Science, Kanagawa University

Student ID Number:202170195

Takashi FUCHINO

Contents

Chapter 1

Introduction

1.1 Research Background

Today, high-quality and complex services are being offered. In line with this, faster communication has become a key requirement for the functionality and quality of network services.

Packet classification is used to determine the behavior of incoming packets in network devices. Packet classification is a fundamental technology that enables packet communications in complex networks and prevents malicious communications. Since packet classification is performed by all devices connected to the network, faster packet classification enables faster network communication. The development of virtualization technologies such as NFV and SDN, and IoT technologies such as smart home and automatic vehicle operation, requires high-speed packet classification technology without using special hardware such as ternary content addressable memory (TCAM) and field programmable gate arrays (FPGA).

Virtualization technologies such as Network Function Virtualization (NFV) are one of the most important technologies in today's IT infrastructure. NFV is a software-enabled technology that enables a single computer to perform the functions of multiple network devices. By virtualizing computer appliances such as firewalls and Unified Threat Management, complex networks can be built, managed, and modified without the need for special hardware. However, high-speed software packet classification is essential for these functions to operate on general-purpose servers.

IoT technologies such as smart houses and autonomous driving are being developed, and a variety of products are connected to the Internet. These products are often difficult to implement security appliances due to space and cost constraints, so software based packet classification is required. For example, most of today's vehicles are controlled and managed by electronic devices such as ECUs. The Controller Area Network (CAN-bus) is known as a communication system for these devices. Today, CAN-bus can be connected to the Internet and other devices not only to manage the vehicle status, but also to provide various types of entertainment and to improve convenience. In addition, the development of vehicles is moving toward automation, which will increase communication with other vehicles and the grid. On the other hand, the physical limitations of vehicles, such as weight and installation space, make it difficult to implement security appliances, and software based packet classification is required.

In general, packet classification is done according to five fields: source, destination address, source and destination port, and protocol. These fields are expressed in prefix patterns such as 133.72.*.* etc., or in a range such as 0-65535. In addition to these representations, some studies have used arbitrary bitmask representations such as *.72.*.141 to represent more complex fields [1]. In virtualized environments such as NFV, more fields are used to represent arbitrary bitmask representation is needed to speed up packet classification.

1.2 Research Classification

Today, various packet classification methods have been developed to maintain network communication quality. Hardware-based packet classification uses Application Specific Integrated Circuit (ASIC), Field Programmable Gate Array (FPGA), Ternary Content Addressable Memory (TCAM), and GPUs for high-speed packet classification. However, due to cost reductions and the use of virtualization technology, there is an increasing demand for software based high-speed packet classification without using these security appliances.

There are several types of software based packet classification methods. We categorized the packet classification studies as shown in Table 1.1 and Table 1.2. They classify each data structure used for packet classification.

Packet classification using decision trees is achieved by searching for a decision tree corresponding to a policy. As shown in Figure 1.1, a decision tree consists of two types of nodes: leaf nodes and internal nodes. A leaf node contains a rule list with one or more rules. When a packet arrives at a leaf node, the rules are matched one by one, starting with the first rule, and the action of the first matched rule is applied. Internal nodes branch for each bit value of interest and direct the packet to the leaf node corresponding to the characteristics of the packet.

The size of the tree and the number of rules contained in the leaf nodes vary depending on which bit or bits are considered for each internal node. Increasing the tree size decreases the number of rules stored in the leaf nodes, as shown on the left side of Figure 1.1. Following a single node halves the search area for matching rules, generally reducing search time but increasing the number of nodes. As a result, the amount of memory required. On the other hand, reducing the tree size reduces the number of nodes and so the amount of memory required, but as shown on the right side of Figure 1.1, the number of rules in a leaf node increase. As a result, the classification time becomes dependent on the number of rules. Therefore, it is necessary to find the optimal rule assignment between the height of the tree and the leaves. In addition, what bits are used for branching will affect the replication of rules. As shown on the left side of Figure 1.2, a leaf node must contain all the rules that satisfy the conditions of the path from the root, so the same rule may be replicated in multiple leaves. Rule replication affects the amount of memory required and should be avoided if at all possible.

Packet classification using decision trees is generally faster because the number of compar-

Table 1.1: Taxonomy of software based packet classification 1.

Rule List	Lucent Bit Vector [2]
	Aggregated Bit–Vector (ABV) [3]
	Cross–Producting [4]
	Recursive Flow Classification [5, 6]
	HybridRFC [7]
	the Rule order Optimizer based on Simulated Annealing [8]
	Takeyama [9]
	Simple Rule Sorting [10]
	Sub-Graph Merging [11]
	Hikin's method [12]
	Hikage's method [13]
	Mohan's method [14]
	Shao's method [15]
	Misherghi's method [16]
	Fumiiwa's method [17]
	Approximate Packet Classifiers [18]
Decision Tree	Hierarchical Intelligent Cuttings (HiCuts) [19]
	EffiCuts [20]
	Modular Packet Classification [21]
	Hypercuts [22]
	Multidimensional Interval Tree (MITree) [23]
	NeuroCuts [24]
	hierarchical hash tree (H-HashTree) [25]
	FROD [26]
	CutSplit [27]
	ByteCuts [28]
	NuevoMatch [29]
	TabTree [30]
	KickTree [31]
	Run-Based Tries Based [32]

isons between rules and packets is independent of the number of rules. However, most decision tree construction methods search for rules starting from the first bit of the address, which increases the size of the decision tree for arbitrary bitmask representations. In addition, early works such as HiCuts [19] and Hypercuts [22] have the problem that the same rule is replicated in several leaves, which increases the memory requirement. EffiCuts [20], CutSplit [27], and others reduce the number of rule replications by constructing smaller trees with more closely related rules.

Table 1.2: Taxonomy of software based packet classification 2.

Tri	Extended Grid–of–Tries (EGT) [33]
	Hierarchical Trie [4]
	Set–Pruning Trie
	Grid of Tries [4]
	Multibit-tries packet classification engine [34]
Geometrical	Area–Based Quadtree [35]
	Fat Inverted Segment Tree [36]
	Grid of Tries [4]
Tuple Space	Tuple Space Search & Tuple Pruning [37]
	CutTSS [38]
	TupleMerge [39]
	Learned Bloom Filter [40]

In addition, classification time may increase if the constructed tree is unbalanced, as shown on the right side of Figure 1.2. Therefore, a method has been proposed that achieves faster packet classification by constructing a decision tree that is similar to a balanced tree while addressing the above problems. Modular Packet Classification [21] combines the structures of an index jump table, decision tree, and rule list to perform classification. It constructs several decision trees with small rule lists as leaf nodes and an index jump table that contains pointers to their root nodes.

TabTree [30] is a method for constructing a more balanced decision tree by branching in bits such that the rules are evenly assigned to the child nodes created. In addition, multiple decision trees are constructed using the TSS method to prevent rule duplication.

KickTree [31] constructs a decision tree by focusing on bits such that at least one rule has $*$ when constructing child nodes, and then constructs a branch for that bit. At this time, the rule whose bits have $*$ is excluded from the search of the decision tree being constructed, and the next sub-tree that is created is used to determine whether or not the rule matches the rule. This process is repeated until all rules are included in one of the decision trees.

In these multiple decision tree construction methods, there is a trade-off between the number of sub-trees and the depth and balance of each tree.

NuevoMatch [29] and NeuroCuts [24] use reinforcement learning to construct decision trees, but differ in how they introduce machine learning.

NeuroCuts uses reinforcement learning to construct a decision tree using heuristic decision tree construction such as HiCuts and EffiCuts as the supervised data.

NuevoMatch is a method for rapidly narrowing the search space by mapping special RMI models to independent rules. Machine learning is used to construct the RMI model to determine which rules to use as keys.

Tri-based packet classification uses a special decision tree to classify packets. Decision trees

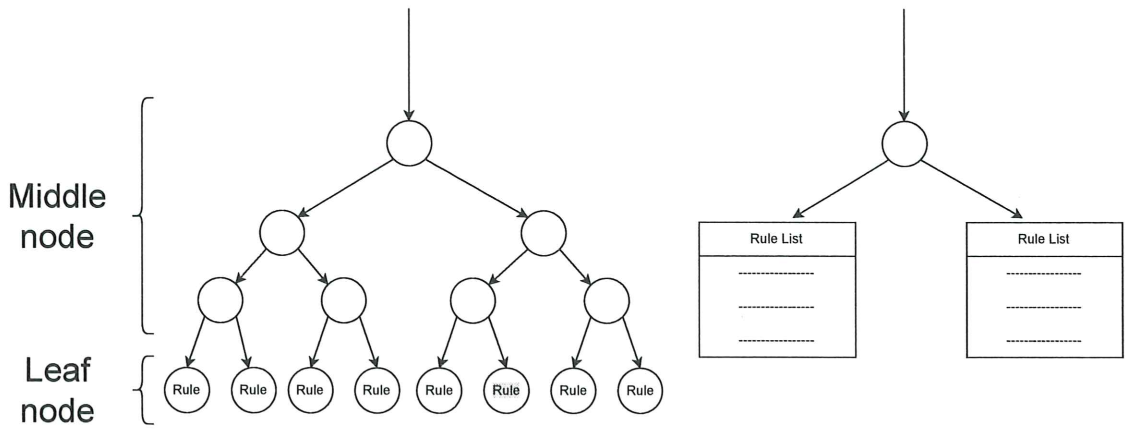


Figure 1.1: Decision Tree.

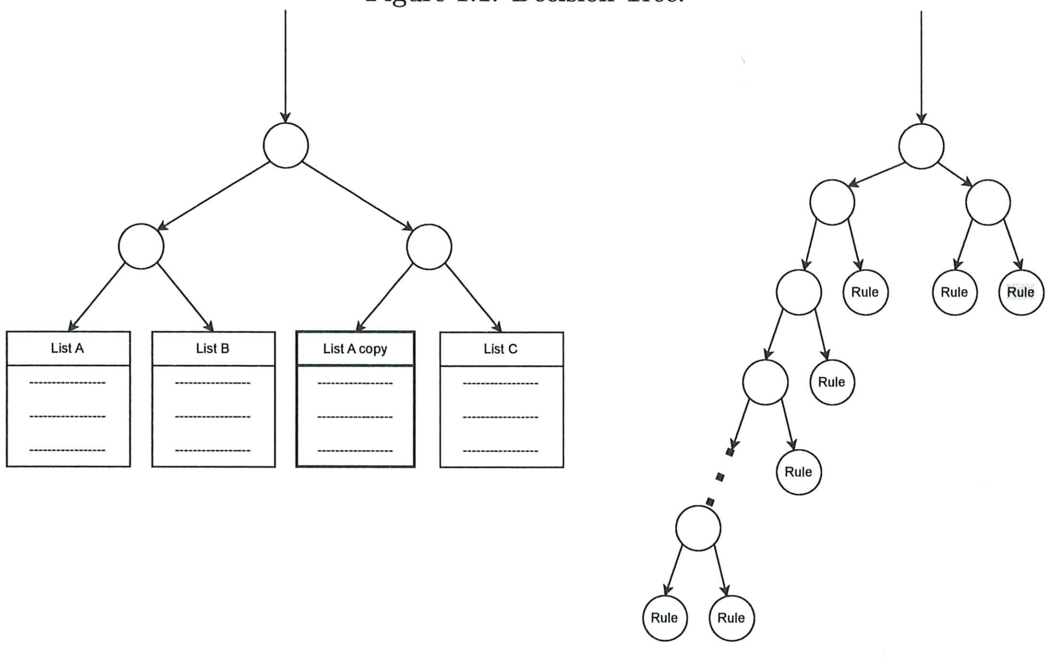


Figure 1.2: Copied rules and Unbalanced tree.

are constructed for each field, and pointers are pointed from the leaf nodes of these decision trees to the roots of the corresponding decision trees for other fields, allowing multiple fields to be searched in a single search. In general, one rule is placed in one leaf node, which tends to increase the size of the tree, so more bits of branching are required in one internal node [4].

The multibit-tries packet classification engine [34] uses reinforcement learning to determine which bits to focus on at the internal nodes to construct smaller decision trees.

Mikawa et al. proposed a method for constructing a decision tree using Run-based [41]. The packet classification is achieved by constructing a tri-tree consisting of bit-lengths and applying the action of the rule with the highest priority among the rules that satisfy the condition. Furthermore, the size of the tree is reduced by pointing each leaf node with a pointer to the corresponding tri-tree by Ishikawa et al. [32].

Geometrical packet classification considers the conditional expressions of rules as a $d-$dimensional geometric space, and places rules as hyperrectangles in that dimension. This is achieved by taking the header information of an arriving packet as the coordinates of each dimension and applying the action of the rule with the highest priority placed at the coordinates. Since the conditional expressions for source and destination addresses described in the rules are prefix expressions, this method performs fast packet classification by constructing a hash map using them as keys and a decision tree that searches from the first bit to the last. However, many of these classification methods suffer from the problem that arbitrary bitmask representations make space partitioning difficult. In addition, as the number of rules placed increases, the size of the classifier constructed by the rules increases, making it impossible to classify correctly. Therefore, when constructing a classifier with a larger number of rules, we sometimes allow a rectangle to contain multiple rules and switch to classification using rule lists or decision trees for detailed classification.

Today, the packet classification problem using decision trees requires improvement of three issues. The first is to address the trade-off between memory and classification speed due to the height of the tree, the number of trees, and the number of rules in the leaf nodes. If the height of the constructed decision tree is tall, the number of rules in the leaf nodes is expected to decrease, but the number of nodes increases in parallel with 2^h, where h is the height of the decision tree, and thus the memory requirement increases. To address this problem, a method to reduce the height of each tree by constructing a decision tree for each similar rule has been considered. However, this requires several trees to be traversed when classifying packets, which increases the classification time. The second is to reduce the number of replicated rules and balance the decision tree. If several leaves require the same rules, the number of rules in the decision tree will increase, which will increase the memory requirement. In addition, the increase in the number of rules in a leaf node leads to an increase in classification time. Therefore, a method is needed that minimizes the number of rules that satisfy both conditions of branching at the internal nodes when constructing the decision tree. On the other hand, extreme branching can lead to unbalanced decision trees and increase the classification time. Therefore, a method to construct a decision tree that is closer to a balanced tree is required. The third is to develop a decision tree construction method for arbitrary bitmask representations. Most decision tree construction methods assume that the input rule list is in prefix notation. In the case of arbitrary bitmask representations, constructing a decision tree by starting from the first bit often results in the duplication of rules. Therefore, there is a need to develop a decision tree construction method for arbitrary bitmask representations.

Packet classification using rule lists is achieved by comparing arriving packets in order from the first rule in the rule list and applying the action of the first matching rule. Packet classification using rule lists requires few hardware resources and is easy to implement. Also, by placing the necessary rules at the top of the list, partial policy changes can be easily made. Rule list is the most basic structure for packet classification, and research has been underway to accelerate packet classification using rule lists. In packet classification using a rule list, as the number of rules increases, the search time increases, and in general, communication latency increases. To minimize latency, the problem of finding an efficient rule list has been studied [2–9]. This problem can be classified as either static or dynamic. The static problem is given a set of packets arriving at a network device and a rule list and requires a rule list that classifies the given set of packets faster. Since the packets arriving at the network device are fixed, the number of packets that match each rule in the rule list can be estimated to some extent. The dynamic problem is to extract the packets arriving at a network device and find a sequence or list of rules with lower latency for that distribution.

Recursive Flow Classification [5, 6], such as HybridRFC [7], divides a list of rules in prefix patterns into fields and constructs a chunk in each field. A chunk is a range of bit strings that can match the same rule. The partitioned chunks are then aggregated and the chunks are repartitioned. By performing this operation on all the segmented fields, the rules that must be matched in each chunk are obtained. This compresses the rule list description, thereby reducing the rule list size and search time. However, as the policy becomes more complex due to arbitrary bitmask notation or an increase in the number of rule lists, the memory requirement increases and the system does not operate properly.

In Approximate Packet Classifiers and other methods, when several network devices perform packet classification, the abstraction level of the conditional expressions of the rules is adjusted according to the location of the network devices to reduce the number of matches across the network. Although this method can speed up the processing of priority packets and distribute the load on the network, when the number of devices performing packet classification decreases, strict packet classification is required, and fast packet classification is no longer possible.

In general, packet classification using a rule list is faster when the number of rules in the rule list is small and the rules with high matching frequency are placed at the top. Therefore, research is being conducted to find a rule list that can perform packet classification faster.

In Tanaka et al.'s method [42], when a rule list is in bitmask notation, they considered them as logical expressions and proposed a method to obtain a rule list with a small number of rules by using the Kwein-McCluskey method.

In the methods of E.W. Fulp [10] and Mohan et al. [14], when there is no precedence relationship between two rules and the frequency of matching is high for a rule placed lower, the rule with the higher matching frequency is placed higher by swapping these rules. However, these methods cannot reduce the latency sufficiently, because the rule with the higher matching frequency cannot be moved any higher if it is preceded by another rule.

Takeyama et al.'s method [9] and Sub-Graph Merging [11] focus on the rules that precede the rule with the highest matching frequency and place them at the top of the list preferentially. This allows the rule with the highest matching frequency to be placed higher in the list. However, the latency may not be sufficiently small because it is not always necessary to give priority to the rule with the highest matching frequency.

Misherghi et al. formulated the rule order optimization problem as an integer programming problem and proposed a method to find a sequence of rules with the minimum latency using a solver [16]. However, the computational complexity of the integer programming solver and the algorithm for formulating the rule order optimization problem into an integer programming problem is of exponential order, and the operation does not terminate when the number of rules increases.

Fumiiwa et al. proposed an optimal solution method for the optimal rule-ordering problem based on the branch-and-bound method. This method finds the rule ordering with the minimum latency from the rule ordering that preserves precedence constraints due to overlap relations. However, there exists rule ordering that holds policy even if it does not hold precedence constraints due to overlap relations, and there are cases in which there is a rule ordering with smaller latency.

Figure 1.3 shows an overview of the static rule list optimization problem. The problem we address in this study is highlighted in gray. We divided the packet classification problem into software based and hardware-based packet classification, and further divided the software based packet classification into different approaches. For rule list-based packet classification, we divided the problem into two parts: an optimization problem for a single rule list, and an optimization problem in which the entire set of rule lists applied in the network is considered as a single classifier for packet classification. The optimization problem for a single rule list can be divided into a dynamic problem and a static problem, and can be further divided into whether the rules are written in prefix pattern or arbitrary bitmask pattern. We then divide the optimization problem into two categories: optimization problems limited to the order of the rules, and the problems involving restructuring. In this paper, we address restructuring in rule list optimization problems and **RORO** in rule reordering. We also address the rule list equivalence decision problem, which is important for rule list optimization. The details of each problem are described below.

1.3 Relaxed Optimal Rule Ordering

Optimal Rule Ordering (**ORO**) takes a rule list and a set of packets as input and finds an ordering of rules that minimizes latency while holding precedence constraints based on overlap relations in the rule list. The overlap relation refers to the relationship between rules that can match the same packet. If the order of the overlapping rules is changed, it may result in an ordering that does not satisfy the policy. Since maintaining the order of overlapping rules preserves the policy, the problem of finding a sequence of rules that minimizes latency while holding this precedence constraint is called **ORO**. Since this problem is known to be **NP**-hard, various heuristics have been proposed to solve it [9–12, 14]. However, even if the precedence constraint based on the overlap relation is not satisfied, there may exist a sequence of rules that holds the policy, and among them, there may be a sequence of rules with smaller latency. Therefore, the problem of finding the ordering with the smallest latency and hold policy without prior constraints based on overlap relations has been studied. Relaxed Optimal Rule Ordering (**RORO**) takes a rule list and a set of packets as input and finds an ordering of rules that minimizes latency while holding policy. It is known that many of the algorithms proposed for solving **ORO** are also executable in **RORO** because **ORO** imposes stronger constraints on **RORO**. In this paper, we show the computational complexity of **RORO** and propose a heuristic solution method for this problem. Computer experiments are conducted to verify the effectiveness of the proposed method and to compare it with previous **ORO** and heuristic solution methods for **RORO**.

1.3.1 Computational Complexity of RORO

ORO has been shown to be **NP**-complete by reducing from job scheduling problem, but **RORO** cannot reduce the job scheduling problem because the number of packets matching the rule may be changed when reordering. In this paper, we show the computational complexity of this problem by reducing from **EXACT COVER BY 3-SETS** and present a heuristic solution for this problem.

1.3.2 Heuristic Algorithms for RORO

Since **ORO** is known to be **NP**-hard, heuristic solutions to this problem have been proposed. These heuristics can also be used for **RORO** to find an ordering of rules with less latency than reordering rules as **ORO**, where two overlapping rules can be reordered if they have the same actions. Swapping overlapping rules may result in a change in the number of packets matching the rules. This causes the number of packets that match the rules to be changed. This phenomenon is called weight fluctuation, and heuristics for **ORO** cannot take into account this weight fluctuation, so there is a problem that rules that can match many packets in the lower levels cannot be placed in the upper levels.

Sub-graph Merging (SGM) [11] traces precedence constraints based on overlap relations and focuses on the rule with the highest average weight of the set of reachable rules. If the focused rule overlaps with other rules, the rule with the highest average weight of the set of reachable rules among the overlapping rules is focused on. This process is repeated until the rule does not overlap with any other rule, and the rule is added to the aligned list and removed from the rule list. This process is repeated until the rule list is empty. This method can find an ordering with less latency because the weights of the rules and the rules required to place the rules in the sorted list are taken into account in sorting the rules. However, the time computation becomes $\mathcal{O}(n^3)$ when the number of rules is n because the precedence constraints are traversed many times. In addition, when calculating the evaluation value of each rule, SGM only considered the rules necessary to place the rule under focus, and thus cannot sufficiently reduce the latency.

The method of Hikin et al. [12] swaps two adjacent rules if they have no overlap and the rule with the higher weight is placed lower. This process is repeated until all rule pairs cannot be swapped. This method is fast because the time complexity is $\mathcal{O}(n^2)$, but the latency is not sufficiently small because it cannot take into account the ordering in which a rule with a large weight is placed higher than the rule with which it has an overlap.

Takeyama et al.'s method [9] focuses on rules with high weights and prioritizes the rules necessary to place them in the sorted list. This method is fast, with a time-computation cost of $\mathcal{O}(n^2)$. However, since it does not consider how many rules are needed to place the rule in focus in the list, it is unable to place the rule with less weight but with fewer dependent rules at the top of the list.

The method of Hikage et al. is based on the divide-and-conquer method [13]. The rules are divided into connected components of prior constraints, and a list is constructed for each connected component from the bottom of the list. Then, for each rule, the average of the rule weights from the end of the list to the rule is used as the evaluation value, and the rule with the lowest evaluation value is added to the top of the sorted list up to the end of the list, and then removed from the original list. This process is repeated until all the lists are empty. This allows the lower-priority rules to be placed lower in the aligned list, thus reducing latency. There are two versions of Hikage et al.'s method that differ in the way they choose which rules to place at the top of the list when building each list from the connected components. The first method uses the average weight of the reachable rules, and the rule with the lowest weight average that is not dependent on any other rule is added to the top of the list. This method can determine the order of the rules in the connected component more precisely and thus can obtain an ordering of rules with less latency, but it requires $\mathcal{O}(n^{2.3728})$ computation because it includes an operation to find the set of reachable rules. The other is a method using single weights, which constructs the list using the weights of the rules themselves instead of weight averaging. This version has a computational complexity of $\mathcal{O}(n^2)$ and can sort rules quickly, but may not reduce latency sufficiently.

Shao et al.'s method [15], is an improvement on SGM that aims to find a sequence of rules with less latency. Comparison of weighted means that include the same rules may fail to properly select the rule that should be placed at the top. Therefore, this method addresses this problem by using an average value that excludes rules included in both sets when comparing weight averages. Also, by removing the transition edge in the prior constraints, redundant search is eliminated. However, in some cases, the rules included in both sets are unnecessary in the comparison of weighted means, while in other cases, they are not, and the latency may not be sufficiently small.

In this study, we describe a method that speeds up SGM and improves on the methods of SGM and Hikage et al. into a method with reduced latency. We also propose a method that improves on the problems of SGM and Shao et al.'s method to obtain a sequence of rules with lower latency.

In SGM, precedence constraints are managed using a two-dimensional array, but using an adjacency list allows faster management of precedence constraints. When tracing the precedence constraints, the next rule to focus on is selected from the rules that precede directly the current rule, but this method may not be able to find an order of rules with sufficiently small latency. In this paper, we propose a method to find a sequence of rules with smaller latency by using an adjacency list to accelerate the SGM and then searching for the next rule to focus on among the rules that are reachable by the rule.

In addition, SGM uses the average weight of the rules reachable from the rule when selecting the rule to place at the top. This method can take into account the rules necessary to place the rule, but it cannot take into account that several heavy rules can be placed at a higher level by placing the precedence rule at a higher level. By placing a rule that has a small weight but precedes several rules with larger weights, the rule can be placed higher, allowing the several rules that preceded it to be placed higher, resulting in an order of rules with smaller latency. Therefore, we propose a rule reordering method that takes into account not only the set of rules reachable from the rule but also the rules that can be placed by placing the rule.

As described above, reordering rules using the average weight of the rule set may not be able to sufficiently reduce latency. Therefore, we propose a method to find an ordering of rules that is expected to reduce the latency, and then compare the difference between the latency in that ordering and the original ordering to find an ordering of rules with smaller latency.

The method of Hikage et al. with $\mathcal{O}(n^2)$ of computational complexity can reorder the rules faster, but it only considers single weights when determining the order of each connected component. Therefore, it cannot take into account the case where a single rule has a large weight but is dependent on several rules with smaller weights, and thus the latency would be smaller if the rules were placed lower. Therefore, we propose a method to find an order of rules with smaller latency while keeping the computational complexity $\mathcal{O}(n^2)$ by creating a list using the average weights of the rule itself and the rules to which it is directly subordinated.

In packet classification using rule lists, it is easy to change a part of the policy by adding a rule at the top. However, such a change may result in a rule with no matching packets. Since such rules are still compared with packets, the number of comparing for the packets matching the rules placed lower in the list increases. Therefore, we propose a method to search for such rules and place them lower than the default rules to obtain an order of rules with lower latency.

Many heuristics for this problem reorder based on precedence constraints based on overlap or dependency relations. However, there exists an ordering that holds policy even if those precedence constraints do not hold, and among them, there may be an ordering of rules with smaller latency. Therefore, we propose a method to find an order of rules with smaller latency by searching for and eliminating precedence constraints that preserve policy even if they do not hold.

An Allow list is a rule list in which all actions of rules other than the default rule are "Allow". The administrator only needs to specify packets that are allowed to communicate and other packets are rejected by the default rule, making it highly resistant to unknown attacks. For this reason, among packet classification using rule lists, the Allow list tends to be adopted in the field of network security such as Firewalls. On the other hand, since unauthorized packets always match the default rule, an increase in the number of denied packets causes a significant increase in latency. Therefore, there is a need for an Allow list with lower latency. Although Allow lists have no precedence constraints with each other except for the default rule, there is an overlapping relationship, which causes weight fluctuation. Previous rule reordering methods cannot sufficiently reduce latency because they cannot account for weight fluctuation. Therefore, we propose a rule reordering method that takes into account the variable number of matching packets by measuring and reordering the number of matching packets regardless of the order of the packets. Computer experiments are conducted to verify the effectiveness of the proposed methods.

1.4 Rule List Reconstruction

Rule list optimization is the problem of minimizing the latency of the rule list while holding policy, given a rule list and a set of packets as input. Since the latency of a rule list with a small number of rules is generally small, most heuristics for this problem reduce the latency by merging rules that differ by only one bit in the input rule list or by reducing the number of rules using the Quine-McCluskey method [18,42]. However, the rule list with the smallest number of rules is not necessarily the rule list with the smallest latency. Therefore, it is necessary not only to merge the rules in the input rule list but also to reconstruct the rule list by generating rules to be placed based on the policy expressed in the rule list and the packet frequency distribution by the input. In this paper, we show the computational complexity of the optimal rule list problem and propose a heuristic solution for this problem. The proposed method constructs a packet space, which is a set of actions and arrival frequencies that should be applied to each packet, and constructs a rule list with lower latency. Since the exact construction of the packet space is computationally exponential in terms of the number of bits, the proposed method constructs an independent rule list from the input rule list and constructs a simplified packet space quickly.

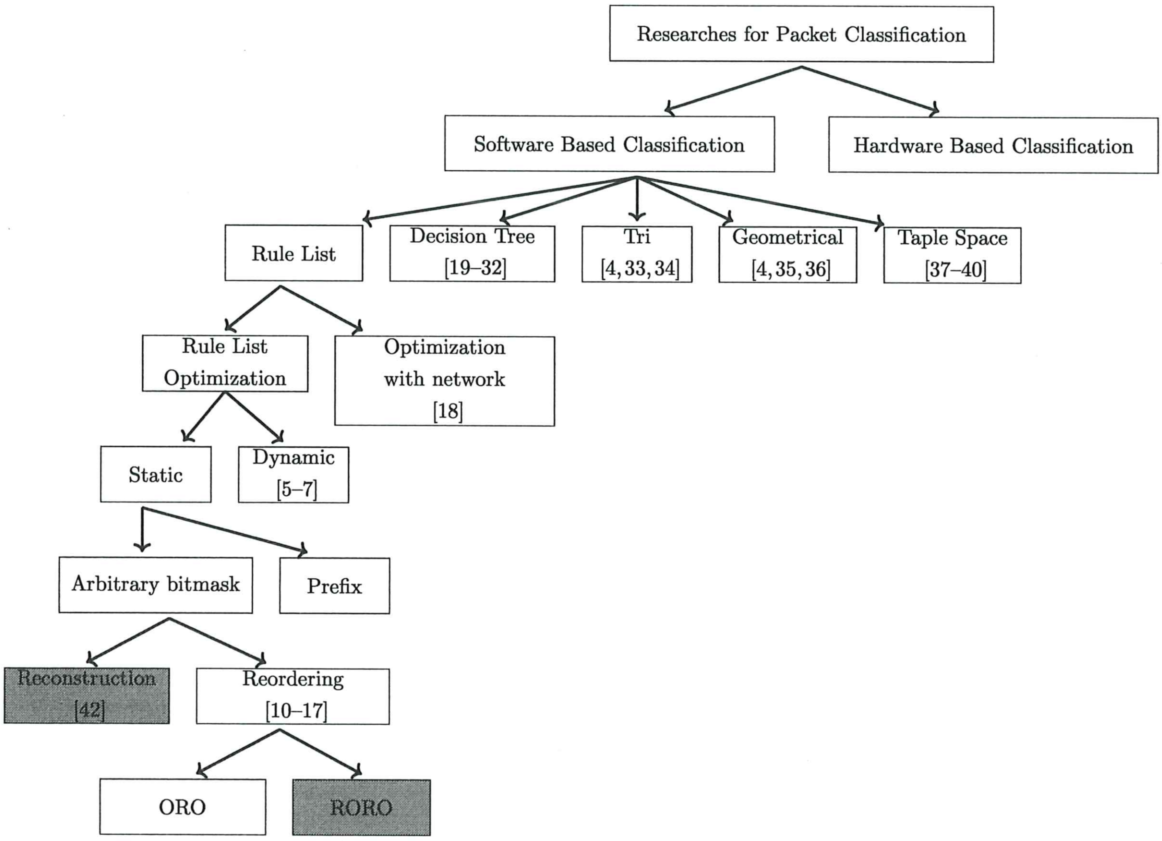


Figure 1.3: Research classification overview.

Chapter 2

Packet Classification Using Rule List

In this chapter, we describe packet classification using rule lists and define an optimization problem that aims to speed up packet classification using rule lists. In Section 2.1, describes packet classification using rule lists and provides definitions for rules and packets. In Section 2.2, we define Optimal Rule Ordering and Relaxed Optimal Rule Ordering, which are problems in finding a sequence of rules with minimum latency.

2.1 Rule list and Packet

Packet classification using a rule list is realized as illustrated in Figure 2.1. Every rule consists of a rule number $i \in \mathbb{N}$, a condition on $\{0, 1, *\}^l$ and an evaluation type on $\{a_1, a_2, \dots a_m\}$ where l is the length of a condition, and the symbol $*$ indicates that the bit matches both 1 and 0. A rule list consists of n rules. In this paper, we assume that there are two actions, $a_i \in \{A, D\}$. A means permission of communication, and D means denial of communication. A packet is a bit string with length l on $\{0, 1\}^l$. The rule is defined in 2.1.1. Examples of rules and a rule list are listed in Table 2.1.

Definition 2.1.1. *Rule Formalization*

Table 2.1: Rule List $\mathcal{R}$.

Filter $\mathcal{R}$	
r_1^A	0010
r_2^A	10**
r_3^D	*01*
r_4^D	1*0*
r_5^A	**01
r_6^A	**00
r_7^D	****

Table 2.2: Reordering by σ.

Filter $\mathcal{R}$	
r_2^A	10**
r_4^D	1*0*
r_5^A	**01
r_6^A	**00
r_1^A	0010
r_3^D	*01*
r_7^D	****

Table 2.3: Policy.

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

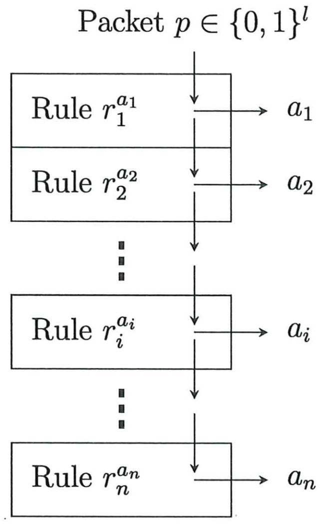


Figure 2.1: Packet classification model.

Table 2.4: Rule List $\mathcal{R}$ with weight.

	Filter $\mathcal{R}$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{F}}$
r_1^A	0010	10
r_2^A	$10**$	5
r_3^D	$*01*$	30
r_4^D	$1*0*$	12
r_5^A	$**01$	25
r_6^A	$**00$	26
r_7^D	$****$	22
$L(\mathcal{R}_\sigma, \mathcal{F}) = 571$		

Table 2.5: Rule List $\mathcal{R}_\sigma$ with weight.

	Filter $\mathcal{R}_\sigma$	$\lvert E(\mathcal{R}_\sigma, i)\rvert_{\mathcal{F}}$
r_2^A	$10**$	5
r_4^D	$1*0*$	12
r_5^A	$**01$	25
r_6^A	$**00$	26
r_1^A	0010	10
r_3^D	$*01*$	30
r_7^D	$****$	22
$L(\mathcal{R}_\sigma, \mathcal{F}) = 570$		

Table 2.6: distribution $\mathcal{F}$.

$0000 \mapsto 22$	$0001 \mapsto 15$
$0010 \mapsto 10$	$0011 \mapsto 30$
$0100 \mapsto 4$	$0101 \mapsto 10$
$0110 \mapsto 9$	$0111 \mapsto 3$
$1000 \mapsto 1$	$1001 \mapsto 1$
$1010 \mapsto 1$	$1011 \mapsto 2$
$1100 \mapsto 5$	$1101 \mapsto 7$
$1110 \mapsto 4$	$1111 \mapsto 6$

$$
\begin{aligned}
r_i^{a_i} &= b_1 b_2 \dots b_l, \\
& b_k \in \{0, 1, *\}, \\
& a_i \in \{A_1, A_2, \dots, A_m\}
\end{aligned}
\tag{2.1}
$$

In the following, the subscription of actions may be omitted for simplification of the notation.

Let $\mathcal{P}$ denote the set of packets. An incoming packet in a network device is compared with each rule in order and the evaluation type of the first matched rule is provided to the packet. We add the default rule r_n^e to the bottom of the list, since all arriving packets match at least one rule. The default rule is the rule that all bits are $*$. Assume that the ordering of n items is a bijection function $\sigma : [n] \to [n]$.

$$
\sigma = (2\ 4\ 5\ 6\ 1\ 3\ 7) \tag{2.2}
$$

For example, In the order 2.2, $\sigma(3) = 6$ means that r_3 is sixth in the list and $\sigma^{-1}(4) = 6$ means that the fourth rule is r_6. We denote the rule list that is sorted in the order of σ by $\mathcal{R}_\sigma$. For example, the rule list in Table 2.2 is the rule list in Table 2.1 sorted by σ. The rule list $\mathcal{R}$ is a function to the set of evaluation types $\{a_1, a_2, a_3, \ldots, a_m\}$, and this function is defined as the policy of $\mathcal{R}$. We use $\mathcal{R}(p)$ to denote an evaluation type for p as the classification result. For example, in the rule list $\mathcal{R}$ in Table 2.1, $\mathcal{R}(0110) = D$. The policy of the rule list $\mathcal{R}$ in Table 2.1 is shown in Table 2.3.

If a packet p exists for $\mathcal{R}(p)$ and σ such that $\mathcal{R}(p) \neq \mathcal{R}_\sigma(p)$, we state that the ordering σ violates the policy, or that a policy violation occurs. There are multiple orders that satisfy the same policy. For example, the rule list in Table 2.1 and the rule list in Table 2.2 that reorders the rule list by order2.2 both satisfy the same policy.

We denote the set of packets that can match the rule r_i without rules with a higher position than r_i by $M(r_i)$.For example, given the rule list $\mathcal{R}$ in Table 2.1, the set of packets that are matched by rule r_5 is

$$M(r_5) = \{0001, 0101, 1001, 1101\}.$$

Since packets are compared with each rule in order and the evaluation type of the first matched rule, packets that match r_i are included in $M(r_i)$, excluding packets that match rules that have higher positions than r_i. We denote this set of packets by $E(\mathcal{R}, i)$. For example, the set of packet matched r_5 is

$$E(\mathcal{R}, 5) = \{0001, 0101\}.$$

Given a set of packets $\mathcal{P}$ and a packet arrival distribution $\mathcal{F} : P \to \mathbb{N}$, we denote the number $\sum_{p \in P} \mathcal{F}(p)$ as $|\mathcal{P}|_\mathcal{F}$.

Given a packet arrival distribution F, a rule list $\mathcal{R}$, the number of packets that actions are determined by the rule r_i^a is called the number of packets evaluated for r_i^a or the weight. We denote the number as $|E(\mathcal{R}, i)|_\mathcal{F}$ and refer to it as the weight of rule r_i. For example, the weight of r_5 in Table 2.1 and Table 2.6 is shown as follows.

$$|E(\mathcal{R}, 5)|_\mathcal{F} = |\{0001, 0101\}|_\mathcal{F} = 25.$$

Regarding a comparison of a packet with a rule as latency 1, on a rule list $\mathcal{R}$, and a packet arrival distribution F, the classification latency $L(\mathcal{R}, \mathcal{F})$ is defined as follows:

Definition 2.1.2. (classification latency)

$$L(\mathcal{R}, \mathcal{F}) = \sum_{i=1}^{n-1} i|E(\mathcal{R}, i)|_\mathcal{F} + (n-1)|E(\mathcal{R}, n)|_\mathcal{F} \tag{2.3}$$

Since the rule list matches at least one rule for every packet that arrives, the last rule is not compared, so the number of packets that match the last rule is $n-1$. The latency when the rule list in Table 2.1 classifies the packet set in Table 2.6 is shown in Table 2.4.

In packet classification using rule lists, packets that match a rule placed lower in the list are compared more frequently, and thus, there is a problem that latency increases when rules with high matching frequency are placed lower in the list. Reducing the number of rules and placing the rule with the higher matching frequency at the upper position generally reduces latency. Therefore, we define the following problem to minimize latency while maintaining policy.

Definition 2.1.3. (Optimal Rule List)

$$\begin{aligned} \text{Input}: &\quad \text{Rule list } \mathcal{R}, \text{ Packet Arrival Distribution } \mathcal{F} \\ \text{Output}: &\quad \text{A rule list } \mathcal{R}' \text{ that holds the policy and minimizes the latency } L(\mathcal{R}', \mathcal{F}) \end{aligned}$$

In the optimal rule list problem, we can consider an optimization problem restricted to the operation of reordering rules. In the following, we define two types of optimization problems: optimal rule ordering problems based on precedence constraints and optimization problems that do not depend on precedence constraints.

2.2 Optimal Rule Ordering

It is important to determine which pair of rules causes a policy violation when these rules are reordered. Therefore, we define the overlap relations on the rules as follows.

Definition 2.2.1. (overlap on rules)
If a packet exists that matches both r_i and r_j, or $M(r_i) \cap M(r_j) \neq \emptyset$, we state that r_i and r_j are overlapped.

If these rules are interchanged, the rules that the packet is matched to may change, causing a violation of the policy. For example, consider a rule list that contains rules r_i and r_j that can match packet p. Before reordering, r_i is placed above r_j and packet p matches r_i. When these rules are replaced, the rule that packet p matches changes to r_j. In packet classification using a rule list, the action of the first matching rule is applied, so reordering causes packet p to be subject to the action of r_j, resulting in a policy violation. Since the policy can be maintained if the precedence constraint based on the overlap relation holds, we define the following problem.

Definition 2.2.2. (Optimal Rule Ordering)

$$\begin{aligned} \text{Input}: &\quad \text{Rule list } \mathcal{R} \text{ and packet arrival distribution } \mathcal{F} \\ \text{Output}: &\quad \text{An order of rules } \sigma \text{ that hold the precedence constraint and minimizes } L(\mathcal{R}_\sigma, \mathcal{F}). \end{aligned}$$

The corresponding decision problem to **ORO** is known to be **NP**-hard, and many heuristic solutions have been proposed. However, there are orders of rules that hold policies even if they do not hold precedence constraints due to overlap relations. For example, if the actions of the overlapping rules r_i and r_j are the same, the policy is preserved even if a packet that matched r_i now matches r_j. In order to find the sequence of rules with the lowest latency regardless of the overlap relation, we define a relaxed optimization problem as follows.

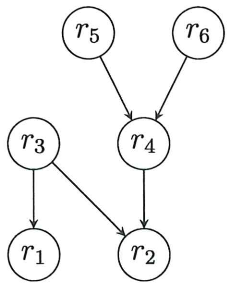


Figure 2.2: Dependent graph for Table 2.4.

Definition 2.2.3. (Relaxed Optimal Rule Ordering(**RORO**))

$$\begin{aligned} \text{Input}: &\quad \text{Rule list } \mathcal{R}, \text{and packet arrival distribution } \mathcal{F} \\ \text{Output}: &\quad \text{An order of rules } \sigma \text{ that hold the policy and minimizes } L(\mathcal{R}_\sigma, \mathcal{F}). \end{aligned}$$

Hamed's proof for **ORO** is not sufficient for **RORO** because it does not consider weight fluctuation. Therefore, this study provides a rigorous proof for this problem.

Many heuristics for **ORO** and heuristics for **RORO** reorder rules while holding precedence constraints by dependency relations in order to preserve policy.

Definition 2.2.4. (dependency on rules) If $r_i^{e_i}$ and $r_j^{e_j}$ are overlapped and evaluation type e_i is different from e_j, we say that r_i and r_j are dependent.

We define the dependent graph $G_{\mathcal{R}} = (V, A)$ on the $\mathcal{R}$ as follow:

$$\begin{aligned} V &= \{1, 2, \ldots, n\} \\ A &= \{\ ik \mid i, k \in V, i < k, D(r_i, r_k) \\ &\qquad \neg\exists j \in V, i < j < k \wedge D(r_i, r_j) \wedge D(r_j, r_k)\} \end{aligned} \tag{2.4}$$

Note that $D(r_i, r_j)$ means that r_i and r_j are dependent. The condition $\neg\exists j \in V, i < j < k \wedge D(r_i, r_j) \wedge D(r_j, r_k)$ in 2.4 is the condition for removing the edge from r_i to r_k when there is more than one path from r_i to r_k. We denote the dependent graph 2.2 of Table 2.4.

Interchanging the dependent rules r_j and r_i will change the action given to packets that match both rules, and may cause a policy violation. Note, however, that there are also orders that hold policy even if they do not hold precedence constraints due to the dependent relation.

Chapter 3

Optimal Rule Ordering

This chapter describes the rule order optimization problem. In packet classification using a rule list, the latency of classification increases as the number of packets that match the rules placed lower in the list increases. Therefore, the order of rules with lower latency can be computed by placing the rule with the higher matching frequency at the higher of the list and the rule with the lower matching frequency at the lower of the list. **ORO** is the problem of finding an order of rules with minimum latency that holds precedence constraints based on overlap relations in order to hold policy. However, there exist rule sequences that hold policy even if these ordering relations do not hold. For example, if rules that can match the same packet both have the same action, the policy is held even if their order relations are interchanged. If the overlapping rules r_i and r_j have the same action, even if the rule that matches packet p changes from r_i to r_j, the action applied is the same and no policy violation occurs. Therefore, the problem of finding a sequence of rules that minimizes latency while holding policy, the rule order optimization problem (**RORO**), has been studied.

In this chapter, we first show that **RORO** is **NP**-hard in Section. 3.1. Then, we propose several heuristics for this problem. In Section 3.2, we describe SGM, which is known as the method that, on average, reduces latency the most in **ORO**, and propose problems with SGM and methods to improve them. In Section 3.3, we describe Hikage et al.'s method, which is known as the method that reduces latency the most in $\mathcal{O}(n^2)$ reordering methods, and explain the problems this method has. Then, we propose a method to improve the problems. In Section 3.4, we propose a method to find a reduced latency order of rules by considering not only the rules reachable from the rule, but also the rules subordinate to the rule.

In Section 3.5,we explain the problems with reordering methods using weight average. Then, we propose a method to find a order of rules with smaller latency by constructing a order that can be expected to have smaller latency and comparing it with previous reordering methods. The effectiveness of the method as demonstrated by experiments is discussed in Section 3.6.

In Section 3.7, we propose a method to auxiliary previous reordering methods to find an order of rules that has less latency. In Section 3.7.1, we propose an method to find a order of rules with lower latency by searching for rules with no matching packets and placing them

Table 3.1: Rule list $\mathcal{R}$.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{F}}$
$r_1^A = 1{*}{*}0$	90
$r_2^A = {*}1{*}1$	70
$r_3^A = 0{*}01$	60
$r_4^A = {*}110$	120
$r_5^A = 1{*}1{*}$	30
$r_6^A = 01{*}{*}$	40
$r_7^A = {*}{*}00$	20
$r_8^D = {*}{*}{*}{*}$	30
$L(\mathcal{R}, \mathcal{F}) = 1630$	

Table 3.2: Packet distribution $\mathcal{F} : \mathcal{P} \to \mathbb{N}$.

$0000 \mapsto 20$	$0001 \mapsto 60$	$0010 \mapsto 10$	$0011 \mapsto 10$
$0100 \mapsto 40$	$0101 \mapsto 30$	$0110 \mapsto 120$	$0111 \mapsto 10$
$1000 \mapsto 10$	$1001 \mapsto 10$	$1010 \mapsto 30$	$1011 \mapsto 30$
$1100 \mapsto 10$	$1101 \mapsto 10$	$1110 \mapsto 30$	$1111 \mapsto 30$

lower than the default rule. Many heuristic solutions to **RORO** reorder the rules according to precedence constraints based on dependency relations. In Section. 3.7.2, we propose a method to reduce the latency of previous reordering methods by searching for and removing precedence constraints that do not affect the policy. To evaluate the effectiveness of these auxiliary methods, we performed computer experiments, the results of which are presented in Section. 3.7.3

In actual environments, there is a trend to use Allow lists because of their resistance to unknown attacks. An Allow list is a rule list in which all actions of rules other than the default rule are Allow. Because of this, Allow lists do not have precedence constraints except for the default rules, the previous reordering method can only order of rules in descending order by weight, which does not sufficiently reduce latency. Therefore, in Section. 3.8.1 we propose a method of calculating the number of matchable packets for each rule and reordering the rules according to this value. Furthermore, we conduct computer experiments to confirm the effectiveness of the proposed method. Finally, in Section. 3.9 we summarize the optimal rule ordering problem and discuss future issues.

3.1 Complexity of RORO

In this section, we show that the decision version of Relaxed Optimal Allow Rule Ordering is **NP**-hard by reducing from **EXACT COVER BY 3-SETS (XC3)**. Relaxed Optimal Allow Rule Ordering (**RAO**) is defined as follows:

Definition 3.1.1. (Decision version of Relaxed Allow rule Ordering (**RAO**))
Input: Allowlist $\mathcal{R}$, packet distribution $\mathcal{F}$, and positive integer K.
Question: Is there an order σ, such that $L(\mathcal{R}_\sigma, \mathcal{F}) \leq K$ and $\forall p \in \mathcal{P}, \mathcal{R}(p) = \mathcal{R}_\sigma(p)$?

Note that an allowlist is a rule list in which all rule actions except the default rule are allow, and the default rule action is deny. For example, consider the problem where $\mathcal{R}$ is Table 3.1, $\mathcal{F}$ is Table 3.2, and the positive integer $K = 770$. The latency of the rule list $\mathcal{R}$ that is reordered by the order $\sigma = (1, 2, 4, 3, 6, 5, 7, 8)$ is $L(\mathcal{R}_\sigma, \mathcal{F}) = 760$. Thus the answer is Yes. In contrast, if $K = 460$, then all packets in the distribution $\mathcal{F}$ must be matched with the first rule, but since no such rule exists the answer is No.

The decision problem, **EXACT COVER BY 3-SETS (XC3)** is defined as follows:

Definition 3.1.2. (**EXACT COVER BY 3-SETS (XC3)**)
Input: Set $S = \{s_1, s_2, \ldots, s_n\}$ and family of subsets $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ such that $|S|$ is multiple of 3 and $\forall i \in [m]$, $|C_i| = 3$.
Question: Is there a subset $\mathcal{D}$ of $\mathcal{C}$ such that $\bigcup_{c \in \mathcal{D}} c = \mathcal{C} \land \forall c, c' \in D,\ c \cap c' = \emptyset$?

This decision problem is known to be **NP**-hard [43]. The following is a concrete example of **XC3**.

$$\begin{aligned} S &= \{1,2,3,4,5,6,7,8,9\} \\ \mathcal{C} &= \{\{5,7,9\},\{4,5,9\},\{3,6,7\}, \\ &\qquad \{1,4,8\},\{7,8,9\},\{2,3,6\}\} \end{aligned} \tag{3.1}$$

For this instance, the answer is Yes because we can take $\mathcal{D} = \{\{1,4,8\},\{2,3,6\},\{5,7,9\}\}$ in the subset $\mathcal{C}$. If $\mathcal{C} = \{\{5,7,9\},\{4,5,9\},\{3,6,7\},\{3,4,8\},\{7,8,9\},\{1,2,3\}\}$, the answer is No.

3.1.1 Reduction from XC3 to RAO

Theorem 3.1.1. *RAO is NP-hard.*

Proof. We show a polynomial-time reducing algorithm f from **XC3** to **RAO**.

f takes inputs an instance S and $\mathcal{C}$ of **XC3** and outputs the instance of an allowlist problem $\mathcal{R}, \mathcal{F}$, and K.

Let the rule length of the allowlist l be $|S|$ and the number of rules n be $|\mathcal{C}| + 1$. Each bit of the rule r_i, except the default rule, is defined as follows:

$$b_j = \begin{cases} `*\text{'} & \text{if } j \in C_i \\ `0\text{'} & \text{otherwise} \end{cases} \tag{3.2}$$

The rule list $\mathcal{R}$ consists of $r_1, r_2, \ldots, r_{n-1}$ that are generated as described above, and the default rule r_n. And let the packet distribution $\mathcal{F}$ be as follows:

$$\mathcal{F}(p) = \begin{cases} 1 & \text{if } \exists! i \in \{1, \ldots, l\},\ p_i = \text{'1'} \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

Note that $\exists!$ means that only one exists. We denote that p_i is the ith bit in the packet p. Furthermore, let the integer $K = \sum_{i=1}^{N} 3i = \frac{3N(N+1)}{2}$, where $N = |S|/3$.

It is clear that f is computable in a polynomial time. Thus, we show that $(S, \mathcal{C}) \in \mathbf{XC3} \iff f(S, \mathcal{C}) \in \mathbf{RAO}$ where $(S, \mathcal{C})$ is an instance of $\mathbf{XC3}$.

We assume that the subset $\mathcal{D} = \{D_1, D_2, \ldots, D_m\}$ of the family $\mathcal{C}$ of subsets of S is an exact covering of S. For the rule list $\mathcal{R} = \langle r_1, r_2 \ldots, r_{|\mathcal{C}|}, r_{|\mathcal{C}|+1}\rangle$, and the packet distribution $\mathcal{F}$ generated by f, there exists an order σ such that all packets with a frequency of 1 are evaluated by the upper $|S|/3$ rules $r_{\sigma^{-1}(1)}, r_{\sigma^{-1}(2)}, \ldots, r_{\sigma^{-1}(|S|/3)}$ of the rule list. Since the weight of these rules is 3, the latency of the rule list $\mathcal{R}_\sigma$ is as follows:

$$L(\mathcal{R}_\sigma, \mathcal{F}) = \sum_{i=1}^{|S|/3} 3i = \frac{|S|(|S|/3+1)}{2} = K$$

Thus, this makes $x \in \mathbf{XC3} \Rightarrow f(S, \mathcal{C}) \in \mathbf{RAO}$ valid.

We also show $f(S, \mathcal{C}) \notin \mathbf{XC3} \Rightarrow x \notin \mathbf{RAO}$ that is the contrapositive of $(S, \mathcal{C}) \in \mathbf{XC3} \Rightarrow f(S, \mathcal{C}) \in \mathbf{RAO}$.

Assume that for a family $\mathcal{C}$ of subsets of a set S, there is no exact covering of S. For the rule list $\mathcal{R} = \langle r_1, r_2 \ldots, r_{|\mathcal{C}|}, r_{|\mathcal{C}|+1}\rangle$, and packet distribution $\mathcal{F}$ generated by f, there is no ordering σ such that all packets with a frequency of 1 are evaluated by the upper $|S|/3$ rules $r_{\sigma^{-1}(1)}, r_{\sigma^{-1}(2)}, \ldots, r_{\sigma^{-1}(|S|/3)}$ of the rule list. So, for any ordering σ, there is at least one packet that is evaluated by the rule that placed $|S|/3+1$ or later. For such an ordering, the following relationship holds.

$$\begin{aligned} K &= \frac{|S|(|S|/3+1)}{2} \\ &= \sum_{i=1}^{|S|/3} 3i \\ &< L(\mathcal{R}_\sigma, \mathcal{F}) \\ &= \sum_{i=1}^{|S|/3} i|E(\mathcal{R}_\sigma, \sigma^{-1}(i))|_{\mathcal{F}} \\ &\quad + \sum_{i=|S|/3+1}^{|\mathcal{C}|+1} i|E(\mathcal{R}_\sigma, \sigma^{-1}(i))|_{\mathcal{F}} \end{aligned}$$

This makes $f(S, \mathcal{C}) \notin \mathbf{XC3} \Rightarrow x \notin \mathbf{RAO}$ valid, and therefore $f(S, \mathcal{C}) \in \mathbf{RAO} \Rightarrow x \in \mathbf{XC3}$ valid. From the above, **RAO** is **NP**-hard because of the existence of a polynomial-time reducing algorithm f from **XC3** to **RAO**. $\square$

Table 3.3: The allowlist: Result of reduction from Eq. (3.1).

$r_1^A = 0000{*}0{*}0{*}$
$r_2^A = 000{*}{*}0000$
$r_3^A = 00{*}00{*}{*}00$
$r_4^A = {*}00{*}000{*}0$
$r_5^A = 000000{*}{*}{*}$
$r_6^A = 0{*}{*}00{*}000$
$r_7^D = {*}{*}{*}{*}{*}{*}{*}{*}{*}$

For example, applying the reduction algorithm f to the instance of the Eq. (3.1), the algorithm outputs the allowlist in Table 3.3, the packet distribution such that 000000001, 000000010, 000000100, 000001000, 000010000, 000100000, 001000000, 010000000, and 100000000 are 1 and the other packets are 0, and the integer $K = 18$.

3.1.2 Reduction from RAO to RORO

Then, we show that a decision version of **RORO** is **NP**-hard. The decision version of Relaxed Optimal Rule Ordering (**RORO**) is defined as follows:

Definition 3.1.3. *(Decision version of **RORO**)*
Input: *Rule list $\mathcal{R}$, packet distribution $\mathcal{F}$, and positive integer K.*
Question: *Is there an order σ, such that $L(\mathcal{R}_\sigma, \mathcal{F}) \leq K$ and $\forall p \in \mathcal{P}, \mathcal{R}(p) = \mathcal{R}_\sigma(p)$?*

Corollary 3.1.1. *Decision version of **RORO** is **NP**-hard.*

Proof. An instance $\mathcal{R}$ of the **RAO** can be regarded as an instance of **RORO** by considering it as a rule list in which the actions of all rules except the default rule are "Allow". And **RAO** and **RORO** clearly make the same decision in the same instance. Therefore, since **RAO** is a limited version of **RORO**, the decision version of **RORO** is **NP**-hard. □

3.2 Improved SGM

In this section, we propose improved methods of SGM. SGM is a method that reduces latency by considering dependencies and placing rules with high weights at the top of the list.

To begin with, we provide an outline of SGM. For rules $r_1, r_2, \ldots, r_n$, SGM makes the reachable rule set $G(r_i)$ for every rule, compares these average weights, and selects the heaviest rule set $G(r_s)$. If the rule set $G(r_s)$ is a singleton set, SGM adds element r_s to the sorted rule list and repeats the above operations until the input rule list becomes empty. Otherwise, SGM

Table 3.4: Rule list $\mathcal{R}$.

Filter $\mathcal{R}_\sigma$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{F}}$
r_1^A =0*101	87
r_2^A =0000*	60
r_3^D =0**01	5
r_4^D =0101*	55
r_5^D =0111*	55
r_6^A =01***	400
r_7^A =00***	60
r_8^A =10*1*	65
r_9^D =*****	50
$L(\mathcal{R}, \mathcal{F}) = 4684$	

Table 3.5: The packet arrival distribution $F : \mathcal{P} \to \mathbb{N}$.

00000 $\mapsto$ 10	00001 $\mapsto$ 50	00010 $\mapsto$ 17	00011 $\mapsto$ 23
00100 $\mapsto$ 20	00101 $\mapsto$ 60	00110 $\mapsto$ 8	00111 $\mapsto$ 8
01000 $\mapsto$ 200	01001 $\mapsto$ 5	01010 $\mapsto$ 20	01011 $\mapsto$ 35
01100 $\mapsto$ 200	01101 $\mapsto$ 27	01110 $\mapsto$ 15	01111 $\mapsto$ 40
10000 $\mapsto$ 8	10001 $\mapsto$ 2	10010 $\mapsto$ 12	10011 $\mapsto$ 13
10100 $\mapsto$ 6	10101 $\mapsto$ 2	10110 $\mapsto$ 12	10111 $\mapsto$ 28
11000 $\mapsto$ 1	11001 $\mapsto$ 13	11010 $\mapsto$ 2	11011 $\mapsto$ 1
11100 $\mapsto$ 3	11101 $\mapsto$ 3	11110 $\mapsto$ 7	11111 $\mapsto$ 2

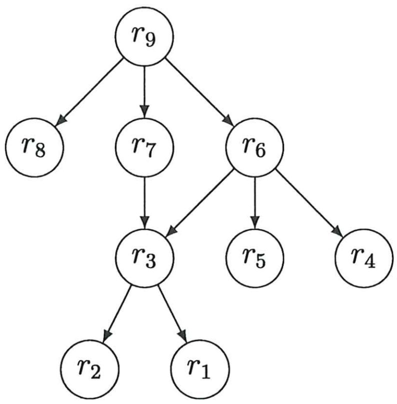


Figure 3.1: The dependent graph in Table 3.4.

selects the heaviest rule set from rule sets $G(r_i), \ldots, G(r_j)$ based on rules $r_i, \ldots, r_j$, which are adjacent to r_s.

SGM is formalized as follows. For each rule r_i, rule set $G(r_i)$ consisting of rules that can reach r_i and its weight $Z(r_i)$ are defined. For instance, for the rule list in Figure 3.1,

$$G(r_7) = \{r_1, r_2, r_3, r_7\},$$

and

$$\begin{aligned} Z(r_7) &= \sum_{r \in G(r_7)} (\text{the weight of } r) \\ &= 14 + 16 + 4 + 12 = 46. \end{aligned}$$

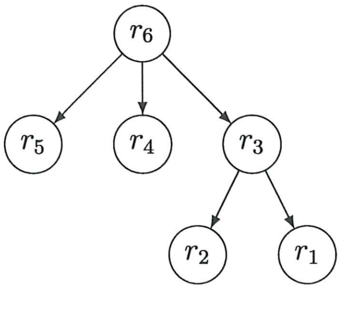


Figure 3.2: Reachable rules from r_6.

The quotient of the sum of weight $Z(r_i)$ divided by its cardinality $|G(r_i)|$ is the average weight of $G(r_i)$. For rule set $G(r_7)$ in Figure 3.1, the average weight is $46/4 = 11.5$.

First, SGM focuses on rule r_i whose $Z(r_i)/|G(r_i)|$ is the maximum, to set a heavier rule in an early position. Then, in order to decide the rule to select, SGM searches rules $r_j, r_k, \ldots, r_l$ that are adjacent to r_i, computes their average weights $Z(r_j)/|G(r_j)|, Z(r_irk)/|G(r_k)|, \ldots, Z(r_l)/|G(r_l)|$, and selects the heaviest rule. Repeating this process, SGM takes rule r_b, which depends on no rule (i.e., it's out–degree $deg^+(r_b)$ is 0). r_b is inserted in the sorted list. SGM repeats the above two processes, selecting rule r_b and inserting it in the sorted list until the input rule list becomes empty.

We demonstrate the selection of a rule in SGM for the rule list in Figure 3.1. First, for rules $r_1, r_2, \ldots, r_9$ in the rule list, we compose the reachable set $G(r_1), G(r_2), \ldots, G(r_9)$ and their average weights. Among the sub–graphs for the rule list in Figure 3.1, the sub–graph $G(r_6)$ is the heaviest, with its average weight being 101/6. Since the out–degree of r_6 is not 0, we focus on sub–graph $G(r_6)$ as shown in Figure 3.2. Since r_3, r_4, and r_5 are adjacent to r_6 in graph $G(r_6)$, we compute their average weights as $Z(r_3)/|G(r_3)| = 34/3$, $Z(r_4)/|G(r_4)| = 12$, and $Z(r_5)/|G(r_5)| = 13$, respectively. Then, the heaviest rule, r_5, is selected. As the out–degree of r_5 is 0, we add r_5 to the sorted list and remove r_5 from the input rule list. For the rule list in Table 3.4, repeating the above process results in the rule list

$$r_5^D, r_4^D, r_2^A, r_1^A, r_3^D, r_6^A, r_8^A, r_7^A, r_9^D.$$

In Figure 1, we show the pseudocode of SGM in [11]. Parameters S, Q, X, C, PROB, and DEP in Figure 1 represent the empty list, an input rule list, an array of length n (which contains $Z(r_i)$ for each rule r_i), an array of length n (which consists of the size of $G(r_i)$), an array of length n (which stores the weight for each rule), and a two–dimensional array representing the preceding relations for the rules.

After SGM inserts a rule in the sorted list, the algorithm should delete the rule from the input rule list, as described above. This update process corresponds to lines 26 to 28 in Figure 1. In this part of the process, the algorithm decrements only C$[r_i]$ of r_i, which is adjacent to r_{select}. It is thought that since r_{select} is inserted in the sorted list, the algorithm removes the preceding relations of r_{select}, that is, the edges contain r_i.

However, there is a possibility that rule r_j exists, which contains r_{select} in $G(r_j)$ and is not directly dependent on rule r_{select}. Thus, the algorithm often falls into an infinite loop.

Therefore, we fix the algorithm such that when removing rule r_i from the graph, the algorithm decrements not only C[r_j] but also all C[r_k], where r_j is adjacent to r_i, and r_k is reachable from r_i. We show the fixed algorithm in Figure 2.

3.2.1 Using the Adjacency List

Since the algorithm in [11] manages the preceding relation with the two–dimensional array DEP[][], a considerable amount of time is consumed to reorder rules when the preceding relation is complex. Identifying adjacent rules for r_i with the two–dimensional array DEP[][] requires at most n steps. For example, consider the loop from 15 to 22 in Figure 1. The algorithm must access DEP[r_1][r_b] to DEP[r_{b_1}][r_b] to decide whether rule r_i is adjacent to r_b or not.

When managing the preceding relation with the adjacency list, we do not search for rules adjacent to rule r_i and can thus reorder rules faster.

3.2.2 Comprehensive Construction of Sub–Graphs

In this section, We propose an improved SGM, that can reorder large–scale rule sets practically.

First, SGM selects rule r_b such that the average weight of sub–graph $G(r_b)$ is maximum. If $G(r_b)$ is not a singleton, SGM searches for rules $r_{i_1}, r_{i_2}, \ldots, r_{i_k}$ that are adjacent to r_b, and compares the average weights of sub–graphs based on those rules, as shown in line 16 in Figure 1. However, a rule $r^{'}$ can exist such that it is not adjacent to r_b, and the average weight of $G(r^{'})$ based on $r^{'}$ is the highest among sub–graphs $G(r_{i_1}), G(r_{i_2}), \ldots, G(r_{i_k})$. Then, selecting rule $r^{'}$ reduces the latency compared to SGM in most cases. When average weight $G(r_i)$, based on r_i (which is adjacent to r_b) is low, and average weight $G(r^{'})$, based on $r^{'}$ (which is reachable from r_b) is large, SGM does not select $r^{'}$ instead of r_i. For instance, for the rule list in Figure 3.1, SGM computes the average weights $G(r_3)$, $G(r_4)$ and $G(r_5)$, because only r_3, r_4, and r_5 are adjacent r_6. Then, since the average weight of $G(r_5)$ is maximum, it selects r_5. However, since the average weight of $G(r_2)$, $Z(r_2)/|G(r_2)| = 16$ is larger than that of $G(r_5)$, $Z(r_5)/|G(r_5| = 13$, we should select r_2 instead of r_5.

For the rule list in 3.1, the latencies of the rule list reordered by SGM and the above method are 4468 and 4364, respectively.

As shown above, we modify the process in line 16 in the pseudocode 1 such that the algorithm compares not only sub–graphs based on rules $r_{i_1}, r_{i_2}, \ldots, r_{i_k}$ that are adjacent to r_b, but also rules $r_{j_1}, r_{j_2}, \ldots, r_{j_l}$ that are reachable from r_b. This algorithm is shown in Figure 3.

The main part of the algorithm 3 is the recursive function *selectMaxWeightRule*() that takes a graph as an input and returns the heaviest sub–graph $G(r_b)$ of the input graph. *select-MaxWeightRule*() shown in the algorithm 4 computes the average weights of $G(r_i)$ for all rules r_i in the input graph, and returns the heaviest sub–graph $G(r_b)$. $SG(r_i)$ in line 6 in Figure 4 denotes the average weight of $G(r_i)$, $Z(r_i)/|G(r_i)|$.

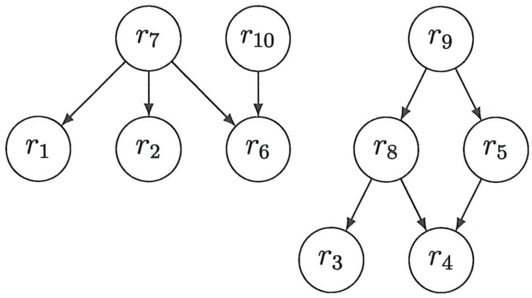


Figure 3.3: The dependent graph in Table 3.6.

The algorithm shown in Algorithm 3 reorders the rules in Table 3.4 as

$$r_2^A, r_1^A, r_5^D, r_4^D, r_3^D, r_6^A, r_8^A, r_7^A, r_9^D,$$

and the latency of this rule list is 4352 while that of SGM is 4406. The proposed algorithm thus decreases latency compared to SGM.

3.3 Improved Hikage's Method

In this section, we propose an improved method of Hikage's method.

Hikage et al. proposed the following rule-reordering method [13]: First, a dependency graph is regarded as an undirected graph and decomposed into connected components. Thereafter, in each component, rules are determined in the order of the rule weights. Subsequently, the smallest rule is moved to a lower position as far as possible. [1]

Furthermore, based on the above approach, they proposed another method that determines an order in the component via each rule weight instead of each rule evaluation, which is the average weight of the rules that are reachable from the rule including itself. The time complexity of their method was $O(n^2)$. In the following, we refer to the latter algorithm of Hikage et al. as Hikage's method. We demonstrate the method in Algorithms 5 and 6.

Algorithm 5 divides the digraph constructed from the dependency relations over the rules into connected components $C_1, C_2, \ldots, C_k$ by regarding the digraph as an undirected graph in line 1, where k is the number of components when a digraph is regarded as an undirected graph. Next, it determines the order in each component C_i by applying Algorithm 6 to each component C_i. Algorithm 6 inserts a rule with an indegree of 0 in the order of the rule weight into a list N that denotes the order of a component. For each component C_i, the list N_i denotes the order of the rules in C_i. Thereafter, in 4 to 6, for all rules r, it computes W, which is the ratio of the sum of weights of j rules $r_I, \ldots, N.\text{last}$ divided by j, where j is the position of the rule r from the tail in N_i. The algorithm determines the order of the rules according to the following process until all of the lists become empty: In line 8, the rule r is selected in the non-empty list

[1] Although the time complexity of the method in [13] was $O(n^2)$, they fixed it as $O(n^{2.3728})$, which is the time complexity of computing the transitive closure of a digraph, where n is the number of nodes.

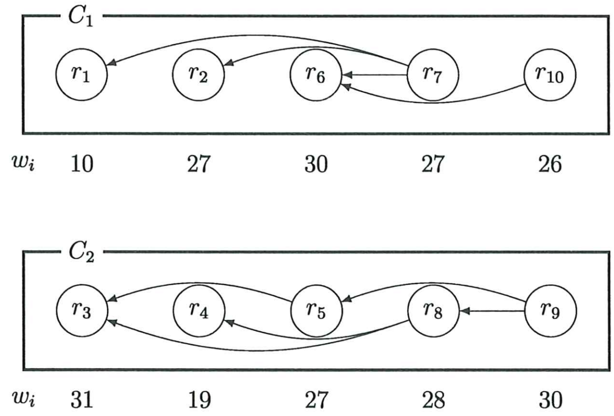
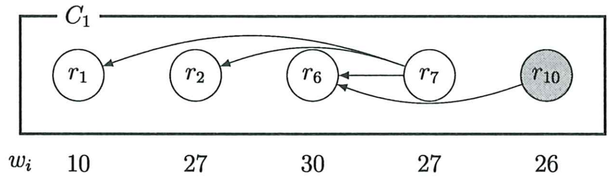
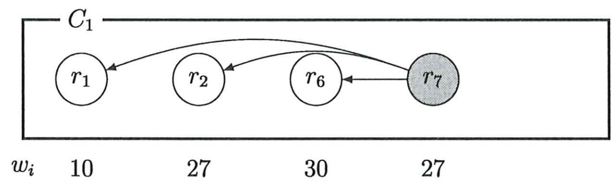


Figure 3.4: Divide rules into two sets of rules (components).



$N_1 = \{\}$

Figure 3.5: Select the lightest rule r_{10} among the rules r_7 and r_{10} with indegrees of 0.



$N_1 = \{r_{10}\}$

Figure 3.6: Select rule r_7 with indegree of 0.

N, for which W is the smallest among the remaining rules. Thereafter, the rules from r to the last rule in N are added into the sorted list $\mathcal{R}'$ and they are removed from N.

Hikage's algorithm 5 is explained in Table 3.6. First, the algorithm divides the graph of the precedence relation illustrated in Figure 3.3, which is constructed from the rule list in Table 3.6, into two components C_1 and C_2, as indicated in Figure 3.4. We explain Algorithm 6 using the component C_1. Two rules r_7 and r_{10} in C_1 exist with indegrees of 0, as shown in Figure 3.5. As w_{10} is less than w_7, the algorithm inserts r_{10} into N_1 and removes it from C_1. The graph in Figure 3.6 is obtained, and because only r_7 has an indegree of 0, the algorithm inserts r_7 into N_1 and removes it from C_1. Similarly, by determining the order of r_1, r_2, and r_6, we obtain the following order:

$$N_1 = [r_6, r_2, r_1, r_7, r_{10}].$$

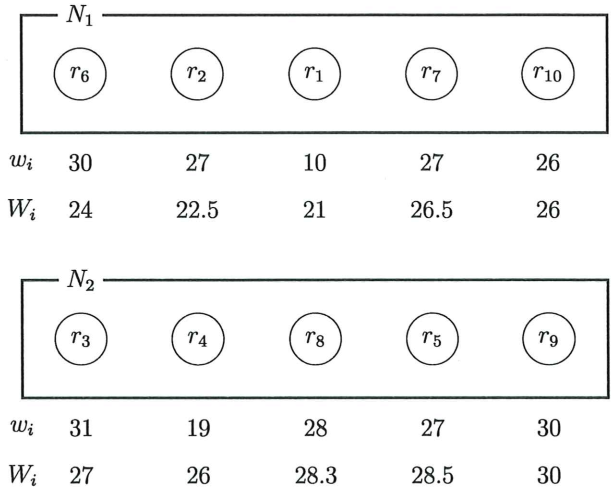


Figure 3.7: Compute W_i for each rule in N_1 and N_2.

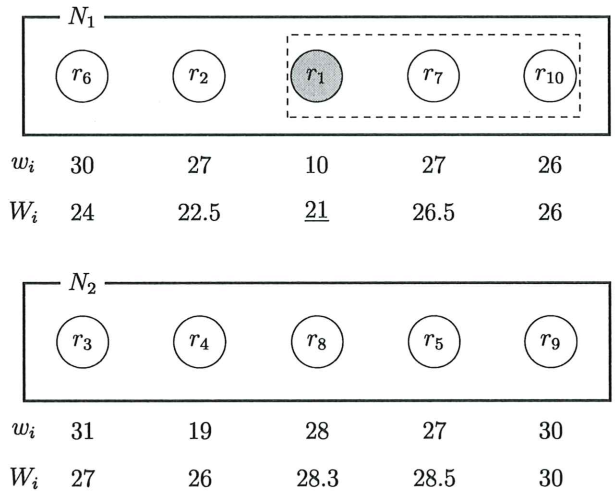


Figure 3.8: Insert rules from r_1 to r_{10} that is last of N_1 into $\mathcal{R}'$.

For C_2, we obtain $N_2 = [r_3, r_4, r_8, r_5, r_9]$.

Subsequently, Algorithm 5 computes W_i for each rule r_i belonging to the list N_i, as illustrated in Figure 3.7, where W_i is the ratio of the sum of the weights of rules from the tail of N_i to r_i, divided by the number of rules. For example, as r_1 in N_1 is the third rule from the tail of N_1 and r_7 exists between r_1 and r_{10} that is the last rule of N_1, $W_1 = (w_1 + w_7 + w_{10} = (10+27+26)/3 = 21$.

The algorithm determines the order of the rules with W_i computed, as explained above. Because $W_1 = 21$ is the smallest in N_1 and N_2, r_1 and the rules from r_1 to the tail rule r_7 are inserted into the sorted list $\mathcal{R}'$, as illustrated in Figure 3.8. Thereafter, the algorithm removes r_1, r_7, and r_{10} from N_1 and updates the values of W_6 and W_2 to $(30 + 27)/2 = 28.5$ and 27, respectively. Subsequently, as $W_4 = (19 + 28 + 27 + 30)/4 = 26$ is the smallest, the algorithm

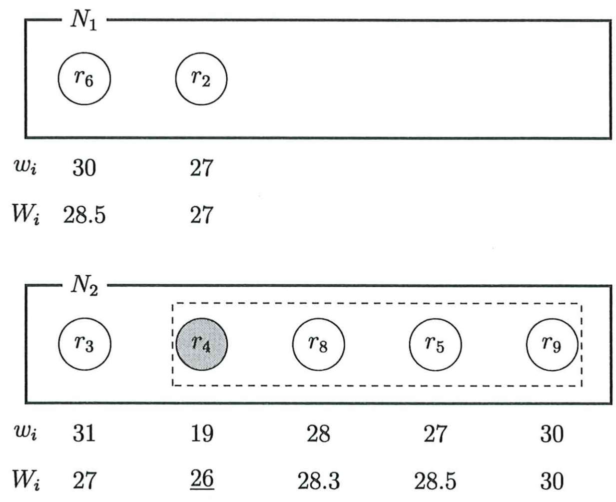


Figure 3.9: Insert rules from r_4 to r_9 that is last of N_2 into $\mathcal{R}'$.

inserts rules r_4, r_8, r_5, and r_9 into $\mathcal{R}'$, as indicated in Figure 3.9. By repeating this process until N_1 and N_2 become empty, the following order is obtained:

$$\mathcal{R}' = [r_3, r_6, r_2, r_4, r_8, r_5, r_9, r_1, r_7, r_{10}].$$

3.3.1 Comparison considering weights of directly dependent rules

As per Algorithm 5, we propose a novel rule-reordering algorithm based on Hikage's method.

As Hikage's method determines the order of the rules based on the single weights in each component, it cannot order appropriately when a heavy rule depends on a light rule. By computing the set of rules that are reachable from each rule, the order of the rules can be determined accurately. However, there is currently no algorithm that computes these sets in $O(n^2)$. Thus, we propose a rule-reordering algorithm that uses the weights of the rules on which the rule is directly dependent instead of the average weights of the rules that are reachable from each rule. We present our method in Algorithms 7 and 8. The difference between Hikage's method and our method is the determination of the order of the rules in each component, as demonstrated in Algorithm 8. Therefore, we only explain this difference.

In lines 1 to 2 of Algorithm 8, for each rule r_i that belongs to the list N, the algorithm sums the weights of the rules on which r_i directly depends and r_i itself, and computes its mean. For example, for the component C_1 in Figure 3.10, as r_7 depends on r_1, r_2, and r_6, $w'_7 = (27 + 10 + 27 + 30)/4 = 23.5$.

In lines 3 to 5, our method determines the order of the rules in a component, as computed by Algorithm 6. For components C_1 and C_2 in Figure 3.10, we obtain the following rule order by applying Algorithm 8:

$$N_1 = [r_6, r_2, r_{10}, r_1, r_7], \quad N_2 = [r_3, r_5, r_4, r_8, r_9].$$

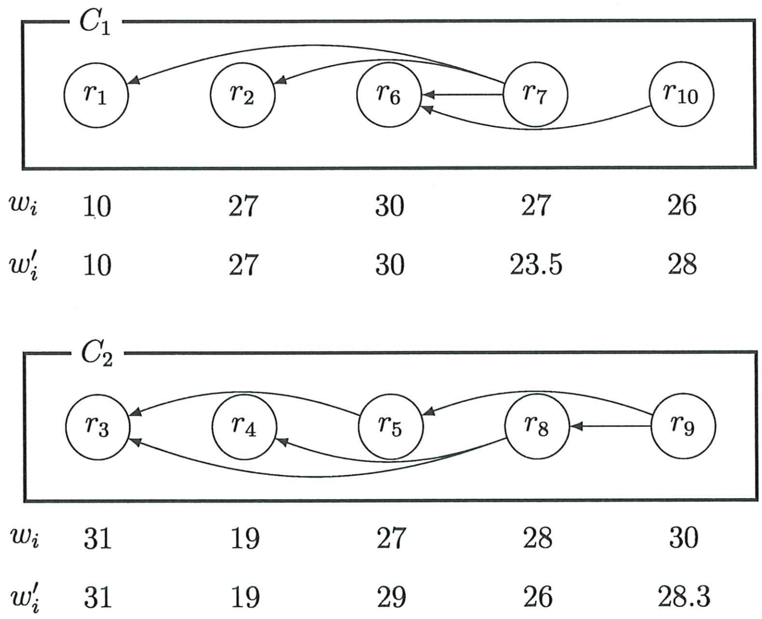


Figure 3.10: For each rule r_i, compute w'_i by adding children weights.

Thereafter, by determining the order of rules as per Algorithm 5, we obtain the following order:

$$\mathcal{R}' = [r_3, r_6, r_5, r_2, r_{10}, r_4, r_8, r_9, r_1, r_7].$$

The latency of the rule list reordered by Hikage's method is 1317, whereas that of our method is 1293.

3.4 A Reordering Method via Dependent Subgraph Enumeration

In general, if a rule that matches many packets is placed at the top of the rule list, many packets that match the rule will be evaluated with fewer comparisons, thus reducing latency. Therefore, it is desirable to place the heavy rules at the top of the rule list, but some rules can not be placed at the top due to dependencies. On the other hand, if several heavy rules depend on a rule, even if the weight of the rule is small, then placing the rule at the top of the list will help to place the heavier rules that depend on the rule at the higher positions in the list and help to reduce the latency. Therefore, we propose a reordering method that takes into account not only the rules that precede the rule but also the rules that depend on it when calculating the evaluation value.

Dependent subgraph enumeration is a reordering method based on the divide-and-conquer method, which enumerates rule sets that can be placed in an aligned list, partitions the rule list into the rule set with the highest average weight in the set and other rule sets, and recursively repeats partitions within each range. The rules that are dependent only on the candidate rules

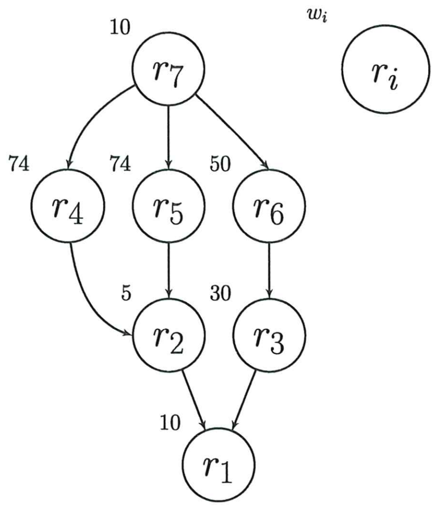


Figure 3.11: The dependent graph in Tabel 3.8.

are focused on in turn, and if the addition of the rule contributes to increasing the average weight of the candidate rules, it is added to the new rule set. Where if there is a rule that is dependent on a rule other than the candidate rule, it is not added to the rule set, because even if the candidate rule is placed in the list, it is not possible to place the focused rule into the sorted list. Based on this idea, dependent subgraphs that determine the range of rule selection are enumerated in order. Because enumerating the entire set of rules would be computationally exponentially expensive, the dependent subgraph enumeration method identifies and excludes rule sets whose addition to the rule set would reduce the average weight, and enumerates only those rule sets with large average weights.

3.4.1 Rule Set Enumeration and Comparison

The algorithm that divides the rule list and adds each rule to the sorted list is shown in Algorithm 9. Algorithm 9 divides the rule list into two lists, the list placed higher in the sorted list, $\mathcal{R}_{upper}$, and the list placed lower in the sorted list, $\mathcal{R}_{lower}$. This operation repeats the above operations until the number of rules in the divided list reaches one, and finally, each rule is placed in a sorted list. The algorithm that returns the set of rules that should be placed at the top of the rule list is shown in Algorithm 10. In Algorithm10, $G(r)$ is the set of rules reachable from rule r, $D(r)$ is the set of rules that are only dependent on r, and $T(r)$ is the set of all rules that have the maximum average weight when r is a candidate. The $S(r)$ is a set of rules constructed simultaneously in the process of finding $T(r)$ and summed with $\{r\}$ for all $S(u)$ such that the average of weight about $T(r) \cup S(u)$ is increasing for all $u \in D(r)$. Let $X(r)$ denote the sum of the weights of the rules in $S(r)$, $Z(r)$ denote the sum of the weights of the rules in $T(r)$, and LD denote the lists in which u are stored in order of increasing weight average of $S(u)$.

Algorithm10 constructs $G(r)$ and $D(r)$ for each rule r in line.1–4. Initialize $S(r)$ as $\{r\}$ and $T(r)$ as $G(r)$. Based on this, $S(r)$ and $T(r)$ are constructed in order from the lower rules.

The line.7 finds a list of $D(r)$ for each r, constructs a rule set $S(u)$ for each rule u that depends only on r, and sorts LD in descending order by the average value. line.8–9 extracts rule u from LD in order, and line.10 finds the union set of $T(r)$ and $S(u)$. In line.11, the average weight of the rule set that $S(u)$ is added to $T(r)$ is compared with the average weight of the rule set before the addition, and if the average weight after the addition is larger than the average weight before the addition, the set is set as the new $T(r)$, and $S(r)$ is also updated at the same time. In this way, all rules that are dependent only on r and that contribute to increasing the average weights are added to $T(r)$. When $T(r)$ has been constructed for all rules, search for the candidate rule r' that maximizes the $T(r)$ average weight in line.15–16. If r' is reachable from all the rules in the rule list, the search is repeated with the list excluding r' from the rule list with line.18. Finally, $T(r')$ is returned in the list structure.

Consider the case where the rule list in Table 3.8 is reordered using the proposed method. First, Algorithm 10 constructs $G(r), D(r), T(r), S(r)$ for each rule r in the rule list, starting from the bottom. In line.7, the rule set of average weight is constructed with r_7 as a candidate, however, since $D(r_7)$ is an empty set, $S(r_7)$ and $T(r_7)$ are not updated and remain at their initial values. r_6, r_5, r_4 are also the same. Then, the algorithm constructs $D(r_3)$ with r_3 as a candidate. Since r_6 is dependent on r_3 and r_6 is not dependent on any other rule, add r_6 to $D(r_3)$. Now compare the average weights of $T(r_3)$ and $T(r_3) \cup S(r_6)$. Since the average weight of $T(r_3)$ is $\frac{10+30}{2} = 20$ and the average weight of $T(r_3) \cup S(r_6)$ is $\frac{10+30+50}{3} = 30$, the rules included in $S(r_6)$ are added to $T(r_3)$ and $S(r_3)$. Then, the algorithm constructs $D(r_2)$ with r_2 as a candidate. Since r_2 is dependent on r_4 and r_5, and each rule is not dependent on any rule except r_2, r_4 and r_5 are added to $D(r_2)$. The algorithm then compares the average weights of $S(r_4)$ and $S(r_5)$ added to each rule. Since the average weights of $S(r_4)$ and $S(r_5)$ are the same, the comparison is performed starting from $S(r_4)$ with the smallest rule number. The average weight of $T(r_2)$ is $\frac{10+5}{2} = 7.5$ and that of $T(r_2) \cup S(r_4)$ is $\frac{10+5+74}{3} = 29.66$, so $S(r_4)$ is added to $T(r_2)$ and $S(r_2)$. The same comparison is made with $S(r_5)$, and $S(r_5)$ is added to $T(r_2)$ and $S(r_2)$. These steps are repeated to construct the rule set in Table 3.12. Since $T(r)$ has the highest average weight among the $T(r)$ that can be placed in the Figure 3.11, $T(r_1)$ is listed and returned to Algorithm9. Algorithm9 takes the list received from Algorithm10 as $\mathcal{R}_{upper}$ and returns the rule set excluding $\mathcal{R}_{upper}$ from the rule list as $\mathcal{R}_{upper}$. Let $\mathcal{R}_{lower}$ be the rule set excluding $\mathcal{R}_{upper}$ from the rule list, and recursively apply Algorithm9 to each list.

Repeat this operation until the number of rules in the list returned by Algorithm10 is 1, and then add each rule to the sorted list to obtain the order

$$\sigma = [1, 2, 5, 3, 4, 6, 7].$$

3.4.2 Time Complexity for the Proposed Method

In this section, we show the time complexity of the proposed method. For each rule r, in the rule u included in $D(r)$, $Z(r)/|T(r)| < Z'(r)/|T'(r)|$ is determined in line. 11. The number of comparisons for line. 11 is at most n, so the time complexity is $O(n)$. Where $T'(r) =$

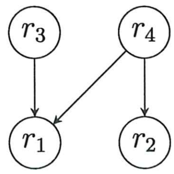


Figure 3.12: The dependency graph where the mean of weights does not work.

$T(r) \cup S(u)$, and $Z'(r)$ is the sum of the weights of the rules included in $T'(r)$. After $T(r)$ has been constructed, search for the candidate rule with the largest average weight in line. 15. The number of comparisons for line. 15 is at most n, so the time complexity is $O(n)$. This operation is repeated until the number of rules in the list that Algorithm 9 receives from Algorithm 10 reaches 1. The number of calls of line. 3 in Algorithm 9 is n at most, so it is $O(n)$. Therefore, the time complexity of the dependent subgraph enumeration method is $O(n^3)$.

3.5 A Reordering Method via Difference of Latency

Many heuristics to the optimal rule ordering problem reorder the rules by following the dependent graph and using the average weight of the reachable rules [11, 13, 44, 45]. However, there are rule lists that cannot be reordered using average weights to sufficiently reduce the latency. For example, in the rule list with the dependent relations in Figure 3.12, the following five orderings are possible, and the order of rules with smaller latency changes depending on the weights of r_1, r_2, r_3, r_4. Where the default rule is omitted because it is placed at the bottom of the list and does not affect latency difference.

(i) r_1, r_2, r_3, r_4
(ii) r_1, r_2, r_4, r_3
(iii) r_1, r_3, r_2, r_4
(iv) r_2, r_1, r_3, r_4
(v) r_2, r_1, r_4, r_3

If the weights of the sink rules r_1 and r_2 are larger than those of the other rules r_3 and r_4, the rule with the larger weights is added first to the top of the sorted list to get an ordering of rules with smaller latency. However, when R_1 and R_2 have smaller weights than R_3 and R_4, simple reordering by weight or average of weights may not result in an ordering of rules with small latency.

In reordering rules, most heuristics that greatly reduce latency use the average of the weight of rule r_i and the weights of the rules that r_i is dependent on as the evaluation value $\mathcal{E}(r_i)$, and find a ordering of rules with smaller latency by placing the rule with the larger weight higher in the order of the rules [11, 44]. For example, in the rule list with the dependent relations in Figure 3.12, the evaluation value of r_4 is $\mathcal{E}(r_4) = \frac{|r_1|+|r_2|+|r_4|}{3}$ because it is the average of the weights of r_4 and the rules to which r_4 is dependent, and the evaluation value of r_3 is

$\mathcal{E}(r_4) > \mathcal{E}(r_3)$. The previous heuristic places r_4 higher than r_3 if $\mathcal{E}(r_4) > \mathcal{E}(r_3)$. The condition for placing r_4 higher than r_3 is as follows.

$$\begin{aligned} &\mathcal{E}(r_4) > \mathcal{E}(r_3) \\ \Leftrightarrow\ &\frac{|r_1| + |r_2| + |r_4|}{3} > \frac{|r_1| + |r_3|}{2} \\ \Leftrightarrow\ &\frac{2|r_2| + 2|r_4| - |r_1|}{3} > |r_3| \end{aligned} \tag{3.4}$$

Now consider the difference in latency of each ordering of (i)$\sim$(v). The order of (ii) and (iii) is a ordering that places r_1 higher than r_2, and the difference in latency between the respective lists $\mathcal{R}_{\text{(ii)}}$ and $\mathcal{R}_{\text{(iii)}}$ is as follows.

$$\begin{aligned} &L(\mathcal{R}_{\text{(ii)}}, \mathcal{F}) - L(\mathcal{R}_{\text{(iii)}}, \mathcal{F}) \\ =\ &|r_1| + 2|r_2| + 3|r_4| + 4|r_3| - (|r_1| + 2|r_3| + 3|r_2| + 4|r_4|) \\ =\ &2|r_3| - (|r_2| + |r_4|) \end{aligned} \tag{3.5}$$

If $|r_3| > \frac{|r_2|+|r_4|}{2}$ in the equation (3.5), then the order of (iii) has smaller latency than (ii). Thus, in the reordering method using average weights, one of the conditions for the weights that do not reduce the latency of the list is as follows.

$$\frac{2|r_2| + 2|r_4| - |r_1|}{3} > |r_3| > \frac{|r_2| + |r_4|}{2} \tag{3.6}$$

When the weight of r_3 is in this range, the reordering method using average weights will select the order (ii) even though the order (iii) has a smaller latency, thus leading to an incorrect selection. By comparing the latency, a better ordering can be found if such an ordering can be eliminated.

In the next section, we generalize this idea so that it can be applied to the comparison of sub lists in a rule list. We propose a method to reduce the latency by replacing the sub lists based on a decision using the difference in latency.

3.5.1 Proposed Method

To generalize the idea of the previous section, consider the difference of latency when the L_1 and L_2 sub lists from the ith to the jth and from $j+1$ to k of the rule list are swapped. Where the rules in L_2 do not depend on any of the rules in L_1. By considering such, the front-back relationship between L_1 and L_2 is not affected by the policy. The latency $L(\mathcal{R}, \mathcal{F})$ of the rule list $\mathcal{R}$ in the ordering before replacing the partial list is as follows. where $L(i)$ is the i-th rule in the list L and $|L|$ is the number of rules in the list L.

$$
\begin{aligned}
L(\mathcal{R},\mathcal{F}) = &\sum_{u=1}^{i-1} u|\mathcal{R}(u)| + \sum_{u=1}^{|L_1|}(i-1+u)|L_1(u)| \\
&+ \sum_{u=1}^{|L_2|}(i-1+|L_1|+u)|L_2(u)| \\
&+ \sum_{u=k+1}^{n-1} u|\mathcal{R}(u)| + (n-1)|\mathcal{R}(n)|
\end{aligned}
\tag{3.7}
$$

Equation (3.7) is an expression that divides the parts of the partial lists L_1 and L_2 in formula (2.3). Each term is the sum of the number of matches for the first rule through the $i-1$th rule, the rules in L_1, the rules in L_2, the rules placed lower than L_2, and the default rule. On the other hand, the latency $L(\mathcal{R}',\mathcal{F})$ of the rule list $\mathcal{R}'$ with L_1 and L_2 swapped is as follows.

$$
\begin{aligned}
L(\mathcal{R}',\mathcal{F}) = &\sum_{u=1}^{i-1} u|\mathcal{R}(u)| + \sum_{u=1}^{|L_2|}(i-1+u)|L_2(u)| \\
&+ \sum_{u=1}^{|L_1|}(i-1+|L_2|+u)|L_1(u)| \\
&+ \sum_{u=k+1}^{n-1} u|\mathcal{R}(u)| + (n-1)|\mathcal{R}(n)|
\end{aligned}
\tag{3.8}
$$

The difference between these latencies is as follows.

$$
\begin{aligned}
&L(\mathcal{R},\mathcal{F}) - L(\mathcal{R}',\mathcal{F}) \\
&= \sum_{u=1}^{|L_1|}(i-1+u)|L_1(u)| \\
&\quad + \sum_{u=1}^{|L_2|}(i-1+|L_1|+u)|L_2(u)| \\
&\quad - \sum_{u=1}^{|L_2|}(i-1+u)|L_2(u)| \\
&\quad - \sum_{u=1}^{|L_1|}(i-1+|L_2|+u)|L_1(u)| \\
&= |L_1|\sum_{u=1}^{|L_2|}|L_2(u)| - |L_2|\sum_{u=1}^{|L_1|}|L_1(u)|
\end{aligned}
\tag{3.9}
$$

If the (3.9) is greater than 0, then $\mathcal{R}'$ has a smaller latency than $\mathcal{R}$. We propose a method to search for a partial list that should be placed at the top using this decision.

Let $D(L_1, L_2)$ be a formula to decide whether the latency would be smaller if L_1 and L_2 were swapped in adjacent partial lists L_1 and L_2 in the rule list, and define it as follows.

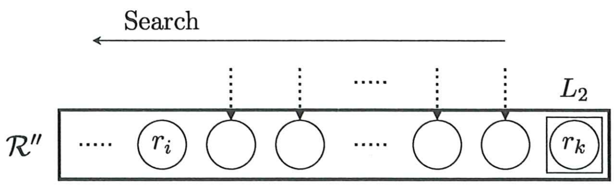


Figure 3.13: Search a dependent rule with r_k in the provisional list.

Definition 3.5.1.

$$D(L_1, L_2) = |L_1| \sum_{k=1}^{|L_2|} |L_2(k)| - |L_2| \sum_{k=1}^{|L_1|} |L_1(k)|$$

Note that the (3.5) is an example of the definition 3.5.1. By letting L_1 be r_2, r_4 and L_2 be r_3, $D(L_1, L_2) = 2|r_3| - (|r_2| + |r_4|)$, which is the same as (3.5). If $D(L_1, L_2)$ is greater than 0, $\mathcal{R}_{(iii)}$, which is a ordering with L_2 placed higher, has a smaller latency.

In the proposed method, in advance, there is an ordering that is based on the average weight of the rules that are expected to reduce the latency. From this ordering, we obtain a partial list L_2 of rules that are candidates for placement at upper positions and determine whether placing them at upper positions reduces the latency by the difference $D(L_1, L_2)$ using the list L_1 of rules that currently exist at the upper positions.

To make a decision using Definition 3.5.1, it is important that none of the rules in L_2 are subordinate to L_1. We describe below how to obtain such a partial list L_2 from $\mathcal{R}'$.

The details of the algorithm are as follows. First, rules are ordered from the top of the rule list $\mathcal{R}'$ sorted by the existing method, and are added to the preliminary list $\mathcal{R}''$ in order. If the rule r_k to be added is not a sink rule, it is added to L_2 as a candidate to be placed upper, and the partial list L_2 is compared with the partial list of $\mathcal{R}'$ to find an ordering with lower latency. As shown in Figure3.13, the rules that r_k depends on are searched in order from the lower rules in the preliminary list. $D(L_1, L_2)$ is used to decide whether L_1 or L_2 should be placed at the upper position.

If $D(L_1, L_2)$ is less than 0, then placing L_2 lower in the list, as shown in the upper part of Figure 3.15, results in lower latency. If $D(L_1, L_2)$ is greater than 0, then L_2 including r_k should be placed next to r_i as shown in the bottom part of Figure 3.15. If $D(L_1, L_2)$ is greater than 0, r_i is added to the top of the list L_2, and the proposed method adds the rules to which r_i depends on *PrecedingSet*. The *PrecedingSet* is the set of rules that must be placed before L_2 to place it on the upper position. This operation is repeated until $D(L_1, L_2)$ is less than 0 or until the top of the list is reached to decide where L_2 should be placed. When $D(L_1, L_2) = 0$, it means that the latency is the same for both orders of $D(L_1, L_2)$. In this case, there may be an order with lower latency that places L_2 higher, and thus the search is restarted with L_2 as the order that places L_2 higher. This operation is repeated for all rules to reorder the rules.

The entire algorithm is shown in Algorithm 11. In Algorithm 11, first, it determines whether the rule r_k in focus is a sink rule. If it is a sink rule, it is added to the preliminary list $\mathcal{R}'$ by line. 1. If it is not a sink rule, use Algorithm 12 to search for a rule that should be placed upper

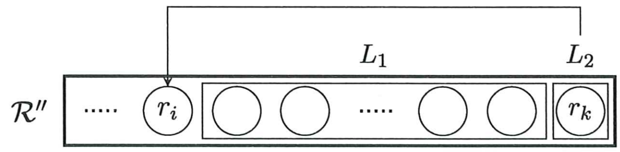


Figure 3.14: Reaching a rule that is dependent on r_k.

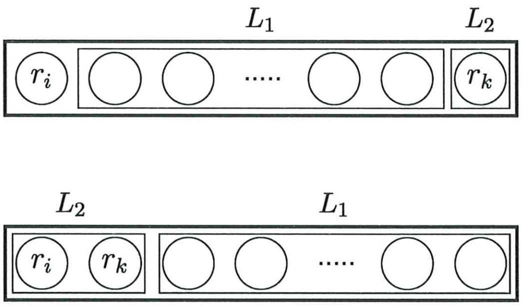


Figure 3.15: The list that places r_k at the end of the list and the list that places r_k at the next to r_i.

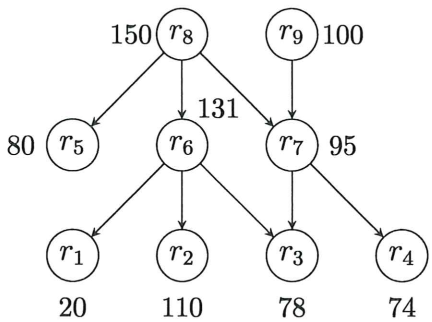


Figure 3.16: Dependency graph of Table .

and reorder the rules. In line. 1, r_k is added to the list L_2 and the position where it should be added to $\mathcal{R}'''$ is decided. Searching for rules that have a dependency relationship with r_k from the subordinate rules of the tentative list by line. 7 to the line. 14, if focused rule r_i depends on r_k, then it searches an ordering with lower latency using the decision formula $D(L_1, L_2)$. If $D(L_1, L_2) \geq 0$, then placing L_2 next to r_i will reduce the latency to less than that of the original ordering, so r_i is added to the top of L_2 at line. 9 and the rules that r_i depends on are added to *PrecedingSet*. Also, at the line. 11, the rules that should be placed lower are added than L_2 to *lowerlist*. By repeating this operation until $D(L_1, L_2) < 0$ or until reaching the first rule, the position of L_2 including r_k is decided and $\mathcal{R}'''$ is returned to Algorithm 11. This operation is performed for all the rules.

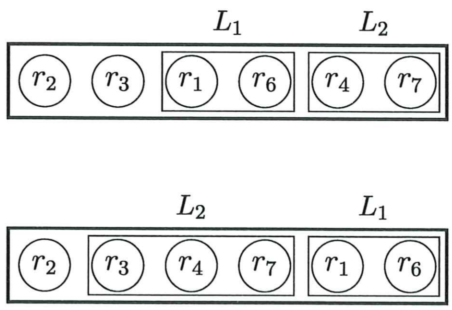


Figure 3.17: The Order by weight sorting in the list and L_2 is upper in the list.

3.5.2 Execution example

We consider reordering a rule list in Table 3.13 with the dependency in Figure 3.16 using the proposed method. First, the rules are reordered using previous heuristics. In this section, we use an improved version of Hikage's method proposed in 3.3. The rule list of Table 3.13 with the dependencies of Figure 3.16 is reordered using the improved Hikage's method to produce the order in Table 3.15.

Then, search for rules that depend on the upper rules, beginning with the first rule. Since r_2, r_3, r_1 are sink rules, they are added to the preliminary list. Since r_6 is not a sink rule, it is searched in order from the lower rules. Since r_6 depends on all the rules in the preliminary list, all the rules in the preliminary list are added to L_2, and r_6 is placed at the end of the list. As a result, L_1 becomes empty. Since $D(L_1, L_2) > 0$, we assume the ordering with L_2 at the top is used as the preliminary list. Since r_4 is also a sink rule, it is added to the preliminary list. r_7 is not a sink rule and does not depend on all the rules in the preliminary list, so r_7 is added to L_2. Then, r_4 and r_3 are added to *PrecedingSet*. Since r_4 is included in *PrecedingSet* but is the lowest in the preliminary list, r_4 is added to the top of L_2 and the search is restarted. As r_3 is included in the *PrecedingSet*, and the lower position of the ordering in Figure 3.17 may reduce the latency, a decision is made using the definition 3.5.1. In this case, $D(L_1, L_2)$ is as follows.

$$D(L_1, L_2) = 2(|r_4| + |r_7|) - 2(|r_1| + |r_6|)$$
$$= 2(74 + 95) - 2(20 + 131) \geq 0 \tag{3.10}$$

This shows that placing L_2 next to r_3 reduces the latency. Therefore, r_3 is added at the beginning of L_2 and the search is restarted. Since r_2 is not included in *PrecedingSet*, the search reaches the top of the preliminary list. Then, a decision is made whether L_2 should be placed at the top or not. In this case, $D(L_1, L_2)$ is as follows.

$$D(L_1, L_2) = |r_3| + |r_4| + |r_7| - 3|r_2|$$
$$= 78 + 74 + 95 - 360 < 0 \tag{3.11}$$

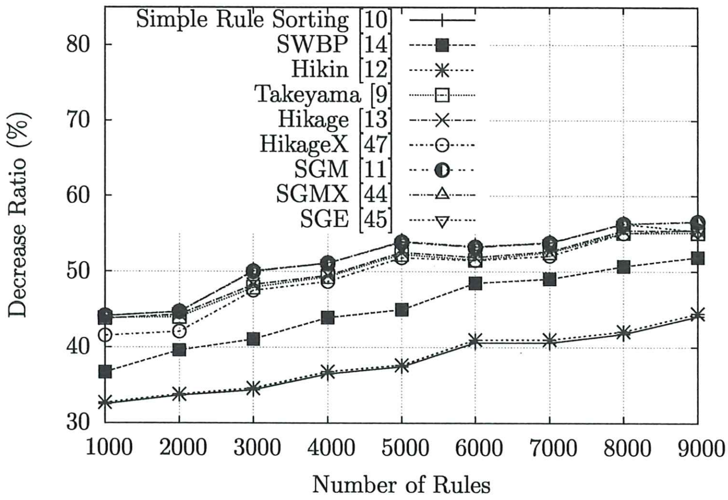


Figure 3.18: The latency of ACL.

As a result, because L_2 should be placed lower than r_2 to minimize the latency, the position where L_2 should be placed is decided. Repeating the operation in the same way results in the ordering shown in Table 3.16. As shown in Table3.16, the latency of the rule list is reduced by reordering using the proposed method.

3.6 Experiments

We demonstrate the effectiveness of the proposed algorithm and reordering algorithms through computer experiments.

The proposed methods were implemented in Java under Ubuntu 22.04.3 LTS on Intel Core i7-8700 with 8 GB of main memory. We used ClassBench which is known as the benchmark tool for the packet classification algorithm. It generates a rule list and a header list based on data obtained from an actual environment. So, ClassBench can build an experimental environment closer to the real environment. We generated 270 rule sets of 1,000 to 9,000 rules using ClassBench [46] with the seed file of the Access Control List (ACL). The evaluation type P or D was added to each rule in the rule list, each with a probability of 1/2. There were 100,000 headers for each rule list. We implemented the methods of simple rule sorting [10], swapping window-based paradigm [14], Takeyama et al. [9], Hikage et al. [13], and three proposed methods that are the improved SGM (SGMX), the improved Hikage's method (HikageX) and the method via dependent subgraph enumeration (SGE). The decrease ratio and reordering times were measured. The averages of 30 trials are depicted in Figs. 3.18 and 3.21.

For clarity, each result is divided into $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ methods, which are shown in Figure 3.18, Figure 3.19, Figure 3.20, Figure 3.21, Figure 3.22 and Figure 3.23 respectively.

As shown in Figure 3.18, the $\mathcal{O}(n^3)$ methods have higher latency reduction than the $\mathcal{O}(n^2)$ methods, and thus find an order of rules with lower latency. Also, as shown in Figure 3.19 and

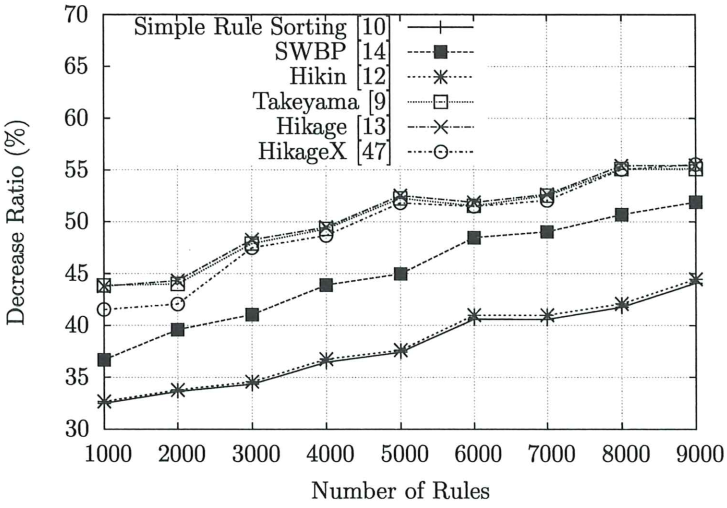


Figure 3.19: The latency of ACL($\mathcal{O}(n^2)$ methods).

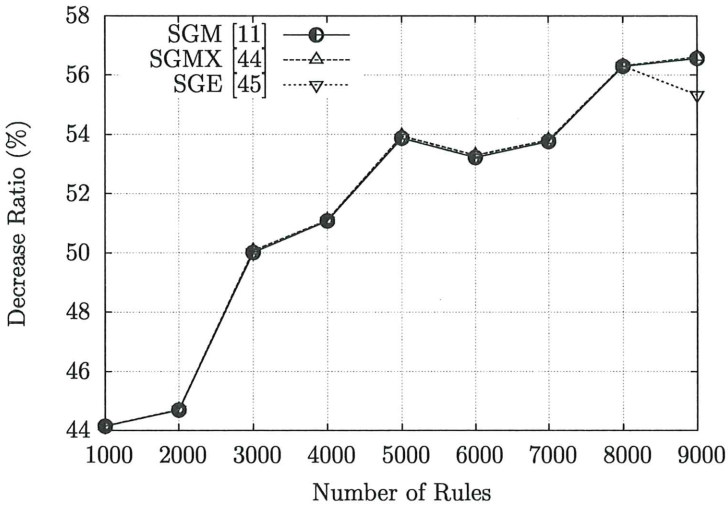


Figure 3.20: The latency of ACL($\mathcal{O}(n^3)$ methods).

Figure 3.20, the proposed method reduces the latency on average, and as the number of rules increases, it reduces the latency more than the previous methods.

As shown in Figure 3.21, the method of $\mathcal{O}(n^2)$ reorders the rules faster than that of $\mathcal{O}(n^3)$ in many cases.

As shown in Figure 3.21, SGMX reorders the rules faster than SGM in many cases. Also, as shown in Figure 3.22, HikageX reorders rules faster than many $\mathcal{O}(n^2)$ methods when the number of rules increases.

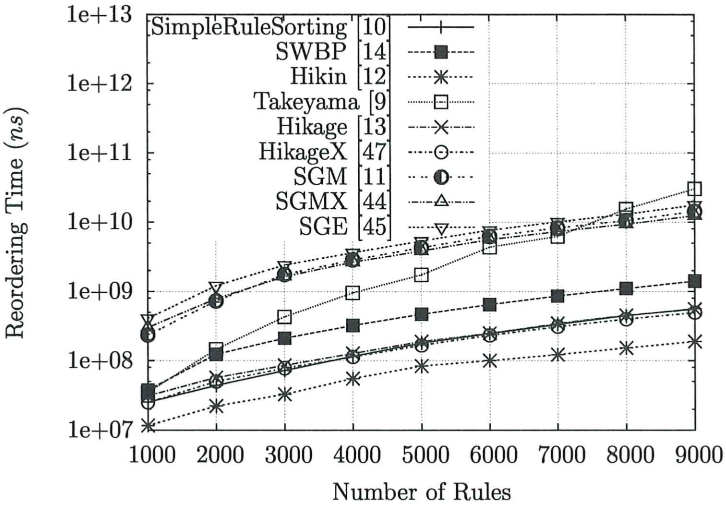


Figure 3.21: The reordering time for Table 3.18.

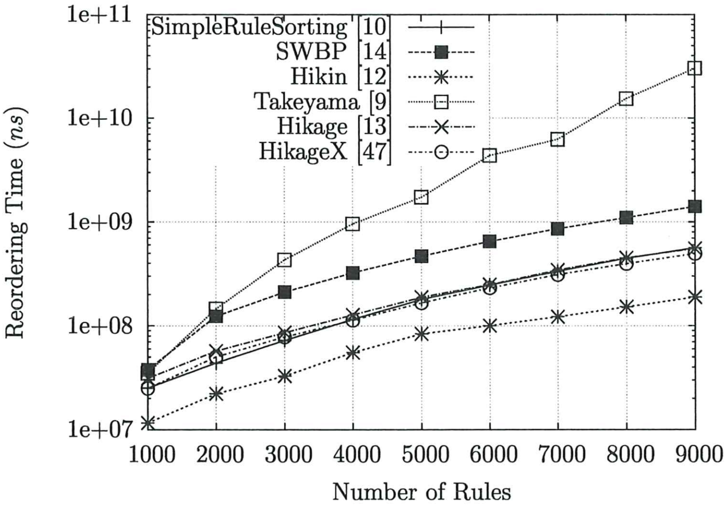


Figure 3.22: The reordering time of ACL($\mathcal{O}(n^2)$ methods).

3.7 Auxiliary Methods

Most heuristic methods for optimal rule ordering according to precedence constraints are based on overlap and dependency relations. However, there exist orders of rules that hold policies without these precedence constraints, and such orders may have lower latency. When all matchable packets match the rule placed higher in the list, the number of packets that match the rule is zero, so placing the rule lower than the default rule does not violate the policy. Also, such a rule does not have any matching packets, but the packets are compared, so the latency will not be sufficiently reduced. Therefore, we propose an auxiliary method to find an order of rules with lower latency by searching for rules with no matching packets and placing them lower than

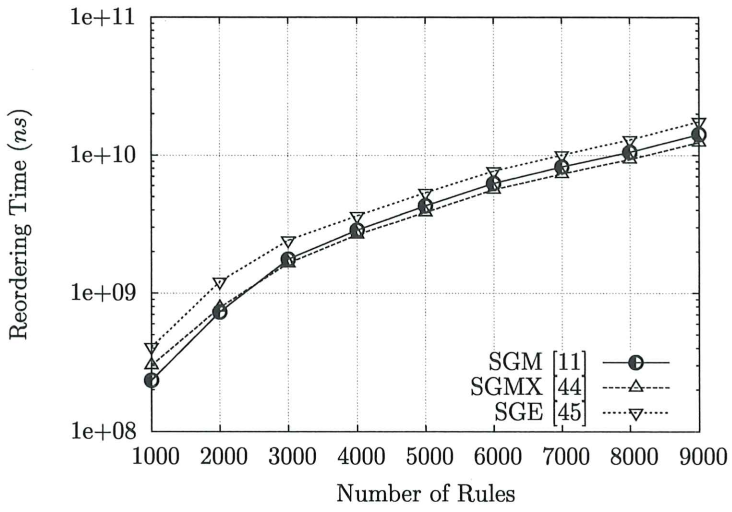


Figure 3.23: The reordering time of ACL($\mathcal{O}(n^3)$ methods).

the default rule.

For overlapping rules r_i and r_j, the set of packets $M(r_i) \cap M(r_j)$ that can match both rules is called the common part of those rules. The set of packets to which the applicable action may change when the dependent rules are interchanged is the common part of those rules. Therefore, when all packets in the common part match the rule placed above, there is no policy violation even if the precedence relation by the dependent relation in focus does not hold. Therefore, we propose an auxiliary method to find a lower latency order of rules by removing the precedence relations that do not affect the policy.

3.7.1 A Reordering Method via Deleting 0 Weights Rules

Reordering rules or adding rules due to policy changes may result in a rule that does not match a packet. If all matchable packets match the rule placed above, there is no packet that matches the rule.

In the case of Table 3.19, the matchable packet set for r_4 is $M(r_4) = \{1100, 1101, 1110, 1111\}$. r_4 depends on r_1, r_2, r_3, and the set of matchable packets for these rules is $M(r_1) = \{1001, 1011, 1101, 1111\}, M(r_2) = \{1000, 1110\}, M(r_3) = \{0110, 0111, 1110, 1111\}$, respectively. This shows that $E(\mathcal{R}, 4) = \emptyset$ since all packets in $M(r_4)$ are included in either $M(r_1), M(r_2)$, or $M(r_3)$, and the number of matching packets with r_4 is zero. Such a rule does not match any packet in any order that holds the policy, but packets that match a rule lower than this rule are also compared with this rule, thus increasing the number of comparisons. Placing such a rule lower than the default rule prevents comparison between the packet and the rule. We call this action the deletion of a rule in rule reordering.

In addition, rules dependent on rule r_i that are removed according to the conditions described above do not violate the policy even if they do not hold precedence constraints due to their dependencies with r_i. This makes it possible to place rules with higher weights that are dependent on r_i at higher positions, and thus reduce the latency. In the case of Table 3.19, removing r_4 reduces the latency in the rule list to $L(\mathcal{R}, \mathcal{F}) = 580$. Also, r_5 and r_6 depend on r_4 and could not be placed higher than r_4 if the rules are reordered according to the dependency relation, but by deleting r_4, the precedence constraint by the dependency relation is removed and they can be placed higher. Thus, the latency is reduced to $L(\mathcal{R}_\sigma, \mathcal{F}) = 345$ when the rules are reordered into the order $\sigma = \{6, 5, 3, 1, 2, 7, 4\}$.

In this section, we propose a method to decide whether the number of packets that match the rule r_i is zero or not. Since the problem of finding packets that match r_i is $\sharp\mathcal{P}$-complete, we propose a method to search for rules with no packets matching r_i using the SAT solver. The conditions of the rules from the top of the rule list to r_i are expressed in Conjunctive Normal Form (CNF), and the solver determines whether there exists a packet that satisfies the following logical equation. If there is no packet with an assignment that is true in the propositional logic formula for $E(\mathcal{R}, r_i) = \emptyset$, we know that $E(\mathcal{R}, r_i) = \emptyset$. Where in the logical variables, false means that the packet does not match the corresponding rule, and true means that it does.

$$\neg r_1 \land \neg r_2 \land \neg r_3 \cdots \land \neg r_{i-1} \land r_i \tag{3.12}$$

For each bit of the rule, a propositional variable with the corresponding bit number if it is 1, or its negation if it is 0, is combined by logical OR to form a clause for each rule. For example, in the Table 3.19, the condition of r_1 is $b_1 \land b_4$ and the condition of r_2 is $b_1 \land \neg b_3 \land \neg b_4$. For the formula (3.12), negation is added to the clauses corresponding to rules from the top of the rule list to r_{i-1}, and the logical product is combined with the clause of r_i to form a logical formula that determines whether the packet matches r_i or not. The logical formula to determine whether or not a packet matching r_4 exists in the rule list in Table 3.19 is as follows.

$$\begin{aligned}
&\neg r_1 \land \neg r_2 \land \neg r_3 \land r_4 \\
&\cong \neg(b_1 \land b_4) \land \neg(b_1 \land \neg b_3 \land \neg b_4) \land \neg(b_2 \land b_3) \land (b_1 \land b_2) \\
&\cong (\neg b_1 \lor \neg b_4) \land (\neg b_1 \lor b_3 \lor b_4) \land (\neg b_2 \lor \neg b_3) \land b_1 \land b_2
\end{aligned} \tag{3.13}$$

When the logical formula (3.13) is determined whether it is satisfiable using MiniSat, the result is UNSAT. Thus, there is no assignment that satisfies the formula (3.13), and the number of packets matching r_4 is zero. This determination is performed for all rules, and the rule whose weight is determined to be 0 is removed from the rule list rule list.

3.7.2 A Rule Reordering Method via Deleting Pre-Constraints that do not Affect Policies

Most of the heuristic methods to solve rule-order optimization problems reorder the rules according to the precedence constraints by the dependency relation. Therefore, we define a set of

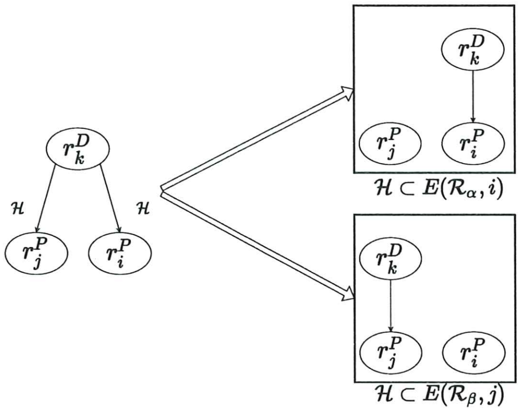


Figure 3.24: Dependent graph of r_i^A, r_j^A, r_k^D(left) and actual restriction(right).

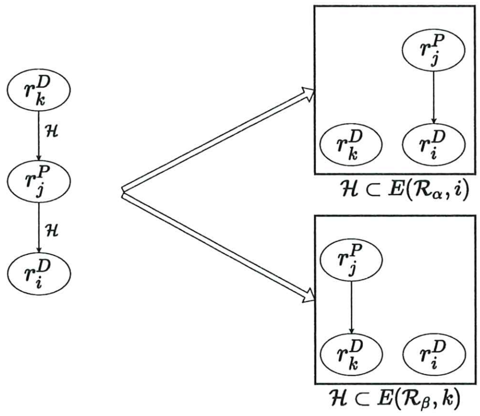


Figure 3.25: Dependent graph of r_i^A, r_j^A, r_k^D(left) and actual restriction(right).

rules such as to encompass the common part $M(r_j) \cap M(r_i)$ corresponding to the prior constraint (j, i).

Definition 3.7.1. (covering)
For a rule set I, C is said to cover I or I is said to be covered by C if the common part $\bigcap_{r \in I} M(r)$ of the set of packets matching the rules belonging to I is contained in the union $M(\mathcal{C}) = \bigcup_{r \in C} M(r)$ of the packets matching the rules belonging to rule set C.

When a single set $C = \{r\}$ covers I, r is called a covering rule of I. For a precedence constraint (j, i), if there exists a set of rules C covering $\{r_j, r_i\}$, we call C a covering rule of (j, i). Also, r is called a covering (j, i) if (j, i) is covered by a covering rule r.

Consider three rules r_i, r_j, r_k with the same common part $\mathcal{H} = M(r_i) \cap M(r_j) = M(r_i) \cap M(r_k) = M(r_j) \cap M(r_k)$. Where the order of these rules before reordering is r_i, r_j, r_k. For example, when the actions of these rules are P, P, D, the dependency relations are on the left side of Figure 3.24. Since these rules are overlapped by a common part $\mathcal{H}$, if either r_i or r_j is placed first, the rule set consisting of only the lower two rules is covered by the rule placed first. Thus, the precedence constraints between the remaining rules can be removed, and the case can actually be divided into precedence constraints such as the one on the right side of Figure 3.24. For example, if we know that r_i is placed above r_k, we know that (k, j) does not affect the policy because (k, j) is covered by r_i, and we can remove (k, j).

Then if the actions of these rules are D, P, D, the dependent relation is on the left side of Figure 3.25. However, since these rules are overlapped in the same packet $\mathcal{H}$, if r_k or r_i is placed on the top, as shown on the right side of Figure 3.25, the rule set consists of only the lower two rules The rule set consisting of only the lower two rules is covered by the previously placed rule. Thus, there exists a sequence of rules that preserves the policy even if it violates the precedence constraint by the dependency relation.

Reordering the rule list in Table 3.20 using the improved version of SGM proposed in 3.2.2 results in the ordering in Table 3.21. Since r_2 covers $(7, 4)$, $(7, 4)$ does not affect the policy when r_2 is placed on higher positions. Reordering the rules with this in view, an ordering of the Table 3.22 can be obtained, which has a lower latency. As shown above, there are cases in which an ordering with lower latency exists that violates the precedence constraint by the dependency relation but still holds the policy. However, methods such as SGM follow the constraints based on the dependency relation, so such a sequence cannot be obtained in principle. Thus, we propose the reordering method that relaxes the precedence constraints by searching for and eliminating the precedence constraints that are covered by the upper-level rules.

The proposed method is based on reordering methods such as SGM, which builds up a sorted list starting from the rule on the top. The proposed method considers this sorted list as the upper-level rule and finds the set of rules covered by it, thereby relaxing the precedence constraint. In the following, we propose two methods for determining coverages, one using a SAT solver and the other using covering rules.

Search for removable precedence constraints

The determination of whether the precedence constraint (j, i) is covered can be converted into a determination of whether there exist packets that do not match the rule in the upper level in the packets that are in the common part $M(r_j) \cap M(r_i)$. This problem corresponds to the determination problem of the satisfiability of a logical formula consisting of the rules placed at the upper level and r_i, r_j. The proposed method determines whether all precedence constraints are covered, and if so, it removes the precedence constraints. This process is repeated each time the sorted list is updated, thereby relaxing the precedence constraints in the reordering process.

When the improved SGM reorders the rules, the algorithm for relaxing the prior constraints using the SAT solver is shown in Algorithm 15. Algorithm 15 first constructs the precedence constraints by dependency relations as an adjacency list $\mathcal{A}$, the same as the improved version of SGM proposed in 3.2.2. The rules to be placed in the sorted list are selected in line 2, added to the sorted list in line 3, and removed from the rule list in line 4. Then, the updated sorted list is used in line 5 to remove precedence constraints that do not affect the policy. This process is repeated until the rule list $\mathcal{R}$ is empty.

The algorithm for removing precedence constraints covered by rules placed at the upper level for precedence constraints in the adjacency list $\mathcal{A}$ is shown in Algorithm 13. In Algorithm 13, at line 2–5, all precedence constraints in $\mathcal{A}$ are determined whether they are covered by the SAT solver, and if they are covered, the corresponding dependent relation is removed from $\mathcal{A}$. Line 3 determines whether (j, i) is covered by the rule placed at the top of (j, i) in line 3. The logical equation used to determine this is constructed as follows.

Construction of decision formula

At first, the rules that can be matched to the common part $M(r_j) \cap M(r_i)$ of the precedence constraint (j, i) in the sorted list are searched and placed in the list L. At first, the rules that can be matched to the common part $M(r_j) \cap M(r_i)$ of the precedence constraint (j, i) in the sorted list are searched and placed in the list L. Whether (j, i) is covered or not corresponds to the existence of packets belonging to the common part $M(r_j) \cap M(r_i)$ that do not match the rules placed in the list L, thus the following logical formula is generated. Where $L(i)$ is the i-th rule in the list L.

$$f((j,i), L) = \neg L(1) \wedge \neg L(2) \wedge \neg L(3) \cdots \wedge \neg L(h) \wedge r_i \wedge r_j \tag{3.14}$$

If there is no packet corresponding to the assignment that would be true in the propositional logic formula (3.14), then the precedence constraint (j, i) is covered by the set of rules located in L. Where in the logical variables, false means that the packet does not match the corresponding rule, and true means that it does.

Then, by transforming the conditions of the rules in the same way as in the 3.7.1, the logical variables are mapped to the bit values of the packet.

For example, in the case of Table 3.20, the condition for r_1 is $\neg b_1 \wedge b_2 \wedge \neg b_3$, and for r_6 is $\neg b_1 \wedge b_4$. By (3.14), negate the clauses corresponding to the rules placed in the list L and take the logical conjunction. Finally, by taking the logical union between the rule r_i and the rule r_j from the precedence constraint (j, i) under consideration, the logical expression determines whether (j, i) affects the policy or not.

In Table 3.20, r_4 and r_6 have precedence constraints based on the dependency relation, but if r_1 and r_3 are placed in the aligned list R, the logical formula to determine if the precedence constraint $(6, 4)$ affects the policy is as follows.

$$\begin{aligned} f((6,4), \mathcal{R}') &= \neg r_1 \wedge \neg r_3 \wedge r_4 \wedge r_6 \\ &= \neg(\neg b_1 \wedge b_2 \wedge \neg b_3) \\ &\quad \wedge \neg(\neg b_1 \wedge b_2 \wedge b_3) \\ &\quad \wedge (b_2 \wedge b_4) \wedge (\neg b_1 \wedge b_4) \end{aligned} \tag{3.15}$$

For every precedence constraint in the rule list, this determination is performed each time the sorted list is updated to determine whether or not each precedence constraint is covered.

This method can be used to determine if a precedence constraint (j, i) is not covered by a single rule, even in complex cases where it is covered by multiple rules.

Time Complexity for determining coverages using SAT solver

In this section, we show the time complexity of the precedence constraint elimination method using SAT solver.

The first step in this method is to construct a common part for each precedence constraint (j, i). The common part is a string of length l consisting of three characters $\{0, 1, *\}$. The computational complexity of this process is $\mathcal{O}(l)$ for the rule pairs r_i and r_j. This process is performed for all precedence constraints. The number of precedence constraints based on dependencies is at most $\frac{1}{2}n^2$ for all rules in the rule list, since the maximum number of precedence constraints is reached when the rule is dependent on all rules placed higher than itself. As a result, the complexity of finding the common part of all dependent rule pairs in the rule list $\mathcal{R}$ is $\mathcal{O}(ln^2)$.

Then, for each precedence constraint (j, i), a list L is constructed from the rules placed at the upper level. If the number of rules in the input rule list is n, the number of rules placed at the top is at most n, so the computational complexity of constructing L is $\mathcal{O}(ln)$. Since this is done for all (j, i) precedence constraints, the computational complexity is $\mathcal{O}(ln^3)$. This process is performed each time a rule is placed in the sorted list, so the computational complexity of this method is $\mathcal{O}(ln^2 + ln^4x) = \mathcal{O}(ln^4x)$ if the SAT solver determines the dependency of a rule by $\mathcal{O}(x)$. Since the computational complexity of the SAT solver is exponential with respect to the literals of the input logic formulas, it is important to be able to find a solution for a realistic size problem.

Search for rules covering precedence constrains

The time complexity of the method that eliminates precedence constraint using the SAT solver is exponential. Therefore, a method that operates in polynomial time while maintaining accuracy as much as possible is required. In this section, we propose a method to search for and remove precedence constraints covered by a single rule.

The proposed method first searches for a covering rule for each precedence constraint (j, i) from the rules that overlap with r_j. When the searched rules are placed in the ordered list by the reordering method, the precedence constraints due to their dependencies are removed. This process is repeated each time the ordered list is updated.

The algorithm for removing precedence constraints using covered rules while selecting rules from the rule list and adding them to the aligned list is shown in Algorithm 16. Algorithm 16 first constructs precedence constraints by dependency relations in the form of an adjacency list $\mathcal{A}$, the same as the method proposed in 3.7.2. line 2 constructs a hash map C whose keys are the rule numbers and whose values are the list of precedence constraints covered by the rule. In line 4–6, same as the method proposed in 3.7.2, the proposed method selects rules to be added to the aligned list, places the selected rules in the sorted list and removes them from the rule list. At this time, the precedence constraints are relaxed by removing from the adjacent list $\mathcal{A}$ the precedence constraints based on the dependency relations belonging to the list that can be obtained from the rule numbers of the selected rules in the map C. This process is repeated until the rule list is empty.

In the case of Table 3.20, the precedence constraint $(7, 4)$ is covered by r_2. This shows that $(7, 4)$ does not affect the policy below r_2.

Time Complexity for the Covered Rule Search Method

In this section, we show the time complexity of the covered rule search method. The method first constructs the common part for each precedence constraint (j, i). The computational complexity is $\mathcal{O}(ln^2)$ as in 3.7.2. Then, for each rule in the rule list, the algorithm determines whether the precedence constraint (j, i) is covered or not. If the rule in focus covers (j, i), a map is constructed with the rule number of the covering rule as the key and the list of precedence constraints covered by the rule as the value, and (j, i) is stored in the list whose key is the rule number in focus. The computational complexity of comparing the common part with the rules and determining whether the rule is covered is $\mathcal{O}(l)$, and the computational complexity of determining which rule covers a single precedence constraint is $\mathcal{O}(ln)$ because the common part is compared with all rules. Since this process is performed for all precedence constraints, the computational complexity of map construction is $\mathcal{O}(ln^3)$. When placing a rule in the sorted list, the precedence constraints whose key value is the rule number of the rule to be placed are deleted. The complexity of repeating this process until the rule list is empty is $\mathcal{O}(n)$. The complexity of this method is $\mathcal{O}(ln^2 + ln^3 + n) = \mathcal{O}(ln^3)$.

3.7.3 Experiments

To demonstrate the effectiveness of the proposed methods, computer experiments were conducted using the Java language.

The PC used for the experiments on the rule reordering method by deleting rules with zero weight was an Intel Core i5-3470 CPU with 3.20GHz x 4 and CentOS release 7.6.1810 as the OS. We generated 100 rule sets of 1,000 to 10,000 rules and $100,000$ headers for each rule list using ClassBench [46]. We applied the method to these rule lists to remove the rules that have no matching packets and measured the latency whether decreases or not. We also used SGM to reorder the generated rule list and the rule list without 0-weighted rules and measured the latency whether decreased or not. The average latency for each number of rules is shown in Figure 3.26. Since it is difficult to see the difference between our method and the proposed method in Figure 3.26, the results are also shown in the Tables 3.23 and 3.24, and the difference in latency is shown in Figure 3.27 and 3.28.

As shown in Table 3.23, by removing the rules with weight 0, the latency is reduced compared to the given rule list. Figure3.28 shows that the proposed method relaxes the precedence

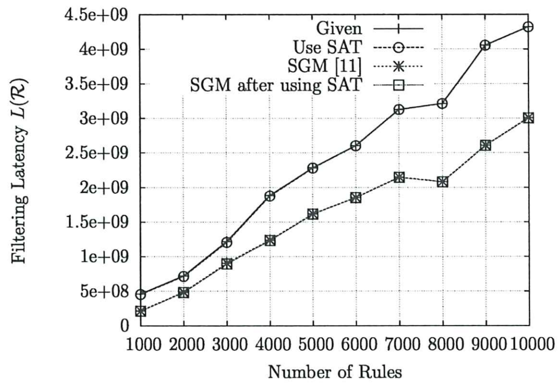


Figure 3.26: The latency of Packet classification.

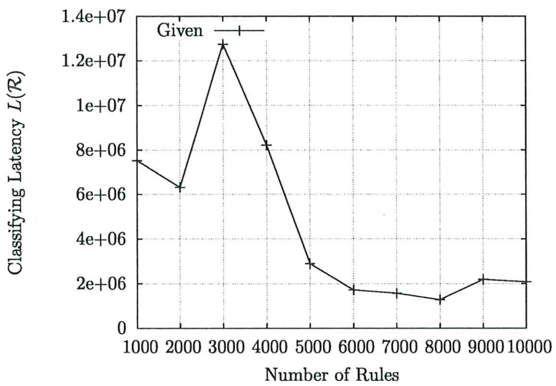


Figure 3.27: The difference of latency.

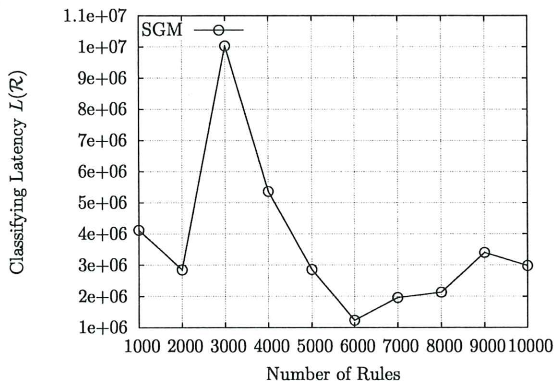


Figure 3.28: The difference of latency using SGM.

constraint due to the dependency relation, and the SGM reduces the latency more. When the number of rules is fewer, removing rules with weight 0 from the rule list before reordering often reduces the comparison frequency of packets matching the lower rules with higher weights. Therefore, the reduction in latency shown in Figure 3.27 is larger than that shown in Figure 3.28 after each rule list is reordered using SGM. However, when the number of rules exceeds 7000, the reduction in latency due to the ability to place rules with heavier weights at the upper positions by relaxing the dependency relation is larger than the reduction in the number of comparisons due to the deletion of rules with zero weights. The PC used for the experiments of the rule reordering method based on the elimination of precedence constraints via dependent relations has 8GB of main memory, an Intel Core i7-8700 CPU, and CentOS release 7.8.2003 as the OS. For the computer experiments, we generated a rule list with 100 $\sim$ 1000 rules and a header list with 100,000 corresponding headers using ClassBench. These rule lists were reordered using SGM and the two proposed methods were used to remove precedence constraints by dependent relations that do not affect the policy. For each result, we measured the latency and the reordering time. We used the SAT solver MiniSat [48] for the satisfiability determination. For each

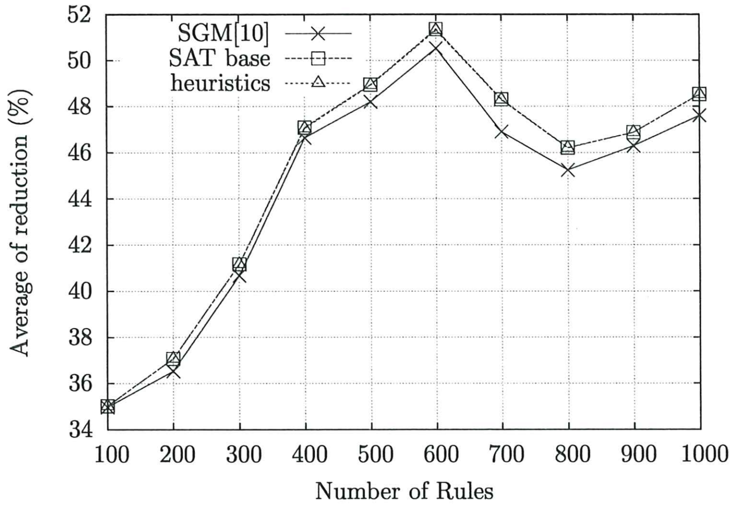


Figure 3.29: The average rate of decrease in ACL.

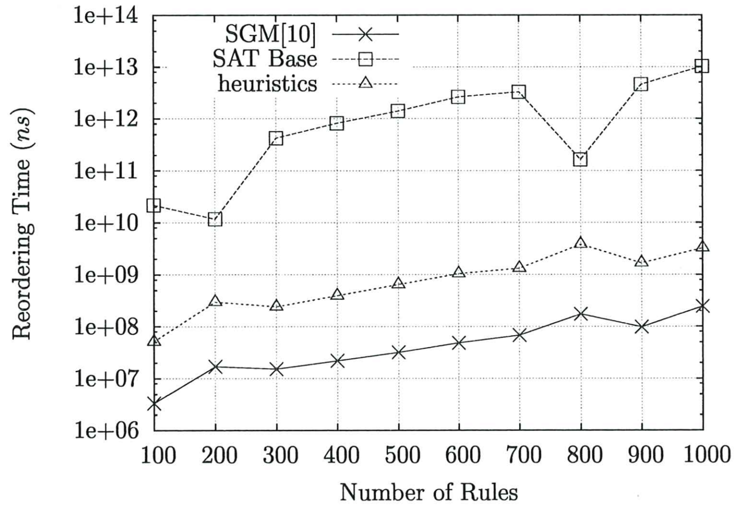


Figure 3.30: The reordering time of ACL.

number of rules, 10 rule lists and their corresponding header lists were tested once, for a total of 100 trials. The average of the reduction rate after reordering with respect to the latency of the generated rule lists and the average reordering time for each methods are shown in Figure 3.29, Table 3.25 and Figure 3.30. Figure3.29 shows the number of rules on the horizontal axis and the rate of reduction of latency on the vertical axis, and Figure3.30 shows the number of rules on the horizontal axis and the reordering time on the vertical axis. As shown in Figure 3.29 and Table 3.25, the proposed method reduces the latency compared to reordering using SGM alone as the number of rules increases.

In addition, for more than 300 rules, the method using the SAT solver reduces the latency more than the covered-rule search method. Since the method using the SAT solver can deter-

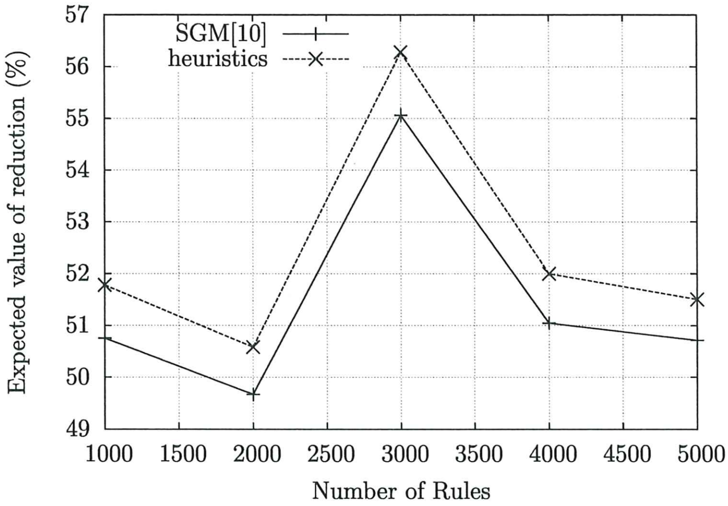


Figure 3.31: The average rate of decrease in ACL that have 1000 to 5000 rules.

mine complex precedence constraints that are covered by several rules, it can eliminate more precedence constraints, which may contribute to the reduction in latency. On the other hand, the covered rule search method cannot determine precedence constraints that are covered by several rules, so it is considered to be less accurate than the method using the SAT solver, and therefore, it did not reduce the latency sufficiently.

We performed computational experiments to demonstrate the effectiveness of the proposed method for rule lists with more than 1000 rules. The average reduction rate of latency when the rule list with 1000 rules is reordered using the SGM and covered rule search methods is shown in Figure 3.31. The horizontal axis is the number of rules and the vertical axis is the latency reduction rate. As shown in Figure 3.31, the covered rule search method reduces the latency more than the SGM for all rule numbers. Although the difference in the reduction rate between the proposed method and SGM is about 0.05% to 1%, this difference is meaningful because it indicates that there actually exists the order of rules with lower latency in the search area extended by the proposed method and such an order is required. As shown in Figure 3.30, the reordering time increases as the number of rules increases for the method using the SAT solver, but only up to 20 times for the covered rule search method. The reordering of rules in the real environment can be done by other computers based on the rule list and the number of evaluated packets, rather than directly on the rule list implemented in the network device so that the latency improvement rate and the reordering time can be considered separately. Therefore, the proposed method is an effective algorithm because the elimination of precedence constraints that do not affect the policy reduces the latency.

3.8 Optimal Allow Rule Ordering

An allowlist is a rule list in which all rule actions except the default rule are allow, and the default rule action is deny. So, in **OAO**, there are no precedence constraints with rules other than the default rule. In general, placing rules that match a large number of packets at the higher of the list tends to decrease the latency. However, the reordering method using the weight calculated before reordering can not account for the rule that will have a bigger weight by weight fluctuation. Thus, these methods can not reduce efficiently the latency.

3.8.1 Greedy Method for OAO

We propose a method that places the rule that is more matched with given packets in the upper place. For each rule, the method counts the number of packets that can be matchable with that and adds the rule with the largest value at the sorted list. Then, the method removes packets that match the added rule from the set of packets and removes the rule from the rule list. This process is repeated until the given rule list is empty.

We explain the proposed method using the rule list $\mathcal{R}$ in Table 3.26 and the packet distribution $\mathcal{F}$ in Table 3.27. For each rule, the algorithm computes the number of matchable packets regardless of the order. In this case, the numbers of the matchable packets are as follows.

$$
\begin{aligned}
|M(r_1)|_{\mathcal{F}} &= |\{1000, 1010, 1100, 1110\}|_{\mathcal{F}} = 90 \\
|M(r_2)|_{\mathcal{F}} &= |\{0101, 0111, 1101, 1111\}|_{\mathcal{F}} = 70 \\
|M(r_3)|_{\mathcal{F}} &= |\{0001, 0101\}|_{\mathcal{F}} = 90 \\
|M(r_4)|_{\mathcal{F}} &= |\{0110, 1110\}|_{\mathcal{F}} = 150 \\
|M(r_5)|_{\mathcal{F}} &= |\{1010, 1011, 1110, 1111\}|_{\mathcal{F}} = 110 \\
|M(r_6)|_{\mathcal{F}} &= |\{0100, 0101, 0110, 0111\}|_{\mathcal{F}} = 200 \\
|M(r_7)|_{\mathcal{F}} &= |\{0000, 0100, 1000, 1100\}|_{\mathcal{F}} = 90
\end{aligned}
$$

Since r_6 has the highest number of matchable packets, it is added to the top of the sorted list. Then the algorithm removes packets $\{0100, 0101, 0110, 0111\}$ that match r_6 from the set of packets and removes r_6 from the rule list $\mathcal{R}$. To determine the rule to be added to the sorted list, for each remaining rule, the algorithm computes the number of packets that match the rule

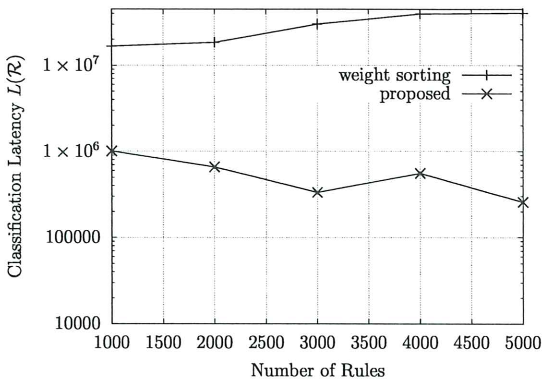


Figure 3.32: The latency of ACL.

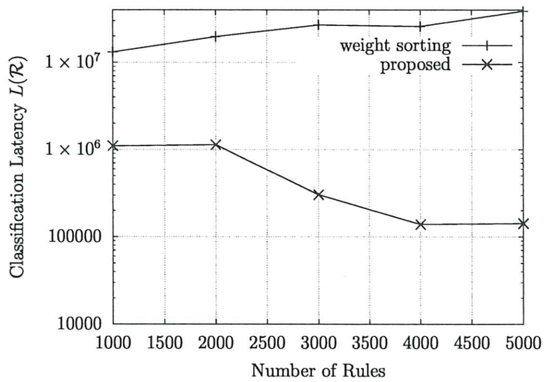


Figure 3.33: The latency of IPC.

in the set of packets $\mathcal{F}'$.

$$
\begin{aligned}
|M(r_1)|_{\mathcal{F}'} &= |\{1000, 1010, 1100, 1110\}|_{\mathcal{F}'} = 90 \\
|M(r_2)|_{\mathcal{F}'} &= |\{1101, 1111\}|_{\mathcal{F}'} = 40 \\
|M(r_3)|_{\mathcal{F}'} &= |\{0001\}|_{\mathcal{F}'} = 60 \\
|M(r_4)|_{\mathcal{F}'} &= |\{1110\}|_{\mathcal{F}'} = 30 \\
|M(r_5)|_{\mathcal{F}'} &= |\{1010, 1011, 1110, 1111\}|_{\mathcal{F}'} = 110 \\
|M(r_7)|_{\mathcal{F}'} &= |\{0000, 1000, 1100\}|_{\mathcal{F}'} = 50
\end{aligned}
$$

At this time, r_5 has the highest number of matchable packets. So, r_5 is added sorted list and removed from the rule list and the packets $\{1010, 1011, 1110, 1111\}$ are removed from the set of packets $\mathcal{F}'$. This process is repeated until the rule list is empty. Table 3.28 is the rule list that is reordered by the proposed method from the rule list in Table 3.26 and the packet distribution $\mathcal{F}$ in Table 3.27. As shown in Table 3.28, the proposed method reduces the latency.

3.8.2 Time complexity of the proposed method

We explain the time complexity of the proposed method. First, the algorithm searches the rule that has the highest number of matchable packets regardless of the order. The time complexity of this process is $\mathcal{O}(nq)$ where n is the number of rules and q is the number of packets in $\mathcal{F}$. Since that process is repeated n times, the time complexity of the proposed method is $\mathcal{O}(n^2q)$ and so, the method is the polynomial time algorithm.

3.8.3 Experiments

We demonstrate the effectiveness of the proposed algorithm through computer experiments. The proposed method was implemented in Java under CentOS release 7.8.2003 on an Intel Core i7-8700 with 8 GB of main memory. ClassBench [46] is known as the benchmark tool for packet

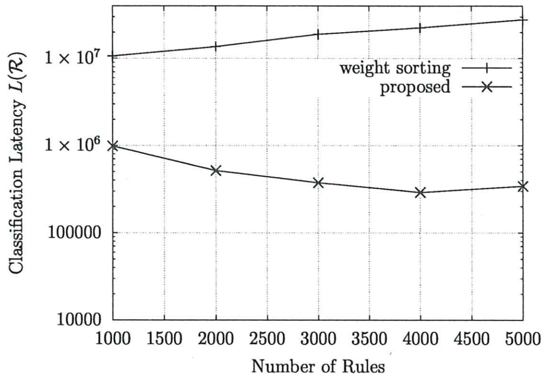


Figure 3.34: The latency of FW.

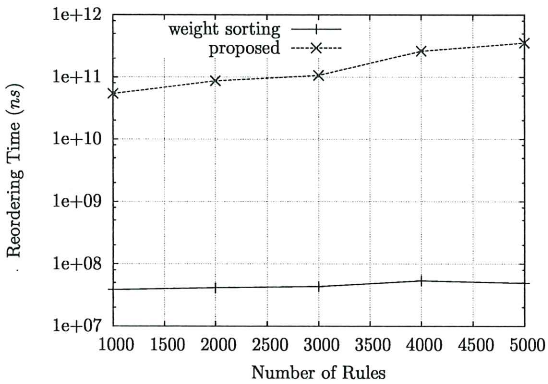


Figure 3.35: The time of ACL.

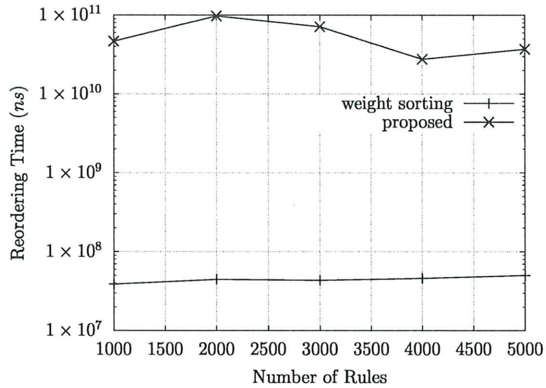


Figure 3.36: The time of IPC.

classification algorithms. It generates a rule list and a header list based on data obtained from actual environments. Therefore it can build an experimental environment closer to the real environment. We generated 50 rule sets of 1,000 to 5,000 rules with the seed file of the Access Control List (ACL), IP chain (IPC), and Fire Wall (FW). There were $100,000$ headers for each rule list. We implemented the methods of the descending order of weight and the proposed method.

The averages of 10 trials are depicted in Figs. 3.32, 3.33, 3.34, 3.35, 3.36, and 3.37. Note that we plotted the latencies and the reordering times on logarithmic scales in these graphs.

In Figs. 3.32, 3.33, and 3.34, the horizontal axes indicate the number of rules, whereas the vertical axes show the latencies. Figs. 3.32, 3.33, and 3.34 show that the proposed method reduced the latency compared to the other method. Furthermore, these graphs show that the proposed method reduces the latency as the number of rules increases. This is because as the number of rules increases, it is more likely that rule matching more packets will be generated in the rule list.

In Figs. 3.35, 3.36, and 3.37, the horizontal axis indicates the number of rules, whereas the vertical axis shows the reordering times. As shown in Figure 3.35, the proposed method

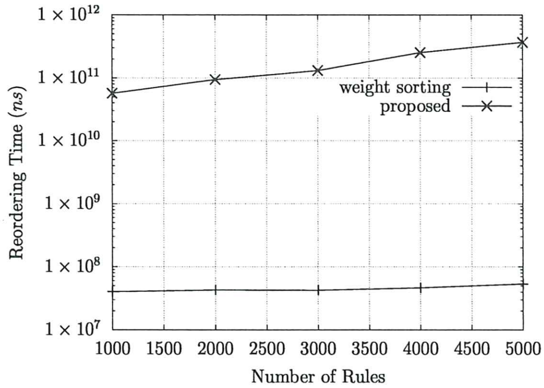


Figure 3.37: The time of FW.

takes about 1000 times longer than the methods of the descending order of weight. As shown in Figs. 3.36 and 3.37, in some cases, the reordering time increased for IPC and FW. This may be due to the inability to find rules that are matchable with many packets, which increased the number of packet matches.

3.9 Conclusion

In this chapter, we proposed several solution methods for **RORO**. To show the effectiveness of these methods, computational experiments were performed.

The future work is to develop a method that takes into account weight fluctuations in rule reordering. In **OAO**, existing heuristic methods sort rules in descending order of their weights at the time of input. However, in practice, when a rule that can match many packets is placed lower, the weight is calculated lower by the rule placed higher.

In general, processing more packets with a single rule results in lower latency, so there is a need to calculate the essential match count and develop a reordering method that uses this count. In addition, although Allow lists tend to be adopted in actual environments, there are cases where the environment is not necessarily limited to Allow lists. When changing partial policies, it is easier to accept the addition of rules with the Deny action. Therefore, it is a future challenge to devise a reordering method that takes into account weight fluctuations for general rule lists.

Algorithm 1: Sub-Graph Merging.

input : Rule List S and Q, Arrays X, C and $PROB$, and Two–dimensional Array DEP

output : Reordered rule list S

```
function policySort(S, Q, X, C, PROB, DEP);
while (Q ≠ φ) do
    /* Select the best rule in Q */
    set r_b to a rule in Q;
    foreach r_j ∈ Q and r_j ≠ r_b do
        if ((X[r_b]/C[r_b]) < (X[r_j]/C[r_j])) then
            set r_b to r_j;
        end
    end
    boolean selected = false;
    while (!selected) do
        /* Check if the selected rule has any dependents */
        if (C[r_b] == 1) then
            add r_b to S and remove r_b from Q;
            set selected to true;
            set r_selected to r_b;
        end
        else
            /* Select the best rule from G*(r_b) */
            bool temp = false;
            for (i = 1; i < b; i++) do
                if (DEP[r_i][r_b] == 1) then
                    if (temp == false) then
                        set r_tmp to r_i;
                        set temp to true;
                    end
                end
                else
                    if ((X[r_tmp]/C[r_tmp]) < (X[r_i]/C[r_i])) then
                        set r_tmp to r_i;
                    end
                end
            end
            set r_b to r_tmp;
        end
    end
    /* Update Data Structures */
    for i = selected + 1 to n − 1 do
        if (DEP[r_selected][r_i] == 1) then
            set DEP[r_selected][r_i] to 0;
            decrement C[r_i] by 1;
            [r_i] = X[r_i] - PROB[r_selected];
        end
    end
end
return S ;
```

Algorithm 2: Fixed SGM algorithm.

input : Weighted Rule List Q, Empty Rule List S, Arrays X, C and $PROB$, and Two–dimensional Array DEP

output: Rulelist S

while $(Q \neq \phi)$ **do**
  /*The same as line 3 to 23 in Figure 1*/

  /* Update Data Structures */
  A = get rules that are reachable from $r_{selected}$;
  for $i = selected + 1$ *to* $n - 1$ **do**
    set DEP$[r_{selected}][r_i]$ to 0;
  end
  foreach $r_i \in A$ **do**
    set DEP$[r_{selected}][r_i]$ to 0;
    decrement C$[r_i]$ by 1;
    X$[r_i]$ = X$[r_i]$ - PROB$[r_{selected}]$;
  end
end
return S;

Algorithm 3: ImprovedSGM.

input : Rule List $\mathcal{R}$

output: Rulelist $\mathcal{R}'$

make an empty list $\mathcal{R}'$;
while $\mathcal{R} \neq \phi$ **do**
  /* select the best rule in $\mathcal{R}$ */
  $G(r_b) \leftarrow$ selectMaxWeightRule($\mathcal{R}$);
  add r_b to $\mathcal{S}$ and remove r_b from $\mathcal{R}$;
end
return $\mathcal{R}'$;

Algorithm 4: selectMaxWeight.

```
input  : Weighted Rule List 𝓡
output: RuleSet R_s
if the size of 𝓡 = 1 then
    return the only one element R_s ∈ 𝓡;
end
set r to a rule in 𝓡;
val_b ← −1;
foreach r_i ∈ 𝓡 do
    set val_i to the average of weights of SG(r_i) ;
    if val_b < val_i then
        val_b ← val_i;
        r_b ← r_i;
    end
end
return selectMaxWeightRule(SG(r_b));
```

Table 3.6: Rule list $\mathcal{R}$.

Filter $\mathcal{R}$	w_i	Filter $\mathcal{R}$	w_i
$r_1^D = *001$	10	$r_6^D = *101$	30
$r_2^D = *000$	27	$r_7^A = 0*0*$	27
$r_3^A = 101*$	31	$r_8^D = **11$	28
$r_4^A = 001*$	19	$r_9^A = *11*$	30
$r_5^D = 1*10$	27	$r_{10}^A = 110*$	26

Table 3.7: Packet distribution F.

$0000 \mapsto 12$	$0001 \mapsto 14$
$0010 \mapsto 9$	$0011 \mapsto 10$
$0100 \mapsto 27$	$0101 \mapsto 6$
$0110 \mapsto 30$	$0111 \mapsto 15$
$1000 \mapsto 15$	$1001 \mapsto 16$
$1010 \mapsto 18$	$1011 \mapsto 13$
$1100 \mapsto 26$	$1101 \mapsto 4$
$1110 \mapsto 27$	$1111 \mapsto 13$

Algorithm 5: Hikage's Method.

input : Weighted Rule List $\mathcal{R}$,Dependent Graph G

output: RuleList $\mathcal{R}'$

1 Regarding G as an undirected graph, divide G into components $C_1, C_2, \ldots, C_k$. ;

2 **foreach** C_i **do**

3     $N_i \leftarrow$ Algorithm6(C_i)

end

4 **for** $i \leftarrow 1$ *to* k **do**

5   **for** $j \leftarrow 0$ *to the size of* N_i **do**

        /*I is a rule number of (N_i.length - k)-th rule*/

6         $W_I \leftarrow \left(\sum_{k=0}^{j} w_I\right) / (k+1)$;

    end

end

7 **while** *there is a non empty list* **do**

8   select the lightest rule r_j according to W_j in some N ;

9   add rules ($r_j, \ldots, N$.length) to $\mathcal{R}'$;

10   remove rules r_j, ..., N.last from N

end

11 **return** $\mathcal{R}'$;

Algorithm 6: Sort component.

input : Weighted Digraph C

output: List of rule number (vertices) in C

1 **while** C *is not empty* **do**

2   select the lightest rule r among the rules with $\deg^-(r) = 0$;

3   add r to N and remove r from C;

end

4 **return** N

Algorithm 7: Hikage's Method based on Comparison using with dependent rules.

input : Weighted Rule List $\mathcal{R}$,Dependent Graph G

output: RuleList $\mathcal{R}'$

/* the same as line 1 in Algorithm 5 */

1 **foreach** C_i **do**

2     $N_i \leftarrow$ Algorithm 8;

end

3 /* the same as lines 4 to 10 in Algorithm 5 */

4 **return** $\mathcal{R}'$;

Algorithm 8: Sort component with children weights.

input : Weighted digraph C

output: List of rule numbers in C

/*S is the children of r_i*/

foreach $r_i \in C$ **do**

  $w'_i \leftarrow \left(\sum_{j \in S} w_j\right) \Big/ |S|$;

end

while *C is not empty* **do**

  select the lightest rule r according to w' among the rules with $\deg^{-}(r) = 0$;

  add r to N and remove r from C ;

end

return N

Table 3.8: Rule list $\mathcal{R}$.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{F}}$
r_1^A =**11	10
r_2^D =1**1	5
r_3^D =0**1	30
r_4^A =111*	74
r_5^A =10**	74
r_6^A =011*	50
r_7^D =****	10
$L(\mathcal{R}, \mathcal{F}) = 1132$	

Table 3.9: The packet arrival distaribution $\mathcal{F} : \mathcal{P} \to \mathbb{N}$.

$0000 \mapsto 1$	$0001 \mapsto 18$	$0010 \mapsto 2$	$0011 \mapsto 3$
$0100 \mapsto 3$	$0101 \mapsto 12$	$0110 \mapsto 50$	$0111 \mapsto 0$
$1000 \mapsto 35$	$1001 \mapsto 2$	$1010 \mapsto 39$	$1011 \mapsto 2$
$1100 \mapsto 4$	$1101 \mapsto 3$	$1110 \mapsto 74$	$1111 \mapsto 5$

Table 3.10: Reordering by SGM.

Filter $\mathcal{R}_\sigma$	$\lvert E(\mathcal{R}_\sigma, i)\rvert_{\mathcal{F}}$
r_1^A =**11	10
r_3^D =0**1	30
r_6^A =011*	50
r_2^D =1**1	5
r_5^A =10**	74
r_4^A =111*	74
r_7^D =****	10
$L(\mathcal{R}_\sigma, \mathcal{F}) = 1074$	

Table 3.11: Better reordering.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}_\sigma, i)\rvert_{\mathcal{F}}$
r_1^A =**11	10
r_2^D =1**1	5
r_5^A =10**	74
r_4^A =111*	74
r_3^D =0**1	30
r_6^A =011*	50
r_7^D =****	10
$L(\mathcal{R}_\sigma, \mathcal{F}) = 1048$	

Algorithm 9: DividingAndConqueringRules.

input : Weighted rule list $\mathcal{R}$

output: Reordered rule list $\overline{\mathcal{R}}$

if $|\mathcal{R}| = 1$ **then**

  return $\mathcal{R}$;

end

// select the rules to be placed upper position ;

$\mathcal{R}_{upper} \leftarrow$ DivideRules($\mathcal{R}$) ;

$\mathcal{R}_{lower} \leftarrow \mathcal{R} \setminus \mathcal{R}_{upper}$;

$\overline{\mathcal{R}}_{upper} \leftarrow$ DividingAndConqueringRules($\mathcal{R}_{upper}$) ;

$\overline{\mathcal{R}}_{lower} \leftarrow$ DividingAndConqueringRules($\mathcal{R}_{lower}$) ;

$\overline{\mathcal{R}} \leftarrow$ concatenate $\overline{\mathcal{R}}_{upper}$ and $\overline{\mathcal{R}}_{lower}$;

return $\overline{\mathcal{R}}$;

Algorithm 10: DivideRules.

input : Weighted rule list $\mathcal{R}$

output: Sub rule list $\mathcal{R}_{upper} \subset \mathcal{R}$

foreach $r \in \mathcal{R}$ **do**
- make $G(r)$ and $D(r)$;
- $S(r) \leftarrow r$;
- $T(r) \leftarrow G(r)$;

end

for $i =$ *size of* $\mathcal{R}$ *to* 1 **do**
- $r \leftarrow$ The i'th Rule in $\mathcal{R}$;
- make list LD of $D(r)$ and sort LD according to $X(u)/|S(u)|$ for $u \in D(r)$;
- **for** $j = 1$ *to size of* LD **do**
  - $u \leftarrow$ The j'th Rule in LD ;
  - $T^{'}(r) \leftarrow T(r) \cup S(u)$;
  - **if** $Z(r)/|T(r)| < Z^{'}(r)/|T^{'}(r)|$ **then**
    - $T(r) \leftarrow T^{'}(r)$;
    - $S(r) \leftarrow S(r) \cup S(u)$;
  - **end**
- **end**

end

$r^{'} \leftarrow$ head of $\mathcal{R}$;

foreach $r \in \mathcal{R}$ **do**
- **if** $Z(r^{'})/T(r^{'}) < Z(r)/T(r)$ **then**
  - $r^{'} \leftarrow r$;
- **end**

end

if $|G(r^{'})| = |\mathcal{R}|$ **then**
- **return** DividingAndConqueringRules($\mathcal{R} \setminus r^{'}$);

end

return $T(r^{'})$ as a rule list;

Table 3.12: Constructed sets.

r_i	$G(r_i)$	$D(r_i)$	$T(r_i)$	$S(r_i)$
r_7	$\{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}$	$\{\}$	$\{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}$	$\{r_7\}$
r_6	$\{r_1, r_3, r_6\}$	$\{\}$	$\{r_1, r_3, r_6\}$	$\{r_6\}$
r_5	$\{r_1, r_2, r_5\}$	$\{\}$	$\{r_1, r_2, r_5\}$	$\{r_5\}$
r_4	$\{r_1, r_2, r_4\}$	$\{\}$	$\{r_1, r_2, r_4\}$	$\{r_4\}$
r_3	$\{r_1, r_3\}$	$\{r_6\}$	$\{r_1, r_3, r_6\}$	$\{r_3, r_6\}$
r_2	$\{r_1, r_2\}$	$\{r_4, r_5\}$	$\{r_1, r_2, r_4, r_5\}$	$\{r_2, r_4, r_5\}$
r_1	$\{r_1\}$	$\{r_2, r_3\}$	$\{r_1, r_2, r_4, r_5\}$	$\{r_1, r_2, r_4, r_5\}$

Algorithm 11: PartialBlockReordering.

input : Weighted rule list $\mathcal{R}'$
output: Reordered rule list $\mathcal{R}''$
for $k = 1$ *to the size of* $\mathcal{R}'$ **do**
  if $\mathcal{R}'(k)$ *is sink* **then**
    add $\mathcal{R}'(k)$ to $\mathcal{R}''$;
  end
  else
    $\mathcal{R}'' \leftarrow$MakeBlockAndReplace($\mathcal{R}', \mathcal{R}'', k$);
  end
end
return $\mathcal{R}''$;

Algorithm 12: MakeBlockAndReplace.

input : Weighted rule list $\mathcal{R}'$, $\mathcal{R}''$,Integer k
output: Reordered rule list $\mathcal{R}'''$

add $\mathcal{R}'(k)$ to the front of L_2;
PrecedingSet $\leftarrow$ the set of rules that $\mathcal{R}'(k)$ depends on;
$L_1 \leftarrow \emptyset$;
lowerlist $\leftarrow \emptyset$;
upperlist $\leftarrow \emptyset$;
$i =$ size of $\mathcal{R}''$;
while $i \geq 1$ **do**
    if $\mathcal{R}''(i) \in$ *PrecedingSet* **then**
        upperlist $\leftarrow$ the rules $\mathcal{R}''(1), \mathcal{R}''(2), \ldots, \mathcal{R}''(i)$;
        if $D(L_1, L_2) < 0$ **then**
            break;
        end
        else
            move r_i to the front of L_2;
            add rules that r_i depends on to *PrecedingSet*;
            add L_1 to the front of *lowerlist*;
            $L_1 \leftarrow \emptyset$;
        end
    end
    else
        add r_i to the front of L_1;
    end
    $i = i - 1$;
end
if $i < 1$ **then**
    if $D(L_1, L_2) < 0$ **then**
        $\mathcal{R}'' \leftarrow L_1 + L_2 + lowerlist$;
    end
    else
        $\mathcal{R}''' \leftarrow L_2 + L_1 + lowerlist$;
    end
end
else
    $\mathcal{R}''' \leftarrow upperlist + r_i + L_1 + L_2 + lowerlist$;
end
return $\mathcal{R}'''$;

Table 3.13: Rule list $\mathcal{R}$.

Filter $\mathcal{R}$	$\lvert r_i \rvert$
$r_1^A = 11{*}0$	20
$r_2^A = 01{*}0$	120
$r_3^A = 010{*}$	78
$r_4^A = 0{*}01$	74
$r_5^D = 10{*}0$	80
$r_6^D = {*}{*}00$	131
$r_7^D = {*}1{*}1$	95
$r_8^A = 1{*}0{*}$	150
$r_9^A = 1{*}11$	100
$r_{10}^D = {*}{*}{*}{*}$	10
$L(\mathcal{R}, \mathcal{F}) = 4883$	

Table 3.14: A packet arrival distribution $\mathcal{F} : \mathcal{P} \to \mathbb{N}$.

$0000 \mapsto 131$	$0001 \mapsto 74$	$0010 \mapsto 10$	$0011 \mapsto 0$
$0100 \mapsto 110$	$0101 \mapsto 78$	$0110 \mapsto 10$	$0111 \mapsto 60$
$1000 \mapsto 20$	$1001 \mapsto 150$	$1010 \mapsto 60$	$1011 \mapsto 100$
$1100 \mapsto 15$	$1101 \mapsto 30$	$1110 \mapsto 5$	$1111 \mapsto 5$

Table 3.15: $\mathcal{R}$' reordered by sub-list merging.

Filter $\mathcal{R}'$	$\lvert r_i \rvert$
$r_2^A = 0{*}01$	120
$r_3^A = 01{*}1$	78
$r_1^A = 01{*}0$	20
$r_6^D = {*}1{*}1$	131
$r_4^A = {*}011$	74
$r_7^D = 0{*}1{*}$	95
$r_5^A = {*}001$	80
$r_8^A = 00{*}{*}$	150
$r_9^A = {*}1{*}0$	100
$r_{10}^D = {*}{*}{*}{*}$	10
$L(\mathcal{R}', \mathcal{F}) = 4550$	

Table 3.16: $\mathcal{R}''$ reordered by the proposed.

Filter $\mathcal{R}''$	$\lvert r_i \rvert$
$r_2^A = 0{*}01$	120
$r_3^A = 01{*}1$	78
$r_4^A = {*}011$	74
$r_7^D = 0{*}1{*}$	95
$r_9^A = {*}1{*}0$	100
$r_1^A = 01{*}0$	20
$r_6^D = {*}1{*}1$	131
$r_5^A = {*}001$	80
$r_8^A = 00{*}{*}$	150
$r_{10}^D = {*}{*}{*}{*}$	10
$L(\mathcal{R}'', \mathcal{F}) = 4495$	

Table 3.17: Rule list $\mathcal{R}$.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R},i)\rvert_{\mathcal{F}}$
r_1^A =1**1	10
r_2^D =1*00	5
r_3^D =*11*	30
r_4^A =11**	0
r_5^A =*10*	40
r_6^A =**11	50
r_7^D =****	10
$L(\mathcal{R},\mathcal{F}) = 670$	

Table 3.18: The packet arrival distaribution $\mathcal{F} : \mathcal{P} \to \mathbb{N}$.

0000 $\mapsto$ 3	0001 $\mapsto$ 1	0010 $\mapsto$ 2	0011 $\mapsto$ 50
0100 $\mapsto$ 20	0101 $\mapsto$ 20	0110 $\mapsto$ 6	0111 $\mapsto$ 0
1000 $\mapsto$ 10	1001 $\mapsto$ 5	1010 $\mapsto$ 4	1011 $\mapsto$ 0
1100 $\mapsto$ 0	1101 $\mapsto$ 0	1110 $\mapsto$ 4	1111 $\mapsto$ 0

Table 3.19: Remove r_4 from search space.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R},i)\rvert_{\mathcal{F}}$
r_6^A =**11	50
r_5^A =*10*	40
r_3^D =*11*	30
r_1^A =1**1	10
r_2^D =1*00	5
r_7^D =****	10
r_4^A =11**	0
$L(\mathcal{R},\mathcal{F}) = 345$	

Table 3.20: Rule list $\mathcal{R}$.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{F}}$
$r_1^A = 010*$	10
$r_2^A = 1{*}01$	20
$r_3^A = 011*$	30
$r_4^D = {*}1{*}1$	5
$r_5^D = 11{*}0$	5
$r_6^A = 0{*}{*}1$	100
$r_7^A = 1{*}0{*}$	120
$r_8^D = {*}{*}{*}{*}$	10
$L(\mathcal{R}, \mathcal{F}) = 1695$	

Table 3.21: Reordering $\mathcal{R}$ by SGM.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}_{SGM}, i)\rvert_{\mathcal{F}}$
$r_3^A = 011*$	30
$r_2^A = 1{*}01$	20
$r_1^A = 010*$	10
$r_4^D = {*}1{*}1$	5
$r_6^A = 0{*}{*}1$	100
$r_5^D = 11{*}0$	5
$r_7^A = 1{*}0{*}$	120
$r_8^D = {*}{*}{*}{*}$	10
$L(\mathcal{R}_{SGM}, \mathcal{F}) = 1560$	

Table 3.22: Better ordering.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}_{better}, i)\rvert_{\mathcal{F}}$
$r_3^A = 011*$	30
$r_2^A = 1{*}01$	20
$r_5^D = 11{*}0$	5
$r_7^A = 1{*}0{*}$	120
$r_1^A = 010*$	10
$r_4^D = {*}1{*}1$	5
$r_6^A = 0{*}{*}1$	100
$r_8^D = {*}{*}{*}{*}$	10
$L(\mathcal{R}_{better}, \mathcal{F}) = 1320$	

Algorithm 13: SearchAndDeletePre-Constraints.

input : constraints $\mathcal{A}$,the middle of sorted list $\overline{\mathcal{R}}$
output: constraints $\overline{\mathcal{A}}$

$\overline{\mathcal{A}} \leftarrow \mathcal{A}$;
foreach *Constraints* (j, i) *in* $\mathcal{A}$ **do**
  $L \leftarrow$ rules that are matchable with the packet in $M(r_j) \cap M(r_i)$;
  CNF $F \leftarrow$ MakeCNFForTheDependency((j, i),L);
  if F *is UNSAT* **then**
    //The judgment is based on the SAT solver;
    Delete (j, i) in $\overline{\mathcal{A}}$;
  end
end
return $\overline{\mathcal{A}}$;

Algorithm 14: SearchCoveringRule.

input : Pre-Constraints$\mathcal{A}$,RuleList$\mathcal{R}$
output: Map C
foreach (j, i) *in* $\mathcal{A}$ **do**
  foreach *Rule* r_c *that is depended on* r_j *or* r_i **do**
    if r_c *overlapped with* r_j **then**
      if r_c *cover* (j, i) **then**
        (j, i) add to the List C_c;
      end
    end
  end
end
return C;

Algorithm 15: SATbased.

input : Weighted rule list $\mathcal{R}$
output: Reordered rule list $\overline{\mathcal{R}}$
Make the adjacency list $\mathcal{A}$ by dependency relation on $\mathcal{R}$;
while $|\mathcal{R}| \neq 0$ **do**
  Select a rule r from $\mathcal{R}$ by SGM;
  add r to $\overline{\mathcal{R}}$;
  Delete r from $\mathcal{R}$;
  $\mathcal{A} \leftarrow$ SearchAndDeletePre-Constraints($\mathcal{A}$,$\overline{\mathcal{R}}$);
end
return $\overline{\mathcal{R}}$;

Algorithm 16: DeleteConstraintsCoveredBySingleRule.

input : Weighted rule list $\mathcal{R}$
output: Reordered rule list $\overline{\mathcal{R}}$
Make the adjacency list $\mathcal{A}$ by dependency relation on $\mathcal{R}$;
Map(RuleNum,List of (j, i)) C $\leftarrow$SearchCoverRule($\mathcal{A}$,$\mathcal{R}$);
for $|\mathcal{R}| \neq 0$ **do**
  $\overline{\mathcal{R}} \leftarrow$ The Rule r_s decided by SGM in $\mathcal{R}$;
  Delete r_s from $\mathcal{R}$;
  foreach *Dependency* D *in* C_s **do**
    Delete D in $\mathcal{A}$;
  end
end
return $\overline{\mathcal{R}}$;

Table 3.23: The result of Delete 0 Weight Rules.

	Given	Proposed
1000	$4.60087e+08$	$4.52561e+08$
2000	$7.20902e+08$	$7.14581e+08$
3000	$1.21889e+09$	$1.20615e+09$
4000	$1.88454e+09$	$1.87632e+09$
5000	$2.28286e+09$	$2.27996e+09$
6000	$2.60218e+09$	$2.60046e+09$
7000	$3.12566e+09$	$3.12409e+09$
8000	$3.21211e+09$	$3.21083e+09$
9000	$4.05510e+09$	$4.05291e+09$
10000	$4.32193e+09$	$4.31985e+09$

Table 3.24: The result of reordered by SGM.

	SGM	Proposed
1000	$2.17318e+08$	$2.13202e+08$
2000	$4.8534e+08$	$4.8249e+08$
3000	$9.04147e+08$	$8.94115e+08$
4000	$1.23941e+09$	$1.23405e+09$
5000	$1.61881e+09$	$1.61595e+09$
6000	$1.85364e+09$	$1.85241e+09$
7000	$2.14838e+09$	$2.14642e+09$
8000	$2.08214e+09$	$2.08001e+09$
9000	$2.60803e+09$	$2.60463e+09$
10000	$3.00534e+09$	$3.00236e+09$

Table 3.25: The average rate of decrease for reordered rule list.

rule	SGM	SATbased	heuristic
100	34.9665	35.0159	35.0159
200	36.5317	37.0713	37.0713
300	40.6770	41.1706	41.1714
400	46.6387	47.0479	47.0867
500	48.1987	48.9199	48.9393
600	50.5280	51.3132	51.3577
700	46.8984	48.2835	48.3015
800	45.2458	46.1945	46.2073
900	46.2897	46.8658	46.8690
1000	47.6016	48.5121	48.5191

Table 3.26: Allow list.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{F}}$
$r_1^A = 1{*}{*}0$	90
$r_2^A = {*}1{*}1$	70
$r_3^A = 0{*}01$	60
$r_4^A = {*}110$	120
$r_5^A = 1{*}1{*}$	30
$r_6^A = 01{*}{*}$	40
$r_7^A = {*}{*}00$	20
$r_8^D = {*}{*}{*}{*}$	30
$L(\mathcal{R}, \mathcal{F}) = 1630$	

Table 3.27: Packet distribution $\mathcal{F} : \mathcal{P} \to \mathbb{N}$.

$0000 \mapsto 20$	$0001 \mapsto 60$	$0010 \mapsto 10$	$0011 \mapsto 10$
$0100 \mapsto 40$	$0101 \mapsto 30$	$0110 \mapsto 120$	$0111 \mapsto 10$
$1000 \mapsto 10$	$1001 \mapsto 10$	$1010 \mapsto 30$	$1011 \mapsto 30$
$1100 \mapsto 10$	$1101 \mapsto 10$	$1110 \mapsto 30$	$1111 \mapsto 30$

Algorithm 17: Proposed Method.

input : Allowlist $\mathcal{R}$ and Packet Distribution $\mathcal{F}$

output: Reorderd allowlist $\mathcal{R}'$

while $\mathcal{R}$ *is not empty* **do**

  $max = 0$;

  foreach $r_i \in \mathcal{R}$ **do**

    foreach $p \in \mathcal{F}$ **do**

      if p *matches* r_i **then**

        $matchcount \leftarrow matchcount + 1$;

      end

    end

    if $max < matchcount \lor max = 0$ **then**

      $maxrule \leftarrow r_i$;

      $max \leftarrow matchcount$;

    end

  end

  foreach $p \in \mathcal{F}$ **do**

    if p *matches* $maxrule$ **then**

      $\mathcal{F} \leftarrow \mathcal{F} \backslash \{p\}$;

    end

  end

  $maxrule$ adds to $\mathcal{R}'$;

  $maxrule$ removes from $\mathcal{R}$;

end

return $\mathcal{R}'$;

Table 3.28: Reordered by proposed.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R}, i)\rvert_{\mathcal{F}}$
$r_6^A = 01{*}{*}$	200
$r_5^A = 1{*}1{*}$	110
$r_3^A = 0{*}01$	60
$r_7^A = {*}{*}00$	50
$r_2^A = {*}1{*}1$	10
$r_1^A = 1{*}{*}0$	0
$r_4^A = {*}110$	0
$r_8^D = {*}{*}{*}{*}$	30
$L(\mathcal{R}, \mathcal{F}) = 1060$	

Chapter 4

Rule List Reconstruction

The rule list optimization problem takes a rule list and a frequency distribution of packets as input and finds a rule list with minimized latency while holding policy. In general, reducing the number of rules results in a rule list with lower latency, so a method that considers the rule list as a logical expression and constructs a rule list with fewer rules using the Kwein-McCluskey method has been proposed [42]. In this chapter, we show the computational complexity of the rule list optimization problem and propose a heuristic solution for this problem.

4.1 Complexity of ORL

We define the optimization problem as follows: given a list and a frequency distribution, construct a list that has the same policy as the given list and minimizes the classification latency.

Definition 4.1.1. (Optimal Rule List(**ORL**))

$$\begin{aligned} \text{Input}: &\quad \text{Rule List } \mathcal{R},\ \text{Packet Arrival Distribution } \mathcal{F} \\ \text{Output}: &\quad \text{Rule List } \mathcal{R}' \text{ such that the following conditions are satisfied} \\ &\quad \mathcal{R}' \equiv \mathcal{R} \text{ and } \forall \mathcal{S} \equiv \mathcal{R}, L(\mathcal{R}', \mathcal{F}) \leq L(\mathcal{S}, \mathcal{F}) \end{aligned}$$

Furthermore, a decision problem version of this optimization problem is defined as follows.

Definition 4.1.2. (Rule List Reconstruction(**RLR**))

Table 4.1: Allow list.

	Filter $\mathcal{R}$	$\lvert E(\mathcal{R},i)\rvert_{\mathcal{F}}$
r_1^A	11*1	2
r_2^A	0*0*	4
r_3^A	11*0	2
r_4^A	1000	1
r_5^D	****	7

Table 4.2: Merge and separate.

	Filter $\mathcal{R}$	$\lvert E(\mathcal{R},i)\rvert_{\mathcal{F}}$
r_1^A	**00	4
r_2^A	11**	3
r_3^A	0*01	2
r_4^D	****	7

Table 4.3: Merge and separate.

	Filter $\mathcal{R}$	$\lvert E(\mathcal{R},i)\rvert_{\mathcal{F}}$
r_1^A	1001	1
r_2^A	**0*	7
r_3^A	111*	2
r_4^D	****	6

Input :	Rule List $\mathcal{R}$, Packet Arrival Distribution $\mathcal{F}$ and Positive integer K
Question :	Is there a Rule List $\mathcal{R}'$ that satisfies the following condition?
	$\mathcal{R}' \equiv \mathcal{R}$ and $L(\mathcal{R}', \mathcal{F}) \leq K$

We prove that this determination problem is NP-hard by reducing from *Min-DNF*. *Min-DNF* is the problem to find the minimum disjunction form of f given an n-variable logic function f as a truth table and is known to be NP-hard. The decision problem version of this optimization problem is as follows.

Definition 4.1.3. (*Min-DNF*)

Input :	n-variable logic function f as a truth table tt and Positive integer K
Question :	Is there a disjunction form with less than or equal to the number of K disjuncts for a logic function?

In the following, we refer to the decision problem version of *Min-DNF* as *Min-DNF*.

For an n-variable oolean function f, define the set$ON(f) \subset 0, 1^n$ as $ON(f) = \{x | f(x) = 1\}$. Furthermore, $OFF(f) = \{0, 1\}^n \setminus ON(f)$. For example, $ON(f) = \{001, 101, 110\}$ for a three-variable logic function f whose truth table is 01000110.

4.1.1 Definition and Lemma for Reduction

In this section, we present definitions and lemma to show the polynomial-time reduction from *Min-DNF* to **RLR**. The strategy of the reduction is to generate from the truth table tt, an instance of *Min-DNF*, a list $\mathcal{R}$ of **RLRs**, a frequency distribution $\mathcal{F}$, and a threshold $\mathcal{K}$ such that the latency is not less than $\mathcal{L}$ unless the Allow list has fewer than K rules. The details of the proofs of some of the corollaries are taken from the literature [49]. Details are given in Appendix C.

Definition 4.1.4. (Rule Merging)
Assume that for several rules $r_i, \ldots, r_j$, the set $M(r_k)$ of packets that can be matched by rule r_k is equal to $M(r_i) \cup \cdots \cup M(r_j)$. In this case, merging r_i and r_j into a single rule r_k is called rule merging.

Definition 4.1.5. (Rule Decomposition)
Dividing a rule r_i into two rules r_j and r_k such that $M(r_j) \cup M(r_k) = M(r_i)$ and $M(r_j) \cap M(r_k) = \emptyset$ is called rule decomposition.

The list in Table 4.2 is produced by decomposing r_2^A into $r_{2,1}^A = 0 * 00$ and $r_{2,2}^A = 0 * 01$ and r_3^A into $r_{3,1}^A = 1100$ and $r_{3,2}^A = 1110$ in Table 4.1 and merging $r_{2,1}^A$, $r_{3,1}^A$, r_4^A, $r_{3,1}^A$, $r_{3,2}^A$ and r_4^A.

Definition 4.1.6. (Merging rules by supplementation)
When it is possible to add a rule merging $r_i, \ldots, r_j$ under r_k by inserting rule r_k above it to hold the policy for several rules $r_i, \ldots, r_j$, the addition and merging of rules by such an operation is called merging rules by supplementation.

Rule r_2^A in Table 4.3 with the Allow action is generated by merging $r_{3,1} = 1100$ and r_4, which are decompositions of r_2 and r_3 in Table 4.1, by supplementing r_1^D with the Deny action.

It should be noted that in the *Min-DNF* to **RLR** reduction, it is not necessarily that the fewer rules in the list the smaller the latency. The following is a complement to be used in the proof.

Lemma 4.1.1. *Given an Allow list R consisting of n allow rules, the latency of reordering the rules in descending order of weight, such as*

$$|E(\mathcal{R}_\sigma, 1)| \geq |E(\mathcal{R}_\sigma, 2)| \geq \cdots \geq |E(\mathcal{R}_\sigma, n-1)| \geq |E(\mathcal{R}_\sigma, n)|$$

is the minimal latency of the Allow list obtained by the rule reordering.

Definition 4.1.7. *Let the list $\mathcal{R}$ consisting of n rules without masks $*$ is as follows.*

$$\mathcal{R} = [r_1^A, r_2^A, \ldots, r_n^A]$$

Let the list $\mathcal{S}$ consisting of K rules that express the same policy as $\mathcal{R}$ be as follows.

$$\mathcal{S}_K = [s_1^A, s_2^A, \ldots, s_K^A]$$

Where for the weight of each rule $|E(\mathcal{S}, i)|$ in $\mathcal{S}$, $|E(\mathcal{S}, 1)| \geq |E(\mathcal{S}_\sigma, 2)| \geq \cdots \geq |E(\mathcal{S}_\sigma, n-1)| \geq |E(\mathcal{S}_\sigma, n)|$ and $|E(\mathcal{S}, i)| \geq 1$. In addition, $\mathcal{R}$ has the default rule r_{n+1}^D with a weight of 0.

Lemma 4.1.2. *From the literature* [49], the lower bound on the latency $L(\mathcal{S}_K, \mathcal{F})$ of $\mathcal{S}_K$ is $2n + \frac{(K-2)(K-1)}{2} - 2^l$.

We denote the upper bound of the latency in $\mathcal{S}_K$ by $\underline{B(K)}$.

Lemma 4.1.3. *From the literature* [49], the upper bound on the latency $L(\mathcal{S}_K, \mathcal{F})$ of $\mathcal{S}_K$ is $2^l + \frac{q(K+2)(K-1)}{2} + \frac{r(r+3)}{2}$. Where $l = min\{l | 2^l \geq \frac{n}{K}\}$ and q and r are is a natural number satisfying the following fomula.

$$n - 2^l = (K-1)q + r(K-1 > r)$$

We denote the upper bound of the latency in $\mathcal{S}_K$ by $\overline{W(K)}$.

Lemma 4.1.4. *For any natural number $n \geq 2, K \geq 1$, $\underline{B(K+1)} \leq \overline{W(K)}$, if $\overline{W(K)} - \underline{B(K+1)}$ is a polynomial order in n and K.*

Lemma 4.1.5. *Assume that for an Allow list $\mathcal{R}$, suppose that $\overline{W(K)} - \underline{B(K+1)} = d > 0$. Let $\mathcal{R}'$ be the Allow list where the rules that cannot be merged with any rule in $\mathcal{R}$, have a weight of 1, and do not contain $*$ are added to $\mathcal{R}$. For $\mathcal{R}'$, $\overline{W(K+1)} - \underline{B(K+2)} = d - 1$.*

Although fewer rules do not necessarily mean lower latency, from the lemma4.1.5, we can add $W(K) - B(K+1) + 1$ of non-mergeable rules so that the difference between the upper bound of latency of an Allow list with size K and the lower bound of latency of an Allow list with size $K+1$ is exactly 1. This allows us to generate an Allow list with lower latency for fewer rules. Also, from the complement4.1.4, this difference can be computed in polynomial time, and this difference fits into a polynomial of size K and $ON(f)$ of size n, which is an instance of *Min-DNF*.

4.1.2 Reduction from *Min-DNF* to RLR

In addition to the previous section, it should be noted that the merging rules by supplementation of the Deny rule may decrease the latency in the reduction from *Min-DNF* to **RLR**. Therefore, the algorithm for reducing from *Min-DNF* to **RLR** is shown in Algorithm18 so that the latency cannot be reduced by such a merging. The list generation in this algorithm first generates a permission list consisting of m_1 permission rules corresponding to each element of $ON(f_{tt})$ from the input truth table *tt*. To this list, the algorithm adds $m_1 m_2$ permission rules corresponding to $OFF(f_{tt})$ in order to avoid latency reduction in the merging of permission rules by supplementing denial rules. Then, to adjust the number of terms K in *Min-DNF* and the latency L in *RLR*, $d+1$ permission rules are added when $d \geq 0$. Where f_{tt} denotes the logic function corresponding to the truth table, and $m_1 = |ON(f_{tt})|, m_2 = |OFF(f_{tt})|$. The loop from the algorithm line. 8 generates permission rules corresponding to $ON(f_{tt})$. Since these rules only have an additional $*$ column of length *ll* at the end, the rules can be merged if the part corresponding to f_{tt} can be merged.

Then the loop from the line. 19 in the algorithm generates dummy Allow rules and adds them to the list. Since the trailing *ll* bits of these dummy Allow rules are not $*$ but a string of bits, at least m_1 supplementation is required to merge the Allow rules with the deny rules generated in the loop of line. 8. Since the rules differ from each other by at least 2 bits, these rules cannot be merged either. Therefore, the merging of Allow rules by supplementing denial rules does not reduce the latency.

Then, in a loop from the line.26, the algorithm adds a permission rule that cannot be merged with any rule and has a weight of 1 so that the list with a latency less than L is limited when the rules corresponding to $ON(f_{tt})$ can be merged into K permission rules or less. In other words, we add rules to fill the difference mentioned in the lemma 4.1.4.

This completes the generation of the *Min-DNF* truth table *tt* and the list $\mathcal{R}$ corresponding to the nonnegative integer K. Assume that the frequency distribution $\mathcal{F}$ for this list is as follows.

$$\mathcal{F}(p) = \begin{cases} 1 & \text{if } p \in \mathcal{P} \\ 0 & \text{otherwise} \end{cases}$$

Where $\mathcal{P}'$ is the packet set obtained by the reduction algorithm *Red*. This is the distribution in which exactly one packet appears that matches only one of each of the rules generated by the loop from line.8 to line.26, and in which the nth and subsequent bits differ from each other by 2 bits.

And we assume that the threshold L is set as $d+1$ if $\overline{W(K)} - \underline{B(K+1)} = d \geq 0$, otherwise $B(K)$.

Theorem 4.1.1. ***RLR** is **NP**-hard.*

proof. 1. *We show* $(tt, K) \in$ *Min-DNF* $\Longleftrightarrow$ *Red*$(tt, K) \in$ ***RLR***.

If there exists a disjunction form with less than or equal to K terms for an n-variable logic function f_{tt} represented by a truth table tt, then $(tt, K) \in$ *Min-DNF* $\Rightarrow$ *Red*$(tt, K) \in$ ***RLR** is* *valid because there exists a disjunction list with equal policy and latency to the allow list $\mathcal{R}$. On the other hand, if there is no disjunction form with K or fewer terms for f_{tt}, then the rules corresponding to $ON(f)$ cannot be merged into K or fewer rules, and there is no list that achieves L or less latency. From this, $(tt, K) \notin$ Min-DNF $\Rightarrow$ Red$(tt, K) \notin$ **LRR** also holds. And since Algorithm 18 is a polynomial-time algorithm for input size, Red is a polynomial-time attribution algorithm from Min-DNF to **RLR**, and **RLR** is **NP**-hard.*

4.2 Allow List Reconstruction

When considering an optimal rule list problem, generally the number of rules is reduced by merging rules that have the same actions, thereby reducing the latency. Therefore, we treat an optimization problem limited to the Allow list and consider how much latency reduction can be expected without merging rules by supplementation.

The optimization problem restricted to the Allow list is defined as follows

Definition 4.2.1. (Optimal Allow List)
Input : Allow list $\mathcal{R}$, Packet Arrival Distribution $\mathcal{F}$
Output : Allow List $\mathcal{R}'$ such that the following conditions are satisfied
$\mathcal{R}' \equiv \mathcal{R}$ and $\forall \mathcal{S} \equiv \mathcal{R}, L(\mathcal{R}', \mathcal{F}) \leq L(\mathcal{S}, \mathcal{F})$

Since this problem has been shown to be **NP**-hard, [49] we propose a heuristic solution for this problem.

In general, when finding an Allow list with lower latency, it is required to generate a rule that matches more packets. We present the following theorem.

Theorem 4.2.1. *In the Allow list optimization problem, the Allow list can be regarded as a logical formula such that the assignment corresponding to the packet to which the Allow action is applied is true. When the main terms of the formula are enumerated, there exists an Allow list with the minimum latency that consists only of the rule $r_q \in Q$ corresponding to the main term in the logical formula. Where Q is the set of rules corresponding to the principal terms of the logical formula corresponding to the Allow list.*

Proof. In the Allow list $\mathcal{R}$, let $F(\mathcal{R})$ be the logical expression for which the allocation corresponding to the packet to which the Allow action is applied is true.

Consider a minimal latency Allow list $\mathcal{R}$ containing t rules r_u that have no correspondence with the main term of $F(\mathcal{R})$. Since $\mathcal{R}$ is a minimal latency Allow list, there is no rule in $\mathcal{R}$ that can be included in a rule in $\mathcal{R}$. All t rules r_u have to be included in at least one rule contained in Q. We assume that $\mathcal{R}'$ be an Allow list in which every rule r_u in $\mathcal{R}$ is replaced by a rule $r_q \in Q$ such that r_u is included. We prove the theorem by showing that the latency of $\mathcal{R}'$ is always less than or equal to the latency of $\mathcal{R}$.

We consider whether the packets in $E(\sigma(u), \mathcal{R})$ increase the number of comparisons at $\mathcal{R}'$. Since $\mathcal{R}'$ is an allow list where r_u in $\mathcal{R}$ is replaced by the rule $r_q \in Q$ that contains the rule, the number of rules in $\mathcal{R}'$ and $\mathcal{R}$ is the same. Also, the replaced rules are placed in the same positions as before. Since the packets in $E(\sigma(u), \mathcal{R}) \setminus E(\sigma(q), \mathcal{R}')$ do not match r_q in $\mathcal{R}'$, they match rules higher than r_q. Therefore, the number of comparisons is decreasing. The packets in $E(\sigma(u), \mathcal{R}) \cap E(\sigma(q), \mathcal{R}')$ match r_q, which is located at the same position as r_u in $\mathcal{R}'$, so the number of comparisons are same. Thus, in $\mathcal{R}'$, all packets that matched the rule replaced from $\mathcal{R}$ match the rule that is placed at the same or higher position than before the replacement, so $L(\mathcal{R}', \mathcal{F}) \leq L(\mathcal{R}', \mathcal{F})$. This means that the latency of the Allow list $\mathcal{R}'$ is the same or less than the Allow list $\mathcal{R}'$ latency. □

The rule contained in Q is called the maximal rule. A maximal rule is a rule such that $M(r_i) \subset B$ and $M(r_i) \subseteq M(r_j)$ when B is the set of packets to which the Allow action is applied, and $M(r_i) \subseteq M(r_j)$ when there is no r_j. From the theorem4.2.1, there exists a rule list with the minimal latency in the allow list constructed only by the maximal rules, so we propose a method to construct an allow list with smaller latency by enumerating the maximal rules.

4.2.1 Allow list Reconstruction Method using Consensus

The maximal rule can be computed by enumerating the principal terms in the logical formulas corresponding to the input Allow list. The consensus method is used to enumerate the principal terms.

The consensus method is an algorithm that enumerates the principal terms from a set of product terms of an input logical expression [50]. The main terms are obtained by constructing product terms combining two variables from pairs of product terms that differ by only one bit except $*$. For two product terms $c = x_1 x_2 \dots x_n$ and $c' = x'_1 x'_2 \dots x'_n$, if the variables differing in value are at most 1 except for $*$, the following product term c'' can be constructed from them.

$$
\begin{aligned}
&x''_i = x_i \sqcup x'_i = x \sqcup * = * \sqcup x = x \\
&x''_i = x_i \sqcup x'_i = x \sqcup \bar{x} = * \sqcup * = * \\
&\text{for } x = 0 \text{ and } 1
\end{aligned}
$$

Where there is at most one variable for which $x_i \sqcup x'_i = x \sqcup \bar{x}$, since the number of variables that differ in value except for $*$ is at most one. The operation of constructing the product term in this way is called a consensus, denoted $c \sqcup c'$.

We describe an algorithm for enumerating the principal terms for a given assignment using the consensus method.

We describe an algorithm for enumerating the principal terms for a given assignment using the consensus method. The algorithm first takes consensus for all given sets of product terms C. Since a product term with more $*$ is required, the generated term is stored in C' if the number of $*$ in the term generated by consensus is greater than the number of iterations j. The product terms in C are then removed such that the product terms in C' are included in the product terms in C'. The product terms contained in C' are added to C. The product terms that are obtained by consensus and have a number of $*$ corresponding to the number of iterations j or more are added to C', and the product terms that could not be generated are added to the output set as the principal terms. Once all pairs of product terms have been consesed, add C' to C and $j = j + 1$. This process is repeated until C is empty. As a result, the principal terms for the input assignments are obtained.

By using the consensus method to enumerate the maximal rules for the Allow list optimization problem, the maximal rules can be enumerated without enumerating the set of packets to which the Allow action applies.

The rule construction method for the maximum using the consensus method is shown in Algorithm19. First, Algorithm19 removes rules that are included in other rules from the input set of rules C. Then, line 3 takes consensus on all pairs of rules contained in C. If the number of $*$ in the rule is larger than j, it is added to the set C' at the line 4. If the rule can not generate a rule with $*$ greater than $j + 1$ by consensus, it is added to the output set C'' and deleted from C at line 5 Then, the rules in C' are added to C, and j is updated by deleting the rules included in C. This process is repeated until C is empty, and the set of maximal rules C'' is output.

From the enumerated set of maximal rules, the Allow list is reconstructed by finding a rule list with lower latency.

By constructing an Allow list, which is a list for a set of maximal rules, formulating it into an integer programming problem, and obtaining the optimal solution, the Allow list with the minimal latency in the Allow list optimization problem can be obtained. However, since the computational complexity of this method is exponential in relation to the number of rules, the operation will not be completed in a realistic time as the number of rules increases. Therefore, a reordering method based on the greedy method shown in 3.8.1 can be used to obtain a sequence of rules with smaller latency.

4.2.2 Experiments

We demonstrate the effectiveness of the proposed algorithm and reordering algorithms through computer experiments.

The proposed methods were implemented in Java under Ubuntu 22.04.3 LTS on Intel Core i7-8700 with 8 GB of main memory. We used ClassBench which is known as the benchmark tool for the packet classification algorithm. We generated 300 rule sets of 100 to 1000 rules and corresponding $100,000$ headers for each rule list using ClassBench [46] with the seed file of the Access Control List (ACL).

For these Allow lists, we applied the reconstruction method using consensus and measured the rate of reduction in latency and the reconstruction time.

The average ratio of latency reduction for each number of rules is shown in Figure 4.1. Figure 4.1 shows the number of rules on the horizontal axis and latency on the vertical axis. As shown in Figure 4.1, the latency reduction rate increases as the number of rules increases, and the input Allow list is significantly reduced in latency.

In addition, the average reconstructing time for each number of rules is shown in Figure 4.2. The horizontal axis is the number of rules and the vertical axis is the average reconstruction time.

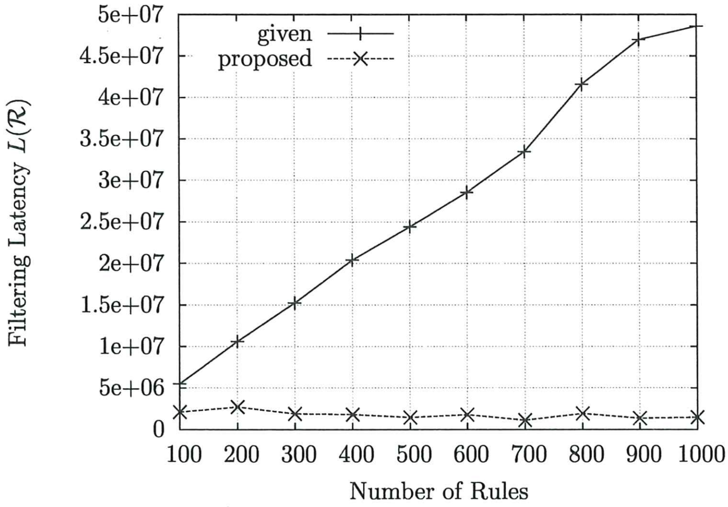


Figure 4.1: The latency of ACL.

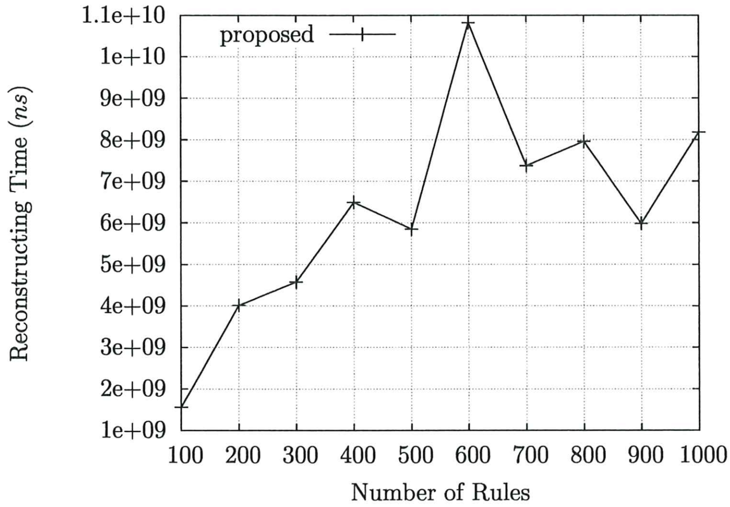


Figure 4.2: The reconstructing time of ACL.

As shown in Figure 4.2, the reconstructing time increases as the number of rules increases. However, the operation time of the proposed method is highly dependent on the policies expressed in the input rule list, so the reconstructing time does not necessarily increase as the number of rules increases.

4.3 Rule List Reconstruction

Previous approaches to the rule list optimization problem have achieved packet classification with fewer rules by merging similar rules in the input rule list. However, such approaches have limited effectiveness, and their performance depends greatly on the input rule list. Therefore, a method to reconstruct rules by focusing on the policy represented by the input rule list is required. It is difficult to obtain the policy expressed in the rule list from the rule list. In addition, it is impractical to maintain the actions to be applied to each packet, because the amount of space calculation is exponential to the bit length. Therefore, by removing overlap relations from the input rule list, a rule list corresponding to a set of packets to which the same action is applied can be obtained. Using this rule list, we propose a method to find a rule list with lower latency.

In a list of rules with overlapping relationships, the packets in the common part match the rules placed above, so the rules placed below are not necessarily the same as the set of packets $M(r_i)$ that can be matched and $E(\mathcal{R}, i)$ that are actually matched. Therefore, in a rule list containing rules with some overlap relations, it is not immediately clear which rule matches how many packets.

In addition, the action that is applied to the packet in the common part is the action of the rule that is placed at the highest level among the rules that are in a dependent relationship. This means that for each rule r_i, not all packets in the set $M(r_i)$ of matchable packets will have the same action applied. This means that for each rule r_i, not all packets in the set $M(r_i)$ of matchable packets will have the same action applied. Therefore, we proposed a method that rewrites the lower-placed rules into rules that do not match the packets in the common part to obtain a rule list that is not dependent on any rules.

4.3.1 Find common parts

First, we explain how to find the common part of two rules r_i and r_j that have an overlap relation. By comparing each bit of each rule and taking the value if they are the same, or the bit value of the other if one is $*$, we can obtain a string $c(r_i, r_j)$ that represents the packet set of the common part.

For example, consider the common part of rules r_i and r_j in Table 4.6. In this case, the 1th bit of $c(r_i, r_j)$ is 0 because the 1th bit of r_i is 0 and the 1th bit of r_j is $*$. Also, the 2th bit of r_i is $*$, but the 2th bit of r_j is 1, so the 1th bit of $c(r_i, r_j)$ is 1. By performing this operation for all bits, a string $c(r_i, r_j) = 01001 * 1111$ representing the packet set of the common part is obtained. This means that the packets contained in the common part $c(r_i, r_j)$ of r_i and r_j are $\{0100101111, 010011111111\}$.

4.3.2 Take Setminus of r_j

To remove the overlap relation between r_i and r_j, we can rewrite either rule so that it does not overlap with the common part. By focusing on the bits that are $*$ in the rewritten rule but not $*$ in the common part of the string and splitting the rule into a rule with 0 and a rule with 1, one of the rules will always not match the common part. This operation can be rewritten by repeating it for each bit to be focused on.

For example, consider the case where rule r_j in Table 4.6 is rewritten to a rule that does not match the common part $c(r_i, r_j)$. In this case, the first bit of r_j is $*$ and the common part is 0, so we know that in $M(r_j)$, the packet with the first bit of 1 is not included in the common part $c(r_i, r_j)$. Therefore, r_j is divided into $r'_{j,1} = 11*01***11$ and $r'_{j,2} = 01*01***11$. This makes $r'_{j,1}$ a rule that does not match the common part. Then, we rewrite $r'_{j,2}$ as a rule that does not match the common part $c(r_i, r_j)$. Since the third bit of $r'_{j,2}$ is $*$ but the common part is 0, we know that the packet with the third bit of 1 in $M(r'_{j,2})$ is not included in the common part. Therefore, $r'_{j,2}$ is divided into two parts: $r'_{j,3} = 11101***11$ and $r'_{j,4} = 01001***11$. This makes $r'_{j,3}$ a rule that does not match the common part. By repeating this operation until the generated rules do not match the packets in the common part, r_j can be rewritten as rules that do not match the common part with r_i. Table 4.7 is the result of rewriting r_j. Similarly, Table 4.8 is the result of rewriting r_i to a rule that does not match $c(r_i, r_j)$.

However, if r_i encompasses r_j, r_j cannot be rewritten because $r_j = c(r_i, r_j)$. In that case, there is no packet that matches r_j, so the policy holds even if r_j is deleted. For this reason, when creating a rule list with no overlap relation, we assume that the rules placed at the lower levels are rewritten.

4.3.3 Algorithm for Removing Overlap

An Algorithm that takes r_i and r_j as input and rewrites r_j into m rules that have no overlap relation with r_i is shown in Algorithm 20. First, Algorithm 20 constructs the common part of r_i and r_j in 5 line to 11 line. Then, rules that do not have overlap relations with the strings in the common part are created in the 6 to 14 lines. As with the Allow list, a rule list with no dependencies does not violate the policy no matter how the rules are reordered above the default rules. Therefore, we propose a method to construct a rule list with no dependencies using Algorithm20, and to sort the rules in order of the number of matching packets using Algorithm 17.

4.3.4 Proposed Method for ORL

First, the proposed method applies the Algorithm20 to the dependent rules r_i and r_j, and rewrites r_j into m rules that do not have an overlap relation with r_i. This operation is performed for all dependent pairs, and the dependent relations in the rule list are deleted. The rules are then sorted in order of the number of matching packets using Algorithm 17. Then, we determine whether the rule r_i^D should be deleted or not. In the $\mathcal{R}'$ rule list sorted using Algorithm 17, whether r_i^D, the ith rule in the list, should be deleted or not is judged based on whether the following high/low relation is satisfied, using the latency expression.

$$
\begin{aligned}
i|E(\mathcal{R}',i)|+&(i+1)\sum_{k=i+1}^{n-1} k(|E(\mathcal{R}',k)|)+(n-1)|E(\mathcal{R}',r_n)| \\
&> \sum_{k=i+1}^{n-1}(k-1)(|E(\mathcal{R}',k)|)+(n-2)(|E(\mathcal{R}',n)|+|E(\mathcal{R}',i)|) \qquad (4.1) \\
\sum_{k=i+1}^{n-1}(|E(\mathcal{R}',k)|)-&(n-2-i)|E(\mathcal{R}',i)|+|E(\mathcal{R}',n)|>0
\end{aligned}
$$

If the inequality 4.1 is satisfied, then r_i^D should be removed from the rule list $\mathcal{R}'$ to reduce the latency. Rewrite the rule list in Table 4.9 to a rule list with no dependencies, using Algorithm 20 to construct a rule list with lower latency. In the case of Table 4.9, r_1^D and r_6^A are dependent, so these rules are entered into Algorithm 20. It outputs the set of rules $S=\{r'^A_{6,1}=11*1, r'^A_{6,2}=1011\}$. By replacing these rules with r_6^A, the overlap relation between r_1^D and r_6^A can be removed. Since no other rule is dependent on r_1, we search for a dependent relation with r_2. Repeating this operation results in a rule list like Table 4.12. The rule list in Table 4.13 is the result of sorting by Algorithm 17 using the packet set in Table 4.11. Then, redundant rules with Deny actions are removed, starting from the lowest rule. For example, the problem of determining whether the latency would be lower if r_3^D were removed is as follows.

$$
\begin{aligned}
&\sum_{k=5+1}^{8-1}(|E(\mathcal{R}',k)|)-(8-2-5)|E(\mathcal{R}',5)|+|E(\mathcal{R}',8)|>0 \\
&(13+10)-16+10>0 \qquad (4.2) \\
&17>0
\end{aligned}
$$

This shows that removing r_3 reduces the latency. Table 4.14 shows the rule list with r_3 removed from the rule list in Table 4.13. As shown in Table 4.14, the latency is lower than that of the rule list in Table 4.13. By repeating this operation for all the Deny rules, we can obtain a rule list with lower latency. In the case of the rule list in Table 4.13, removing all the Deny rules results in a rule list with lower latency. Table 4.15 shows the rule list of Table 4.13 when all the Deny rules are removed. As shown in Table 4.15, the latency is reduced compared to Table 4.9.

4.3.5 Experiments

To demonstrate the effectiveness of the proposed method, we performed computer experiments. The proposed method was implemented in Java under Ubuntu 22.04.3 LTS on Intel Core i7-8700 with 8 GB of main memory. We generated 30 rule sets of 100 to 1,000 rules using ClassBench [46] with the seed file of the Access Control List (ACL).

We applied the proposed method to the generated rule lists and measured the latency and reconstruction time. The latency of the generated rule list and the latency of the reconstructed rule list are shown in Figure4.3. Figure 4.3 shows the number of rules on the horizontal axis

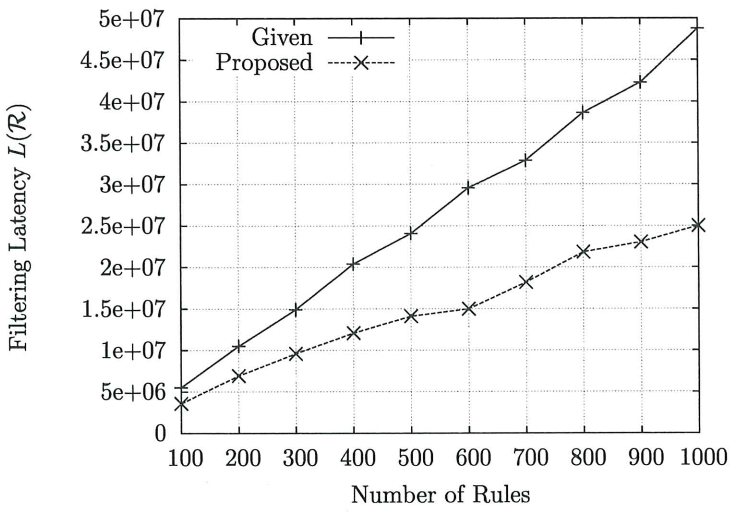


Figure 4.3: The latency of ACL.

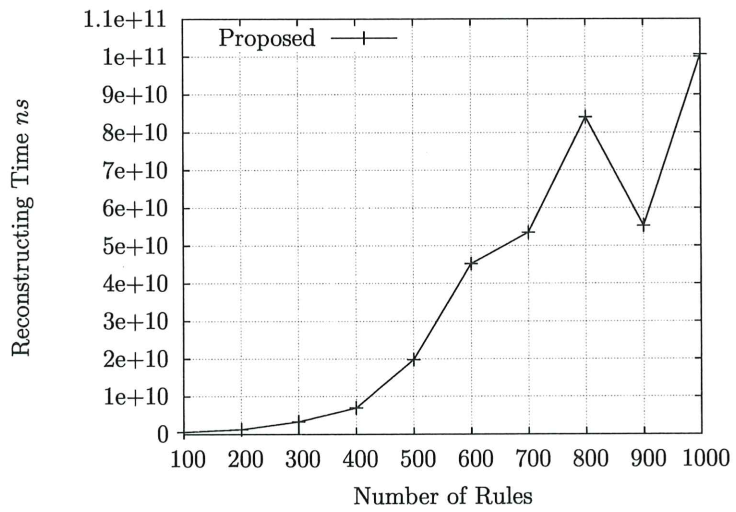


Figure 4.4: The reconstructing time of ACL.

and latency on the vertical axis. As shown in Figure 4.3, the latency is decreases as the number of rules increases.

Figure 4.4 shows the reconstruction time. The horizontal axis is the number of rules and the vertical axis is the average reconstruction time. As shown in Figure 4.4, the reconstruction time increases as the number of rules increases. However, the operation time is faster in the part with 900 rules, which is thought to be due to the fact that the number of rules to be divided was fewer due to the dependencies in the rule list.

4.4 Conclusion

In this section, we show the computational complexity of the rule list optimization problem and propose a heuristic solution method for OAL and general rule list reconstruction. In the reconstruction method for general rule lists, we proposed a method to construct a rule list with lower latency by removing dependency relations so that the rule with the highest matching frequency can be placed at the top. By removing the dependency relation, the proposed method can determine whether a rule with the Deny action should be removed or not from the latency expression because it can be seen that the rule with the Deny action matches the default rule when it is removed. As the number of packets that match the default rule increases, the fewer rules there are, the lower the latency will generally be. However, in the proposed method, the number of rules increases due to the deletion of dependencies. Future work is to propose a method for determining whether dependency relations increase latency due to an increase in the number of rules when removed. Another future work is to develop a method to construct a rule list that can satisfy the same policy with fewer rules.

Algorithm 18: Red.

input : Truth table tt,Integer K
output : Rulelist $\overline{\mathcal{R}}$, Packet distribution $\mathcal{P}'$, Integer L
$d \leftarrow \overline{W(K)} - \underline{B(K)}$;
if $d < 0$ **then**
    $L \leftarrow \underline{B(K)}$;
else
    $L \leftarrow d + 1$;
end
$ll \leftarrow 2m_1m_2$;
$\mathcal{R} \leftarrow$ an empty list;
$\mathcal{P}' \leftarrow \emptyset$;
$i \leftarrow 1$;
forall $x \in ON(f_{tt})$ **do**
    set b to the empty string ϵ and p to ϵ;
    $b \leftarrow b + +x, p \leftarrow p + +x$;
    $b \leftarrow b + +*^{ll}, p \leftarrow p + +1^{ll}$;
    $b \leftarrow b + +0^{ll}, p \leftarrow p + +0^{d}$;
    set the condition of r_i to b;
    add r_i^A to $\mathcal{R}$;
    $p_{n+2i-1} \leftarrow 0, p_{n+2i} \leftarrow 0$;
    add p to $\mathcal{P}'$;
    $i \leftarrow i + 1$;
end
$j \leftarrow 0$;
forall $x \in OFF(f_{tt})$ **do**
    set b to ϵ;
    $b \leftarrow b + +x$;
    $b \leftarrow b + +0^{ll+d}$;
    set the condition of r_i to b;
    add r_i^A to $\mathcal{R}$;
    $i \leftarrow i + 1, j \leftarrow j + 1$;
end
while $i \leq m_1m_2 + d$ **do**
    set b to ϵ, and p to ϵ;
    $b \leftarrow b + +1^{n}, p \leftarrow p + +1^{n}$;
    $b \leftarrow b + +0^{ll}, p \leftarrow p + +0^{ll}$;
    $b \leftarrow b + +1^{d}, p \leftarrow p + +1^{d}$;
    $b_{2j+1} \leftarrow 0, b_{2(j+1)} \leftarrow 0$;
    set the condition of r_i to b;
    add r_i^A to $\mathcal{R}$;
    $p_{2j+1} \leftarrow 0, p_{2(j+1)} \leftarrow 0$;
    add p to $\mathcal{P}'$;
    $i \leftarrow i + 1, j \leftarrow j + 1$;
end
add r_i^D to $\mathcal{R}$;
return $(\mathcal{R}, \mathcal{P}', L)$;

Table 4.4: Allow list $\mathcal{R}$.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R},i)\rvert_{\mathcal{F}}$
$r_1^A = 0{*}00$	10
$r_2^A = 0{*}11$	27
$r_3^A = {*}1{*}1$	31
$r_4^A = 0{*}10$	19
$r_5^A = 10{*}1$	27
$r_6^D = {*}{*}{*}{*}$	27
$L(\mathcal{R},\mathcal{F}) = 503$	

Table 4.5: Packet Arrival Distribution $\mathcal{F} : \mathcal{P} \rightarrow \mathbb{N}$.

$0000 \mapsto 4$	$0001 \mapsto 19$	$0010 \mapsto 2$	$0011 \mapsto 10$
$0100 \mapsto 6$	$0101 \mapsto 5$	$0110 \mapsto 5$	$0111 \mapsto 17$
$1000 \mapsto 7$	$1001 \mapsto 13$	$1010 \mapsto 1$	$1011 \mapsto 14$
$1100 \mapsto 9$	$1101 \mapsto 10$	$1110 \mapsto 3$	$1111 \mapsto 16$

Algorithm 19: Consensus method.

input : The Product terms Set$C$
output: The Prime terms SetsC''

j=0;
remove $c \in C$ such that $c \subseteq c' \in C$;
while C *is not empty* **do**
  foreach $c \in C$ **do**
    foreach $c' \in C$ **do**
      $c'' \leftarrow c \sqcup c'$;
      if *num of* $* \in c'' > j$ **then**
        add c'' to C';
      end
    end
    if *c could not take the consensus* c'' *such that num of* $* \in c'' > j$ **then**
      add c to C'';
      remove c from C;
    end
  end
  add C' to C;
  remove $c \in C$ such that $c \subset c' \in C$;
  $j = j + 1$;
end
return C'';

Table 4.6: The common part between r_i and r_j.

r_i	0∗00∗∗11∗∗
r_j	∗1∗01∗∗∗11
$c(r_i, r_j)$	01001∗1111

Table 4.7: Rewrites r_j into the rules that do not match $c(r_i, r_j)$.

$r'_{j,1}$	11∗01∗∗∗11
$r'_{j,3}$	01101∗∗∗11
$r'_{j,7}$	01001∗0∗11
$r'_{j,8}$	01001∗1011

Table 4.8: Rewrites r_i into the rules that do not match $c(r_i, r_j)$.

$r'_{i,1}$	0000∗∗11∗∗
$r'_{i,3}$	01000∗11∗∗
$r'_{i,7}$	01001∗110∗
$r'_{i,8}$	01001∗1110

Algorithm 20: Separate(r_i, r_j).

input : Overlapped rules r_i, r_j

output: rule set S

$common \leftarrow empty$;
$jmask \leftarrow emptySet$;
for $k = 1$ *to* l **do**
    if $b_{i,k} = b_{j,k}$ **then**
        $common \leftarrow b_{i,k}$;
    end
    else
        if $b_{i,k} = *$ **then**
            $common \leftarrow b_{j,k}$;
        end
        else
            add k to $jmask$;
            $common \leftarrow b_{i,k}$;
        end
    end
end
foreach $n \in jmask$ **do**
    $S_n \leftarrow Setminus(common, r_j, n)$;
    add S_n to S;
end
return S;

Algorithm 21: Setminus(r_i, r_j, K).

input : Overlapped rules r_i, r_j,Integer T

output: rule r'

$r' \leftarrow$ *empty*;

for $k = 1$ *to* l **do**

  if $k < T$ **then**

    $r' \leftarrow b_{i,k}$;

  else if $k = T$ **then**

    $r' \leftarrow \neg b_{j,k}$;

  else

    $r' \leftarrow b_{j,k}$;

  end

end

return r';

Table 4.9: Rule list $\mathcal{R}$.

Filter $\mathcal{R}$		$\lvert E(\mathcal{R}, i)\rvert$
r_1^D	0∗0∗	24
r_2^A	10∗0	8
r_3^D	1∗1∗	23
r_4^D	1∗00	9
r_5^A	100∗	13
r_6^A	∗1∗1	34
r_7^A	∗∗10	35
r_{def}^D	∗∗∗∗	10
$L(\mathcal{R}, \mathcal{F}) = 729$		

Table 4.10: Rewrite r_6^A.

r_1^D	0∗0∗
r_2^A	10∗0
r_3^D	1∗1∗
r_4^D	1∗00
r_5^A	100∗
$r'^A_{6,1}$	11∗1
$r'^A_{6,2}$	0111
r_7^A	∗∗10
r_{def}^D	∗∗∗∗

Table 4.11: Packet Arrival Distribution $\mathcal{F} : \mathcal{P} \rightarrow \mathbb{N}$.

$0000 \mapsto 4$	$0001 \mapsto 9$	$0010 \mapsto 25$	$0011 \mapsto 10$
$0100 \mapsto 6$	$0101 \mapsto 5$	$0110 \mapsto 10$	$0111 \mapsto 24$
$1000 \mapsto 7$	$1001 \mapsto 13$	$1010 \mapsto 1$	$1011 \mapsto 14$
$1100 \mapsto 9$	$1101 \mapsto 10$	$1110 \mapsto 3$	$1111 \mapsto 6$

Table 4.12: Remove Overlap in Table 4.9.

r_1^D	0∗0∗
r_2^A	10∗0
$r_3'^D$	111∗
$r_3'^D$	1011
r_4^D	1∗00
$r_5'^A$	1001
$r_{6,1}''^A$	1101
$r_{6,2}'^A$	0111
$r_{7,1}'^A$	0∗10
$r_{7,2}'^A$	1010
r_{def}^D	∗∗∗∗

Table 4.13: Sort Table 4.12 in order of decreasing weight.

Filter $\mathcal{R}'$		$\lvert E(\mathcal{R}, i)\rvert$
$r_{6,1}'^A$	0∗10	35
$r_{5,2}'^A$	0111	25
r_1^D	0∗0∗	24
r_2^D	1∗1∗	24
r_3^D	1∗00	16
$r_{4,1}'^A$	1001	13
$r_{5,1}''^A$	1101	10
r_{def}^D	∗∗∗∗	10
$L(\mathcal{R}', \mathcal{F}) = 551$		

Table 4.14: Delete r_3 from Table 4.13.

Filter $\mathcal{R}'$		$\lvert E(\mathcal{R}, i)\rvert$
$r_{6,1}'^A$	0∗10	35
$r_{5,2}'^A$	0111	25
r_1^D	0∗0∗	24
r_2^D	1∗1∗	24
$r_{4,1}'^A$	1001	13
$r_{5,1}''^A$	1101	10
r_{def}^D	∗∗∗∗	26
$L(\mathcal{R}', \mathcal{F}) = 534$		

Table 4.15: Removed Deny rule from Table 4.14.

Filter $\mathcal{R}'$		$\lvert E(\mathcal{R}, i)\rvert$
$r_{6,1}'^A$	0∗10	35
$r_{5,2}'^A$	0111	25
$r_{4,1}'^A$	1001	13
$r_{5,1}''^A$	1101	10
r_{def}^D	∗∗∗∗	26
$L(\mathcal{R}', \mathcal{F}) = 460$		

Algorithm 22: Rulelist_Reconstruction($\mathcal{R}, \mathcal{F}$).

input : Rule list $\mathcal{R}$, Packet distribution $\mathcal{F}$

output: Rule list $\mathcal{R}'$

$i = 1$;

List $L \leftarrow \mathcal{R}$ except default rule r_{def};

while $i =$ *size of* $L - 1$ **do**

    r_i adds to L';

    for $j = i + 1$ *to size of* L **do**

        relation $\leftarrow$ RelationCheck(r_i, r_j);

        if *relation* $= O$ **then**

            $S \leftarrow$Separate(r_i, r_j);

            S add to L';

        end

        else if *relation* $= N$ **then**

            r_j adds to L';

        end

    end

    $L \leftarrow L'$ $i \leftarrow i + 1$;

end

$\mathcal{R}' \leftarrow L$;

Default rule adds to $\mathcal{R}'$;

$\mathcal{R}' \leftarrow$ OAO_Greedy($\mathcal{R}', \mathcal{F}$);

$W = 0$;

$defweight = |E(\mathcal{R}', n)|$;

for $i =$ *size of* $\mathcal{R}' - 1$ *to* 1 **do**

    if *the action of* r_i *is Deny* **then**

        if $W - (n - 1 - i)|E(\mathcal{R}', i)| + defweight \geq 0$ **then**

            $defweight \leftarrow defweight + |E(\mathcal{R}', i)|$;

            r_i remove from $\mathcal{R}'$;

        end

        else

            $W \leftarrow W + |E(\mathcal{R}', i)|$;

        end

    end

    else

        $W \leftarrow W + |E(\mathcal{R}', i)|$;

    end

end

return 0;

Algorithm 23: RelationCheck(r_i, r_j).

input : Rule r_i, Rule r_j
output: String C, O or N
iscovered $\leftarrow$ *true*;
for $k = 0$ *to bitlength of* r_i **do**
  $b_{ik} \leftarrow k$th bit of r_i;
  $b_{jk} \leftarrow k$th bit of r_j;
  if $b_{ik} \neq b_{jk}$ **then**
    if $b_{jk} = *$ **then**
      iscovered $\leftarrow$ *false*;
    end
    else if $b_{ik} \neq *$ **then**
      return N;
    end
  end
end
if *iscovered* **then**
  return C;
end
else
  return O;
end

Chapter 5

Conclusion

In this paper, we describe the problem of acceleration of packet classification using rule lists and present the computational complexity of the problem and several methods to solve it.

First, the paper shows the reduction from **XC3** to **RORO** via **RAO**. This clarified the computational complexity of the optimal rule ordering problem and theoretically demonstrated the difficulty of constructing a polynomial algorithm and the necessity of a heuristic method as well.

In addition, we proposed methods to deal with the following problems in solving the optimal rule ordering problem.

- In SGM, when deciding which rule to place in the next sorted list, we have been focusing on the rules that are directly related to the preceding rules in order, but we proposed a method to decide which rule to focus on next from the set of rules reachable from that rule.

- In SGM, only the rules that are required to place the rule under focus in the sorted list are considered, but it is not possible to consider the rules that can be placed by placing the rule in the sorted list. Therefore, we proposed the method $\mathcal{O}(n^3)$, which can find an order of rules with lower latency than SGM by considering not only the rules reachable from the rules but also the rules that are dependent on those rules.

- When the method of Hikage et al. constructs a list for each connected component of a dependent relation, the method with time complexity $\mathcal{O}(n^2)$ constructs the list using only weights of single rules. Therefore, we proposed a method to find the order of rules with lower latency while maintaining $\mathcal{O}(n^2)$ by constructing the list using the average weights of the rules to which the rules are directly dependent, instead of the weights of the single rules.

- Most of the heuristic methods for **RORO** trace the precedence constraints and reorder the rules using reachable weighted averages. However, there are some orderings that cannot be taken into account by the weighted average-based rule reordering method. Therefore, we proposed a method to find ordering with lower latency by using the difference in latency.

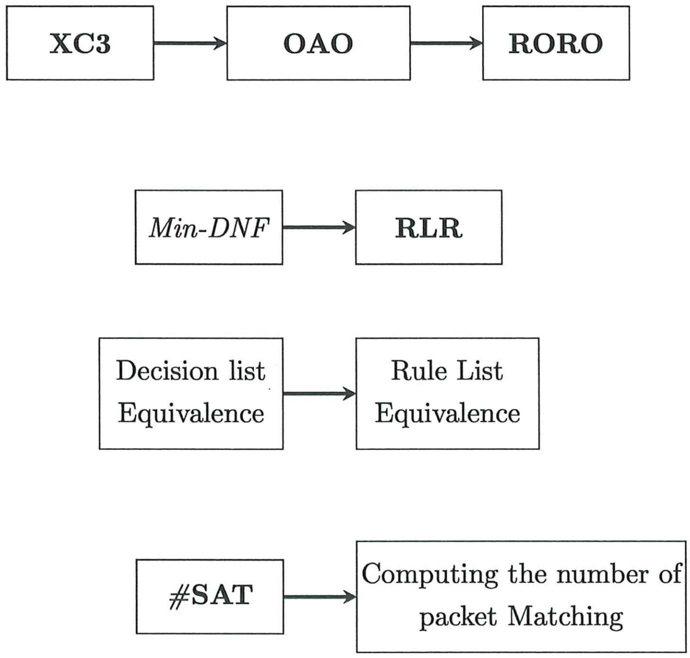


Figure 5.1: The Reduction Relation.

- We proposed a method that aims to reduce the number of packet comparisons by searching the rules that have no matching packets and placing them lower than the default rules.
- We proposed a method to find an order of rules with lower latency by relaxing the precedence constraint by removing dependencies that no longer affect the policy due to rules placed higher in the order.
- We proposed a heuristic method for solving the rule-order optimization problem for Allow lists.

We also showed a reduction from *Min-DNF* to **RLR**, which demonstrates the computational complexity of the optimal rule list problem and the importance of heuristics for this problem. We show the relation between the reduction of the optimization problems for acceleration of packet classification using rule lists, including the problem presented in this paper, and the results in Figure 5.1.

In addition, we proposed two methods for solving the rule list optimization problem: a consensus-based method for OAL and a general rule list reconstruction method that removes the dependent relations. In the consensus-based method, the Allow list is regarded as a logical formula, and by constructing a maximal rule, one rule is able to match more packets and obtain an Allow list with lower latency. In the consensus-based method, the Allow list is regarded as a logical formula, and by constructing a maximal rule, one rule is able to match more packets and obtain an Allow list with lower latency.

In Appendix A, we proposed a method to solve the rule list equivalence decision problem using the SAT solver by transforming it into a satisfiability problem.

With this method, we obtained a problem related to the optimization problem of acceleration of packet classification using rule lists.

The issues related to the optimal rule ordering problem are described below.

- There are cases in which SGM reduces latency when the dependencies become more complex. In such cases, the $\mathcal{O}(n^3)$ method is required to find an order of rules with lower latency.

- When swapping rules that are overlap relations but not dependencies, weight fluctuations occur. This can result in a sequence of rules with lower latency since the number of matching packets increases when a rule with a lower weight is placed higher. However, existing methods of rule reordering cannot take such an ordering into account.

- For finding rules with no matching packets and removing dependencies that do not affect the policy, we proposed a method that uses the SAT solver and a method that operates in polynomial time. The method using a solver has a time complexity that is exponential to the number of rules, so it cannot finish its operation in a realistic time as the number of rules increases. The polynomial-time method is less accurate than the solver-based method. Therefore, there is a need to propose a method that operates in polynomial time with the same level of accuracy as the method using the SAT solver.

We present issues for the rule list optimization problem. In the rule list reconstruction method that removes dependencies from the input rule list, the rule with the largest number of matched packets is placed on top of the rule list, resulting in a rule list with smaller latency. However, since the number of rules increases due to the removal of dependencies, the latency cannot be reduced sufficiently when the number of packets that match the default rules increases. Future work is needed to develop a method to find a rule list with lower latency while maintaining fewer rules. Furthermore, we need to develop the method that is able to find the rule list with minimum latency.

Allow list reconstruction using the consensus method is exponential in space computation with respect to the number of bits, so the operation does not terminate in realistic time for allow lists with complex policies. Therefore, it is necessary to develop a polynomial-time algorithm that can find the maximal rule.

Harada et al. proposed a rule list equivalence decision using data structures such as ZDD, which is faster than the rule list equivalence decision method in Appendix A. However, the decision time and the amount of memory required for rule list equivalence judgment using ZDD increase as the number of rules increases. Therefore, devising a faster rule list equivalence decision method is a future issue.

Acknowledgment

I would like to express my sincere gratitude to my supervisor, Professor Ken TANAKA. I am grateful to Dr. Kenji Mikawa from Informatics Biotechnology Engineering, Maebashi Institute of Technology for his assistance. My sincere thanks to Professors Leo NAGAMATSU and Haruhiko KAIYA from the Field of Information Sciences, Graduate School of Science, Kanagawa University for their helpful comments. I am grateful to Dr. Takashi HARADA from the School of Information, Kochi University of Technology for his assistance. Finally, I would like to thank the members of the TANAKA laboratory for their support.

Appendix A

Deciding Equivalence of Rule Lists

This chapter describes the rule list equivalence problem. In **ORO**, the policy equivalence between the reordered rule list and the input rule list is determined by the overlap relation. This chapter describes the rule list equivalence problem. In **ORO**, the policy equivalence between the reordered rule list and the input rule list is determined by the overlap relation. If the number of rules is n and the number of bits in a rule is l, the rule list policy equivalence decision in **ORO** is $O(n^2 l)$. However, the rule list policy equivalence decision in reordering and rule list optimization problems that do not depend on overlap relations or dependency relations is more complicated. In general, given two decision lists, the problem of determining their equivalence is known to be **coNP**-complete. Since the equivalence decision of a decision list can be attributed in polynomial time to the equivalence decision of a rule list, the equivalence decision of two rule lists $\mathcal{R}_1$ and $\mathcal{R}_2$ is also **coNP**-complete.

Thus, it is generally difficult to determine in polynomial time whether a policy violation has occurred when the rules are reordered or the rule list is reconstructed. On the other hand, for relatively small rule lists, it is possible to determine whether policy violations have occurred by constructing logical expressions corresponding to each rule and transforming them into satisfiability problems. In the following, we show how to construct a logic formula corresponding to a rule list and propose a method for determining the equivalence of rule list policies.

In A.1, we explain how to convert a rule list into a logical expression. In A.2, we explain the transformation to a satisfiability problem and propose a method for constructing instances to be input to the SAT solver. Finally, in A.4, we summarize and discuss future issues.

A.1 Transformation into a Satisfiability Problem

When determining policy equivalence, we focus on packets to which the Allow action is applied. The packets that match the rule r_i of Allow can be expressed as follows, where each rule is regarded as a propositional variable.

$$\neg r_1 \land \neg r_2 \land \cdots \land \neg r_{i-1} \land r_i \tag{A.1}$$

Table A.1: Rule list $\mathcal{R}$.

Filter $\mathcal{R}$	$\lvert E(\mathcal{R},i)\rvert_{\mathcal{F}}$
r_1^D =0*00	4
r_2^A =*100	4
r_3^A =0*01	15
r_4^A =01**	5
r_5^D =00*1	20
r_6^D =*1*1	40
r_7^D =**10	52
r_8^A =**11	45
r_9^A =10*0	60
r_{10}^D =****	10
$L(\mathcal{R},\mathcal{F})=1761$	

Table A.2: Reconstructing $\mathcal{R}$.

Filter $\hat{\mathcal{R}}$	$\lvert E(\hat{\mathcal{R}},i)\rvert_{\mathcal{F}}$
r_1^A =1011	45
r_2^D =1*1*	64
r_3^A =1*00	64
r_4^D =001*	31
r_5^D =1*01	27
r_6^A =0*01	15
r_7^D =****	2
$L(\hat{\mathcal{R}},\mathcal{F})=726$	

Table A.3: The packet arrival distaribution $\mathcal{F}:\mathcal{P}\to\mathbb{N}$

$0000\mapsto 1$	$0001\mapsto 8$	$0010\mapsto 11$	$0011\mapsto 20$
$0100\mapsto 3$	$0101\mapsto 7$	$0110\mapsto 5$	$0111\mapsto 0$
$1000\mapsto 60$	$1001\mapsto 10$	$1010\mapsto 2$	$1011\mapsto 45$
$1100\mapsto 4$	$1101\mapsto 17$	$1110\mapsto 39$	$1111\mapsto 23$

However, in each propositional variable, 1 indicates that the corresponding rule is satisfied and 0 indicates that it is not. Such a formula is constructed for all rules with Allow action, and then they are combined by logical OR to form a formula for whether or not each packet is applied with Allow action. At this time, using the distribution rule, the formula can be shortened as follows when $i<j$.

$$\begin{aligned}&(\neg r_{i-1}^D\wedge r_i^A)\vee(\neg r_{i-1}^D\wedge\neg r_i^A\cdots\neg r_{j-1}^D\wedge r_j^A)\\&=(\neg r_{i-1}^D\wedge r_i^A)\vee(\neg r_{i+1}^D\wedge\cdots\neg r_{j-1}^D\wedge r_j^A)\end{aligned}\tag{A.2}$$

To determine whether a packet matches a rule, the conditions of the rule are converted to logical expressions. For each bit in the rule, a propositional variable with the corresponding bit number is concatenated with the logical conjunction of its negation if the variable is 1 or 0, resulting in a clause for each rule. For example, the condition for r_2 in Table A.1 is $b_2\wedge\neg b_3\wedge\neg b_4$ and the condition for r_4 is $\neg b_1\wedge b_2$.

From (A.1) and (A.2), it can be seen that, in each rule in the rule list, by negating the clause corresponding to the rule with the action of *Deny* and combining them by logical conjunction, it is a logical expression to determine whether the variable assignment corresponding to a packet is adapted to Allow. This is a logical expression that determines whether or not the variable assignment corresponding to the packet is applicable to Allow. The logical expression $T_{\mathcal{R}}$ that determines whether a packet is given the Allow action in the rule list in Table A.1 is as follows.

$$\begin{aligned} T_{\mathcal{R}} =& \neg(\neg b_1 \wedge \neg b_3 \wedge \neg b_4) \wedge (b_2 \wedge \neg b_3 \wedge \neg b_4) \wedge (\neg b_1 \wedge \neg b_3 \wedge b_4) \wedge (\neg b_1 \wedge b_2) \\ & \wedge \neg(\neg b_1 \wedge \neg b_2 \wedge b_4) \wedge \neg(b_2 \wedge b_4) \wedge \neg(b_3 \wedge \neg b_4) \wedge (b_3 \wedge b_4) \wedge (b_1 \wedge \neg b_2 \wedge \neg b_4) \end{aligned} \tag{A.3}$$

Using the distribution rule for logical expressions, the shortening is as follows.

$$\begin{aligned} T_{\mathcal{R}} =& (\neg(\neg b_1 \wedge \neg b_3 \wedge \neg b_4) \wedge b_2 \wedge \neg b_3 \wedge \neg b_4) \vee (\neg b_1 \wedge \neg b_3 \wedge b_4) \vee (\neg b_1 \wedge b_2) \\ & \vee (\neg(\neg b_1 \wedge \neg b_2 \wedge b_4) \wedge \neg(b_2 \wedge b_4) \wedge (b_3 \wedge \neg b_4) \wedge b_3 \wedge b_4) \vee (b_1 \wedge \neg b_2 \wedge \neg b_4) \end{aligned} \tag{A.4}$$

A.2 Determination using SAT Solver

Whether the policies in the rule list $\mathcal{R}_1$ and $\mathcal{R}_2$ are equivalent or not corresponds to whether $T_1 \equiv T_2$ is constant true or not. In other words, if $(T_1 \vee T_2) \wedge (\neg T_1 \vee \neg T_2)$ is unsatisfiable, we can show that $\mathcal{R}_1$ and $\mathcal{R}_2$ are equivalent.

The satisfiability problem is the problem of determining whether there exists an assignment of values to variables such that all constraints are satisfied when the set of variables, the domain of each variable, and the set of constraints among variables are input-solved.

In this study, we use Sugar [51,52], a SAT-type constraint solver, to encode the constructed logical expressions into satisfiability problems. Since Sugar has a predicate iff, there is no need to convert the formula to $(T_1 \vee T_2) \wedge (\neg T_1 \vee \neg T_2)$. Also, since Sugar does not require the input logic formula to be a CNF, there is no need to convert the logic formula created from the rule list to the CNF required for the input of a general SAT solver.

A.3 Time Complexity of Proposed Method

If the number of rules is n and the bit length is w, when constructing a logical expression, the computational complexity to construct a logical expression from a single rule is $O(w)$ when all bit values are 0 or 1. Since this is constructed for all rules, the computational complexity to convert the conditions of all rules into a logical expression is $O(wn)$. These are combined according to (A.1) and (A.2). Therefore, the computational complexity of the method to construct the logic equation to be input to Sugar is $O(wn)$.

A.4 Conclusion

The rule list equivalence problem essentially corresponds to a search for the action to which packet p applies, and to a search for which rule in the rule list the packet p matches. Calculating the number of packets matching a rule is generally a difficult problem because it is exponential in the number of bits when done accurately. However, in the optimal rule ordering problem, reordering with weight fluctuation requires a more accurate match frequency for each rule in each order. To address this problem, Misherghi's method constructs a packet space. A packet space is a set of packets that can match the same rule. The packet space is constructed rapidly by using data structures based on ZDDs and BDDs and can be considered as an applied version of the solution method for rule list equivalence decisions using ZDDs proposed by Harada et al. In the optimal rule list problem, if it is possible to calculate the frequency of packet matching of the generated rules, it will be possible to determine whether the rules contribute to latency reduction or not. Thus, research on the rule list equivalence decision problem is an important problem for the rule list optimization problem.

Appendix B

Policy equivalence determination for multi-valued rule lists

In this paper, we present an equivalence decision algorithm for a rule list policy consisting of only two rules, A and D, that apply to packets. Here, we propose an equivalence decision algorithm for a rule list policy consisting of rules that do not limit actions to only A and D, as shown in Table B.1. Hereafter, rule lists whose actions are not restricted to P and D are referred to as the multi-valued rule lists. For a multi-valued rule list in which there are three or more candidates of actions to be applied to a packet, such as $A_1, A_2, A_3, \cdots, A_m$, we construct logical formulas as described in the previous section for the number of actions. For a multi-valued rule list, construct logical expressions $T_1, T_2, \cdots, T_m$ and compare whether the actions are consistent with A_i.

For the rule lists $\mathcal{R}_1$ and $\mathcal{R}_2$, $T_{11}, T_{12}, \cdots, T_{1m}$ and $T_{21}, T_{22}, \cdots, T_{2m}$, $2m$ logical expressions are constructed and m satisfiability problems $(T_{1i} \vee T_{2i}) \wedge (\neg T_{1i} \vee \neg T_{2i})$ is unsatisfiable, we can determine the policy equivalence of $\mathcal{R}_1$ and $\mathcal{R}_2$.

The computational complexity of the construction is $O(mwn)$, because m, the number of actions, is required by the logical formula in the previous section.

Table B.1: multi-valued rule lists $\mathcal{R}$

Filter $\mathcal{R}$
r_1^A =1011
r_2^D =1$*$1$*$
r_3^B =1$*$00
r_4^C =001$*$
r_5^A =1$*$01
r_6^B =0$*$01
r_7^D =$****$

Appendix C

Lower and Upper bounds for Rule List Latency

For an Allow list $\mathcal{R}$, the upper bound of the latency of an Allow list $W_{K+1} \equiv \mathcal{R}$ with size K is $\overline{L(W_K)}$ and that of an Allow list $B_K \equiv \mathcal{R}$, and the lower bound of the latency of the Allow list of size K is $L(B_K)$. We show that the order of the Allow lists with the minimal latency is in descending order of weight.

Lemma C.0.1. *Let $\mathcal{R}$ be an Allow list such that the weight of the default rule r^e_{n+1} is 0 and the weights of other rules are at least 1. Let w_i be the weight of the i-th rule in the rule list $\mathcal{R}_\sigma$ under the order σ and frequency distribution $\mathcal{F}$, then for the sequence σ of rules with the minimal latency in the Allow list $\mathcal{R}$, we have $w_1 \geq w_2 \geq \dots w_n > w_n + 1$.*

Proof. We prove this by contradiction. Let τ be an order and v_i be the ith rule weight of the Allow list $\mathcal{R}_\tau$ in the frequency distribution $\mathcal{F}$. Assume that $v_1 \geq v_2 \geq \dots \geq v_{n+1}$ not order τ minimizes the latency of the rule list.

The latency in the frequency distribution $\mathcal{F}$ of $\mathcal{R}_\tau$ is

$$\sum_{i=1}^{n} iv_i$$

because the weight of the default rule is zero. By the assumption, there exists $K(1 \leq i < n)$ such that $v_k < v_{k+1}$. Since the actions of all rules except the default rule are the same, the rules $r_{\tau^{-1}(k)}$ and $r_{\tau^{-1}(k+1)}$ are interchangeable and the reordering of these rules The latency of the rule is

$$\sum_{i=1}^{k-1} iv_i + kv_{k+1} + (k+1)v_k + \sum_{i=k+2}^{n+1} iv_i.$$

Since $v_k < v_{k+1}$, $kv_{k+1} + (k+1)v_k < kv_k + (k+1)v_{k+1}$, the latency of the rule list reordered by rules $r_{\tau^{-1}(k)}$ and $r_{\tau^{-1}(k+1)}$ is lower than that of the rule. This is inconsistent with the fact that the latency of the rule list sorted by τ is the lowest. $\square$

Thus, the Lemma C.0.1 is true.

Definition C.0.1. *The Allow list consisting of K rules and having no * rule list*

$$\mathcal{R} = [r_1^A, r_2^A, \ldots, r_n^A]$$

and representing the same policy is denoted as

$$S_K = [s_1^A, s_2^A, \ldots, s_K^A].$$

From the Lemma C.0.1, we do not lose generality by assuming the above.

Definition C.0.2. . *The list of the number of packets (weights) evaluated for each rule in the rule list $S_K = [s_1^A, s_2^A, \ldots, s_K^A]$ is denoted by*

$$\overline{S_K} = [\overline{s_1^A}, \overline{s_2^A}, \ldots, \overline{s_K^A}].$$

Because of this, the latency of the rule list S_K is

$$L(S_K, \mathcal{F}) = \sum_{i=1}^{K} i\overline{s_i}.$$

Since rule s_1 is the first rule in the rule list, its weight is equal to $|M(s_1)|$, the number of packets that can match s_1. This makes $\overline{s_1}$ a power of 2.

First, a lower bound on the latency of the Allow list is shown.

Theorem C.0.1. *The lower bound on the latency of the Allow list S_K is $2n + \frac{(K-2)(K-1)}{2} - 2^l$. Where $l = max\{l \in \mathbb{N} | 2^l \leq n - K + 1\}$.*

Proof. Let $\mathcal{T}_K$ is an Allow list where the weight $\overline{t_1}$ of rule t_1 is 2^l, the weight $\overline{t_2}$ of rule t_2 is $n - \overline{t_1} - (K-2)$, $\overline{t_3}, \overline{t_4}, \ldots, \overline{t_K}$ is 1. Where, $l = \max\{l \in \mathbb{N} | 2^l \leq n - K + 1\}$. Then, the latency $L(\mathcal{T}_K, \mathcal{F})$ of $\mathcal{T}_K$ is as follows.

$$\begin{aligned}
L(\mathcal{T}_K, \mathcal{F}) &= \sum_{i=1}^{K} i\overline{t_i} \\
&= \overline{t_1} + 2\overline{t_2} + \sum_{i=3}^{K} i\overline{t_i} \\
&= \overline{t_1} + 2(n - \overline{t_1} - (K-2)) + \sum_{i=3}^{K} i \\
&= \overline{t_1} + 2(n - \overline{t_1} - (K-2)) + \sum_{i=1}^{K} i - 3 \\
&= \overline{t_1} + 2n - 2\overline{t_1} - 2K + 4 + \sum_{i=1}^{K-2} i + (K-1) + K - 3 \\
&= \overline{t_1} + 2n - 2\overline{t_1} - 2K + 4 + \sum_{i=1}^{K-2} i + (K-1) + K - 3 \\
&= \overline{t_1} + 2n - 2\overline{t_1} + \sum_{i=1}^{K-2} i \\
&= \overline{t_1} + \frac{(K-1)(K-2)}{2} + 2n - 2\overline{t_1}
\end{aligned} \tag{C.1}$$

We show Theorem C.0.1 by proving that there is no Allow list whose latency is smaller than $\mathcal{T}_K$.

Show that for any Allow list S_K, the following holds.

$$\sum_{i=1}^{K} i\overline{t_i} \leq \sum_{i=1}^{K} i\overline{s_i}$$

Thus, we show the following.

$$(\overline{t_1} - \overline{s_1}) + 2(\overline{t_2} - \overline{s_2}) \leq \sum_{i=3}^{K} i(\overline{s_i} - \overline{t_i})$$

Where

$$\sum_{i=1}^{K} i\overline{t_i} = \sum_{i=1}^{K} i\overline{s_i}.$$

From the above, it follows that

$$(\overline{t_1} - \overline{s_1}) + 2(\overline{t_2} - \overline{s_2}) = \sum_{i=3}^{K} i(\overline{s_i} - \overline{t_i}).$$

As a result, we show the following.

$$(\overline{t_2} - \overline{s_2}) \leq \sum_{i=1}^{K} (i-1)(\overline{s_i} - \overline{t_i})$$

If $(\overline{t_2} - \overline{s_2}) \leq 0$, the weight of each rule $|E(S_K, i)| \geq 1$, so the formula (C) clearly holds. If $(\overline{t_2} - \overline{s_2}) > 0$, we show that the formula (C) holds.

Let the set of subscripts in $(\overline{s_3} - \overline{t_3}), (\overline{s_4} - \overline{t_4}), \ldots (\overline{s_K} - \overline{t_K})$ where the difference is more than 1 be $\mathcal{I}$ is as follows.

$$\begin{aligned}(\overline{t_2} - \overline{s_2}) &\leq \sum_{i \in \mathcal{I}} (i-1)(\overline{s_i} - \overline{t_i}) \\ &= \sum_{i \in \mathcal{I}} (\overline{s_i} - \overline{t_i}) + \sum_{i \in \mathcal{I}} (i-2)(\overline{s_i} - \overline{t_i})\end{aligned} \tag{C.2}$$

Thus, it is sufficient to show the following.

$$(\overline{t_2} - \overline{s_2}) \leq \sum_{i \in \mathcal{I}} (\overline{s_i} - \overline{t_i}) + \sum_{i \in \mathcal{I}} (i-2)(\overline{s_i} - \overline{t_i})$$

where $\overline{t_2} - \overline{s_2} = \sum_{i \in \mathcal{I}} (\overline{s_i} - \overline{t_i})$ and at i greater than 3, $\overline{s_i} \geq 1$, $\overline{t_i} = 1$, the inequality (C) holds. $\square$

Then, we show the upper bound on the latency of the Allow list S_K.

Theorem C.0.2. *The upper bound on the latency of the Allow list S_K is $2^l + \frac{q(K+2)(K-1)}{2} + \frac{x(x+3)}{2}$. Where, $l = min\{l|2^l \geq \frac{n}{K}\}$ and q and x are natural numbers satisfying the following equations.*

$$n - 2^l = (K-1)q + x \qquad (K - 1 > x)$$

Proof. We assume that U_K is an Allow list where the rule u_1 weight $\overline{u_1}$ is 2^l, $\overline{u_2}, \overline{u_3}, \ldots, \overline{u_{x+1}}$ is $q+1$, and $\overline{u_{x+2}}, \ldots, u_K$ is q. Where $l = \min\{l|2^l \geq \frac{n}{K}\}$ and q and x are natural numbers satisfying the following equations.

$$n - 2^l = (K-1)q + r \qquad (K - 1 > r)$$

The latency of U_K is as follows.

$$\begin{aligned} L(U_K, \mathcal{F}) &= \sum_{i=1}^{K} i\overline{u_i} = \overline{u_1} + \sum_{i=2}^{r+1} i\overline{u_i} + \sum_{i=r+2}^{K} i\overline{u_i} \\ &= 2^l + \sum_{i=2}^{r+1} i(q+1) + \sum_{i=r+2}^{K} iq \\ &= 2^l + \sum_{i=2}^{K} iq + \sum_{i=2}^{x+1} i \\ &= 2^l + \frac{q(K+2)(K-1)}{2} + \frac{x(x+3)}{2} \end{aligned} \tag{C.3}$$

We show that there is no Allow list whose latency is greater than U_K, and we show that the theorem C.0.2. For any Allow list S_K, we show the following holds.

$$\sum_{i=1}^{K} i\overline{s_i} \leq \sum_{i=1}^{K} i\overline{u_i}$$

If $\overline{s_i} = \overline{u_i}$ for any i, then the inequality (C) holds. If there exists $h > 1$ such that $\overline{s_h} < \overline{u_h}$, then $\overline{u_i} - \overline{u_{i+1}} \leq 1$ for any 2 or more i and $\overline{s_j} \geq \overline{s_{j+1}}$ for any j, the following two conditions that are

$$\overline{s_1} \geq \overline{u_1}, \overline{s_2} \geq \overline{u_2}, \ldots, \overline{s_{h-1}} \geq \overline{u_{h-1}}$$

and

$$\overline{s_{h+1}} \leq \overline{u_{h+1}}, \overline{s_{h+2}} \leq \overline{u_{h+2}}, \ldots, \overline{s_K} \leq \overline{u_K}$$

are satisfied. Let $\mathcal{I}$ be the set of subscripts such that $\overline{s_h} > \overline{u_h}$ with $1 \leq i \leq h-1$, $\mathcal{I}$ be the set of subscripts such that $\overline{s_h} < \overline{u_h}$ with $h \leq j \leq K$. The following inequalities are shown by assuming $\mathcal{J}$.

$$\begin{aligned} \sum_{i \in \mathcal{I}} i\overline{s_i} + \sum_{j \in \mathcal{J}} j\overline{s_j} &\leq \sum_{i \in \mathcal{I}} i\overline{u_i} + \sum_{j \in \mathcal{J}} j\overline{u_j} \\ \sum_{i \in \mathcal{I}} i(\overline{s_i} - \overline{u_i}) &\leq \sum_{j \in \mathcal{J}} j(\overline{u_j} - \overline{s_j}) \end{aligned} \tag{C.4}$$

Then let $\mathcal{Y}$ be the set of subscripts such that $\overline{s_y} = \overline{u_y}$ as follows.

$$\begin{gathered}
\sum_{i\in[1...K]} \overline{s_i} = \sum_{i\in[1...K]} \overline{u_i} \\
\sum_{i\in\mathcal{I}} \overline{s_i} + \sum_{j\in\mathcal{J}} \overline{s_j} + \sum_{y\in\mathcal{Y}} \overline{s_y} = \sum_{i\in\mathcal{I}} \overline{u_i} + \sum_{j\in\mathcal{J}} \overline{u_j} + \sum_{y\in\mathcal{Y}} \overline{u_y} \\
\sum_{i\in\mathcal{I}} \overline{s_i} + \sum_{j\in\mathcal{J}} \overline{s_j} = \sum_{i\in\mathcal{I}} \overline{u_i} + \sum_{j\in\mathcal{J}} \overline{u_j} \\
\sum_{i\in\mathcal{I}} (\overline{s_i} - \overline{u_i}) = \sum_{j\in\mathcal{J}} (\overline{u_j} - \overline{s_j})
\end{gathered} \tag{C.5}$$

Thus, the following holds.

$$(h-1)\sum_{i\in\mathcal{I}} (\overline{s_i} - \overline{u_i}) \leq h \sum_{j\in\mathcal{J}} (\overline{u_j} - \overline{s_j}) \tag{C.6}$$

As a result, The inequality(C) is satisfied by the following two formulas that are

$$\sum_{i\in\mathcal{I}} i(\overline{s_i} - \overline{u_i}) \leq (h-1)\sum_{i\in\mathcal{I}} (\overline{s_i} - \overline{u_i}) \tag{C.7}$$

and

$$h\sum_{j\in\mathcal{J}} (\overline{u_j} - \overline{s_j}) \leq \sum_{j\in\mathcal{J}} j(\overline{u_j} - \overline{s_j}) \tag{C.8}$$

□

Appendix D

Research Achievement

D.1 Journals and Transactions

1. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "Acceleration of Packet Classification Using the Difference of Latency,"IPSJ Journal,Vol. 64, No. 9, pp. 1217-1226,Sep.,2023(in Japanese)

2. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "Computational Complexity of Allow Rule Ordering and Its Greedy Algorithm,"IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E106-A, No.9, pp.1111-1118,Sep., 2023

3. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "A Rule Reordering Method via Deleting Pre-Constraints that do not Affect Policy," B - Abstracts of IEICE TRANSACTIONS on Communications (Japanese Edition) ,Vol. J104-B, No.10, pp.783-791, Jul., 2021

4. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "A Rule Reordering Method via Dependent Subgraph Enumeration,"D - Abstracts of IEICE TRANSACTIONS on Information and Systems (Japanese Edition),Vol. J103-D, No.4, pp.228-237, Apr., 2020

D.2 Conference proceedings

1. T. Fuchino, T. Harada, K. Tanaka, "Accelerating Packet Classification via Direct Dependent Rules," 12th International Conference on Network of the Future(NoF2021) , Oct. 06 - 08, 2021

2. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "Acceleration of Packet Classification Using Adjacency List of Rules,"The 28th International Conference on Computer Communication and Networks(ICCCN2019), Jul. 29 - Aug. 1, 2019

D.3 Technical Reports

1. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "A Rule Reordering Method via Deleting Dependencies Unaffecting the Policy,"Forum on Information Technology,Vol. 2020-09, No. 19, pp. 61-62, Sep., 2020

2. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "Deciding Equivalence of The Rule List Policies via SAT solver," IEICE Technical Report, Institute of Electronics, Information and Communication Engineers, vol. 119, no. 329, pp. 13-19,Dec.,2019

3. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "A Rule Reordering Method via Deleting 0 Weights Rules," IEICE Technical Report, Institute of Electronics, Information and Communication Engineers,vol. 119, no. 249, pp. 47-52,Oct.,2019

4. T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, "A Reordering Method via Rules Pairing based on Average Weights," IEICE Technical Report, Institute of Electronics, Information and Communication Engineers,Vol.118, No.295, pp.31-36, 15,Nov.,2018

Bibliography

[1] Thilan Ganegedara, Weirong Jiang, and Viktor K Prasanna. A scalable and modular architecture for high-performance packet classification. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1135–1144, 2013.

[2] 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.

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

[4] 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.

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

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

[7] Wenjun Li, Dagang Li, Yongjie Bai, Wenxia Le, and Hui Li. Memory-efficient recursive scheme for multi-field packet classification. *IET Communications*, 13(9):1319–1325, 2019.

[8] 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.

[9] 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.

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

[11] 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 International Conference on*, pages 1–6, Aug 2009.

[12] 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.

[13] 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).

[14] 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.

[15] 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).

[16] 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.

[17] Ryosuke Fumiiwa and Toshinori Yamada. Exact algorithm for sorting rules in firewall. Technical Report 15, Technical Committee on Circuits and Systems (CAS), 2019.

[18] Vitalii Demianiuk, Kirill Kogan, and Sergey Nikolenko. Approximate packet classifiers with controlled accuracy. *IEEE/ACM Transactions on Networking*, 29(3):1141–1154, 2021.

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

[20] 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.

[21] 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.

[22] 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.

[23] Sorrachai Yingchareonthawornchai, James Daly, Alex X. Liu, and Eric Torng. A sorted partitioning approach to high-speed and fast-update openflow classification. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, pages 1–10, 2016.

[24] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 256–269. 2019.

[25] Yu-Hsiang Lin, Wen-Chi Shih, and Yeim-Kuan Chang. Efficient hierarchical hash tree for openflow packet classification with fast updates on gpus. *Journal of Parallel and Distributed Computing*, 167:136–147, 2022.

[26] Longlong Zhu, Jiashuo Yu, Jiayi Cai, Jinfeng Pan, Zhigao Li, Zhengyan Zhou, Dong Zhang, and Chunming Wu. Frod: An efficient framework for optimizing decision trees in packet classification. In *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2022.

[27] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2645–2653. IEEE, 2018.

[28] James Daly and Eric Torng. Bytecuts: Fast packet classification by interior bit extraction. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2654–2662. IEEE, 2018.

[29] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. Scaling open {vSwitch} with a computational cache. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1359–1374, 2022.

[30] Wenjun Li, Tong Yang, Yeim-Kuan Chang, Tao Li, and Hui Li. Tabtree: A tss-assisted bit-selecting tree scheme for packet classification with balanced rule mapping. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–8. IEEE, 2019.

[31] Yao Xin, Yuxi Liu, Wenjun Li, Ruyi Yao, Yang Xu, and Yi Wang. Kicktree: A recursive algorithmic scheme for packet classification with bounded worst-case performance. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pages 23–30, 2021.

[32] Y.Ishikawa, T.Harada, K.Tanaka, and K.Mikawa. Decision tree construction based on run-based tries with pointers. *B- IEICE TRANSACTIONS on Communications (Japanese Edition)*, 103(2):48–56, 2020.

[33] 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.

[34] Hasibul Jamil and Ning Weng. Multibit tries packet classification with deep reinforcement learning. In *2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. IEEE, 2020.

[35] 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.

[36] 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.

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

[38] Holden Gordon, Christopher Batula, Bhagyashri Tushir, Behnam Dezfouli, and Yuhong Liu. Securing smart homes via software-defined networking and low-cost traffic classification. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1049–1057. IEEE, 2021.

[39] James Daly, Valerio Bruschi, Leonardo Linguaglossa, Salvatore Pontarelli, Dario Rossi, Jerome Tollet, Eric Torng, and Andrew Yourtchenko. Tuplemerge: Fast software packet processing for online packet classification. *IEEE/ACM transactions on networking*, 27(4):1417–1431, 2019.

[40] Meiyi Yang, Deyun Gao, Chuan Heng Foh, Yajuan Qin, and Victor CM Leung. A learned bloom filter-assisted scheme for packet classification in software-defined networking. *IEEE Transactions on Network and Service Management*, 19(4):5064–5077, 2022.

[41] 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.

[42] Ken Tanaka and Suguru Itoh. Reconstruction method of filtering rules in order to mitigate load of network gears. *B-IEICE TRANSACTIONS on Communications (Japanese Edition)*, 88(5):905–912, 2005.

[43] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.

[44] Takashi Fuchino, Takashi Harada, Ken Tanaka, and Kenji Mikawa. Acceleration of packet classification using adjacency list of rules. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2019.

[45] Takashi Fuchino, Takashi Harada, Ken Tanaka, and Kenji Mikawa. A rule reordering method via dependent subgraph enumeration. *D - Abstracts of IEICE TRANSACTIONS on Information and Systems (Japanese Edition)*, J103-D(4):228–237, 04 2020.

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

[47] Takashi Fuchino, Takashi Harada, and Ken Tanaka. Accelerating packet classification via direct dependent rules. In *2021 12th International Conference on Network of the Future (NoF)*, pages 1–8, 2021.

[48] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[49] Takashi Harada and Ken Tanaka. Computational complexity of whitelist constructing. Technical Report 249, Technical Committee on Theoretical Foundations of Computing (COMP), Oct 2019.

[50] Arlene Fink, Jacqueline Kosecoff, Mark Chassin, and Robert H Brook. Consensus methods: characteristics and guidelines for use. *American journal of public health*, 74(9):979–983, 1984.

[51] Naoyuki Tamura and Mutsunori Banbara. Sugar: A csp to sat translator based on order encoding. *Proceedings of the Second International CSP Solver Competition*, pages 65–69, 2008.

[52] Naoyuki Tamura, Tomoya Tanjo, and Mutsunori Banbara. System description of a sat-based csp solver sugar. *Proceedings of the Third International CSP Solver Competition*, pages 71–75, 2008.