IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 18, 2018, accepted December 6, 2018, date of publication January 1, 2019, date of current version January 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2889194

A Novel Approach to Rule Placement in Software-Defined Networks Based on OPTree

WENJIE LI[1], ZHENG QIN[1], (Member, IEEE), KEQIN LI[2], (Fellow, IEEE), HUI YIN[3], AND LU OU[1], (Member, IEEE)

[1]College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China
[2]Department of Computer Science, State University of New York, New Paltz, NY 12561, USA
[3]College of Computer Engineering and Applied Mathematics, Changsha University, Changsha 410022, China

Corresponding author: Zheng Qin (zqin@hnu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grants 61472131, 61772191, and 61472132; in part by the Science and Technology Key Projects of Hunan Province under Grants 2015TP1004, 2016JC2012; in part by the Hunan Province Cooperative Innovation Center for the Construction & Development of Dongting Lake Ecological Economic Zone; and in part by the Science and Technology Projects of ChangSha City under Grants kq1804008, kq1801008.

ABSTRACT Software-defined networks (SDNs) are a trend of research in networks. Rule placement, a common SDN operation, becomes a challenging problem due to the capacity limitation of devices in which a large number of rules need to be deployed. Prior works mostly consider rule placement in a single device. However, the position relationships between neighbor devices also have influences on rule placement and should be considered. Our basic idea is to classify the devices position relationships into two categories: the serial relationship and the parallel relationship, and we present novel strategies for rule placement based on the two different position relationships. There are two challenges of implementing our strategies: to check whether a rule is contained by a rule set or not and to check whether a rule can be merged with other rules or not. To handle the challenges, we propose a novel data structure called OPTree to represent the rules, which is convenient to check whether a rule is covered by other rules. We design an insertion algorithm and a search algorithm for OPTree. Extensive experiments show that our approach can effectively reduce the number of rules while ensuring placed rules work. On the other hand, the experimental results also demonstrate that it is necessary to consider the position relationships between neighbor devices when placing rules.

INDEX TERMS Position relationship, rule placement, SDN.

I. INTRODUCTION

A. MOTIVATION AND PROBLEM STATEMENT

In Software-Defined Networks, there are a lot of devices to control the flow of data packets. *e.g.*, firewalls, switches, and routers. In these devices, there are many rules to control their functions. The network administrator changes the flow of data packets by modifying the rules in these devices. If the network administrator wants to prohibit/permit some data packets to go through the network, he creates a new rule, and places the rule into a suitable device. *rule placement* is an operation that chooses a suitable device and places a rule into it.

In general, network administrator simply places a rule into the first position of the rule set in the chosen device. However, the operation is constrained by the limited capacity of devices, especially when more and more devices use the Ternary Content Addressable Memories(*TCAMs*) to store the rules [1]–[3]. *TCAMs* have good performances at the cost of hight prices [4]. We cannot unlimitedly insert rules into a device. As the matter of fact, we can reduce the number of rules in a device when placing rules to improve the device performance. When a rule is placed into a device, if it's a redundant/conflicting one with other rules in the device, it could be removed. In addition, two rules might be merged into a new one to reduce the number of rules in the device. Therefore, reducing the number of rules as many as possible is a critical requirement when ensure the placed rule working for rule placement.

In this paper, we focus on the rule placement of access control list (ACL) in firewalls. We try to minimize the total number of rules in the SDN without altering the total effects of the rules.

The position relationship between neighbor devices is ignored in the existing researches. Obviously, the position relationships between neighbor devices have influences on

rule placement. For example, if a rule has been placed into a device, the rule may change the set of data packets which arrive at the next device according to the position relationship. The next device may have rules that are not applicable to the new set of packets. In this paper, we consider the influence of the position relationships between neighbor devices on rule placement.

B. TECHNICAL CHALLENGES

There are two key challenges for proposing an effective approach to rule placement. First, there are many complex relationships between the rules in a device [5], such as conflicts and redundances. When we place a rule, it is easy that the rule become a redundant/conflicting one for other rules and cannot work. Rule placement should ensure the placed rules can work in a device. It is hard to check whether a rule can work in a device or not. As a result, Applegate *et al.* [6] proved that the 2-D range-ACL compression problem is NP-hard and Kogan *et al.* [7] proved that the problem of the prefix-ACL rules compression with an arbitrary number of dimensions is an NP-hard problem. Second, there may be some rules that can be merged, so we can reduce the size of rules in devices by merging them. However, it is hard to find which rules can be merged, efficiently.

C. LIMITATION OF PRIOR WORK

Recently, there are some effective works on rule placement. Casado *et al.* [8] proposed an approach for distributing a centralized firewall policy by placing rules for packets at their ingress switches. Yuan *et al.* [9] presented a method that the edge switch configurations realize the firewall policy. However, these approaches, which do not enforce rule-table constraints on the edge switches or place rules on the internal switches, may make the load on ingress switches very heavy. DIFANE [10] and vCRIB [11] leveraged all switches in the network to enforce an endpoint policy. Specifically, DIFANE proposed a ''rule split and caching'' approach that increases the path length for the first packet of a flow. Kanizo *et al.* [12] presented the Palette distribution framework for decomposing large SDN tables into small ones and then distributing them across the network. Kang *et al.* [13] viewed the network as ''one big switch'' and proposed a heuristic rule placement algorithms that distribute forwarding policies across general SDN networks while managing rule space constraints. Nguyen *et al.* [14] proposed a novel approach for rule placement using trading routing. All these works focus on the rule placement of forwarding policies, which are deployed in router or switch. Li *et al.* [15] proposed a heuristic algorithm for rule placement which focused on the wired networks with dynamic topologies. Ashraf [16] presented the minimum rule application (MIRA), a mixed integer linear programming-based model, which re-calculates flow distribution dynamically while minimizing the number of rule installations, but the main concern is the rule minimization problem in a single device. Kannan *et al.* [17] proposed Raptor, a scalable rule placement scheme that supports multi-path routing as well as immediate failure-recovery to a backup path without policy violation. Chen and Lin [18] proposed the rule placement scheme by considering the tradeoff of TCAM space utilization and the bandwidth consumption in SDN networks.

Angelos *et al.* [19] proposed a novel placement algorithm, which dynamically decides whether a new flow rule should be placed in a hardware (expensive) or a software (cheap) table. The goal of the algorithm is to increase the utilization of the software-based table, without introducing performance degradation in the network in terms of significant delay and packet loss.

Similar to our solution, Zhang *et al.* [20] proposed an Integer Linear Programming (ILP) based solution for placing rules on switches for a given firewall policy. However, the work does not consider the influence of position relationship between neighbor devices on rule placement.

D. OUR APPROACH

In this paper, we propose a novel approach for rule placement. In our approach, we first take the relationships of neighbor devices into consideration when placing rules. We classify the relationship of neighbor devices into two categories: the serial relationship and the parallel relationship, and propose the rule placement strategies for both categories. To overcome the challenges of implementing our placement strategies, we propose a novel data structure called *OPTree* and also design the insertion algorithm and query algorithm for OPTree.

E. KEY CONTRIBUTIONS

In this paper, we extend and reinforce our work in [21] to expatiate our approach in more detail and further improve our approach. The main contributions of this paper are as follows:

(1) To the best of our knowledge, we first consider the influences of the position relationship between neighbor devices on rule placement. According to the real condition in network, we classify the position relationship between neighbor devices into the serial relationship and the parallel relationship.

(2) We propose the rule placement strategies for different position relationships, respectively.

(3) To overcome the challenges of implementing our strategies, we propose a novel data structure called OPTree to represent the rules in devices and design the insertion algorithm and the search algorithm for OPTree.

(4) We analyze the time complexity and conduct experiments to examine our approach.

Compare with our prior work, we have some changes in this paper as follows.

1) We define the problem of the rule placement and provide a formal definition of the problem of the rule placement.

2) We introduce the prefixes of rules and propose some operations of prefixes.

3) We analyze the factors influencing rule placement.

4) We propose the pseudo-code of the insertion algorithm of OPTree and search algorithm of OPTree, respectively.

F. PAPER ORGANIZATION

This paper is organized as follows: in Section II, we present the background and notations. We propose the principles of rule placement and define the problem of rule placement In Section III. In Section IV, we propose our approach in detail. To overcome the challenges of implementing our strategies, we propose a novel data structure called OPTree in Section V. In Section VI, to evaluate our approach, we perform experiments and discuss experimental results. We conclude our paper in Section VII.

II. BACKGROUND

A. SOFTWARE-DEFINED NETWORK

Software-Defined Network (SDN) is a novel network architecture that proposed by CleanSlate research group of Stanford University [22], [23]. Its goal is to achieve control of the hardware forwarding rules through software programming and finally achieve the purpose of a flow of free control. The architecture of SDN contains three layers: the device layer, the control layer, and the application layer. The rule placement is related with the control layer and the device layer.

Figure 1 shows a sketch of the device layer and control layer in SDN. There is at least one controller in the control layer, and there are many devices in the device layer. The rules in devices match every data packet arrives at the devices. If a rule matches a data packet in a device, the device will execute the action of the rule for the data packet, the data flow is composed of a set of data packets that go through the devices. The controller manages the rules in devices by the control flow. When the network administrator wants to change the flow of data packets, he can create or modify a rule in the controller and place it into a device in the device layer by control flow.

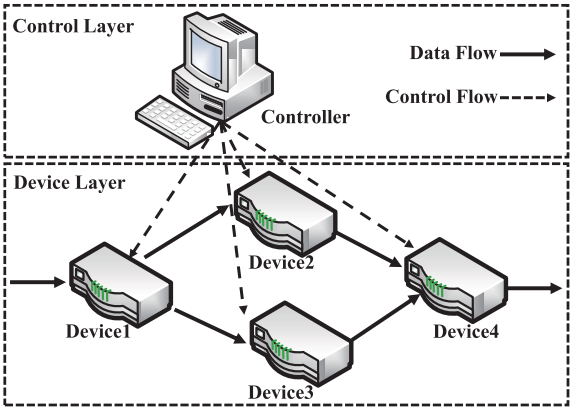


FIGURE 1. A sketch of the device layer and control layer in SDN.

B. RULES IN SOFTWARE-DEFINED NETWORK

The SDN controls the flow of data packets by rules. The rules are represented as a matching table which is created by controller and placed into devices. Figure 2 shows the architecture of the matching table. The matching table denotes a

	Priority	Matching Fields					Action Field
Rule	Pri	SIP	SP	DIP	DP	...	Action
r1	1	36.10.95.0/24	*	34.198.49.48/32	733:734	...	Access
r2	2	36.10.95.0/24	*	34.198.49.48/32	*	...	Drop
...	...	36.10.95.0/24	80	34.198.49.48/32	*	...	Forward

FIGURE 2. The architecture of matching table.

rule set, in which a rule is composed of three kinds of fields: one priority field, some matching fields, and one action field. The priority field specifies the order of matching rule. The higher priority rule will be matched before the lower priority rule. The matching fields specify how the packet header will be matched with this rule. The action field specifies the action to be enforced on the matched data packets. There are many kinds of actions, such as permitting or prohibiting data packets to go through the device, forwarding data packets to other devices, or modifying the head content of data packets, *etc.*

Since an important function of the network is access control, there are many devices in SDN for access control, *e.g.*, firewall. In this paper, we focus on the rule placement of ACL rules in firewalls. ACL is a kind of matching table which is deployed into the firewall. In general, the values of priority fields are same in ACL rules. There are five matching fields in ACL rules, and the value of ACL rules' action field is *accept* or *drop*, we use the following shorthand: a (*Accept*), d (*Drop*).

C. PREFIXES OF RULES

It is a trend to use *TCAMs* to perform high-speed packet classification. So the matching fields' values of rules are consisting of an array of ternary elements, such as $\{0, 1, *\}$, in which $*$ is a wildcard character that matches both 0 and 1. For example, a matching field's value is [4, 7], then we use $1**$ to denote it. we call the array of ternary elements as *prefix*. A prefix can denote a range, but a range might not use a prefix to denote it. For example, if a range is [2, 8], then we must use three prefixes to denote it: 0010, $01**$, and 1000. Prefix has the following two important properties.

1). Any prefix can denote a range, but a range might not be denoted by a prefix.

2). Given two prefixes p_i and p_j, the relationship between p_i and p_j is any of the following two cases:

a. $p_i \cap p_j = \phi$

b. $p_i \cap p_j = p_i$ or $p_i \cap p_j = p_j$

In other words, if the intersection of p_1 and p_2 is not empty, then p_1 is a subset of p_2 or vice versa.

D. SOME IMPORTANT NOTATIONS AND OPERATIONS OF PREFIXES

In this paper, we use p to denote a prefix value of rule's matching field and use $P(r, k)$ to denote the prefix value of the kth matching field in r, in which r is a rule. In this subsection, we present some notations and operations of prefixes as follows.

1) COVERAGE OF PREFIXES

Each prefix denotes a range, we use $R(p)$ to denote the range represented by prefix p. If $R(p_i) \in R(p_j)$, we say p_j **covers** p_i. Given two prefixes p_i and p_j, if p_i does not cover p_j and p_j does not cover p_i, then $R(p_i) \cap R(p_j) = \phi$.

2) DIFFERENCE SET OF PREFIXES

Given two prefixes p_i and p_j, if p_i **covers** p_j, we use the $\boldsymbol{M(p_i, p_j)}$ to denote the difference set between p_i and p_j. $M(p_i, p_j)$ is the minimum prefix set that satisfies $R(M(p_i, p_j)) \cup R(p_j) = R(p_i)$, where $R(M(p_i, p_j))$ denotes the range union of all ranges denoted by the prefixes in $M(p_i, p_j)$. For example, $p_1 = 1 * * * *$, $p_2 = 100 * *$, $M(p_1, p_2) = \{11 * **, 101**\}$. Obviously, the equation $R(p_2) \cup R(M(p_1, p_2)) = R(p_1)$ holds.

3) COMPARISON OF PREFIXES

Each prefix p denotes a range, the upper bound of a prefix p is the value computed by replacing all $*$ of p with 1 and the lower bound of a prefix is the value computed by replacing all $*$ of p with 0. We use $\boldsymbol{p^u}$ to denote the upper bound of p, and use $\boldsymbol{p^l}$ to denote the lower bound of p. For example, $p = 10 * *$, $p^u = 1011$, $p^l = 1000$, the range denoted by p is $[p^l, p^u]$. Given two prefixes p_i and p_j, if $p_i^u \geq p_j^u$, we say $p_i \geq p_j$; otherwise, we say $p_i \leq p_j$.

4) MINIMUM COMMON PREFIX

Given two prefixes p_i and p_j, the **Minimum Common Prefix** of p_i and p_j is a prefix p that $R(p)$ is the minimum range among the ranges denoted by prefixes that satisfy the condition: $R(p_i) \in R(p)$ and $R(p_j) \in R(p)$, we use $\boldsymbol{MCP(p_i, p_j)}$ to denote it. For example, $p_1 = 100 * *$, $p_2 = 11 * **$, $MCP(p_1, p_2) = 1 * * * *$.

5) MERGENCE OF PREFIXES

Given two prefixes p_i, p_j, if p_i and p_j have same number of $*$ and only the last bits before $*$ of p_i and p_j are different, then $p_i \cup p_j$ can be represented by one prefix, we say p_i and p_j can be **Merged**. For example, $p_1 = 10 * *$ and $p_2 = 11 * *$ can be merged because $p_1 \cup p_2 = 1 * * *$. *i.e.*, $R(p_1) \cap R(p_2) = \phi$, $M(MCP(p_1, p_2), p_1) = p_2$.

E. SOME OPERATIONS OF RULES

In this subsection, we state some operations of rules, these operations will be used in the following sections.

1) COVER

Given two rules $r_{i,k}$, $r_{j,k}$, and assume that $i \leq j$ and the numbers of the matching fields of the two rules both are n. If $r_{i,k}$ and $r_{j,k}$ satisfy the following condition,

$$\forall m, F(r_{i,k}, m) \cap F(r_{j,k}, m) = F(r_{j,k}, m)\ (1 \leq m \leq n)$$

then we say $r_{i,k}$ **covers** $r_{j,k}$, and use $r_{i,k} \supseteq r_{j,k}$ to denote it.

2) CONTAIN

Given a rule r and a rule set R, if any data packet can match r, and the data packet can match a rule r_i at least in R, then we say R **contains** r, use $r \in R$ to denote it. Note that if $A(r_i) \neq A(r)$, we use $r \nsubseteq R$ to denote it. Furthermore, Given two rule sets R', R, if for each rule r' in R', $r' \in R$ holds, then we say R' is a subset of R, and use $R' \subseteq R$ to denote it.

3) MERGE

Given two rules r_i, r_j and a rule set R, and assume that the numbers of the matching fields of the two devices both are n, and $A(r_i) = A(r_j)$. If $M(M(MCP(r_i, r_j), r_i), r_j) = \emptyset$ or $M(M(MCP(r_i, r_j), r_i), r_j) \subseteq R$ holds, we say r_i and r_j can be **merged**, we use $r_i \oplus r_j$ to denote the rule merged by r_i with r_j.

III. PROBLEM DEFINITION

In this section, we define the problem of rule placement. Table 1 shows the relevant notations used in the problem formulation.

TABLE 1. Some definition of notation.

Symbol	Description
N	the network
D_i	the ith device in the device layer of N
$C(D_i)$	the capacity of D_i
$R(D_i)$	the Rule Set of D_i
$L(R(D_i))$	the number of Rule Set of D_i
r'	the rule that would been placed into device
$D(r')$	the data packets which match with r'
$R(D_i, r')$	rule set that r' has been placed into D_i
$r_{i,j}$	the ith rule in $R(D_j)$
$F(r_{i,j}, k)$	the value of the kth matching field in $r_{i,j}$
$A(r_{i,j})$	the value of action field in $r_{i,j}$
$P(r_{i,j}, k)$	the prefix value of the kth matching field in $r_{i,j}$
$Re(D_i, D_j)$	the relationship between D_i and D_j
Re_s	the serial relationship
Re_p	the parallel relationship

A. THE PRINCIPLES OF RULE PLACEMENT

The network N is composed of a set of devices, and D_i denotes the ith device of the device layer in N. Every device has a rule set denoted as $R(D_i)$. $L(R(D_i))$ denotes the number of $R(D_i)$. In this paper, rules can perform a high-speed data packets classification by using TCAMs. However, it is very expensive and the capacity is limited (the size of the TCAM is usually $1k \sim 2k$ [4]). *The first principle of rule placement is $L(R(D_i))$ cannot exceed the capacity of D_i,* i.e., $L(R(D_i)) \leq C(D_i)$, *and we choose the device D where $L(R(D))$ is the minimal.*

In this paper, we assume that the rules are optimized in given devices, which means all rules in devices are working. If there is a rule r which cannot work in the device, r should be removed from the device as a redundant rule. *The second principle of rule placement is that all rules can work after rule placement.*

B. THE PROBLEM DEFINITION OF RULE PLACEMENT

In this paper, we focus on the rule placement in SDNs. The rule placement problem can be defined as follows: given a network N, there are k devices $(D_1, D_2, \ldots, D_k)$ in N. The rule placement is a problem in finding a suitable device D_i $(1 \leq i \leq k)$ and place r' into it. Figure 3 shows an example of rule placement. Given four devices (D_1, D_2, D_3, D_4), D_1 denotes a router, and the others denote firewalls, r' denotes the rule which will be placed. The process of rule placement is to choose a suitable device from the three devices (D_2, D_3, D_4) and place r' into it.

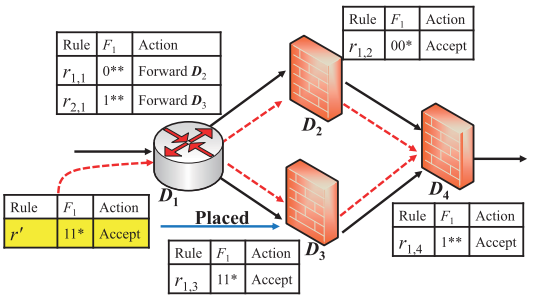


FIGURE 3. An example of rule placement.

In this example, we check each device to find a suitable device that satisfies the above principles of rule placement. Obviously, D_2 is not a suitable device because r' cannot work when r' is placed into D_2. The reason is that the data packets arriving at D_2 do not match r'. As the matter of fact, r' can work in both D_3 and D_4, and r' can merge with $r_{1,3}$ into a rule in D_3, so we finally choose D_3 to place the rule r'. We use $Pr(D_i, r')$ to denote the profit that r' is placed into D_i, and we transform the rule placement problem into the problem of computing $Pr(D_i, r')$ for each device and choosing the device D_i which $Pr(D_i, r')$ is maximal. The computational formula is as follows:

$$Pr(D_i, r') = \begin{cases} -\infty & when\ r'\ cannot\ work\ in\ D_i \\ L(R(D_i)) + 1 - L(R(D_i, r')) \end{cases}$$

IV. SOLUTION APPROACH

In this section, First we classify the rule set in devices. In this paper, our key idea is to consider the position relationship of neighbor devices when placing rules. Second, we classify the position relationship between neighbor devices into two categories: the serial relationship and the parallel relationship. Finally, we propose the rule placement strategy for each category of position relationships respectively.

A. THE CLASSIFICATION OF RULE SET

Given a device D and a flow of the data packets F, and we use $Dir(F)$ to denote the direction of F, we use D_{prev} to denote the device where placed in front of D according to $Dir(F)$ and use D_{next} to denote the device placed in behind of D according to $Dir(F)$, so the $Dir(F)$ indicates that data packets are passed through D_{prev}, D and D_{next}, successively. When data packets arrive at D, and some of them are allowed to go through D, and others are prohibited to go through D. We use $P_{in}(D)$ to denote the set of data packets that arrive at D and use $P_{out}(D)$ to denote the set of data packets that are allowed to go through D. In this paper, we pay close attention to the placement of the ACL rule, since the value of action field is either *accept* or *drop*, We use $R(D)$ to denote the ACL rules in D, and we use $R_a(D)$ and $R_d(D)$ to denote the rules in which the value of action field are *accept* and *drop*, respectively. There are some properties as follows.

1) $R(D) = R_a(D) \cup R_d(D)$.
2) $P_{in}(D) = \bigcup P_{out}(D_{prev})$.
3) Each data packet in $P_{out}(D)$ can match a rule at least in $R_a(D)$.
4) Each data packet in $P_{in}(D)$-$P_{out}(D)$ can match a rule at least in $R_d(D)$.

According to the principles of rule placement in III-A, we should check whether r' can work in D_i or not when we would place r' into D_i. If r' satisfies one of the following two conditions, then r' cannot work in D_i.

1) There does not exist data packet which can match r' in $P_{in}(D_i)$, we use $r' \notin P_{in}(D_i)$ to denote it.
2) There does exist a rule r in $R(D_i)$ that $r \supseteq r'$ and $A(r) = A(r')$.

B. THE POSITION RELATIONSHIP BETWEEN NEIGHBOR DEVICES

In this subsection, we propose a formal definition of the position relationship between neighbor devices.

Given two devices D_i and D_j, we assume that the flow direction of the data packets is from D_i to D_j. In other word, if $P_{out}(D_i) \supseteq P_{in}(D_j)$, then we call the position relationship between D_i and D_j is a **Serial Relationship**, *i.e.*, $Re(D_i, D_j) = Re_s$. There is a serial relationship between D_1 and D_2 have as shown in Figure 4(a).

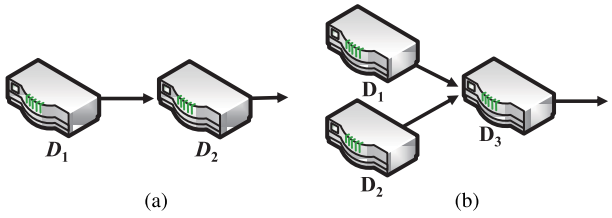


FIGURE 4. Two categories of the position relationships between devices. (a) The serial relationship. (b) The parallel relationship.

If there is no data packet that from D_i to D_j, then we call the position relationship between D_i and D_j is a **parallel relationship**, *i.e.*, $Re(D_i, D_j) = Re_p$. There is a parallel relationship between D_1 and D_2 as shown in Figure 4(b).

C. THE FACTORS INFLUENCING RULE PLACEMENT

According to the above principles of rule placement, there are three factors influencing rule placement as follows.

1) INTERNAL FACTOR OF DEVICE

The factor refers to the rules which are in a device. Figure 5 shows an example of rules in a device. In this example, every data packet which can match r' can also match $r_{1,3}$, r' is the redundant rule for $r_{1,3}$, r' cannot work in D_3, so we cannot place r'_1 into D_3. r' can be merged with $r_{1,1}$ into a new rule in D_1, as a result, one rule is reduced when r'_1 is placed into D_1. This example shows that the rules in a device have direct influences on rule placement.

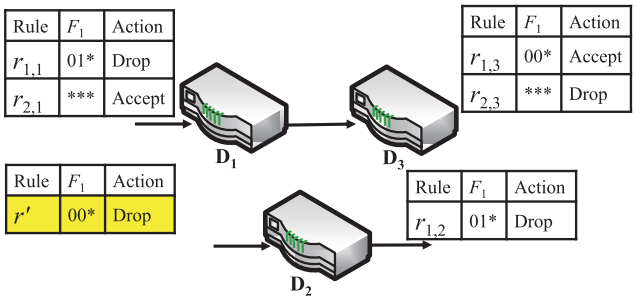


FIGURE 5. An example of rules in devices.

2) EXTERNAL FACTOR OF DEVICE

This factor refers to the data packets which arrive at the device. If there does not exist a data packet that can match r', then r' cannot work in the device. For example, in figure 3, r' cannot be placed into D_2 because there does not exist data packet that can match r' which arrives at D_2.

3) THE POSITION RELATIONSHIP OF THE NEIGHBOR DEVICES

According to the flow of data packets, we assume that two devices D_i and D_j, where D_i is in front of D_j. When a rule r is placed into D_i, r may change the flow of data packets which go through D_i, and these data packets are parts of which arrive at D_j. Thus it might make some rules in D_j cannot work when r is placed into D_i. So the position relationship of the neighbor devices has influences on rule placement.

For example, in figure 5, r' can be placed into D_1 because r' can be merged with $r_{1,1}$ in D_1, and r' can also be placed into D_2 because r' can be merged with $r_{1,2}$ in D_2. Obviously, the position relationship between D_1 and D_3 is a serial relationship. If we place r' into D_1, $P_{in}(D_3)$ may change because $P_{out}(D_1)$ has been changed, then $r_{1,3}$ cannot work in D_3, and we can remove $r_{1,3}$ from D_3. Therefore, we can reduce two rules after r' has been placed into D_1.

D. THE RULE PLACEMENT STRATEGY FOR THE PARALLEL RELATIONSHIP

In this section, we propose the rule placement strategy for the parallel relationship. According to the problem definition of rule placement in section III, the key of our strategy is to computing $Pr(D, r)$ for each device and place the rule into the device which has the larger $Pr(D, r)$.

Given two device D_i and D_j, and there is a parallel relationship between D_i and D_j, *i.e.*, $Re(D_i, D_j) = Re_p$. The rule placement strategy for the parallel relationship contains the following two steps.

Step 1, we check the data packets $D(r')$ which match with r' are contained by $P_{in}(D_i)$ or not. There are two cases as follows.

Case 1. $D(r') \notin P_{in}(D_i)$: In this case, r' cannot work in D_i because any packet which arrives at D_i cannot match with r', and we cannot place r' to D_i, *i.e.*, $Pr(D_i, r') = -\infty$.

Case 2. $D(r') \in P_{in}(D_i)$: In this case, r' can work in D_i because there exists packets which arrive at D_i can match with r'. We put r' into a rule set R', and go to step 2 to compute $Pr(D_i, r'')$ further for each rule r'' in R'.

Step 2, we check each rule r'' in R' is contained by $R(D_i)$ or not. There are tree cases as follows.

Case 1. $r'' \in R(D_i)$: In this case, r'' can not work in D_i because that r'' is a redundance rule for $R(D_i)$, *i.e.*, $Pr(D_i, r'') = -\infty$, and we remove r'' from R'.

Case 2. $r'' \subseteq R(D_i)$: In this case, r'' is a conflict rule for $R(D_i)$ because that there exists a rule r which cover r'' and $A(r'' \neq A(r)$, note that r maybe a rule or a union set of some rules. We should make a choice by our network security requirements. If we choice r'' and r is a rule, we set $Pr(D_i, r'') = 1$, and if we choice r'' and r is a union set of some rules, we set $Pr(D_i, r'') = n$, n is the number of rules. If we choice r, it means that r'' is not satisfied with our network security requirements, we set $Pr(D_i, r'') = -\infty$, and we remove r'' from R'.

Case 3. $r'' \notin R(D_i)$: In this case, r' can be placed into D_i, assume that there exists a rule r in D_i can merge with r'' and use r_{merge} to denote the merged rule, *i.e.*, $r_{merge} = r \oplus r''$ and use r_{merge} to replace the r and r''. Note that r_{merge} maybe also merge with other rules, so this is a constant cyclic process until no rules can be merged. We make $Pr(D_i, r'') = n$, and n is the number of the merged rules.

We make $Pr(D_i, r') = max(Pr(D_i, r'')), r'' \in R'$.

We use the same strategy to compute $Pr(D_j, r')$ for D_j and compare $Pr(D_i, r')$ with $Pr(D_j, r')$. There are three cases as follows.

Case 1. $Pr(D_i, r') = Pr(D_j, r') = -\infty$: In this case, r' can not place into D_i and D_j, we skip it.

Case 2. $Pr(D_i, r') = Pr(D_j, r') \neq$-$\infty$: In this case, we compare $L(R(D_i)$ with $L(R(D_j)$, and place r' into the the device D when $L(R(D))$ is minimal.

Case 3. $Pr(D_i, r') \neq Pr(D_j, r')$: In the case, we place r' into the device D when $Pr(D, r')$ is maximum.

E. THE RULE PLACEMENT STRATEGY FOR THE SERIAL RELATIONSHIP

Given two device D_i and D_j, and there is a serial relationship between D_i and D_j, and D_i is a D_{prev} for D_j. *i.e.*, $Re(D_i, D_j) = Re_p$. The rule placement strategy for the serial relationship is similar to that of the rule placement strategy for the parallel relationship. The difference between the rule placement strategy for the serial relationship and

the rule placement strategy for the parallel relationship as follows.

We use the rule placement strategy for the parallel relationship to compute $Pr(D_i, r')$ and $Pr(D_j, r')$, if $Pr(D_j, r') \leq P(D_i, r')$. According to the rule placement strategy for the parallel relationship, we should place r' into D_i. However, if r' has been placed into D_i, $P_{out}(D_i)$ maybe change, so we should recompute $P_{in}(D_j)$, and use the same strategy to compute $Pr(D_j, r')$ for D_j, and compare $Pr(D_i, r')$ with $Pr(D_j, r')$. The process of comparison is the same with the process of comparison in the rule placement strategy for the parallel relationship.

V. OPTree

In this paper, we propose the strategies of rule placement in section IV. The key of the strategies is checking the rule which would be placed can work or not in a device. However, It is hard to implement to the strategies, there are some challenges in implementing the strategies are as follows.

1) How to check whether the rule is a redundancy rule for a rule set or not.

2) How to check whether the rule can be merged by other rules or not.

FDD is a good data structure to denote the ACL rule that was proposed by Liu and Gouda [24] and widely used in rule compression [25]–[27]. However, the rule needs to be split according to the paths of FDD when checking whether a rule is contained by a rule set or not, and the split operation is a time-consuming operation. In this paper, we propose a novel data structure called OPTree to overcome the challenges.

A. THE PROPERTIES OF OPTree

OPTree is a minimal ordered predicate tree which satisfies the following properties. We use T to denote the OPTree.

1). OPTree is a multi-way tree, it has a root vertex, several leaf vertices, and several non-leaf vertices. We use V to denote the vertex of OPTree.

2). Each vertex except the leaf vertices has a data field which match a field of a rule. We use F_i to denote the ith field in a rule, and use $F(V_i)$ to denote the data field of V_i, and use $D(F(V_i))$ to denote the range value of $F(V_i)$. *e.g.*, if $F(V_i)$ denote the *Source Port*, then $D(F(V_i)) = [0, 2^{16} - 1]$. Each leaf vertex has a data field which match the action field of a rule. We use $L(T)$ to denote the hight of OPTree, if a rule has k fields, then $L(T) = k + 1$ holds.

3). Each vertex except the leaf vertices has one or more children vertices, and the vertex V_i has an edge $e_{i,j}$ with its children vertex V_j. We use $I(e_{i,j})$ to denote the label of $e_{i,j}$. If V_i has n children vertices, then $I(e_{i,j}) \subset D(F(V_i))$ $(1 \leq j \leq n)$ holds, and $D(F(V_i)) = \bigcup_{j=1}^{n} I(E_{i,j})$ holds.

4). The edges of V_i are arranged in order. *e.g.*, if $p \leq q$, then $I(E_{i,p}) \leq I(E_{i,q})$ holds, and E_p is on the left side of E_q.

5). We use *Path* to denote predicate path which contains all edges of a traversal paths that starts from root vertex and ends to a leaf vertex. **If a rule r whose predicate is contained by the union of some path predicates, then there must exist a path that contains the predicate of r in OPTree.**

6). Given two predicate paths $p_i = \{I(e_{i,1}), I(e_{i,2}), \cdots, I(e_{i,n})\}$ and $p_j = \{I(e_{j,1}), I(e_{j,2}), \cdots, I(e_{j,n})\}$, if for each prefix $I(e_{i,k})$ and $I(e_{j,k})$, $I(e_{i,k}) \in I(e_{j,k})$ holds, we say p_i is redundant to p_j. **There is no redundant predicate path in OPTree.**

Given two rules r_1 and r_2, in which $r_1 = \{F_1 = 0**, F_2 = 0** \rightarrow \{accept\}\}$ and $r_2 = \{F_1 = 1**, F_2 = *** \rightarrow \{accept\}\}$. Figure 6 shows the three kinds of trees created by r_1 and r_2. Figure 6(a) shows a tree T_a that is not an OPTree, because T_a does not satisfy the 5th property. We use r_q to denote a rule, in which $r_q = \{F_1 = ***, F2 = 0** \rightarrow \{accept\}\}$. Obviously, r_q's predicate is contained by the union of some path predicates, but there is not a predicate path that contains the predicate of r_q in T_a. Figure 6(b) shows a tree T_b that is not an OPTree, because T_b does not satisfy the 6th property, the rightmost predicate path $Path_r$ contains the leftmost predicate path $Path_l$, in other word, $Path_r$ and $Path_l$ are redundant predicate paths. Figure 6(c) shows an OPTree.

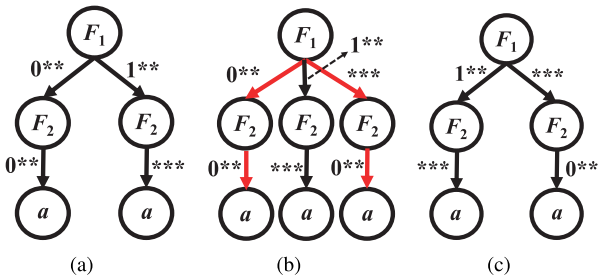


FIGURE 6. Three kinds of tree that created by r_1 and r_2. (a) Not OPTree. (b) Not OPTree. (c) An OPTree.

Obviously, it can overcome the challenges of implementing our strategies by using OPTree to represent the rule set. First, according to the 5th property, given a rule set R and a rule r, if there is a predicate path that contains the predicate of r in OPTree that represent R, then r is contained by R. Second, given two rules r_1, r_2 and a rule set R, r_1 has the same value of the action field with r_2. We use r_{mcp} to denote a rule, which the prefix of each matching field is the **Minimum Common Prefix** of r_1 and r_2 and use R' to denote the rule set, in which the predicate of each rule is an element of the union of the **Difference set** of r_{mcp} with r_1 and the the **Difference set** of r_{mcp} with r_2. For each rule r in R', if there is a predicate path that contains the predicate of r in OPTree that represent R, then r_1 can be merged with r_2.

B. THE INSERTION ALGORITHM OF OPTree

The insertion algorithm of OPTree has three steps, and each step has an algorithm to implement it. The algorithms are described in detail as follows.

Step 1: Convert the rule to a direct predicate path and insert the path into the OPTree. The pseudo-code of the algorithm is shown in Algorithm 1.

Algorithm 1 InsertDirectPath(T, V_c, r_s, i)

Input: T, V_c, r_s, i
Output: T which contains the predicate path of the rule r

```
if i > r_s.length then
    return;
else
    if Edges(V_c) == 0 then
        add a vertex V_new,and label it as F(i + 1);
        add a edge E_new that from V_c to V_new;
        set I(E_new) = P(r_s, i);
        InsertDirectPath(T, V_new, r_s, i + 1);
    else
        for j = 1 to Edges(V_c) do
            if j + 1 > Edges(V_c)|| (I(E_j) < P(r_s, i)&&I(E_{j+1}) > P(r_s, i)) then
                add a vertex V_new and label it as F(i + 1);
                add a edge E_new that from V_c to V_new in order;
                set I(E_new) = P(r_s, i);
                InsertDirectPath(T, V_new, r_s, i + 1);
                Break;
            else if I(E_j) == P(r_s, i) then
                InsertDirectPath(T, V(E_j), r_s, i + 1);
                Break;
```

Step 2: When a direct predicate path has been inserted into the OPTree, the path may merged with other paths, so the main function of step 2 is to find which paths can be merged and merge them into a merged predicate path and insert the merged path into the OPTree by using the algorithm 1. The pseudo-code of the algorithm is shown in Algorithm 2.

Step 3: After the direct predicate path and the merged predicate path have been inserted into the OPTree, we should check the OPTree and remove the redundance predicate path according to the 6th property of OPTree. The pseudo-code of the algorithm is shown in Algorithm 3.

Figure 7 shows the process of how r_2 is inserted into OPTree, which is constructed by r_1, r_1 and r_2 are the two rules in Figure 6.

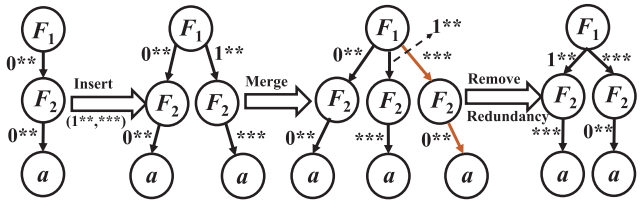


FIGURE 7. An example of Inserting a rule into an OPTree.

C. THE SEARCH ALGORITHM OF OPTree

In this paper, we need to check whether a rule r is contained by a rule set R in devices or not when implementing our strategy of rule placement. We use an OPTree T to represent R and use a predicate path *path* to represent r. So we can use the search process in OPTree to represent the checking process. Obviously, the search operation of OPTree is extremely efficient according to the 5th and 6th properties of OPTree. The pseudo-code of the search algorithm of OPTree is shown in Algorithm 4. Note that we use **Binary-Search** in the algorithm according to the 4th property of OPTree.

Algorithm 2 InsertMergePath(T, r_s)

Input: T, r_s
Output: T that has inserted the merge predicate path

```
set path_m = getMergedPath (T, T.root, r_s, 1);
if path_m.length==T.levels.size then
    create r_m that format as {path_m → {∗}}
    InsertDirectPath(T, T.root, r_m, 1)
/* Find the paths which can be merged in
   T, and merge them into a path                    */
Path getMergedPath (T, V_c, r_s, i)
    create a empty predicate Path path_m to denote the merged
    rule.
    if i > n then
        return path_m;
    for j = 1 to Edges(V_c)
        if I(E_j) == P(r_s, i) then
            Continue;
        else if M(I(E_j), P(r_s, i)) ≠ φ then
            path_m.add(I(E_j) ∩ P(r_s, i));
            getMergedPath (T, V(E_j), r_s, i + 1);
        else if M(I(E_j), P(r_s, i) ≠ φ&&M(MCP(I(E_j), P(r_s, i)), I(E_j)) == P(r_s, i)&&existMerged == false then
            set existMerged = true;
            path_m.add(MCP(I(E_j), P(r_s, i));
            getMergedPath (T, V(E_j), r_s, i + 1);
```

D. ANALYSIS OF THE SEARCH ALGORITHM

In this section, we analyze the time complexity of the search algorithm. Since T is an ordered tree, we can use **Binary-Search** to find the edge quickly. Assume that the OPTree T has m levels and the number of nodes in every level is not more than n, then the time complexity of search algorithm is $O(m\log n)$.

E. ANALYSIS OF THE INSERTION ALGORITHM

In this section, we analyze the time complexity of the insertion algorithm. Assume that each non-leaf vertex of OPTree T has n children nodes. When a rule r has m matching fields, the insertion algorithm needs three steps and we analyze the time complexity of each step as follows.

For step 1, the best case is that there is not an edge e that $I(e) = p_1$ in the first level of OPTree, then we only need to check the edges at the first level. Thus the time complexity is $O(\log n)$. The worst case is that there is an

Algorithm 3 RemoveRedundancePath(T, V_c, r_s, i)

Input: T, V_c, r_s, i
Output: T which not exist redundance path
set E_c is the first edge which start from V_c;
set $path_r$ =`findRePath` (T, V_c, r_s, i, E_c);
if $path_r$ *is not null* **then**
  `removePath` $(T, V_c, E_c, path_r, 1)$;
`/* Find redundance path in T */`
Path `findRePath` (T, V_c, r_s, i, E_c)
  create a empty path $path_r$
  if $i > T.levels.size$ **then**
    return $path_r$
  else
    for $j = 1$ **to** $Edges(V_c)$
      if $M(I(E_j), P(r_s, i)) \neq \phi$ **then**
        $path_r$.add$(I(E_j) \cap P(r_s, i))$;
        `findRePath` $(T, V(E_j), r_s, i+1, E_j)$;
`/* Remove path from T */`
void `removePath` $(T, V_c, E_c, path_r, i)$
  if $i \geq path_r.size$ **then**
    return
  else if V_c *is a terminate node* **then**
    remove V_c, E_c from T
  else
    for $j = 1$ **to** $Edges(V_c)$
      if $I(E_j) == p_i$ **then**
        `removePath` $(T, V(E_j), E_j, path_r, i+1)$
        break;

Algorithm 4 OPTreeSearch(T, r)

Input: OPTree T, query rule $r = \{p_1 \in F_1, p_2 \in F_2, \cdots, p_n \in F_n \rightarrow \{*\}\}$
Output: *True* if r is in T; otherwise, *False*
if $T.root == NULL$ **then**
  return *False*
else
  return `checkPath` $(T.root, r, 1)$
`/* Check query rule path in T */`
bool `checkPath` (V_c, r_q, i)
  if $i \geq r_q.length$ **then**
    return *True*
  for $j = 1$ **to** $Edges(V_c)$
    if $P(r_q, i) \subseteq I(E_j)$ **then**
      return `checkPath` $(V(E_j), r_q, i+1)$
  return *False*;

edge e that $M(I(e), p_1) \neq p_1$ in each level of OPTree, then we need to check edges in each level, thus the time complexity is $O(m\dot{\log}n)$. Therefore, the time complexity of step 1 is $O(k\dot{\log}n)(1 \leq k \leq m)$.

For step 2, the best case is that the predicate path *path* which is inserted by step 1 cannot merge with each edge in first level, then we need to check whether the left edge and the right edge can merge with *path*, thus the time complexity is $O(2)$. The worst case is that each edge can merge with *path* in ith level and there is an edge e that $M(I(e), p_k) = p_k || M(I(e), p_k) = I(e)(1 \leq k \leq m, k \neq i)$ in other levels, so the time complexity is $O(n) + O(k\dot{\log}n)(1 \leq k \leq m)$.

For step 3, the best case is that there is not an edge e that satisfies $M(I(e), p_1) = I(e)$ or $M(I(e), p_1) = p_1$ in first level, then we need to check the left edge and the right edge, so the time complexity is $O(1)$. On the other hand, if the OPTree has m levels and each level has n nodes, the worst case that checks each predicate path is $O(mn)$.

VI. EXPERIMENTAL EVALUATION

In this section, we perform our experiments and evaluate the performance of our approach. In our approach, we consider the influence of the position relationship between neighbor devices on rule placement. Therefore, in our experiment, we use two devices and change the position relationship between them to evaluate our approach.

A. DATA SET GENERATION

First, we use the rule generation tool *ClassBench* proposed in [28], which is widely used in rule generation to generate the rule sets of the two devices. The sizes of the generated rule sets range from 100 to 1000 with the step length is 100. For each size, we generate 20 rule sets.

Second, we also use *ClassBench* to generate the rule sets that would be placed, and the sizes of the generated data sets range from 10 to 100 with the step length is 10, and for each size, we generate 10 data sets. We use A to denote the size of the rule set.

Note that each field of a rule in data set generated by *ClassBench* is represented as a range. So we need to transform the range to one or more prefixes. Thus the size of a transformed set usually is larger than the original one. We use the sizes of the transformed data sets as the metrics in our experiments.

B. IMPLEMENTATION DETAILS

We perform our experiments on desktop PC running Windows 7 Professional with 32GB memory and 4 cores of Intel(R) Xeon(R) processor(3.3GHz) and implement our experiments using C++.

In this section, we perform three kinds of experiments that use the same data set and only change the position relationship between the two devices. The first kind of experiments is the devices with the parallel relationship, the second kind of experiment is the devices with the serial relationship, and the last kind of experiment is a comparison experiment that compares the results of the first kind of experiments and the results of the second kind of experiments.

To evaluate the efficiency of our approach, we computer the two indicators of the experiments: the number of rules after rule placement and the number of rules that reduced after rule placement in the three kinds of experiments. Note that the

number of rules after rule placement is that the total number of rules in the device when placed into a device, so the number of rules after rule placement may more than the number of rules in the device before rule placement. It indicates that our approach is effective when the number of rules after rule placement less than the number of rules in device before rule placement plus the number of rules that would be placed.

C. PARALLEL RELATIONSHIP

The experimental results show that our approach can reduce the size of rules in devices with the parallel relationship. For this set of experiments, we set $A = 10,50,100$ respectively. With $A = 10$, when the total number of rules in the two devices is 200, our approach reduces them to 189 on average; when the total number of rules in the two devices is 1000,our approach reduces them to 974 on average; when the total number of rules in the two devices is 2000, our approach reduces them to 1978 on average. On an average, the number of reduced rules by our approach is 35. With $A = 50$, when the total number of rules in the two devices is 200, our approach reduces them to 232 on average; when the total number of rules in the two devices is 1000, our approach reduces them to 1011 on average; when the total number of rules in the two devices is 2000, our approach reduces them to 1998 on average.On an average, the number of reduced rules by our approach is 47. With $A = 100$, when the total number of rules in the two devices is 200, our approach reduces them to 224 on average; when the total number of rules in the two devices is 1000, our approach reduces them to 1039 on average; when the total number of rules in the two devices is 2000, our approach reduces them to 2011 on average. On an average, the number of reduced rules by our approach is 68. Figure 8(a) shows the number of rules that reduced after rule placement, and Figure 8(b) shows the total number of rules after rule placement.

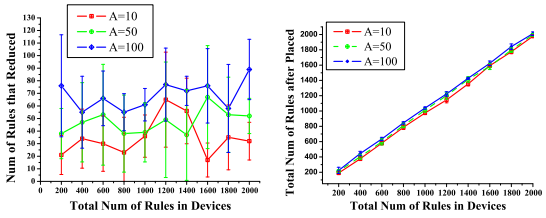


FIGURE 8. Parallel relationship. (a) Number of rules that reduced. (b) Total number after rule placement.

D. SERIAL RELATIONSHIP

The experimental results show that our approach can reduce the size of rules in devices with the serial relationship. We use the same data set in the experiments with the experiments of parallel relationship. With $A = 10$, when the total number of rules in the two devices is 200, our approach reduces them to 173 on average; when the total number of rules in the two devices is 1000, our approach reduces them to 968 on average; when the total number of rules in the two devices is 2000, our approach reduces them to 1969 on average. On an average, the number of reduced rules by our approach is 43. With $A = 50$, when the total number of rules in the two devices is 200, our approach reduces them to 207 on average; when the total number of rules in the two devices is 1000, our approach reduces them to 1002 on average; when the total number of rules in the two devices is 2000, our approach reduces them to 1989 on average.On an average, the number of reduced rules by our approach is 56. With $A = 100$, when the total number of rules in the two devices is 200, our approach reduces them to 215 on average; when the total number of rules in the two devices is 1000, our approach reduces them to 1027 on average; when the total number of rules in the two devices is 2000, our approach reduces them to 2003 on average. On an average, the number of reduced rules by our approach is 77. Figure 9(a) shows the number of rules that reduced after rule placement, and Figure 9(b) shows the total number of rules after rule placement.

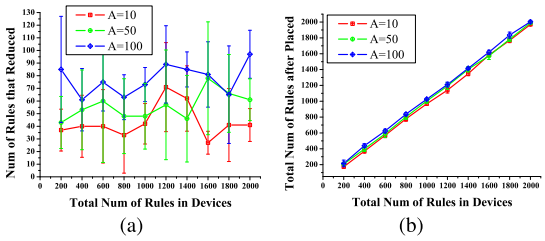


FIGURE 9. Serial relationship. (a) Number of rules that reduced. (b) Total number after rule placement.

E. SERIAL RELATIONSHIP VS. PARALLEL RELATIONSHIP

The experimental results show that the size of rule reduction with serial relationship is more than the size of rule reduction with parallel relationship, which indicates that considering the influence of the position relationship of neighbor devices on rule placement is necessary . In this experiment, we set $A = 50$, and the data sets are exactly the same for different relationships, we use *Parallel* to denote the experiment with the parallel relationship, and use *Serial* to denote the experiment with the serial relationship. When the total number of rules in the two devices is 200, *Parallel* reduces them to 212 on average, *Serial* reduces them to 207 on average; when the total number of rules in the two devices is 1000, *Parallel* reduces them to 1011 on average, *Serial* reduces them to 1002 on average; when the total number of rules in the two devices is 2000, *Parallel* reduces them to 1998 on average, *Serial* reduces them to 1989 on average. On an average, the number of reduce rules by *Parallel* is 47, the number of reduce rules by *Serial* is 56. Figure 10 shows the result of the experiment.

Note that in our experiments, we only change the position relationships between devices to prove our approach can reduce the number of rules in difference position relationships. Because we use the same data, the curves change the same trend, and there are some subtle differences from one curve to another in Figure 10.

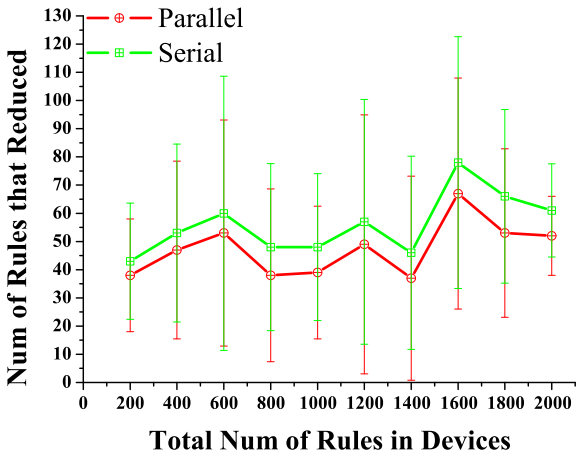


FIGURE 10. Parallel relationship VS. serial relationship.

VII. CONCLUSIONS

In this work, we propose a novel rule placement strategy for different position relationships of neighbor devices, respectively. To overcome the challenges of our strategy implementation, we propose a new data structure called OPTree to represent the rules in devices, which is convenient to check whether a rule is covered by the existed rules. We design two algorithms for OPTree: insertion and search algorithms. In our experimental results, we have shown that our approach can reduce the size of rules in the device after rule placement with different position relationships. Furthermore, the size of rule reduction with the serial relationship is less than the size of rule reduction with the parallel relationship, which indicates that our approach is effective and it is necessary to consider the influence of the position relationship of neighbor devices on rule placement.

REFERENCES

[1] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE J. Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006, doi: 10.1109/JSSC.2005.864128.

[2] Y. Sun and M. S. Kim, "Tree-based minimization of TCAM entries for packet classification," in *Proc. 7th IEEE Consum. Commun. Netw. Conf. (CCNC)*, Las Vegas, NV, USA, Jan. 2010, pp. 1–5.

[3] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, Chicago, IL, USA, 2014, pp. 175–180.

[4] B. Agrawal and T. Sherwood, "Modeling TCAM power for next generation network devices," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Austin, TX, USA, Mar. 2006, pp. 120–129.

[5] M. G. Gouda and X.-Y. A. Liu, "Firewall design: Consistency, completeness, and compactness," in *Proc. 24th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Tokyo, Japan, Mar. 2004, pp. 320–327.

[6] D. L. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang, "Compressing rectilinear pictures and minimizing access control lists," in *Proc. ACM-SIAM Symp. Discrete Algorithms*, New Orleans, LA, USA, 2007, pp. 1066–1075.

[7] K. Kogan, S. Nikolenko, W. Culhane, P. Eugster, and E. Ruan, "Towards efficient implementation of packet classifiers in SDN/openflow," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, Hong Kong, 2013, pp. 153–154.

[8] M. Casado *et al.*, "Rethinking enterprise network control," *IEEE/ACM Trans. Netw.*, vol. 17, no. 4, pp. 1270–1283, Aug. 2009, doi: 10.1109/TNET.2009.2026415.

[9] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "FIREMAN: A toolkit for firewall modeling and analysis," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, May 2006, p. 213.

[10] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *Proc. ACM SIGCOMM Conf.*, New Delhi, India, 2010, pp. 351–362.

[11] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan, "vCRIB: Virtualized rule management in the cloud," in *Proc. 4th USENIX Conf. Hot Topics Cloud Comput.*, Boston, MA, USA, 2012, pp. 23–29.

[12] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *Proc. IEEE INFOCOM*, Turin, Italy, Apr. 2013, pp. 545–549.

[13] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," in *Proc. ACM Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, Santa Barbara, CA, USA, 2013, pp. 13–24.

[14] X. N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Optimizing rules placement in OpenFlow networks: Trading routing for better efficiency," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, Chicago, IL, USA, 2014, pp. 127–132.

[15] H. Li, P. Li, and S. Guo, "MoRule: Optimized rule placement for mobile users in SDN-enabled access networks," in *Proc. IEEE Global Commun. Conf.*, Austin, TX, USA, Dec. 2014, pp. 4953–4958.

[16] U. Ashraf, "Rule minimization for traffic evolution in software-defined networks," *IEEE Commun. Lett.*, vol. 21, no. 4, pp. 793–796, Apr. 2017, doi: 10.1109/LCOMM.2016.2636212.

[17] P. G. Kannan, M. C. Chan, R. T. B. Ma, and E.-C. Chang, "Raptor: Scalable rule placement over multiple path in software defined networks," in *Proc. IFIP Netw. Conf. (IFIP Netw.) Workshops*, Stockholm, Sweden, Jun. 2017, pp. 1–9.

[18] Y.-W. Chen and Y.-H. Lin, "Study of rule placement schemes for minimizing TCAM space and effective bandwidth utilization in SDN," in *Proc. 6th Int. Conf. Future Internet Things Cloud Workshops (FiCloudW)*, Barcelona, Spain, Aug. 2018, pp. 21–27.

[19] A. Mimidis-Kentis, A. Pilimon, J. Soler, M. Berger, and S. Ruepp, "A novel algorithm for flow-rule placement in SDN switches," in *Proc. 4th IEEE Conf. Netw. Softwarization Workshops (NetSoft)*, Montreal, QC, Canada, Jun. 2018, pp. 1–9.

[20] S. Zhang, F. Ivancic, C. Lumezanu, Y. Yuan, A. Gupta, and S. Malik, "An adaptable rule placement for software-defined networks," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Atlanta, GA, USA, Jun. 2014, pp. 88–99.

[21] W. Li, Z. Qin, H. Yin, R. Li, L. Ou, and H. Li, "An approach to rule placement in software-defined networks," in *Proc. 19th ACM Int. Conf. Modeling, Anal. Simulation Wireless Mobile Syst. (MSWiM)*, 2016, pp. 115–118.

[22] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[23] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 3, pp. 1617–1634, 3rd Quart., 2014, doi: 10.1109/SURV.2014.012214.00180.

[24] A. X. Liu and M. G. Gouda, "Diverse firewall design," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. (DSN)*, Atlanta, GA, USA, Jun./Jul. 2004, pp. 595–604.

[25] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 490–500, Apr. 2010, doi: 10.1109/TNET.2009.2030188.

[26] A. X. Liu, E. Torng, and C. R. Meiners, "Firewall compressor: An algorithm for minimizing firewall policies," in *Proc. IEEE INFOCOM*, Phoenix, AZ, USA, Apr. 2008, pp. 176–180.

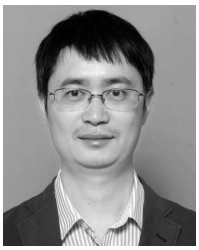
[27] A. X. Liu, E. Torng, and C. R. Meiners, "Compressing network access control lists," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 12, pp. 1969–1977, Dec. 2011, doi: 10.1109/TPDS.2011.114.

[28] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," in *Proc. IEEE INFOCOM*, Miami, FL, USA, Mar. 2005, pp. 89–99.

WENJIE LI received the B.S. and M.S. degrees in software engineering from Hunan University, China, in 2010 and 2013, respectively, where he is currently pursuing the Ph.D. degree with the College of Computer Science and Electronic Engineering. His main interests include network management, network rules optimization, and NoSQL database.

ZHENG QIN received the Ph.D. degree in computer software and theory from Chongqing University, China, in 2001. He is currently a Professor of computer science and technology with Hunan University, China. He has accumulated rich experience in products development and application services, such as in the area of financial, medical, military, and education sectors. His main interests include computer network and information security, cloud computing, big data processing, and software engineering. He is a member of the China Computer Federation and ACM.

KEQIN LI is currently a SUNY Distinguished Professor of computer science with the State University of New York. He has published over 600 journal articles, book chapters, and refereed conference papers. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU–GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, the Internet of Things, and cyber-physical systems. He is an IEEE Fellow. He has received several best paper awards. He is currently serving or has served on the editorial boards of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON CLOUD COMPUTING, the IEEE TRANSACTIONS ON SERVICES COMPUTING, and the IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING.



HUI YIN received the B.S. degree in computer science from Hunan Normal University, China, in 2002, the M.S. degree in computer software and theory from Central South University, China, in 2008, and the Ph.D. degree from the College of Information Science and Engineering, Hunan University, China, in 2018. He is currently an Assistant Professor with the College of Applied Mathematics and Computer Engineering, Changsha University, China. His interests include information security, privacy protection, and applied cryptography.

LU OU received the B.S. degree in computer science from the Changsha University of Science and Technology, in 2009, and the M.S. and Ph.D. degrees in software engineering from Hunan University, in 2012 and 2018, respectively. Her research focuses on security, privacy, optimization, and big data. She is a member of the IEEE.