Exploring Dynamic Rule Caching Under Dependency Constraints for Programmable Switches: Theory, Algorithm, and Implementation

Xinhao Deng, *Student Member, IEEE*, Mingwei Xu, Qi Li, *Senior Member, IEEE*, Weijie Wu, Yuan Yang, Menghao Zhang, Yu Zhou, *Member, IEEE*, and Jianping Wu, *Fellow, IEEE*

***Abstract*—Ternary Content Addressable Memory (TCAM) enables fast lookup and is widely used by routers and switches to support policy-based forwarding. Due to high cost and small capacity, only a small subset of important rules can be cached in TCAM, so determining it is critical to increasing the hit ratio. This is more challenging than traditional caching problems because of complicated rule dependency relationships. Existing works are based on heuristics and they don't work well under all practical scenarios. Worse still, the lack of fundamental understanding of the design space, complexity, and optimality makes all explorations in mystery. In this paper, we use a modeling-based method to formulate the problem, prove its complexity, and propose DROPS, a dynamic rule caching framework with a much higher hit ratio. In particular, we deduce the rule selection problem into a multi-dimensional rule space transformation problem. Thus, we are no longer limited by using the intrinsic rules; rather, we can transform original rules into "new rules" equivalently without rule dependency. We design non-trivial rule placement and update algorithms and implement them in programmable switches. In the experimental evaluation, we show that our method outperforms all existing methods.**

***Index Terms*—TCAM, rule caching, programmable switches, software defined networks.**

I. Introduction

THE EMERGING programmable switch allows network programmers to directly customize data plane algorithms to implement flexible policy-based packet forwarding at line rate [1]. Under such a paradigm, the control plane and the data plane are separated, and a centralized controller is responsible to control one or multiple switches. The flexibility of programmable switches can be widely applied in many areas, e.g., congestion control [2], network monitoring [3], load balancing [4], [5] and encrypted traffic analysis [6].

TABLE I
AN EXAMPLE OF RULE TABLE

Rule	Priority	Src IP	Dst IP	Pattern
r_1	2	10.0.0.0	10.0.0.6/31	011*
r_2	1	10.0.0.0	10.0.0.4/30	01**
r_3	0	10.0.0.0	10.0.0.0/29	0***

Efficient storage of rules in programmable switches significantly impacts the packet forwarding performance [7]. In Table I, we show an example of the rule table. Each rule specifies a pattern on multiple fields in the packet header. Typically, these fields include source and destination IP address, source, and destination port number, and protocol type. The rule's pattern specifies which packets match the rule. A special case here is that pattern fields of different rules might overlap. Let us take rules r_1 and r_2 for example. Their pattern fields are `011*` and `01**`, respectively, and they do have an overlap. Imagine that a packet whose header is `0110`. It matches both r_1 and r_2. In such a case, the switch will select the rule with the highest priority (in this case, r_1) to perform the corresponding action.

To support more fine-grained and flexible forwarding policies in real-world scenarios (e.g., traffic classification [8], [9], anomaly detection [10], [11], and attack mitigation [12]), we would like to store as many rules as possible in the Ternary Content Addressable Memory (TCAM) [13] of the programmable switch. As modern network services increase in size and complexity, the switches need to support `100K` rules or more [14], which exceeds the capacity of existing switches significantly. Facing this challenge, the collaboration between a switch and the slow path (e.g., a server) has become a promising research direction [7], [15], [16], [17]. Today, TCAM is used as a cache to store only part of the rules, while the whole set of rules is stored in the slow path [15]. For an incoming packet, the switch will try to match it in TCAM first; if it fails, the switch will send the packet to the slow path. This guarantees that every packet will match the correct rule. The slow path processes packets by software with a large capacity, incurring low power consumption but slow matching speed. Existing studies [18], [19] show that patterns of flows often

Manuscript received 10 December 2023; revised 28 May 2024; accepted 31 May 2024. Date of publication 3 July 2024; date of current version 21 August 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62132011 and 62221003. The associate editor coordinating the review of this article and approving it for publication was A. Detti. *(Corresponding author: Qi Li.)*

Xinhao Deng, Mingwei Xu, Qi Li, and Jianping Wu are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 102100, China (e-mail: dengxh23@mails.tsinghua.edu.cn; xumw@tsinghua.edu.cn; qli01@tsinghua.edu.cn; jianpingwu@cernet.edu.cn).

Weijie Wu resides in Redwood City, CA 94061 USA (e-mail: wuwjpku@gmail.com).

Yuan Yang is with the Department of Computer Science, Tsinghua University, Beijing 102100, China (e-mail: yangyuan_thu@tsinghua.edu.cn).

Menghao Zhang is with the School of Software, Beihang University, Beijing 100191, China (e-mail: zhangmenghao0503@gmail.com).

Yu Zhou is with the System Department, StepFun Inc., Palo Alto, CA 94306 USA (e-mail: zhouyu@stepfun.com).

Digital Object Identifier 10.1109/TNSM.2024.3422092

satisfy Zipf distribution. Therefore, if we can properly store a few "hot" rules which can cover a large portion of the flows in TCAM, the "hit ratio" of TCAM for rule lookup should be high enough and then the overall forwarding performance is still satisfactory.

Naturally, a key question is how to determine the "hot rules" to store in TCAM. At first glance, this might seem a trivial optimization problem, however, a number of specific features make it very challenging. To name a few:

- *Rule dependency problem.* Directly selecting hot rules may result in incorrect forwarding. Let us recall the packet with header 0110 that we mentioned before. If we only place r_2 in TCAM, the packet will be matched to r_2 without a further lookup in the slow path; however, it should be matched to r_1. This indicates that if we store a low-priority rule *r* in TCAM, then we will have to store many higher-priority rules (which might be "cold") whose match field has an overlap with *r*. We call this issue the "rule dependency problem".
- *Traffic dynamics.* Intuitively, we want to store the popular rules in TCAM. Note that traffic can be highly dynamic and the popularity of rules varies over time. Thus, dynamic update to TCAM entries is essential. However, the rule update in TCAM is slow [20], [21], so we need to design a dedicated online update algorithm.

The community has realized the importance of better utilization of TCAM, and improvement algorithms [7], [15], [22] have been recently proposed. However, they are heuristic approaches with vague physical meanings, and they do not work well under all practical scenarios. A critical missing piece of understanding is that we do not fundamentally understand the design space of our problem. For example, we are unaware of what the fine-grained rules really mean, how we can capture their dependency, and how we can use a subset of rules to represent a critical region of the space.

As we mentioned before, the set of fine-grained rules is generated by different applications and they can be arbitrary. In fact, this set is not necessarily the only option to represent our rule space. It does not make sense to totally rely on raw input to decide the rules placed in TCAM. Let us still consider the example in Table I. Let's assume under our flow distribution, rule r_3 has a very high hit ratio while r_1 and r_2 are rarely hit. If we decide to store r_3 in TCAM, we will have to store r_1 and r_2 in TCAM due to the rule dependency. However, we can remove the rule dependency based on the equivalent transformation of rules. We show the transformed patterns of rules r_1, r_2, and r_3 below. There won't be any dependency on r_3, so we can safely save r_3 in TCAM. This way, a large portion of incoming packets will still correctly find a match in TCAM, yet we save a large amount of TCAM capacity.

Rule	Original Pattern	Transformed Pattern
r_1	011*	011*
r_2	01**	010*
r_3	0***	00**

In the above example, we somehow "magically" removed the rule dependency based on the transformed rules from the raw inputs r_1, r_2, and r_3. However, in reality, transforming the original rules into rules without rule dependency is non-trivial. In order to systematically achieve such efficiency by transforming original rules, the fundamental is really to understand our rule space and choose the best representation of rules; only via this way can we not rely on any specific input format, but choose the best entries for TCAM. Realizing the lack of these fundamental understandings, In this work, we utilize a rigorous modeling approach to tackle the problem. We formally formulate the problem as a mathematical framework, which helps us fully understand the design space by deducing it to the space transformation problem. Then we prove that the problem is NP-hard. Furthermore, we design a series of algorithms that adaptively generate independent rules based on traffic distribution.

The contributions of our work are three-fold:

- *Formulation*: We represent the rule space as a non-standard multi-dimensional space, and TCAM rule selection is a rule space transformation problem over this space. We then prove its NP-hardness.
- *Algorithm designs*: We propose DROPS (dynamic rule caching for programmable switches), which generates independent rules for dynamic rule caching based on clear physical meanings of rule caching. DROPS is based on flow-aware decision trees that make full use of global information (i.e., traffic distribution) of packages to further improve the hit ratio.
- *Implementation & evaluation*: We prototype DROPS with Barefoot Tofino switches EdgeCore Wedge 100BF-32X. Evaluations based on both simulations and real-world experiments demonstrate that our method significantly outperforms all existing approaches.

We would like to emphasize that this research not only shows a near-optimal solution to programmable switches but more importantly, it provides an important framework and insights to understand caching problems with dependency constraints.

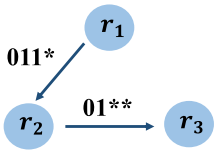
The rest of this paper is organized as follows. Section II states our problem, constraints, and challenges. Section III formulates the problem in the mathematical framework and conducts an in-depth theoretical analysis. Section IV proposes the designs of DROPS. Section V presents evaluation results by simulations and experiments. Section VI states related work and Section VII concludes.

II. Problem Statement

In this section, we formally define the cache rule placement and update problem. We also state the rule dependency constraint and the TCAM insertion/update challenges.

A. Rule Placement and Update

For each rule, the number of matching packets in a specific time window can be counted. Let *N* denote the number of rules, and *C* denote the number of rules that can be placed in TCAM. The problem we will address is to select *C* rules out of the *N* rules to be placed in TCAM, such that the total number of matching packets of the TCAM



Rule	Pattern
r_1	011*
r_2	01**
r_3	0***

Fig. 1. Rule dependency graph of Table I.

is maximized during a particular time window. There are a couple of constraints/challenges when we solve the above problem which we will state below.

B. Rule Dependency

One of the most important constraints is rule dependency. We have shown a simple case in Table I, where rules r_1 and r_2 have an overlap and a packet is possible to match both rules. In case that happens, the packet needs to be matched with r_1 since it has a higher priority. Formally, we say that a rule r_1 "depends on" r_2 if they have an overlap in the rule space and r_1 has a higher priority than r_2. We can construct the *rule dependency graph* [15] to represent such a relationship. In Figure 1, we show the rule dependency graph of Table I where a node represents a rule, and a directed edge captures a direct dependency between a pair of rules. The value of an edge is a set of flows that both nodes can match.

The set of the rules that depend on a given rule *r* is called the *dependent-set* of *r* [15]. As shown in Figure 1, the dependent set of r_3 contains r_1 and r_2. Formally, the constraint we have in TCAM rule placement is: if a rule *r* is placed in TCAM, then all rules in the dependent set of *r* need to be placed in TCAM as well. A simple method of "inserting all rules in the dependent set when placing a rule in TCAM" will induce a huge space overhead. For example, according to our empirical study of the Equinix datacenter routing table and the CAIDA traffic trace collected from the Equinix datacenter on March 15, 2018 [23], we find that, for the top 10K popular rules, the average number of dependent rules associated with each rule is about 143.

C. TCAM Insertion and Update

The rule placement in TCAM oftentimes is not a one-time setting. Rather, the flow distribution changes dynamically over time, therefore, the "popular" rules also change. A measurement in [7] shows that the rules generated by ClassBench [24] are cached in TCAM with a hit ratio of 87%, however, it drops to 48% after one hour. This requires us to dynamically update the rules in TCAM according to the change in flow distribution.

However, inserting/updating a rule in TCAM is often very expensive, as it might change the relative priorities of the rules in TCAM.[1] Therefore, there is a difficult trade-off in keeping real-time popular rules in TCAM vs. keeping the TCAM stable, and our problem becomes designing a dynamic TCAM cache management scheme that achieves a high hit ratio and low update cost simultaneously.

In summary, the challenging problem we are facing is to design an efficient dynamic cache mechanism, which selects a subset of rules to put into TCAM from all rules, and adjusts the selection in real-time according to flow popularity dynamics, subject to rule dependency constraints, so as to maximize the hit ratio of TCAM and minimize the overhead of rule insertion/update. In the next section, we will formulate this problem using a mathematical framework.

[1]This is determined by the TCAM memory management feature. A TCAM query does a parallel search and returns the first position of the rules that match a given entry. As rules may have dependency relationships, it is therefore mandatory to store higher-priority rules in the rule set before lower-priority ones. In order to maintain the order of rule priority in TCAM, it takes a lot of time to move the entries. A measurement in [25] shows that inserting a single ACL rule for a $1K$ rule set generated by ClassBench [24] requires up to 466 rule moves, resulting in a huge time overhead.

TABLE II
MAJOR NOTATIONS USED IN THE PAPER

Notation	Meaning
r	The rule
f	The flow
s	The independent rule
c	The independent hypercube
p	The point in the multi-dimensional space
N	Number of rules
M	Number of flows in a time window
C	Number of rules can be placed in TCAM
T	Number of independent rules
P_i	Number of packets associated with flow i
W_i	Weight of point i
$\mathcal{O}$	Set of original rules
$\mathcal{N}$	Set of independent rules
$\mathcal{S}$	Set of rules placed in TCAM in the last time window
$\mathcal{R}$	Set of rules deleted from TCAM in the current time window
$\mathcal{A}$	Set of rules selected to be inserted in the current time window
$\mathcal{D}_i$	The dependent-set of rule i
$\mathcal{F}_\mathcal{S}$	The flow set that rule set $\mathcal{S}$ can math
$\mathcal{B}$	Set of independent hypercubes selected in the last time window
$\mathcal{I}$	Set of independent hypercubes unselected in the current time window
$\mathcal{E}$	Set of independent hypercubes selected in the current time window
$\mathcal{H}_\mathcal{B}$	The points that independent cube set $\mathcal{B}$ can cover

III. PROBLEM FORMULATION

In this section, we formulate mathematical frameworks to characterize the TCAM rule placement and update the problem. We first formulate the problem as an optimization. Then we demonstrate that the rule space can be represented as a non-standard multi-dimensional space and TCAM rule selection as a rule space transformation problem over this space. We further reduce the rectilinear picture compression problem [26], a known NP-Complete problem, into this problem, to prove its NP-hardness. In Table II, we summarize the major notations in this paper.

A. Rule Caching: An Initial Model

Let us start by considering the simplest case. We want to choose *C* rules out of the *N* rules in a rule set, such that for any incoming packet, we have the highest probability of finding its matched rule in the TCAM entries rather than the backend

storage, or in other words, to achieve the maximal "hit-ratio."[2] Note that the flow distribution is dynamic, let us first consider achieving the maximum hit ratio for a particular time window.

We consider a list of rules $r_1, r_2, \ldots, r_N$. Let $f_1, f_2, \ldots, f_M$ denote the M flows in the current time window. Let $\mathcal{S}$ denote the rule set placed in TCAM at the end of the last time window, and $\mathcal{R}$ and $\mathcal{A}$ denote the rule set deleted from TCAM, and selected to be inserted in the current time window, respectively. In the stage of cache initialization, set $\mathcal{S}$ and set $\mathcal{R}$ are both empty. Obviously, the number of deleted rules cannot exceed the number of existing rules in TCAM. After the rule update, the number of rules in TCAM cannot exceed the TCAM capacity C. Therefore, we have:

$$|\mathcal{R}| \leq |\mathcal{S}|, \tag{1}$$

$$|(\mathcal{S} - \mathcal{R}) \cup \mathcal{A}| \leq C. \tag{2}$$

Now we consider rule dependency. If a rule r is placed in TCAM, all rules in the dependent set of rule r should also be placed in TCAM. Let $\mathcal{D}_i$ denotes the dependent-set of rule i. The following formula represents the rule dependency constraint:

$$x \in (\mathcal{S} - \mathcal{R}) \cup \mathcal{A},\ \forall x \in \mathcal{D}_y \text{ and } y \in (\mathcal{S} - \mathcal{R}) \cup \mathcal{A}. \tag{3}$$

Let $\mathcal{F}_{\mathcal{S}}$, P_i denote the flow set that rule set $\mathcal{S}$ can match, and the number of packets associated with flow i, respectively. The objective is to maximize the number of packets in the current time window that can be covered by the flow set represented by the selected rule set in TCAM. Then the rule placement and update problem can be formulated as follows:

$$\textbf{max} \ \sum_i P_i \cdot \mathbf{I}\Big(f_i \in \mathcal{F}_{(\mathcal{S}-\mathcal{R})\cup\mathcal{A}}\Big), \tag{4}$$

$$\boldsymbol{s.t.}\ |\mathcal{R}| \leq |\mathcal{S}|, \tag{5}$$

$$|(\mathcal{S} - \mathcal{R}) \cup \mathcal{A}| \leq C, \tag{6}$$

$$x \in (\mathcal{S} - \mathcal{R}) \cup \mathcal{A},\ \forall x \in \mathcal{D}_y \text{ and } y \in (\mathcal{S} - \mathcal{R}) \cup \mathcal{A}, \tag{7}$$

where $\mathbf{I}(\cdot)$ is an indicator function.

Up till now, we have formally formulated our caching problem in mathematical forms. One challenge that we face under this formulation, is the huge overhead caused by rule dependency and TCAM insertion and update. To overcome this difficulty, imagine that now we are sure each rule is independent of the other. In such a scenario our problem can be greatly simplified since we can remove all dependency constraints. So the next question is, can we do some transformation on the original set of rule inputs, such that we can work on an "equivalent" set of other rules which are independent of each other?

B. Independent Rule Set Generation

Now we generate independent rules to remove all dependency constraints and realize rule placement according to independent rules. Let $\mathcal{O}$ denote the original rule set, and $\mathcal{N}$ denote a new rule set with independent rules that are independent of each other. We utilize an independent rule generation algorithm to transform rule set $\mathcal{O}$ into the rule set $\mathcal{N}$. The pseudo-code of the independent rule set generation is shown in Algorithm 1. We generate sub-rules based on the rule r in the original rule set. For example, based on rule with pattern `10**`, we can generate all sub-rules with pattern `10**`, `100*`, `101*`, `1000`, `1001`, `1010` and `1011`. The sub-rules of rule r have the same forwarding actions as the rule r. Then we delete sub-rules that overlap with the rules in the dependent set of r. Finally, the remaining sub-rules are added to the rule set $\mathcal{N}$.

Algorithm 1: Independent Rule Set Generation

```
Input:
  O: the original rule set
Output:
  N: the new rule set with independent rules
Procedures:
N ← ∅
for r1 ∈ O do
    Φ ← Calculate all sub-rules of r1
    for r2 ∈ Φ do
        if r2 overlaps with rules in D_r1 then
            Φ ← Φ − r2
    N ← N + Φ
return N
```

The sub-rules in the rule set $\mathcal{N}$ are defined as *independent rules* because they do not overlap with higher priority rules. By replacing the original rule set $\mathcal{O}$ with the independent rule set $\mathcal{N}$, we can remove constraint (7). Now we can have the following formulation:

$$\textbf{max} \ \sum_i P_i \cdot \mathbf{I}\Big(f_i \in \mathcal{F}_{(\mathcal{S}-\mathcal{R})\cup\mathcal{A}}\Big), \tag{8}$$

$$\boldsymbol{s.t.}\ |\mathcal{R}| \leq |\mathcal{S}|, \tag{9}$$

$$|(\mathcal{S} - \mathcal{R}) \cup \mathcal{A}| \leq C. \tag{10}$$

Note that, a rule set will generate a large number of independent rules. We can only select and place a small part of the independent rules in TCAM. However, it is still not clear how we can choose independent rules according to the problem above. Therefore, we will continue transforming the problem to another equivalent form and deepen our understanding of multi-dimensional space.

C. Problem Transformation

Now we can transform rule caching into independent rule caching. Different from the original rule set $\mathcal{O}$, the size of the independent rule set $\mathcal{N}$ increases exponentially. We can achieve correct forwarding based on a small number of independent rules. However, it is difficult to determine which independent rules are selected due to the complicated representation of rules. We analyze the design space in a graphical way.

Figure 2 shows the example of transforming the problem. The fields in the packet header (e.g., source and destination IP

[2]"Hit-ratio" denotes the ratio of the number of packets matched in TCAM to the total packets.

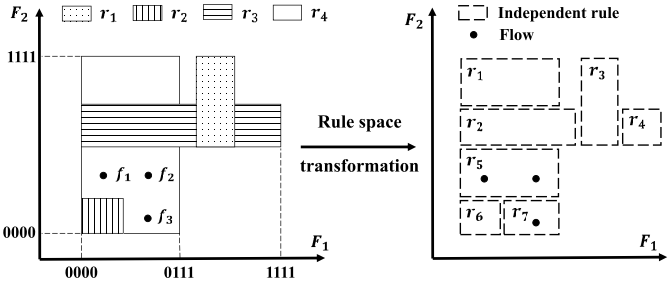
Rule Table

Rule	F_1	F_2	Priority
r_1	10**	1***	4
r_2	00**	00**	3
r_3	****	10**	2
r_4	0***	****	1

Flow Table

Flow	F_1	F_2	Weight
f_1	0010	0100	20
f_2	0101	0100	50
f_3	0101	0010	10

(a) Rule table and flow table



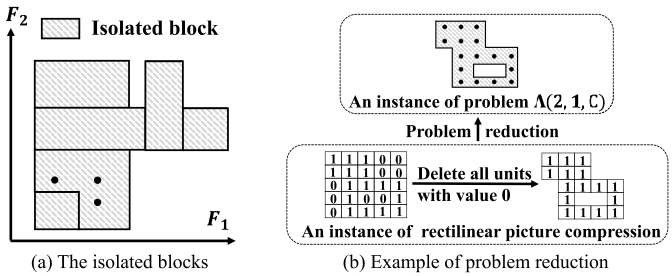
(b) Multi-dimensional rule space transformation

Fig. 2. Example of problem transformation.

addresses, source, and destination port numbers, and protocol numbers) represent the dimensions in the geometric space. A packet is represented as a point in this space, and a rule is a hyper-cube. Therefore, we can transform the original problem into the rule space transformation problem in the multi-dimensional space. For simplicity but without loss of generality, we consider 4-bit patterns and 2-tuple rules in this paper.

As shown in Figure 2(b), each rule with two 4-bit patterns can be converted into a 2-tuple line in the two-dimensional space. Therefore, rules can be transformed into rectangles in space. For example, rule r_1 with patterns $10**$ and $1***$ can be transformed into a rectangle of size 4×8, because r_1 is from 1000 to 1011 in F_1 dimension, and from 1000 to 1111 in F_2 dimension. The priority of the rule determines the coverage of the intersection of the rectangle. When two rectangles intersect, the rectangle with higher priority will cover the intersection. In this case, a flow can be imagined as a point in the two-dimensional space occupied by its highest priority matching rule. In the flow table in Figure 2(a), the weight of a point represents the number of packets associated with a flow. For example, flow f_2 contains 50 packets in the current window.

Figure 2(b) shows an example of multi-dimensional rule space transformation. The original rule space can be transformed into an independent rule space. Thus, rule selection for TCAM is simplified into selecting top C independent rules with the largest sum of weights of points covered, where C is the number of rules that can be placed in TCAM. For example, instead of directly selecting four rules in the original rule space to cover three points associated with flow f_1, f_2, and f_3, we can select two rules r_5 and r_7 in the transformed rule space, which saves a large amount of TCAM capacity. In particular, different from the packet classification problem that is solved by rule space dividing, our paper aims to efficiently realize dynamic rule caching by removing rule dependency among rules. Thus, the packet classification methods [27], [28], [29] cannot be applied to our problem.



(a) The isolated blocks (b) Example of problem reduction

Fig. 3. Example of theoretical analysis.

Given a list of independent hypercubes $c_1, c_2, \ldots, c_T$ associated with independent rules in the multi-dimensional space, let $p_1, p_2, \ldots, p_M$ denote the M points corresponding to the flows appearing in the current time window. There are three independent hypercube sets $\mathcal{B}$, $\mathcal{I}$ and $\mathcal{E}$. Set $\mathcal{B}$, $\mathcal{E}$ represent the independent hypercube set selected in the last time window, and newly selected ones in the current time window, respectively. Set $\mathcal{I}$ represents the set of independent hypercubes in $\mathcal{B}$ that are unselected in the current time window. Let $\mathcal{H}_{\mathcal{B}}$, W_i denote the points that independent hypercube set $\mathcal{B}$ can cover in the multi-dimensional space, and the weight of point i, respectively.

Obviously, c, p, $\mathcal{B}$, $\mathcal{I}$, $\mathcal{E}$, $\mathcal{H}$ and W in this problem correspond to s, f, $\mathcal{S}$, $\mathcal{R}$, $\mathcal{A}$, $\mathcal{F}$ and P in the original problem respectively. Now we can formalize the multi-dimensional space point coverage problem:

$$\max \sum_i W_i \cdot \mathbf{I}\Big(p_i \in \mathcal{H}_{(\mathcal{B}-\mathcal{I})\cup\mathcal{E}}\Big), \tag{11}$$

$$s.t. \quad |\mathcal{I}| \le |\mathcal{B}|, \tag{12}$$

$$|(\mathcal{B}-\mathcal{I})\cup\mathcal{E}| \le C. \tag{13}$$

Our objective is to maximize the sum of the weights of points covered by independent hypercubes. We can traverse each point in the multi-dimensional space to count the sum of weights of points covered by independent hypercubes.

D. Theoretical Analysis

We now analyze the computational complexity of the independent rule space transformation problem in the multi-dimensional space. We can divide the multi-dimensional space into multiple subspaces along the boundaries of each hypercube, such that the hypercube division in each subspace does not affect each other. These subspaces are called *isolated blocks*. Figure 3(a) shows the isolated blocks of Figure 2(b). The isolated blocks do not cross each other, and obviously, each independent hypercube only appears in one isolated block. Let us use $\Lambda(k, B, C)$ to represent the problem of selecting C independent hypercubes associated with independent rules in the transformed space from B isolated blocks in the k-dimensional space, which aims to maximize the sum of weights of the number of covered points. We can have the following theorem.

Lemma 1: Problem $\Lambda(2, 1, C)$ is an NP-Hard problem.

Proof: This can be proved by reducing the rectilinear picture compression problem [26], a known NP-Complete problem,

to problem $\Lambda(2, 1, C)$. Given a positive integer C and a matrix with values being either 0 or 1, the rectilinear picture compression problem is whether we can find C or fewer rectangles to only cover all units with the value being 1. As shown in Figure 3(b), for an instance of a rectilinear picture compression problem, now we start to construct the instance of problem $\Lambda(2, 1, C)$. First, we delete all units with the value being 0. Then we can transform the instance into an isolated block in the two-dimensional space. Finally, we convert each unit with the value being 1 in the instance of the original problem to a point with weight being 1 in the instance of problem $\Lambda(2, 1, C)$. Obviously, we can complete the construction of the instance of problem $\Lambda(2, 1, C)$ in polynomial time. Now we prove the equivalence of these two instances. Problem $\Lambda(2, 1, C)$ aims to find a solution that selects C or fewer rectangles that cover the most points in the space (i.e., maximize the sum of weights of the covered point). If a solution of the instance of problem $\Lambda(2, 1, C)$ can cover all points, then the rectilinear picture compression problem has a corresponding solution that only can cover all units with a value being 1, or it does not have a solution otherwise. For a solution of the instance of the rectilinear picture compression problem, there is a corresponding solution that can cover all points of the instance of problem $\Lambda(2, 1, C)$. Therefore, we can reduce the rectilinear picture compression problem to problem $\Lambda(2, 1, C)$. Since the rectilinear picture compression problem is NP-Complete, problem $\Lambda(2, 1, C)$ is NP-Hard. ■

Theorem 1: Problem $\Lambda(k, B, C)$ is an NP-Hard problem.

Proof: We can prove it by contradiction. Assume problem $\Lambda(k, B, C)$ is not an NP-hard problem, there will be a solution for problem $\Lambda(k, B, C)$ with the polynomial time complexity. Since problem $\Lambda(2, 1, C)$ is a case of problem $\Lambda(k, B, C)$, there must be a solution with the polynomial time complexity for problem $\Lambda(2, 1, C)$. However, according to Lemma 1, problem $\Lambda(2, 1, C)$ is an NP-Hard problem, which contradicts the assumption. ■

According to Theorem 1, we know that problem $\Lambda(k, B, C)$ is NP-hard. Therefore, we design an efficient algorithm based on clear physical meanings of rule caching to solve this problem.

IV. DROPS Design

In this section, we propose DROPS (dynamic rule caching for programmable switches), a dynamic rule caching framework that realizes efficient rule placement under rule dependency constraints for programmable switches.

A. Overview of DROPS

According to the analysis in Section III-D, in order to realize efficient rule caching in TCAM, we need to transform the original rule space into the independent rule space, and then select the top C independent hypercubes associated with independent rules with the largest sum of weights of covered points for TCAM, where C is the number of rules that can be placed in TCAM. Note that the weight and spatial distribution of points are dynamically changing because the flow distribution is dynamic. Therefore, the transformation of rule space needs to update dynamically with the flow distribution. To efficiently cache independent rules, we first build a flow-aware decision tree to reduce rule dependency by dividing rule space. Meanwhile, the tree can capture the current flow distribution and further transform the rule space into the independent rule space according to the flow distribution. By selecting the top C independent rules generated by the tree with the largest sum of weights for the covered points, we solve the original optimization problem in Section III.

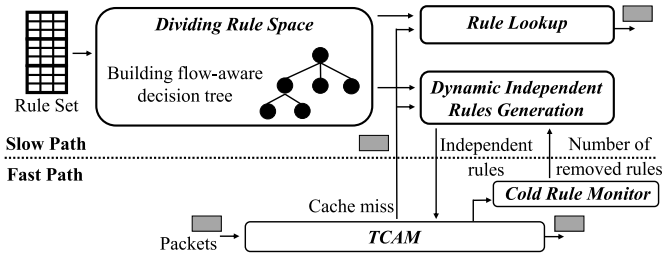
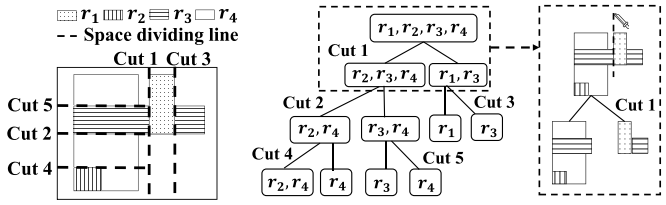


Fig. 4. The framework of DROPS.

Figure 4 shows the framework of DROPS, which consists of three main modules. The *dividing rule space* module builds a flow-aware decision tree to divide the multidimensional space corresponding to the rule set to reduce rule dependency. Based on the flow-aware decision tree built by the *dividing rule space* module, the *dynamic independent rules generation* module captures the distribution of flow in the slow path, and generates independent rules for TCAM according to the number of removed rules in the TCAM monitored by *cold rule monitor* module. The *rule lookup* module enables fast rule matching to forward the packets in the slow path if these rules cannot be cached in the fast path. Note that, the fast path means that packets are processed by the dedicated high-speed hardware switch with TCAM, and the slow path indicates that the packets are processed by software switches.

The *dividing rule space* module divides the multidimensional space corresponding to the rule set by utilizing the flow-aware decision tree. Note that, the pattern of a rule (e.g., source and destination IP address prefixes) represents the dimension in the geometric spaces of the packet header and thus we can use a hypercube in this multi-dimensional space to represent the rule. Thus, we can transform the rule space by dividing the rule space so that we reduce the number of rule dependency relationships. Different from packet classification schemes [27], [28], [29] built upon a decision tree, our flow-aware decision tree transforms the rule space with heavy rule dependency to that with reduced rule dependency, and generates independent rules according to the current flow distribution, which cannot be achieved by packet classification schemes.

The *dynamic independent rules generation* module captures flow distribution and generates independent rules associated with each subspace in the leaf node of the tree instead of all rule space. Due to the locality of flow distribution, we only need to select a small number of leaf nodes (i.e., rule subspace) to generate independent rules. It assigns higher priorities to the



(a) Space division scheme (b) Building the flow-aware decision tree

Fig. 5. The example of rule space division.

leaf node if the corresponding space contains the points with a large sum of weights. Thus, we can generate independent rules based on flow distribution. Since there does not exist rule dependency in independent rules, the insertion and update of rules in TCAM will not incur significant movement of TCAM entries. Therefore, DROPS can efficiently generate and place independent rules for TCAM.

Meanwhile, the *rule lookup* module lookups at the decision tree and identifies the correct rule in leaf nodes if the corresponding rule is not cached in TCAM. Thus, we can enable fast rule matching for packet forwarding in the slow path.

Algorithm 2: Dividing Rule Spaces

Input:
$\mathcal{O}$: the rule set
H: the height of decision tree
L: maximum number of rules associated with leaf nodes
Output:
Γ: the flow-aware decision tree

```
Procedures:
Function BuildTree(Γ, O, H, L):
    if |O| ≤ L or Γ.h ≥ H then
        return
    end
    n ← Number of dimensions of the rule space
    S1, S2 ← ∅
    for i ∈ [1, n] do
        N1, N2 ← Divide the rule set evenly on the dimension i
        if ||N1| − |N2|| < ||S1| − |S2|| then
            S1, S2 ← N1, N2
        end
    end
    Γ.lchild ← NewTreeNode(S1)
    Γ.rchild ← NewTreeNode(S2)
    BuildTree(Γ.lchild, S1, H, L)
    BuildTree(Γ.rchild, S2, H, L)
Function Main:
    Γ ← NewTreeNode()
    BuildTree(Γ, O, H, L)
    return Γ
```

B. Dividing Rule Space

We build a flow-aware decision tree to reduce rule dependency by dividing rule space, in which each node is associated with a subspace of multi-dimensional space corresponding to the rule set. In Section III-C, we have shown that a pattern of a rule (e.g., source and destination IP address prefixes) represents a dimension in the geometric space associated with the packet header. According to one value in a certain dimension, we can divide the multi-dimensional space into two non-overlapping subspaces, which can further be divided. For example, as shown in Figure 5(a), we can divide a two-dimensional space based on the rule set in Figure 2(a), and the space dividing lines (i.e., dashed lines in the figure) can effectively divide the space.

The *dividing rule space* module in DROPS only considers the boundaries of the hypercubes corresponding to the rules as the space dividing lines. Let us take a rule r with pattern fields `10**` and `1***` as an example. The upper of the hypercube corresponding to rule r in the first and second dimensions are `1011` and `1111`, respectively, while the lower boundaries in the first and second dimensions are both `1000`. Note that, we need to build a balanced decision tree since the balanced tree with the smaller maximum depth has better lookup performance [30]. To achieve this, DROPS divides the rule space by selecting the boundary that can evenly divide rules.

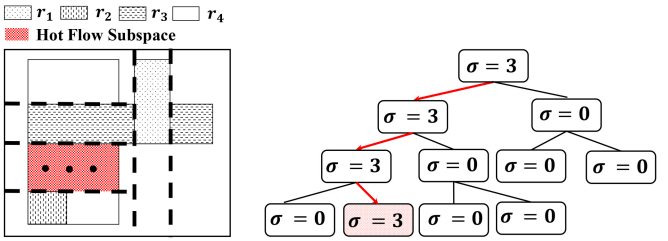
Figure 5(b) shows an example of the rule space dividing process. First, we divide the entire rule space, which represents the root of a tree, into two subspaces along the boundary of the rule r_1 on the dimension F_1. It leads to the creation of two children. If a rule intersects a child's subspace, it is added to that child. For example, rules r_2, r_3 and r_4 intersect in the first subspace (i.e., the first half in this space), and thus they are all added to the first root's child. If a rule intersects in multiple subspaces, it is added to each corresponding child, e.g., rule r_3 is added to the first and second children. We repeatedly divide each subspace until the number of rules does not exceed L or the depth of the tree reaches H.

The pseudo-code of the rule space division algorithm is shown in Algorithm 2. For a given rule set $\mathcal{O}$, the algorithm outputs the flow-aware decision tree representing the division result of the corresponding multi-dimensional space associated with the rules in rule set $\mathcal{O}$. We utilize a recursive tree-building algorithm to divide the space. Given a node Γ, we search all dimensions and calculate division values, i.e., the boundary that divides rules evenly in each dimension. Then we select the dimension d if there exists the smallest difference in the number of rules associated with child nodes after division in d. Finally, we divide node Γ based on the boundary value of rule r in dimension d, and recursively divide the child nodes until the number of rules does not exceed L or the node's depth reaches H. Therefore, for a node with a depth of H, the number of rules associated with it may exceed L. DROPS constraints the tree depth to ensure real-time lookup performance. We analyze the impact of parameters L and H in Section V-B.

DROPS can adapt to rule changes and enable real-time rule updates. Specifically, when a rule changes, DROPS locates the smallest tree node containing the change and incrementally updates the subtree corresponding to the node. Leveraging the advantages of the balanced tree, the rule space division of DROPS is efficient and can be performed real-time.

C. Independent Rule Generation for Rule Caching in TCAM

Since the flow distribution changes over time, we need to dynamically update rules in TCAM. Therefore, we periodically generate new independent rules that can be placed in TCAM according to the flow-aware decision tree in each



(a) Hot flow subspace (b) Locating the hot flow subspace

Fig. 6. The example of independent rule generation.

time window. The *cold rule monitor* module identifies and removes cold rules in TCAM according to the number of flows associated with the rules during each time window, and the *dynamic independent rules generation* module generates the same number of independent rules to that of the removed cold rules. Note that, similar to [7], [31], [32], we utilize a hash table-based *cold rule monitor* module to count the hits of rules in TCAM during each time window.

Let λ denote the number of new independent rules to be generated based on the flow distribution in the last time window. The flow distribution in the time window can be converted into multiple points in the multi-dimensional space. Thus, we can easily count the sum of weights of points associated with each subspace of the leaf node in the flow-aware decision tree. Due to the locality of the flow distribution [33], a small number of leaf nodes always contain most packets, i.e., the sum of weights of points. To generate λ independent rules, we select the top C leaf nodes with the largest sum of weights of points in the flow-aware decision tree. Note that DROPS generates independent rules based on the traffic that did not hit the cache. Therefore, the newly generated independent rules by DROPS are different from the cached rules.

We define the *hot flow subspace* as the subspace associated with the leaf node with the maximum sum of weights of points. To fast locate the hot flow subspace, each node Γ in the flow-aware decision tree is associated with an integer σ:

$$\Gamma.\sigma = \begin{cases} \sum_i W_i \cdot \boldsymbol{I}(p_i \in \Gamma) & \Gamma \text{ is a leaf node,} \\ max(\Gamma.lchild.\sigma, \Gamma.rchild.\sigma) & \text{else.} \end{cases} \tag{14}$$

Note that, if Γ is a leaf node, the value of the associated σ is the sum of weights of points in Γ. Otherwise, the value of σ is the largest σ in the children of Γ.

Figure 6 shows an example of independent rule generation. We start from the root node of the flow-aware decision tree and search recursively along the child node with the largest σ, which allows us to quickly locate the hot flow subspace and generate the corresponding independent rule. We reset the σ of the leaf node to 0 and update the corresponding nodes according to Equation (14).

The pseudo-code of the independent rule generation algorithm is presented in Algorithm 3. For the flow set $\mathcal{F}$ in the slow path in a time window, we count the number of packets (i.e., the sum of weights of points) associated with each leaf node in the decision tree (the function *LeafNodeLocate* is given in Algorithm 4), and then update the value σ of all the non-leaf node. We utilize the independent rule generation function λ times to generate λ independent rules since we removed λ rules in TCAM and we can insert λ new independent rules in TCAM. To achieve this, we locate the hot flow subspace by recursively searching the child node with maximum σ and calculate the independent rule associated with the hot flow subspace. Especially, for the leaf node with more than one rule, we calculate the independent rule associated with the largest subspace based on the point with the largest weight in the hot flow subspace. The greedy strategy of DROPS cannot achieve optimal independent rule generation but ensures the real-time

Algorithm 3: Independent Rule Generation

Input:
Γ: the flow-aware decision tree
λ: the number of independent rules that need to be generated
$\mathcal{F}$: the flow set in a time window
Output:
$\mathcal{U}$: the generated independent rule set

```
Procedures:
Function GenerateRule(Γ):
    if Γ is a leaf node then
        r ← CallIndependentRule(Γ)
        Γ.σ ← 0
        return r
    end
    Γ.σ ← 0
    if Γ.lchild.σ < Γ.rchild.σ then
        r ← GenerateRule(Γ.lchild)
    else
        r ← GenerateRule(Γ.rchild)
    end
    Γ.σ ← max(Γ.lchild.σ, Γ.rchild.σ)
    return r
Function Main:
    U ← ∅
    for f ∈ F do
        LeafNodeLocate(Γ, f)// Update the leaf node
            containing f
    end
    for i ∈ [0, λ − 1] do
        U ← U + GenerateRule(Γ)
    end
    return U
```

Algorithm 4: Rule Lookup in Slow Path

Input:
Γ: the decision tree
p: the packet
Output:
r: the matched rule

```
Procedures:
Function LeafNodeLocate(Γ, p):
    if Γ is a leaf node then
        Γ.σ ← Γ.σ + 1
        r ← locate the matched rule
        return r
    end
    if p in Γ.lchild.ranges then
        r ← LeafNodeLocate(Γ.lchild, p)
    else
        r ← LeafNodeLocate(Γ.rchild, p)
    end
    Γ.σ ← max(Γ.lchild.σ, Γ.rchild.σ)
    return r
```

TABLE III
THE RULE SET AND TRACES IN EXPERIMENT

Rule Set	Number of Rule	Policy	Traces Source	Number of Packet	Number of Flow
Equinix	709K	DIP, real-world	CAIDA, real-world	87,514,817	1,094,110
FWD	180K	DIP, real-world	ClassBench, synthetic	30,911,835	5,844,575
ACL	100K	5-tuple, synthetic	ClassBench, synthetic	10,997,553	61,632
FW	88K	5-tuple, synthetic	ClassBench, synthetic	10,569,720	469,756

generation of independent rules. Finally, we set the value of σ in the leaf node associated with the hot flow subspace to 0.

D. Rule Lookup in Slow Path

Now we design a rule lookup algorithm to realize fast packet forwarding in the slow path. For an incoming packet, TCAM will be looked up for rule matching. If the packet does not match the rules in the TCAM, it will be processed by the *rule lookup* module to match rules based on the flow-aware decision tree. Since all rules in the rule set are stored in the decision tree, packets can always match the correct rule. Based on our analysis in Section III-C, a packet is represented as a point in the rule space. Therefore, the rule lookup in the slow path is equivalent to locating the point in the rule space. Note that, since the decision tree is a balanced structure, the lookup module can quickly locate the leaf node containing the point from the tree. Thus, it performs efficient packet classification according to the rule corresponding to the located leaf node.

The pseudo-code of the locating leaf node algorithm is shown in Algorithm 4. For each packet f in $\mathcal{F}$, we recursively traverse the tree from the root node to locate a leaf node. According to the rule space division algorithm (see Section IV-B), we locate a point associated with the packet header of f in the multi-dimensional space associated with the current node Γ and use the corresponding value of the point in the division dimension of node Γ to locate the next child node. We can repeatedly look up the nodes in different division dimensions until we can identify a leaf node. Therefore, we can quickly match the rules corresponding to the fields in the packet header of each packet.

V. EVALUATION

In this section, we evaluate DROPS in comparison with existing schemes through simulations. Moreover, we prototype our framework in a hardware programmable switch and use a real testbed built upon the hardware switches to demonstrate the effectiveness and performance of our framework.

A. Experimental Setup

Implementation. We first compare DROPS with existing schemes through software simulation, including dependent-set (DS) and cover-set (CS) based method of CacheFlow [15], which is designed with a greedy algorithm based on rule dependence, and isolate rule (IR) based method of T-Cache [7], which calculates isolate-rule for each individual hot flow to eliminate rule dependence. For DROPS, we set the height of the flow-aware decision tree H to 30 and the maximum number of rules associated with the leaf nodes L to 3. We implement them using Python. In particular, we use 2000 LoCs to implement DROPS. Moreover, we implement simulators on the commercial cloud server with Ubuntu 20.04.1-LTS operating system to run all these methods.

We prototype DROPS and the existing schemes in a programmable 32 $\times$ 100 Gbps switch EdgeCore Wedge100BF-32X. We use the P4-16 language to implement packet processing and forwarding, and we set the matching type to ternary to ensure that the lookup table is handled by TCAM. By setting the default matching action to the CPU port, the unmatched packets will be punted to the slow path. In particular, the independent rules generated by DROPS are assigned the same priority value.

Datasets. We show the rule set and packet traces used in our experiments in Table III. We use four rule sets in experiments. The Equinix datacenter routing table [23] includes 709K IP prefixes. The CAIDA traffic traces were collected from an OC-192 backbone link of a Tier 1 ISP at Equinix datacenter on March 15, 2018 from 13:00 to 13:30 UTC. We use 1-minute window traffic with 87 million packets and 30-minutes window traffic with 2 billion packets for static and dynamic experiments, respectively. We utilize the standard benchmark of ClassBench [24] to generate various types of rule sets, which is widely used [7], [28], [29], [34]. We generate two types of rule sets, i.e., Access Control List (ACL), and Firewall (FW) datasets, each of which is consistent with the real-world rule sets. Meanwhile, we obtain a rule set of Forwarding (FWD), which is the real routing table from a real-world Cisco router configuration on a Stanford backbone network [35]. FWD has around 180K rules that are used to forward packets according to destination IP addresses. For the above three rule sets, we use ClassBench to generate corresponding packet traces, in which the flow volume associated with each rule follows the Zipf distribution. We replay these traces to measure the performance.

B. Simulation Results

Cache Hit Ratios without Rule Update. Suppose the traffic information is known in advance. We measure the best-case cache hit ratio of four schemes on four rule sets. Figure 7 shows the TCAM hit ratio on four rule sets. Figure 7(a) shows the cache hit ratios on the rule set of Equinix. When the size of TCAM is 100, DS, CS, IR and DROPS achieve 37.69%, 37.80%, 52.20%, 62.55% hit ratio, respectively. We observe that the hit ratio of DS and CS are significantly lower than IR and DROPS, because DS and CS retain the rule dependency, which causes a large number of cold rules to be cached in TCAM. Figure 7(b) shows the cache hit ratio on the rule set of FWD. In this case, DROPS significantly

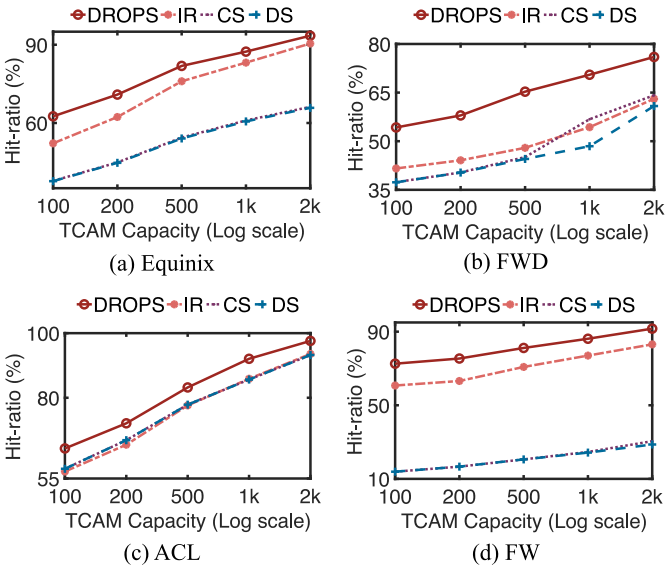


Fig. 7. Comparison of cache hit ratio in the absence of rule update.

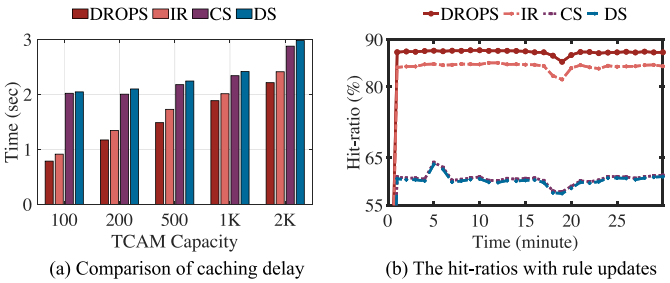


Fig. 8. Evaluation of caching delays and hit ratios with rule updates.

outperforms other methods, because DS, CS, IR ignore the spatial locality of traffic while DROPS can accurately identify the hot flow subspaces and generate independent rules with a high hit ratio. For example, when the TCAM size is 2000, DS, CS, and IR achieve only 60.85%, 64.18% and 63.08% TCAM hit ratio, respectively, while DROPS can achieve a hit ratio more than 77.99%. Thus, DROPS achieves the highest hit ratio. Note that, since IR does not consider the overall flow distribution and only generates independent rules based on a single flow heuristically, the achieved hit ratio is lower. Figure 7(c) shows the cache hit ratio on the multi-field rule table, i.e., ACL. In this case, our scheme outperforms DS, CS, and IR even though the TCAM hit ratio of DROPS is close to other methods due to the high locality of traces with 10 million packets and only 61K flow. When the TCAM size is 2000, all methods achieve hit ratios of more than 93%. Similarly, as shown in Figure 7(d), DROPS and IR can achieve hit ratios of more than 83% on the rule set of FW with TCAM capacity 2k. Due to strong dependency, DS and CS only achieve hit ratios of 28.69% and 30.47%.

The Caching Delays during Bootstrapping. In this experiment, we measure the delay of filling caching in TCAM. Note that, when the cache is empty, all new arriving flows will trigger cache misses, which incurs a delay to select rules and fill the cache in TCAM. Figure 8(a) shows that DROPS

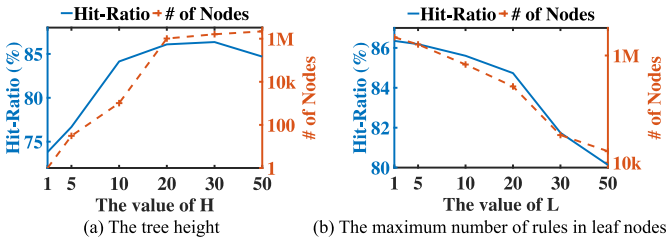


Fig. 9. The Impact of Parameter Setting of Decision Tree.

incurs a much shorter delay than DS, CS, and IR to fill the cache to increase the hit ratio on the rule set of Equinix with 709K rules. For instance, when TCAM size is 100, DS, CS, and IR need 2.05, 2.02, and 0.91 seconds. respectively to achieve a high TCAM hit ratio, while our scheme only needs 0.79s. In particular, the implementation using lower-level languages, such as C and C++, can speed up the process. The results are reasonable because DROPS only needs to generate independent rules based on the balanced decision tree, which requires much less computation time than DS, CS, and IR. Although DROPS requires 5x memory of DS and CS to build the flow-aware decision tree, the memory overhead can be ignored due to the adequate memory of a slow path (e.g., CPU).

Cache Hit Ratios with Rule Update. In order to compare DROPS with DS, CS, and IR, the above experiments are carried out under a relatively static situation. Now we run the four systems in a dynamic environment and the results are shown in Figure 8(b). Under 709K rules with 30 minutes high-speed traffic flows with 2 billion packets, DROPS can achieve an average TCAM hit ratio of about 88.3% with 1K TCAM entries, while the average TCAM hit ratio of IR, CS and DS are only 83.9%, 61.6% and 59.8% with the same TCAM capacity. The reason is that the competition among rules in TCAM is more intensive for DS and CS, which leads to frequent caching and eviction of rules in a dynamic flow environment. Since DROPS generates independent rules based on hot flow subspaces, each rule potentially has a higher hit rate than rules generated by other schemes such as IR.

The Impact of Parameter Settings. We measure the effect of parameter settings of the flow-aware decision tree on DROPS performance. As shown in Figure 9(a), we show DROPS performance on the different heights of decision tree H. DROPS achieves the maximum hit ratio 86.35% when the tree height H is 30. When the tree height H is small, the decision tree cannot sufficiently divide the rule space so that it loses the ability to sense heat flow subspaces. Furthermore, the hot flow subspaces will be small with large tree height H, and we can only achieve the local optimum of the hit ratio, thus reducing the overall performance. Meanwhile, too large tree height will incur many nodes. For instance, when the tree height is 50, the number of nodes is 2,190,821.

Figure 9(b) shows the effect of the maximum number of rules in the leaf nodes L setting on DROPS performance. We observe that the hit ratios slightly change with different L. For instance, the cache hit ratio is 86.35% with a value of L 1, and the cache hit ratio reaches 84.74% with a value of L 10.

TABLE IV
THE PERFORMANCE OF REAL TESTBED

Scheme	Delays of cold start	Throughput	Hit ratio
DS	3555.08 ms	24.56 Gbps	60.59%
CS	3230.37 ms	25.97 Gbps	63.61%
IR	1418.42 ms	33.02 Gbps	82.14%
DROPS	**1357.43 ms**	**36.38 Gbps**	**88.45%**

However, adopting a too-small value of L will make too many nodes in the decision tree, which will incur a large overhead. For example, when the value of L is 3, the number of nodes is 1607727, while the number of nodes is 16383 with the value of L 50.

C. Real Testbed Results

In the real-world experiment, we set a Barefoot Tofino switch chip as the fast path and the Intel Xeon Gold 6258R CPU as the slow path. We set the egress port of the default matching action of the switch to the CPU port so that packets that do not match the cache will be sent to the slow path. The software part of all schemes is implemented using Python 3.8, while the hardware part is implemented using the P4-14 language. The data plane TCAM has 1024 entries. We replay the traces by using DPDK [36].

We evaluate the performance of DROPS, DS, CS, and IR in the real testbed. We use real-world traffic traces and rules and measure the cold start delay,[3] the total hit ratio, and the throughput, which validates our simulation results and demonstrates the effectiveness and feasibility of DROPS. To cope with cold starts, we set a time window of 1 second in the beginning. All schemes calculate rules based on traffic distribution during the time window and insert the rules into the TCAM. The TCAM will not evict any existing rules until it is full. The delay of cold start includes the calculation delay in the software part and the rule insertion delay in the hardware part.

As shown in Table IV, DROPS and IR only take 1357.43 milliseconds and 1418.42 milliseconds to complete the cold start process, while DS and CS consume 3555.08 milliseconds and 3230.37 milliseconds, respectively. Since the DS and CS schemes maintain dependencies and priorities among rules, rule insertion causes rule movement in the TCAM, resulting in significant time overhead. Compared to IR, DROPS incurs a smaller time overhead in rule calculation. Therefore, DROPS is more efficient in cold starts compared to existing schemes. Furthermore, we observe that DROPS significantly improves the hit ratios and the throughput. It achieves 88.45% hit ratio, while DS, CS and IR only achieve 60.59%, 63.61%, and 82.14% hit ratio, respectively, which is consistent with our simulation results (see Section V-B). Meanwhile, DROPS achieves 48.1%, 40.1%, and 10.2% higher throughput than DS, CS and IR, respectively. The throughput difference is related to the cache hit ratio and rule update delay of different schemes. DROPS eliminates rule dependencies by constructing a flow-aware decision tree and generates independent rules in real-time based on dynamic traffic distribution, resulting in a higher cache hit ratio and reducing the number of packets processed by the slow path. Furthermore, DROPS has higher efficiency in rule calculation and updating, reducing the time overhead of rule updates. Therefore, DROPS significantly outperforms the existing schemes.

[3] *Cold start* denotes the procedure of placing the initial rules.

VI. RELATED WORK

Rule Compression. Compacting rule tables have been extensively studied to alleviate the TCAM memory limit [18], [37], [38], [39], [40]. For example, Dong et al. [18] construct compressed rules for evolving flows to improve TCAM utilization. However, it cannot handle high dynamic flows with a high churn rate of rules, which is well addressed in this paper. Furthermore, multi-dimensional packet classification rules can be compressed through decision trees [27], [28], [29]. HiCuts [27] cuts the space of each node in one dimension to create multiple equal-sized subspaces to separate rules, and NuevoMatch [14] applies neural networks to the virtual network switch. NeuroCuts [29] utilizes reinforcement learning to build a decision tree for packet classification. RQ-RMI [41] accelerates packet classification based on the Range-Query RMI machine learning model. Rottenstreich et al. [42] utilize spare resources at the fast data plane for cooperative rule caching. Unfortunately, they fail to remove rule dependency and generate independent rules for efficient rule caching. Kang et al. [43] design the policy transformation in Software Defined Networks (SDN), which also fails to remove rule dependency and cannot be applied to programmable switches.

Efficient Rule Placement. Traditional static flow placement [44], [45], [46] is unable to efficiently utilizes TCAM. Rule dependency [15], [47] and popular rule placement [15], [48], [49] have been studied to address this issue. Rottenstreich et al. [48] improved hit ratios by increasing the number of switches and designing a rule placement algorithm based on cooperative caching among switches. Kang et al. [49] achieve efficient rule placement through hierarchical rule management and optimized rule distribution. These methods still cannot effectively eliminate rule dependencies and do not support efficient dynamic rule caching, making them unsuitable for real-world deployment in programmable switches. Katta et al. [15] design a CacheFlow system, which optimizes TCAM hit ratios by splicing dependency chains and caching smaller groups of rules. However, they do not consider the overhead of rule updates and cannot achieve efficient rule caching in TCAMs because it will incur significant entry updating in TCAMs due to rule dependency. Wu et al. [50] apply the dynamic programming algorithm to generate and place rules without dependencies, which results in a significant time overhead that cannot be realistically applied to programmable switches.

Dynamic Rule Update. A number of studies have been proposed to eliminate the unnecessary rule moves incurred by rule updates. Ding et al. [22] update rules for the purpose of minimizing the rule update overhead when selecting cached

rules. However, it sacrifices update latency to achieve high TCAM's hit ratios. Mercury [51] maps a logical TCAM flow table into two physical flow tables with different capacities. The small flow table is used for rule updates to reduce update overhead, and the rules of the small flow table are regularly migrated to the large flow table for matching traffic. He et al. [25] performed an in-depth analysis of the TCAM update problem and design a rule update algorithm based on the partial order theory to reduce the number of TCAM moved entries. T-Cache [7] selects an isolated rule that has the most extensive coverage for individual hot flow, which eliminates the rule dependency. However, generated rules in T-Cache cannot effectively improve the hit ratios due to disregarding the flow distribution. To sum up, these methods incur significant computation overhead or resource consumption. DROPS addresses these issues by constructing the flow-aware decision tree, which allows us to maximize hit ratios of TCAM, and significantly reduce the update overhead.

VII. Conclusion

In this paper, we utilize a modeling-based method to formulate the rule caching problem in programmable switches and prove its NP-hardness. In order to solve the problem, we propose DROPS built upon efficient algorithms to transform all rules into independent rules that are non-overlapping with each other and cache independent rules in TCAM in the granularity of independent rules. DROPS ensures that rules cached in switches always with a higher hit ratio and rule insertion will not interfere with cached rules. We use experiments with datasets and real hardware switch testbed to demonstrate that DROPS significantly outperforms all existing methods, which sheds light on developing a general framework for efficient rule caching in real hardware switches.

References

[1] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[2] Y. Li et al., "HPCC: High precision congestion control," in *Proc. ACM Spec. Interest Group Data Commun.*, 2019, pp. 44–58.

[3] X. Gao et al., "Switch code generation using program synthesis," in *Proc. Annu. Conf. ACM Spec. Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, 2020, pp. 44–61.

[4] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proc. Symp. SDN Res.*, 2016, pp. 1–12.

[5] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Spec. Interest Group Data Commun.*, 2017, pp. 15–28.

[6] M. Zhang et al., "Poseidon: Mitigating volumetric DDoS attacks with programmable switches," in *Proc. NDSS*, 2020, pp. 1–18.

[7] Y. Wan et al., "T-cache: Dependency-free ternary rule cache for policy-based forwarding," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2020, pp. 536–545.

[8] X. Deng et al., "Robust multi-tab Website fingerprinting attacks in the wild," in *Proc. IEEE Symp. Security Privacy (SP)*, 2023, pp. 1005–1022.

[9] X. Deng, Q. Li, and K. Xu, "Robust and reliable early-stage Website fingerprinting attacks via spatial-temporal distribution analysis," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2024, pp. 1–15.

[10] Y. Qing et al., "Low-quality training data only? A robust framework for detecting encrypted malicious network traffic," in *Proc. NDSS*, 2024, pp. 1–18.

[11] C. Fu, Q. Li, M. Shen, and K. Xu, "Detecting tunneled flooding traffic via deep semantic analysis of packet length patterns," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2024, pp. 1–15.

[12] Q. Li et al., "Dynamic network security function enforcement via joint flow and function scheduling," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 486–499, 2022.

[13] F. Long, Z. Sun, Z. Zhang, H. Chen, and L. Liao, "Research on TCAM-based OpenFlow switch platform," in *Proc. Int. Conf. Syst. Inform. (ICSAI2012)*, 2012, pp. 1218–1221.

[14] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "Scaling open {*vSwitch*} with a computational cache," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI 22)*, 2022, pp. 1359–1374.

[15] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proc. Symp. SDN Res.*, 2016, pp. 1–12.

[16] R. Li, Y. Pang, J. Zhao, and X. Wang, "A tale of two (flow) tables: Demystifying rule caching in openflow switches," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.

[17] S. Yang et al., "FISE: A forwarding table structure for enterprise networks," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 2, pp. 1181–1196, Jun. 2020.

[18] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, "Wire speed packet classification without TCAMs: A few more registers (and a bit of logic) are enough," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, pp. 253–264, 2007.

[19] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[20] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, 2013, pp. 43–48.

[21] X. Jin et al., "Dynamic scheduling of network updates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 539–550, 2014.

[22] Z. Ding, X. Fan, J. Yu, and J. Bi, "Update cost-aware cache replacement for wildcard rules in software-defined networking," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, 2018, pp. 457–463.

[23] "The CAIDA UCSD Anonymized Internet traces sampler," Caida, Dataset, 2018. [Online]. Available: http://www.caida.org/data/passive/passive_sampler_dataset.xml

[24] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.

[25] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast TCAM updates," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 217–230, Feb. 2018.

[26] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang, "Compressing rectilinear pictures and minimizing access control lists," in *Proc. 18th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2007, pp. 1066–1075.

[27] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," presented at the Hot Interconnects VII, 1999.

[28] W. Li, X. Li, H. Li, and G. Xie, "Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2018, pp. 2645–2653.

[29] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *Proc. ACM Spec. Interest Group Data Commun.*, 2019, pp. 256–269.

[30] Z.-J. Lin and C.-Y. Yu, "An improved support vector machines with balanced binary decision tree for multi-class classification," *J. Chin. Comput. Syst.*, vol. 35, no. 5, pp. 1124–1127, 2014.

[31] X. Wen et al., "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2016, pp. 179–188.

[32] V. S. Srinivasavarma and S. Vidhyut, "A TCAM-based caching architecture framework for packet classification," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 1, pp. 1–19, 2020.

[33] A. Appadurai, "The production of locality," in *Counterworks: Managing the Diversity of Knowledge*, vol. 204. New York, NY, USA: Routledge, pp. 204–225, 1995.

[34] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2003, pp. 213–224.

[35] "Stanford backbone router forwarding configuration." 2014. [Online]. Available: http://tinyurl.com/oaezlha

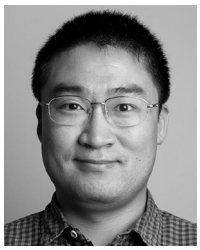
[36] I. Cerrato, M. Annarumma, and F. Risso, "Supporting fine-grained network functions through Intel DPDK," in *Proc. 3rd Eur. Workshop Softw. Defined Netw.*, 2014, pp. 1–6.

[37] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proc. Int. Conf. Distrib. Comput. Netw.*, 2013, pp. 439–444.
[38] 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.
[39] C. R. Meiners, A. X. Liu, and E. Torng, "Topological transformation approaches to TCAM-based packet classification," *IEEE/ACM Trans. Netw.*, vol. 19, no. 1, pp. 237–250, Feb. 2011.
[40] O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 864–878, Apr. 2017.
[41] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "A computational approach to packet classification," in *Proc. Annu. Conf. ACM Spec. Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, 2020, pp. 542–556.
[42] O. Rottenstreich, A. Kulik, A. Joshi, J. Rexford, G. Rétvári, and D. S. Menasché, "Data plane cooperative caching with dependencies," *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 3, pp. 2092–2106, Sep. 2022.
[43] N. Kang, J. Reich, J. Rexford, and D. Walker, "Policy transformation in software defined networks," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2012, pp. 309–310.
[44] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet classifiers in ternary CAMs can be smaller," in *Proc. Joint Int. Conf. Meas. Model. Comput. Syst.*, 2006, pp. 311–322.
[45] F. Chang, W.-c. Feng, and K. Li, "Approximate caches for packet classification," in *Proc. IEEE INFOCOM* , 2004, pp. 2196–2207.
[46] J. Xu, M. Singhal, and J. Degroat, "A novel cache architecture to support layer-four packet classification at memory access speeds," in *Proc. IEEE INFOCOM Conf. Comput. Commun. 19th Annu. Joint Conf. Comput. Commun. Soc. (Cat. No. 00CH37064)*, 2000, pp. 1445–1454.
[47] J.-P. Sheu and Y.-C. Chuo, "Wildcard rules caching and cache replacement algorithms in software-defined networking," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 1, pp. 19–29, Mar. 2016.
[48] O. Rottenstreich, A. Kulik, A. Joshi, J. Rexford, G. Rétvári, and D. S. Menasché, "Cooperative rule caching for SDN switches," in *Proc. IEEE 9th Int. Conf. Cloud Netw. (CloudNet)*, 2020, pp. 1–7.
[49] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the 'one big switch' abstraction in software-defined networks," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol.*, 2013, pp. 13–24.
[50] J. Wu, Y. Chen, and H. Zheng, "Approximation algorithms for dependency-aware rule-caching in software-defined networks," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, 2018, pp. 1–6.
[51] H. Chen and T. Benson, "The case for making tight control plane latency guarantees in SDN switches," in *Proc. Symp. SDN Res.*, 2017, pp. 150–156.

Xinhao Deng (Student Member, IEEE) is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace, Tsinghua University. He is interested in research on network security and privacy.

Mingwei Xu received the B.S. and Ph.D. degrees from Tsinghua University, Beijing, China, where he is currently a Full Professor with the Department of Computer Science. He is the Winner of National Science Foundation for Distinguished Young Scholars of China. He is an Executive Director of China Institute of Communications. His research interests include Internet architecture, Internet routing, and Cybersecurity.

Qi Li (Senior Member, IEEE) received the Ph.D. degree from Tsinghua University, where he is currently an Associate Professor with the Institute for Network Sciences and Cyberspace. His research interests include network and system security, particularly Internet security, mobile security, and machine learning security. He is currently an Editorial Board Member of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, *ACM Transactions on Privacy and Security*, and *ACM Digital Threats: Research and Practice*, and has served as an Organization or Program Committees for various premier conferences.



Weijie Wu received the B.Sc. degree in electronics from Peking University, and the Ph.D. degree in computer science from The Chinese University of Hong Kong. He worked as an Independent Researcher in this paper. He is also a Principal Software Engineer with Roblox Corporation, CA, USA. Before that, he held research or faculty positions at ZeroEx Inc., Huawei Technologies, Shanghai Jiao Tong University, and National University of Singapore. His research interests include modeling and performance evaluation of computer networks, resource allocation in networked systems, and network economics.

Yuan Yang received the B.Sc., M.Sc., and Ph.D. degrees from Tsinghua University, where he is currently an Assistant Researcher with the Department of Computer Science and Technology. He was a Visiting Ph.D. Student with The Hong Kong Polytechnic University. His major research interests include computer network architecture and routing protocols.

Menghao Zhang received the B.S. and Ph.D. degrees in computer science from Tsinghua University in 2016 and 2021, respectively. He is currently an Associate Professor with the School of Software, Beihang University. His research interests include programmable networks, high-performance networks, networked systems, and network security.

Yu Zhou (Member, IEEE) received the B.S. degree from the School of Information and Communication Engineering, Beijing University of Posts and Telecommunications, Beijing, China, in 2016 and the Ph.D. degree from the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include software-defined networking and programmable data planes.

Jianping Wu (Fellow, IEEE) is a Computer Network Expert, an Academician of the Chinese Academy of Engineering in 2015, an International Fellow of the Royal Academy of Engineering, U.K., in 2019, and the Chair Professor at Tsinghua University. He is also serving as the Director for National Engineering Research Center of Next Generation Internet Core Technologies, China, the Director for CERNET National Network Center, and the Chair for CERNET Technical Board. He is one of the pioneers of the Internet development and cyberspace security engineering in China. He received one award of the Second Prize of the National Award for Technological Invention, and three awards of the Second Prize of the National Award for the Advancement in Science and Technology. He was selected the Outstanding Youth by the National Natural Science Foundation of China in 1998. He received the Chair Professorship Award for Cheung Kong Scholars Program jointly established and sponsored by Ministry of Education and Hong Kong Cheung Kong Infrastructure Holdings Limited in 2000. He received the Prize for Scientific and Technological Progress of Ho Leung Ho Lee Foundation in 2008, the Jonathan B. Postel Service Award of ISOC in 2010, and was inducted into Internet Hall of Fame in 2017. He was honored the National Award for Excellence in Innovation in 2017, and Tsinghua University Award for Outstanding Contributions in 2021.