RuleOut Forwarding Anomalies for SDN

Shaoke Xi, *Student Member, IEEE*, Kai Bu, *Member, IEEE*, Wensen Mao, Xiaoyu Zhang, Kui Ren, *Fellow, IEEE*, Xinxin Ren

***Abstract*—Reliable Software-Defined Networking (SDN) should mitigate forwarding anomalies due to cross-plane rule inconsistencies. Most existing countermeasures either inject probe packets to infer forwarding correctness or collect packet traces to detect forwarding anomalies. They, however, cannot detect or filter forwarding anomalies for production packets in real time. In this paper, we propose RuleOut as the first attempt to automatically throttle SDN forwarding anomalies. It disambiguates dependent rules via augmenting their matching fields with unique tags. Leveraging source routing, we further bind each packet with the tag sequence corresponding to rules the packet should match. RuleOut thus renders each packet to match at most one rule on each switch. This completely addresses the root cause of forwarding ambiguity. To implement RuleOut, we develop a non-overlapping rule dependency graph, a series of algorithms for incremental rule update and tag generation upon it, and various optimization techniques toward scalability and efficiency. We prototype RuleOut on the Ryu controller and Open vSwitch and evaluate its performance over public rule sets such as Stanford, Internet2, and Airtel1. RuleOut can use tags of only several bits long to disambiguate thousands to millions of rules and generate tags fairly fast within a few milliseconds.**

***Index Terms*—Software-Defined Networking, forwarding fault, source routing.**

I. Introduction

Software-Defined Networking (SDN) has long been susceptible to forwarding anomalies due to rule inconsistencies between the control and data planes [1]–[16]. In SDN, the control plane (i.e., the controller) translates network management polices into packet forwarding rules [17]. It then populates these rules to data-plane switches. Once the rules that actually take effect on the switches differ from what have been populated by the controller, a rule inconsistency occurs and violates network management policies. The so caused forwarding anomalies can lead to performance degradation and security breaches [11].

SDN forwarding anomalies become a practical concern because various root causes of rule inconsistencies have been found on commercial switches. For example, some switches acknowledge rule installation to the controller earlier than rules take effect [2]. This leads to transient rule inconsistencies. Furthermore, permanent rule inconsistencies can be induced by certain commercial switches that do not correctly enforce the priority order of rules [11], [18]. They may simply regard the latest installed rules as the higher priority ones [18] or incorrectly enforce rules with a large priority as the lowest-priority ones [11]. Permanent inconsistencies are also destined when bit-flip errors occur on Ternary Content-Addressable Memory (TCAM), where rules are stored [19].

Unlike abrupt network malfunctions such as switch crashes and link failures, forwarding anomalies caused by rule inconsistencies are stealthy and hard to pinpoint. No matter how thoroughly the correctness of the controller codebase [11], [20]–[22], applications [23], [24], and rules [25]–[29] is verified, we still cannot ensure that rules will faithfully take effect on switches. A well-explored solution is to practice on-switch rules using probe packets [3]–[5], [14]–[16], [19], [30]. Each probe packet is crafted to match a specific rule. If the processing result conforms to the specific rule's action, we consider the rule correctly installed. Otherwise, we consider it missing and leading to a rule inconsistency. However, probe packets do not reveal the essential processing results of production packets. Another line of research thus directly tracks the forwarding traces of production packets [1], [6]–[10], [12], [13], [31]–[35]. Solutions of this kind either mirror forwarding states to trace collectors (e.g., the controller) [1], [12], [13], [31], [35] or encode switch identities [6], [32]–[34] or proofs [7]–[10] into packet headers. They require additional parties to collect and analyze packet traces or even require additional computation capability from switches. More importantly, they defer detection of forwarding anomalies way after mis-forwarded packets have fled the network.

In this paper, we present RuleOut as the first attempt to detect and filter SDN forwarding anomalies in a complete, proactive, and deployment-friendly way. Being complete, RuleOut enables switches to directly verify forwarding correctness of each production packet instead of indirectly inferring it via probe packets. Being proactive, RuleOut enables switches to detect and filter anomalies in real time without postponing their exposure after they reach the incorrect destinations. Toward practicality, we implement completeness and proactiveness of RuleOut with high deployment easiness. In particular, RuleOut can directly apply to off-the-shelf switches without requiring additional computation capability.

The key idea of RuleOut is to disambiguate dependent rules via augmenting their matching fields with unique tags. It completely addresses the root cause of SDN forwarding anomalies—a packet may match more than one rule on a switch. Traditionally, switches use rule priorities to arbitrate such forwarding ambiguity. The highest-priority rule dominates when more than one rule matches the same packet. Once the highest-priority rule is missing, its lower-priority successor will take over and lead to a stealthy forwarding anomaly. After RuleOut augments the matching field of each of dependent rules with a unique tag, a packet can match at

S. Xi, K. Bu, W. Mao, X. Zhang, and K. Ren are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: {shaokexi, kaibu, wsmao, xiaoyu.zhang, kuiren}@zju.edu.cn). Corresponding Author: Kai Bu.

S. Xi, K. Bu, and K. Ren are also with ZJU-Hangzhou Global Scientific and Technological Innovation Center, China.

X. Ren is with the GTTX Network Technology, China (e-mail: renxinxin@gttx.com).

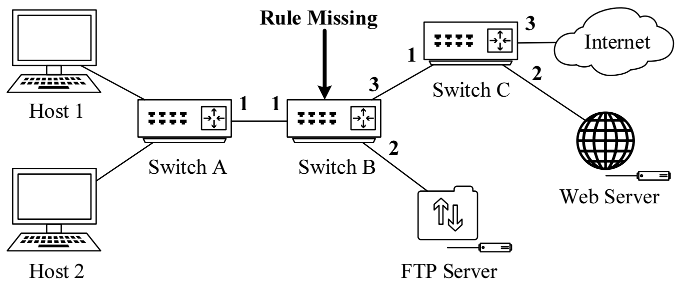


Fig. 1: Example of SDN forwarding anomalies.

TABLE I: Example rule sets on switches in Figure 1. Rules are indexed in descending order of priority and marked as ✓ if they are successfully installed and as ✗ otherwise.

Switch	No.	Match	Action	Status
A	1	ip_dst = 10.15.201.*	output: 1	✓
B	1	ip_dst = 10.15.201.37	output: 2	✗
	2	ip_dst = 10.*.*.*	output: 3	✓
C	1	ip_proto = http, port_dst = 80	output: 2	✓
	2	ip_dst = 10.*.*.*	output: 3	✓

most one rule on a switch. Whenever the rule supposed to process a packet is missing, the packet can be either discarded or reported to the controller for further diagnosis. We develop a non-overlapping rule dependency graph for assigning unique tags to dependent rules. We have also explored a series of optimization techniques (e.g, incremental graph construction, unified matching-space operation, rule clustering) to boost efficiency while preserving rule semantics.

We prototype the RuleOut service and agent on the Ryu [36] controller and Open vSwitch [37]. The RuleOut service inside the controller maintains the non-overlapping rule dependency graph and generates unique tags across dependent rules. The RuleOut agent is attached to endhosts. Borrowing the idea from source routing, the agent queries the server for the sequence of tags to pre-mark packets of each flow. We validate RuleOut performance using three typical data sets—Stanford [25], Internet2 [19], and Airtel1 [38]. The results show that RuleOut can use tags of only 4 bits long to disambiguate thousands to millions of rules. RuleOut is also extremely fast in that both rule graph update and tag generation can be delivered within a few milliseconds.

In summary, we make the following major contributions against SDN forwarding anomalies.

- Disambiguate dependent rules by augmenting their matching fields with unique tags (Section III). This completely removes the forwarding ambiguity as a packet can match at most one rule on a switch.
- Integrate the disambiguation technique into SDN in a non-intrusive way (Section III). Inspired by source routing, we develop an agent that enables endhosts to query and pre-mark packets per flow with a tag sequence corresponding to the sequence of rules supposed to match. Any missing rule will trigger a mismatch over packets it should match. This way, we can easily reveal forwarding anomalies without any modification of off-the-shelf switches.
- Propose a non-overlapping rule dependency graph for efficiently tagging rules (Section IV). We develop efficient algorithms and optimization techniques for graph update and tag generation in an incremental, efficient way.
- Implement disambiguation-based RuleOut as the first attempt for complete, proactive, and deployment-friendly detection and filtering of SDN forwarding anomalies (Section III and Section IV). We prototype RuleOut on the Ryu controller and Open vSwitch with about 4,000 lines of Python code. Evaluation results in Section V over the Stanford [25], Internet2, [19], and Airtel1 [38] data sets demonstrate that RuleOut can use tags of only several bits to disambiguate thousands to millions of rules and generate tags fairly fast within milliseconds.

II. Problem

In this section, we investigate the root causes and impacts of SDN forwarding anomalies. We also review existing countermeasures and underline their limitations in completeness, proactiveness, and deployment easiness.

A. SDN Forwarding Anomalies

The control-data plane inconsistency is widely regarded as the root cause of SDN forwarding anomalies [6]. Specifically, it occurs when the installed rules on data-plane switches do not conform to what have been issued by the control-plane controller. Such inconsistencies tend to make the underlying forwarding behavior deviate from the controller's regulation. Figure 1 showcases an SDN forwarding anomaly due to a rule missing fault. As shown in Table I, rules on Switch B intend to direct FTP connections to the FTP server (with IP address of 10.15.201.37) via outport 2 while other packets to Switch C via outport 3. In the given example, however, Rule 1 on Switch B is missing. Packets from Host 1 to the FTP server will match Rule 2 on the FTP server and be incorrectly forwarded to Switch C.

Unlike abrupt network malfunctions such as switch crashes and link failures, a series of subtle and stealthy malfunctions can also cause rule inconsistencies yet are hard to pinpoint. Both switch software and hardware may lead to transient or permanent inconsistencies. For example, the OpenFlow protocol requires that switches send barrier messages to the controller to acknowledge successful rule installation [39]. However, switches tend to send barrier messages earlier than rules take effect [2]. This leads to transient rule inconsistencies. Switches may also violate OpenFlow protocol specifications for optimizing efficiency [18]. For example, certain commercial switches do not correctly enforce the priority order of rules [11], [18]. They may simply regard the latest installed rules as the higher priority ones [18] or incorrectly enforce rules with a large priority as the lowest-priority ones [11]. Such permanent rule inconsistencies can even be introduced by hardware bit-flip errors in TCAM that store rules [19].

Once a rule inconsistency occurs, corresponding forwarding anomalies may breach network validity and security. For example, they can violate network access control, break traffic isolation, or deviate intended forwarding paths [1], [3], [4],

[6]. As vendors keep their switch implementation proprietary, users need effective and efficient countermeasures against forwarding anomalies toward reliable SDN.

B. Countermeasures and Limitations

A fundamental countermeasure using forwarding proofs, however, cannot directly apply to SDN switches. It is inspired by path validation for securing the next generation of Internet [40]–[43]. The key idea is that forwarding devices add cryptographic proofs to packets they process. Such proofs are then used to verify whether packets have followed designated forwarding paths. Bringing path validation to SDN, SDNsec [7] and REV [8], [9] require the controller to assign each packet with a proof and each switch a cryptographic key. As a packet traverses a switch, the switch updates the proof using its key. Before the packet exits SDN, the controller verifies whether its proof has been correctly updated in order by all the switches it is supposed to traverse. However, current SDN switches do not support cryptographic functions.

Countermeasures that are more feasible to off-the-shelf SDN switches can be classified into two categories, probing and tracing. Probing based solutions inject crafted probe packets to the data plane and check whether they are processed by the desired rules. However, such solutions reveal the forwarding behaviors of probe packets instead of production packets. Tracking based solutions, therefore, track the forwarding states of production packets and reactively troubleshoot forwarding anomalies once they occur. Such solutions face a trade-off challenge of troubleshooting accuracy and resource overhead for tracking forwarding states.

Probing examines rule correctness through injecting probe packets to the data plane and verifies their forwarding correctness. Specifically, to verify whether a rule has been successfully installed on a switch, the controller crafts a probe packet that matches the rule as the highest-priority one among all it can match on the switch. If the probe packet follows the rule's action, the controller regards the rule as correctly installed. A fundamental challenge is how to minimize the number of probe packets. ATPG [19] addresses this challenge by associating forwarding correctness with reachability. That is, if probe packets do reach the intended destination, the controller regards all associated rules from the source to the destination as correct. However, given that different rules may have the same forwarding action, being forwarded to the intended destination does not necessarily guarantee that the packet matches all intended rules en route. Monocle [3] then generates probe packets that enumerate matching possibilities on a switch. RuleScope [4], [5] and FADE [30] further enhance Monocle by detecting priority-swapping faults and associating one probe packet with rules on multiple switches, respectively. Finally, RuleChecker [14], [15] boosts the speed for generating probe packets by taking the entire flow table (instead of a set of dependent rules therein) as the input altogether. It is 5$\times$ faster than Monocle and 20$\times$ faster than RuleScope.

Debugging the network proactively, probing based solutions suite more for verifying relatively steady configurations. This makes them inevitably vulnerable to the following limitations.

- The number of probe packets tends to be huge because they have to exhaust the matching space of each rule. Specifically, the scale of probe packets required by a rule expands exponentially in terms of the number of wildcards in the rule [4]. It is time-consuming to generate so many probe packets. Injecting them into the network also consumes bandwidth and switch processing resources.
- Further and foremost, probe packets cannot reveal the actual forwarding states of production packets, which are the key purpose of network debugging.

Tracing debugs forwarding correctness by collecting forwarding traces of production packets per se. Related solutions require that switches either mirror forwarding states to trace collectors (e.g., the controller) [1], [12], [13], [31], [35] or encode their identities into packet headers [6], [32]–[34]. For example, NetSight [1] uses switches to encapsulate every packet header along with processing metadata (e.g., switch ID and output port) into the so called postcards. Then switches send the postcards to the controller/servers for future query. For simplifying trace collection, CherryPick [32], PathDump [33], and SwitchPointer [34] enable switches to directly encode their identities into packet headers. Such encoded information is also exported to storage for query. Without squeezing extra states into packet headers, SPHINX [35] and FOCES [12], [13] directly report on-switch flow counters to the controller for anomaly detection. For example, after collecting the counters of a specific flow from the switches the flow traverses, if these counters show noticeable increase or decrease on a certain hop, it is highly likely that hop encounters a forwarding anomaly. Recently, PAZZ [16] marries tracing and probing to collect forwarding traces of both production packets and probe packets toward a more systematic detection framework. It is, however, expensive to track the forwarding traces of all packets. An interactive debugging solution in [31], VeriDP [6], and DynaPFV [44] enhance efficiency by collecting aggregate statistics of packets or traces of interested packets (e.g., suspicious traffic [31]).

Albeit directly revealing the forwarding states of production packets, tracing based solutions share some of probing's limitations as well as show some of their own.

- The controller/servers need to store and process large-scale forwarding traces. It tends to be overloaded and turns into a bottleneck.
- The essence of reactiveness delays the detection and troubleshooting of forwarding errors after the affected traffic has already fled the network. Recently, BFPR [10] leverages programmable switches to maintain a Bloom filter based path tracer in packet headers. The receiver can directly use path tracers to verify forwarding correctness. However, it applies to only programmable switches and delays filtering forwarding anomalies to receivers instead of filtering them by switches on the fly.

To address the limitations of both probing and tracing solutions, we suggest that a practically efficient solution should simultaneously feature the following properties.

Completeness: Every possible forwarding error of production packets can be detected.

TABLE II: Qualitative comparison of RuleOut proposed in this paper with existing solutions.

Principle	Solution	Completeness	Proactiveness	Deployability
Probing	ATPG [19]; Monocle [3]; RuleScope [4], [5]; FADE [30]; RuleChecker [14], [15]	✗	✗	✓
Tracing	NetSight [1]; CherryPick [32]; PathDump [33]; SwitchPointer [34]; SPHINX [35] FOCES [12], [13]; PAZZ [16]	✓	✗	✓
Disambiguating	RuleOut (this paper)	✓	✓	✓

Proactiveness: Each mis-forwarded packet can be filtered immediately upon its occurrence.
Deployability: The solution should be readily applicable to off-the-shelf SDN switches.

In this paper, we present RuleOut as the first attempt toward filtering SDN forwarding anomalies with all the preceding properties guaranteed (Table II). RuleOut disambiguates dependent rules by augmenting their matching fields with unique tags. It then pre-marks incoming packets with a sequence of tags that associate with rules to be matched. This enables switches to automatically detect and filter any packet matching incorrect rules. Furthermore, RuleOut achieves such automatic filtering of forwarding anomalies without switch hardware change.

III. Overview

In this section, we present the key ideas of RuleOut. It disambiguates dependent rules by augmenting their matching fields with unique tags. This requires only lightweight additional support from endhosts to mark packet headers. Then the ineffectiveness of any faulty rule leads to an unsuccessful matching with the tagged packet. This not only avoids misforwarding but also localizes the rule fault.

A. Motivation

We observe that a forwarding anomaly may occur when a packet can match more than one rule on a switch. This instantly motivates disambiguating rules as a fundamental solution against SDN forwarding anomalies. If rules are crafted such that a packet match at most one, the packet cannot incorrectly match others if the supposed one happens to not take effect. However, traditional dependent rules with overlapping matching fields and different priorities share common matching packets. Upon multi-rule matching, the highest-priority rule dominates. When we disambiguate each rule from its dependent rules, even if the highest-priority rule is missing, a packet cannot match lower-priority ones and thus avoids misforwarding. Take the two rules on Switch B in Table I for example. Rule dependencies are essentially because of wildcards in rules. However, it is prohibitively expensive to split a wildcarded rule by exhausting the matching range of wildcards. For example, the 3-wildcarded Rule 2 on Switch B in Table I should be split into $2^8 \times 3 = 768$ rules to uniquely match every ip_dst address in the 10.0.0.0/24 subnet. The associative explosion of rule count renders it infeasible.

B. Idea

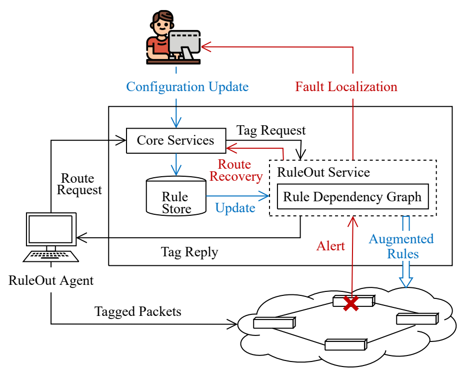
Based on the motivation of rule disambiguation, we propose augmenting the matching field of each of dependent rules with a unique, non-wildcard tag. This can resolve the overlapping matching fields of any traditionally dependent rules. Borrowing the idea from source routing, we pre-tag a packet (in VLAN/MPLS or other unused header fields) with a sequence of tags of rules that it should match along the expected forwarding path. The unique challenge to our solution is, however, that tags should be assigned at a rule granularity while source routing only needs a switch granularity. After tagging rules and packets, we can successfully enforce a unique packet-rule matching on every enroute switch. Once a packet fails to match any tagged rules on a switch, it reveals a rule fault and can be filtered or reported to the controller. A potential forwarding anomaly is thus avoided in real time.

TABLE III: Motivating example of RuleOut. Rules in Table I for switches in Figure 1 are augmented with tags in the matching fields to de-overlap dependent rules.

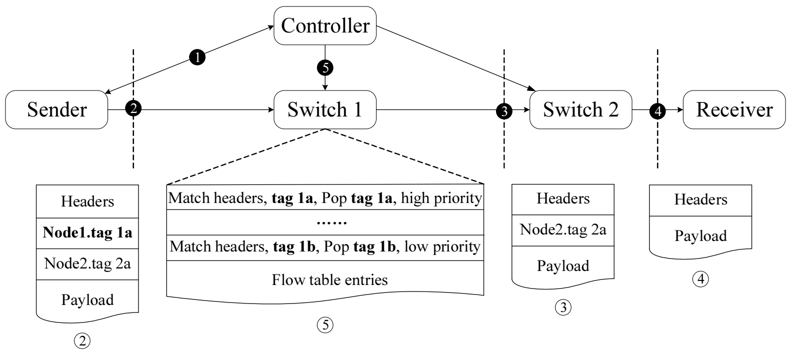
Switch	No.	Match	Action
A	1	ip_dst = 10.15.201.* tag = 0x1	output: 1 pop tag
B	1	ip_dst = 10.15.201.37 tag = 0x1	output: 2 pop tag
	2	ip_dst = 10.*.*.* tag = 0x2	output: 3 pop tag
C	1	ip_proto = http, port_dst = 80 tag = 0x1	output: 2 pop tag
	2	ip_dst = 10.*.*.* tag = 0x2	output: 3 pop tag

We illustrate the effectiveness of our proposal against forwarding anomalies by revisiting the example in Figure 1. As shown in Table III, we augment the matching fields of Rule 1 and Rule 2 on Switch B with tag 0x1 and tag 0x2, respectively. The tags help render the two rules non-overlapping although their original versions share an overlapping ip_dst field. Meanwhile, the packet with destination address 10.15.201.37 for connecting Host 1 with the FTP server is pre-tagged with the sequence (0x1, 0x1), which indicates that the packet should match Rule 1 on Switch A and Rule 1 on Switch B. Tags are stripped from the packet hop by hop using the pop actions supported by switches. This way, even if the higher-priority Rule 1 becomes missing as in Table I and the packet's destination address matches Rule 2's ip_dst field, the packet will not incorrectly match Rule 2 because tag 0x1 it carries is different from tag 0x2 required by Rule 2.

In this paper, we implement the idea of rule disambiguation through RuleOut, the first countermeasure against SDN forwarding anomalies in a complete, proactive, and deployment-friendly way. RuleOut simultaneously satisfies these three properties outlined in Section II-B as follows. First, it satisfies completeness by resolving the overlapping matching fields of dependent rules. Whenever a rule happens to not take effect, its missing is not hidden by its dependent rule with



(a)



(b)

Fig. 2: (a) RuleOut architecture consisting of the server (i.e., the dashed box) on the controller and the agent on endhosts (i.e, senders and receivers). (b) Packet forwarding with RuleOut. ❶ The sender communicates with the RuleOut service with its augmented RuleOut agent and gets the tag sequence of the route for a packet from the RuleOut service on the controller. ❷ The sender instruments the packet header with the tag sequence (②) and sends the packet to the first switch on the forwarding path. Note that tags occupy traditionally unused fields of the packet header; we append them to other header fields (②) for simplicity. ❸ Each switch matches the packet with flow table entries (⑤), strips the outermost tag used for matching (③), and forwards the processed packet to the successor switch until ❹ the packet arrives at the receiver (④). ❺ Upon rule updates, the controller populates the corresponding augmented flow table entries to switches.

a lower priority. In other words, any associative forwarding anomaly can be detected. Second, RuleOut satisfies proactiveness because it pre-tags production packets in accordance with the rules to be matched by them. Any forwarding anomaly can be directly revealed by production packets without using additional probing packets or collecting their traces. Third, RuleOut is readily applicable to off-the-shelf switches because it requires no special switch support.

C. Architecture

Figure 2(a) demonstrates the architecture of RuleOut. Its key add-ons to SDN are a RuleOut server on the controller and a RuleOut agent on the endhost. The RuleOut server aims to disambiguate dependent rules, compile flow-rule bindings, and localize faulty rules.

First, the RuleOut service leverages the non-overlapping rule dependency graph for rule disambiguation. The traditional rule dependency graph is essentially a directed acyclic graph (DAG), tracking the dependency among rules with overlapping matching fields. We adapt it into a non-overlapping version where the out edges of each rule node share no common matching space. Our rule disambiguation then augments the matching fields of dependent rules with unique tags, which are generated by the RuleOut service.

Second, the RuleOut service collaborates with the RuleOut agent to enforce flow-rule bindings. Specifically, the RuleOut agent sends the first packet of a flow to the controller. The controller decides the forwarding path of this flow according to the routing policy. The RuleOut service then identifies the sequence of rules to be matched along the forwarding path. Then it replies the corresponding sequence of tags of the matched rules to the RuleOut agent, which pre-marks each packet in the flow with the tag sequence before sending it to SDN.

Third, after the packet enters SDN, it is expected to match exactly the series of rules with the same tags as it carries.

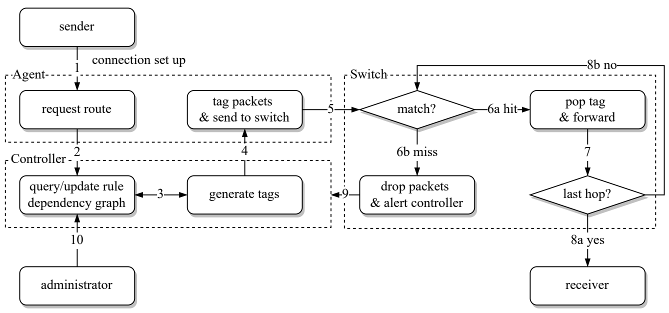


Fig. 3: RuleOut workflow.

Figure 2(b) showcases the packet forwarding process with RuleOut. Once any expected rule becomes ineffective, the packet fails to match the expected tag and trigger an alert to the controller. The alert message contains the unmatched packet header. Exercising the packet header using local rules, the controller can easily localize the faulty rule.

IV. Design

In this section, we present the core of RuleOut design—tag generation using the non-overlapping rule dependency graph. The non-overlapping rule dependency graph maintains an invariant that the matching space of each rule is covered by that of its mutually non-overlapping out edges. This offers not only fast incremental graph construction but also efficient tag assignment for disambiguating dependent rules. We accordingly develop various algorithms and optimization techniques.

RuleOut workflow. For ease of understanding, Figure 3 sketches the workflow of RuleOut to showcase where tag generation, packet tagging, packet-rule matching, and rule tagging fit into the big picture. Specifically, when the sender sets up a new connection, its agent sends the route request to the controller (Step 1). Upon receiving the request, the controller identifies the forwarding path of the connection and asks the RuleOut service for a tag sequence, which can disambiguate rule matching along the chosen path (Step 2-3). The RuleOut service generates the tag sequence over a series of rules using a non-overlapping dependency graph (Section IV-A, Section IV-B, and Section IV-C) (Step 10). The graph can be incrementally constructed upon the administrator updates routing policies. Rules are augmented along with graph construction and populated from the controller into switches subsequently. Meanwhile, the controller replies the tags to the agent, which embeds these tags into packet headers before sending packets out (Step 4). The tagged packets are expected to match augmented rules at each switch along the forwarding path until leaving the network (Steps 6a-8). The mismatching events indicate forwarding anomalies (Step 6b) and trigger alerts to the controller (Step 9).

A. *Observation of Rule Dependency Graph*

Since RuleOut detects forwarding anomalies through disambiguating dependent rules, it is necessary to identify the dependency relationship of rules. A typical way uses the rule dependency graph based on the geometric model and header space algebra [25]. Each graph node represents a rule while a directed edge from a high-priority rule (child node) to a low-priority rule (parent node) indicates the dependency between them [45]. Formally speaking, let R^{M} denote the matching space of rule R. R^{M} covers the set of packet headers that match rule R [46]. A dependency exists between any pair of rules R_i and R_j with $R_i^{\mathrm{M}} \cap R_j^{\mathrm{M}} \neq \emptyset$. Assume that R_i has a higher priority than R_j does. A directed edge E from R_i to R_j exists. We refer to R_i and R_j as E^{src} and E^{dst}, respectively. We also define the matching space of E as the intersection of R_i^{M} and R_j^{M}, that is, $E^{\mathrm{M}} = R_i^{\mathrm{M}} \cap R_j^{\mathrm{M}}$ [46]. We add an all-wildcard rule R_0 with the lowest priority into the dependency graph for ease of design. Any rule node is at least a direct child of R_0 if it does not overlap with any other lower-priority rules.

Given that a rule takes effect on packets that match none of its higher-priority dependent rules, we have Definition 1.

Definition 1. Exclusive Matching Space R^{EXM} *of rule R is the set of packet headers that 1) belongs to the matching space of R yet 2) does not belong to the matching spaces of R's higher-priority dependent rules. Let $R^{\mathrm{in}} = \{E \mid E^{\mathrm{dst}} = R\}$ represent the set of edges from R's higher-priority dependent rules. We can define R^{EXM} as:*

$$R^{\mathrm{EXM}} = R^{\mathrm{M}} - \bigcup_{E \in R^{\mathrm{in}}} E^{M}. \tag{1}$$

We observe that rule update can be essentially modelled as variation of exclusive matching spaces. This observation motivates efficient and incremental update of the rule dependency graph instead of re-building it from scratch upon inserting or deleting rules. Specifically, an inserted rule subtracts exclusive matching spaces of its lower-priority dependent rules. In contrast, a deleted rule increases them. Let ΔR^{in} denote the set of edges from rule R's newly updated higher-priority dependent rules to R. Then the variation of R's exclusive matching space because of ΔR^{in} can be defined as follows.

$$\Delta R^{\mathrm{EXM}} = \bigcup_{E \in \Delta R^{\mathrm{in}}} (-1)^i \times E^{\mathrm{M}}, \tag{2}$$

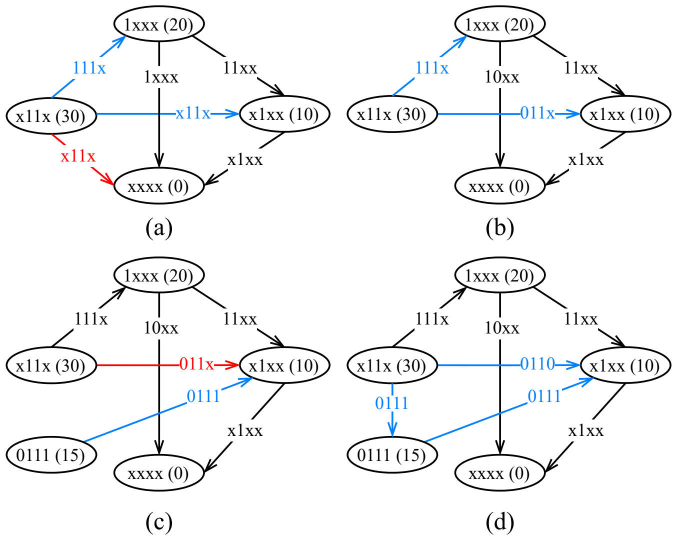


Fig. 4: Comparison of (a) the traditional rule dependency graph and (b) its non-overlapping version. Each ellipse represents a rule node annotated with its matching space and priority (in the parentheses). The traditional rule dependency graph adds a directed edge for any pair of dependent rules, containing overlapping or redundant edges. We observe that it is sufficient to build a non-overlapping dependency graph by making each node's out edges non-overlap. The observation is further verified through examples in (c) and (d).

where i is an indicator for increasing or decreasing the exclusive matching space given an added or deleted edge E:

$$i = \begin{cases} 1, \text{ if } E \text{ is added;} \\ 0, \text{ if } E \text{ is deleted.} \end{cases}$$

Based on Equation 2, we further observe that to optimize graph update, the ideal incremental update solution should minimize graph changes to only vertices and edges related to the added or deleted rules. Such an ideal solution requires that the matching spaces of all edges connected with a rule node are non-overlapping. According to the second union item in Equation 2, if edges are overlapping, adding or deleting an edge does not necessarily increase or decrease the exclusive matching space. However, the existing rule dependency graph does not support the non-overlapping property we expect. As shown in Figure 4(a), the traditional rule dependency graph maintains a directed edge between any two overlapping nodes. Consider, for example, when adding the rule node for x11x(30). The traditional rule dependency graph adds three edges from it to dependent rules. However, the edge from x11x(30) to xxxx(0) does not vary the exclusive matching space of xxxx(0) because the edge's matching space is already included by an existing edge to xxxx(0) (i.e., the edge from x1xx(10) to xxxx(0)). It is, therefore, unnecessary to add such edges. Similarly, the other two edges from 1xxx(20) and x1xx(10) to xxxx(0) overlap with each other's matching space. Their overlapping leads to redundant updates of exclusive matching spaces.

B. Non-overlapping Rule Dependency Graph Construction

We continue with investigating the example in Figure 4 to explore how to build a non-overlapping rule dependency graph. To make the in-edges to x1xx(10) non-overlapped (Figure 4(a)), it is sufficient to make the related out-edges from 1xxx(20) and x11x(30) to x1xx(10) non-overlapped (Figure 4(b)). To double check the effectiveness of the preceding intuition, we insert 0111(15) as shown in Figure 4(c). Traditionally, the insertion of 0111(15) introduces several edges connecting it to rules with dependency (e.g., xxxx(0)). We showcase only the introduced edge from 0111(15) to x1xx(10) to concentrate our discussion on edges related to x1xx(10). In this case, this new edge 0111 overlaps an existing edge 011x from x11x(30) to x1xx(10). This overlapping can be solved using another new edge from x11x(30) to 0111(15) (Figure 4(d)); this edge actually helps to non-overlap both out-edges from x11x(30).

Motivated by the example in Figure 4, we present Theorem 1 to formally prove that the sufficient condition to build a non-overlapping rule dependency graph is that out edges of any node should have non-overlapping matching spaces.

Theorem 1. *Consider a rule dependency graph $G = (R_v, E)$ where R_v denotes the set of vertices each corresponding to a rule and E denotes the set of edges each connecting a pair of vertices. G contains no ambiguity, that is, dependent rules in G have different priorities. If the matching spaces of any vertex's out edges are non-overlapping, the matching spaces of any vertex's in edges are also non-overlapping. Formally speaking, given $R^{\text{out}} = \{E \mid E^{\text{src}} = R\}$, $\forall R \in R_v$ and $\forall (E_i, E_j) \in R^{\text{out}}$ where $E_i \neq E_j$, if the following condition holds:*

$$E_i^{\text{M}} \cap E_j^{\text{M}} = \emptyset, \tag{3}$$

then $\forall R' \in R_v$, $\forall (E_m, E_n) \in R'^{\text{in}}$ where $E_m \neq E_n$, the following condition also holds:

$$E_m^{\text{M}} \cap E_n^{\text{M}} = \emptyset. \tag{4}$$

Proof. We prove Theorem 1 by contradiction. Assume that Condition 3 holds while Condition 4 does not so that $E_m^{\text{M}} \cap E_n^{\text{M}} \neq \emptyset$. Let $R_m = E_m^{\text{src}}$ and $R_n = E_n^{\text{src}}$. When $E_m^{\text{M}} \cap E_n^{\text{M}} \neq \emptyset$, we have $(R_m^{\text{M}} \cap R'^{\text{M}}) \cap (R_n^{\text{M}} \cap R'^{\text{M}}) \neq \emptyset$ and thus $R_m^{\text{M}} \cap R_n^{\text{M}} \neq \emptyset$. This indicates the dependency between R_m and R_n. Without loss of generality, consider when R_m has a higher priority than R_n has. There should be an edge E_k from R_m to R_n. For R_m's two out edges E_m and E_k, their matching spaces have the following relationship:

$$E_m^{\text{M}} \cap E_k^{\text{M}} = (R_m^{\text{M}} \cap R'^{\text{M}}) \cap (R_m^{\text{M}} \cap R_n^{\text{M}}) \neq \emptyset, \tag{5}$$

which contradicts with Condition 3. Therefore, Conditions 3 and 4 should simultaneously hold. This proves Theorem 1. □

Following Theorem 1, we develop Algorithm 1 for incremental construction of the non-overlapping rule dependency graph. The key invariant to maintain is that all out edges of a rule node share mutually non-overlapping matching spaces. Given rule R, its out edges R^{out}, and rule R_i with $E_i = (R, R_i) \in R^{\text{out}}$, we compute E_i^{M} as:

$$E_i^{\text{M}} = R^{\text{M}} \cap R_i^{\text{M}} - \bigcup E_j^{\text{M}}, \tag{6}$$

Algorithm 1: Incremental Rule Insertion

```
Input: Non-overlapping dependency graph
       $G = (R_v, E)$, Inserted rule $R$
Output: Updated non-overlapping dependency graph
        $G'$
1  Insert (Matching space: $M$, Root: $R_0$, Rule: $R$) for
   $E \in R_0^{in}$ do
2      if $M \cap E^M \neq \emptyset$ then
3          $R' \leftarrow E^{src}$;
4          if $R'.priority > R.priority$ then
5              new $E' \leftarrow < R', R >$;
6              $E'^M \leftarrow M \cap E^M$;
7              $R'^{out} \leftarrow R'^{out} \cup \{E'\}$;
8              $R^{in} \leftarrow R^{in} \cup \{E'\}$;
9              $E^M \leftarrow E^M - (M \cap E^M)$;
10             if $E^M = \emptyset$ then
11                 $R'^{out} \leftarrow R'^{out} - \{E\}$;
12                 $R_0^{in} \leftarrow R_0^{in} - \{E\}$;
13         else
14             Insert($M \cap E^M, R', R$);
15             $M \leftarrow M - (M \cap E^M)$;
16 if $M \neq \emptyset$ then
17     new $E' \leftarrow < R, R_0 >$;
18     $E'^M \leftarrow M$;
19     $R^{out} \leftarrow R^{out} \cup \{E'\}$;
20     $R_0^{in} \leftarrow R_0^{in} \cup \{E'\}$;
21 Insert($R^M, G.R_0, R$);
```

where $E_j = (R, R_j) \in R^{\text{out}}$ and R_j has a higher-priority than R_i does. As shown in Figure 4(b), the matching space of the edge from 1xxx(20) to xxxx(0) is equal to $1xxx \cap xxxx - 11xx = 10xx$. The computation of Equation 6 is recursively enforced during incremental update. Initially, we start from the lowest-priority all-wildcard rule R_0. Given that the initial R_0^{in} is empty, we directly add an edge from the added R to R_0 (lines 17-21). Later on, each inserted rule R needs to be compared with existing edges (lines 2-16). Once it introduces edges from or to some existing nodes, the corresponding edges' matching spaces are updated. If all the out edges cannot cover the entire matching space of R, we complete it by adding an edge from R to R_0 (lines 17-21).

Rule deletion in Algorithm 2 is relatively straightforward to handle. The key is to check whether deleting rule R necessitates connecting previously isolated nodes or incrementing the matching spaces of existing edges. Such cases happen when R's in edges and out edges share overlapping matching spaces (lines 5-9). After adding these edges, we can delete rule R and all its in edges and out edges (line 10 and lines 11-13).

C. Tag Generation

Given the non-overlapping rule dependency graph, tag generation preserves an invariant that any two dependent rules should be assigned with different tags. To make tags fit into packet headers, we need to reuse tag values as much as possible. Since non-overlapping rules cannot match the same packet, they can be assigned with the same tag without

Algorithm 2: Incremental Rule Deletion

```
Input: Dependency graph $G = (R_v, E)$, Deleted rule $R$
Output: Updated dependency graph $G'$
1  for $E_i \in R^{out}$ do
2      $R_i \leftarrow E_i^{dst}$;
3      for $E_j \in R^{in}$ do
4          $R_j \leftarrow E_j^{src}$;
5          if $E_i^M \cap E_j^M \neq \emptyset$ then
6              new $E \leftarrow < R_j, R_i >$;
7              $E^M \leftarrow E_i^M \cap E_j^M$;
8              $R_j^{out} \leftarrow R_j^{out} \cup \{E\}$;
9              $R_i^{in} \leftarrow R_i^{in} \cup \{E\}$;
10     $R_i^{in} \leftarrow R_i^{in} - \{E_i\}$;
11 for $E \in R^{in}$ do
12     $R' \leftarrow E^{src}$;
13     $R'^{out} \leftarrow R'^{out} - \{E\}$;
```

inducing matching ambiguity. In contrast, nodes on the same path may have dependencies and they should be assigned with different tags. Algorithm 3 shows the incremental tag generation process along with increment rule insertion. Because a rule node may be inserted into any position of a path, we need to search both directions toward the highest-priority child node (lines 1-5) and the lowest-priority parent node (lines 6-10) for identifying the assigned tags on this path. We use set S to gather potentially overlapping nodes on the path (lines 11-13). Then, we choose the minimum value not used by any nodes in the set S as R's tag. This greedy strategy minimizes the tag value and guarantees sufficiently short tag fields.

D. Optimization

Lazy minus of matching spaces. The frequent set minus operations during header space analysis [25] for matching space update turn out to be very expensive. Consider xxxx−0000=1xxx∪x1xx∪xx1x∪xxx1 for example. Subtracting a small set from a large space leads to an explosion in space fragments. Subsequently, it needs to exhaust all fragments to determine whether an element (e.g., 0001) matches one of them. This leads to a high overhead in time and space. We introduce a lazy minus technique to address the issue. It records equations instead of solving them. Specifically, we record a header space object in the format of $P = \bigcup P_i$, where $P_i = P_i.hs - P_i.sub$, $P_i.hs$ is a binary string representing a matching space, and $P_i.sub$ is a list of binary strings representing the area subtracted from $P_i.hs$. We compute the minus between two header space objects as:

$$\begin{aligned} P - Q &= e \in P \wedge e \notin Q \\ &= e \in (P_1 \cup ... \cup P_n) \wedge e \notin (Q_1 \cup ... \cup Q_m) \\ &= \bigvee_{i=1}^{n} (e \in P_i \wedge (\bigwedge_{j=1}^{m} e \notin Q_j)). \end{aligned}$$

This way, we maintain only expressions instead of many space fragments they may result in. This also confines the scale of computation upon rule update.

Algorithm 3: Incremental Tag Generation

```
Input: Dependency graph $G = (R_v, E)$, Inserted rule $R$
Output: $Tags = \{R : value\}$
1  UpDFS (Graph: G, Rule: R, Set: S) for $R_i \in R^{children}$ do
2      if $R_i \notin S$ then
3          $S \leftarrow S \cup \{R_i\}$;
4          UpDFS($G, R_i, S$);
5  DownDFS (Graph: G, Rule: R, Set: S) for $R_j \in R^{parents}$ do
6      if $R_j \notin S$ then
7          $S \leftarrow S \cup \{R_j\}$;
8          DownDFS($G, R_j, S$);
9  $S \leftarrow \emptyset$;
10 UpDFS($G, R, S$);
11 DownDFS($G, R, S$);
12 $R.tag \leftarrow min_T$ where $\forall s \in S, s.tag \neq T$;
```

Rule clustering by potential dependency. The time complexity of incremental rule node insertion or deletion is $O(E)$, where E denotes the number of directed edges in the graph. Among all the edges, however, only a small group of them may have dependency with the inserted/deleted rule. To quickly narrow down the target group, we cluster rules by their potential dependency. We choose the ip_dst field as a clustering reference as most forwarding rules specify the destination IP address. Maintaining a group for each individual IP address, however, is expensive. We follow the commonly used prefix tree structure and select IP prefixes of destination addresses for classifying rules. Our experiment results show that a 16-bit prefix tree for the ip_dst field can speed up graph construction by more than 100 times.

Graph division. We explore further efficiency by dividing a large dependency graph into smaller subgraphs. The division reference should be a matching field with a limited matching range. In RuleOut implementation, we choose the in_port field. If a rule does not specify in_port, we insert a replicate of it into each subgraph and maintain their status globally.

V. Evaluation

In this section, we prototype RuleOut and report the evaluation results.

Methodology. We implement the RuleOut service and agent in Python with about 4,000 lines of code using the Ryu controller [36] and Open vSwitch [37]. We evaluate RuleOut performance on three publicly available and widely used data sets—Stanford, Internet2, and Airtel1—which are collected from the Stanford campus network [25], Internet2 nationwide network [19], and SDN-based ONOS-controlled network [38], respectively. The Stanford data set contains 16 routers with more than 757,000 forwarding rules and 1,500 ACL rules. The Internet2 data set contains 9 routers with approximate 100,000 forwarding rules. We use scripts to translate these static router configurations into OpenFlow rules. The Airtel1 data set contains about 14,000,000 dynamic insertion and

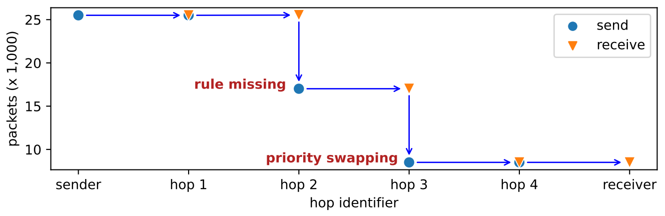


Fig. 5: Effectiveness for detecting and filtering forwarding anomalies. Switches at hop 2 and hop 3 have emulated rule inconsistency due to rule missing and priority swapping, respectively.

removal traces of IPv4 forwarding rules, which are generated by an upper layer SDN-IP application. Some of the updates are triggered by link failure events. We run all the experiments on an Ubuntu 18.04 system with a quad-core Intel(R) Core(TM) i7-7700 CPU @ 3.6 GHz, a 1 MB L2-cache, and a 16 GB DRAM.

Metrics. In particular, we evaluate feasibility and efficiency of RuleOut using the following metrics.

- Anomaly filtering rate characterises the effectiveness of RuleOut for detecting and filtering forwarding anomalies. It quantifies the percentage of packets filtered with respect to packets encountered forwarding anomalies.
- Tag length indicates how long tags are to differentiate dependent rules in a large rule set. A practically efficient solution should limit tag overhead in packet headers.
- Dependency intensity measures the percentage of rules with dependency in a given rule set. It indicates to what extent the control-data plane inconsistency may affect forwarding correctness.
- Graph construction/update time measures how fast our non-overlapping rule dependency graph can model a rule set. Since we build the graph incrementally, we focus on the time for inserting or deleting a rule given a dependency graph of different sizes.
- Tag generation time quantifies how fast to tag a rule after it is inserted to the dependency graph.
- Tag query time evaluates the swiftness for finding the matching rules for a packet. Then the sequence of tags from these rules is returned to the querying endhost.

Results. Extensive experiment results show that RuleOut can accurately detect and filter forwarding anomalies. For all the verified rule sets containing thousands to millions of rules, RuleOut uses only 4-bit tags to disambiguate dependent rules. It can complete not only rule insertion and deletion but also tag generation and query within milliseconds.

A. Anomaly Filtering Rate

To demonstrate the effectiveness of RuleOut for detecting and filtering forwarding anomalies, we introduce a metric called anomaly filtering rate. It quantifies the percentage of packets filtered with respect to packets encountered forwarding anomalies. We emulate rule inconsistency due to rule missing and priority swapping on different hops and count the packets each hop receives and sends. If no forwarding anomaly occurs, then no packet would be dropped; the number of received packets and that of sent packets on each hop

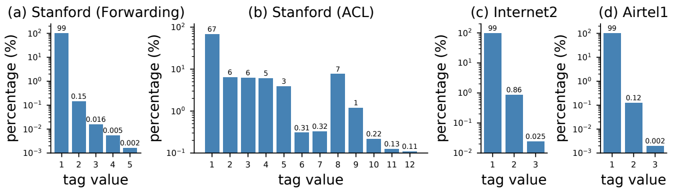


Fig. 6: Distribution of tag values in different rule sets.

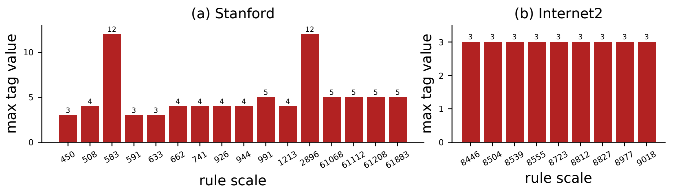


Fig. 7: Maximum tag values for disambiguating different scale of rules on (a) 16 routers in the Stanford data set and (b) 9 routers in the Internet2 data set. Routers are sorted in ascending order of rule scale.

should be identical. We emulate rule errors in a controlled way such that a known number of packets fails to match with them and get filtered. The emulated scenario is chosen from the network topology of Stanford University, with each switch/router populated with rules in the Stanford rule set. Then we randomly choose a routing path through which two endhosts (i.e., sender and receiver) perform a regular network perf test.

Figure 5 reports packet counts at each hop. In the emulation, the sender initially sends 25,501 packets in total. These packets are supposed to reach the receiver via four hops of routers. We emulate rule missing and priority swapping on hop-2 and hop-3 routers, respectively. Both types of rule inconsistency make affected packets match with no rule and thus get filtered. We pre-craft the number of to-be-affected packets on the sender as 8,511 on both hop-2 and hop-3 routers. The number of sent packets on the hop-2 router (i.e, the number of received packets on the hop-3 router) is exactly 25,501−8,511=16,990. Similarly, the hop-3 router filters 8,511 more packets due to the priority-swapping fault. This shows that RuleOut successfully detects forwarding anomalies and achieves a 100% anomaly filtering rate.

B. Tag Length

As discussed in Section IV-C, feasibility of RuleOut depends on sufficiently short tags that can fit in packet headers. Tag length further depends on the scale of tag values for disambiguating dependent rules. Figure 6 shows the distribution of tag values RuleOut generates to disambiguate Stanford, Internet2, and Airtel1 rules. To our surprise, 12 tags (except a default one for tagging the lowest-priority all-wildcard rule R_0) are sufficient for disambiguating all these sets containing thousands to millions of rules. Such short tags require only up to 4 bits to encode. This encouraging result is mainly because that a very small proportion of matching fields overlap in realistic configurations. More than 99% of forwarding rules (Figures 6(a), (c), and (d)) and 67% of ACL rules (Figure 6(b)) are assigned with tag value one; when these rules are inserted,

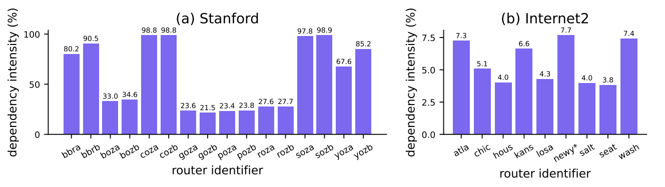


Fig. 8: Dependency intensity of rules on (a) 16 routers in the Stanford data set and (b) 9 routers in the Internet2 data set.

they are directly dependent on the all-wildcard rule R_0. Moreover, ACL rules share more dependencies than forwarding rules do because they specify constraints on more header fields toward finer-grained forwarding behaviors. Therefore, in Figure 6(b), ACL rules require more tags to differentiate.

To further verify the property that tag length depends on dependency complexity instead of rule scale, we measure the maximum tag value for disambiguating rules on each router in the Stanford and Internet2 data sets. Specifically, for each of the 16 routers in the Stanford data set and the 9 routers in the Internet2 data set, we build its non-overlapping rule dependency graph, generate tags for disambiguating dependent rules, and report the maximum tag value in Figure 7. While sorting routers in ascending order of their rule scale, we do not observe a noticeable increase of maximum tag value with rule scale. As shown in Figure 7(a), the Stanford router with 450 rules requires the maximum tag value of 3 while the one with 61,883 rules requires only 5. For all Internet2 routers in Figure 7(b), they require the same maximum tag value of 3 to encode 8,446∼9,018 rules. In contrast, rule dependency complexity decides tag length. For example, the two Stanford routers with 583 rules and 2,896 rules require a maximum tag value of 12, which is much larger than 5 used by routers with more than 60,000 rules. This is because they happen to hold some ACL rules that have complex intrinsic dependencies (Figure 6(b)).

C. Dependency Intensity

Note that short tags do not necessarily indicate that only a small portion of rules overlap with others. To address this potential concern, we measure dependency intensity of the Stanford and Internet2 rule sets. Dependency intensity quantifies the percentage of rules that are dependent on at least one other rule. If any such rule becomes faulty, it might induce forwarding anomalies. Figure 8 reports the measurements over the Stanford and Internet2 rule sets. The results show that routers in the Stanford data set and the Internet2 data set can install up to 98.9% and 7.7% dependent rules, respectively. For the Stanford data set, dependency intensity ranges from 21.5% to 98.9%. All are over 20% and half are higher than 60%. The Internet2 data set shows a relatively smaller scale of dependency intensity. Routers therein install rules with dependency intensity ranging from 3.8% to 7.7%. This is because that the Internet2 data set does not publicize ACL rules, which tend to involve more dependencies. In practical network applications, more and more emerging services such as stock trading and online gaming necessitate highly reliable traffic forwarding [47]. Even sporadic forwarding anomalies may cost customers' significant profits and providers' troubleshooting efforts.

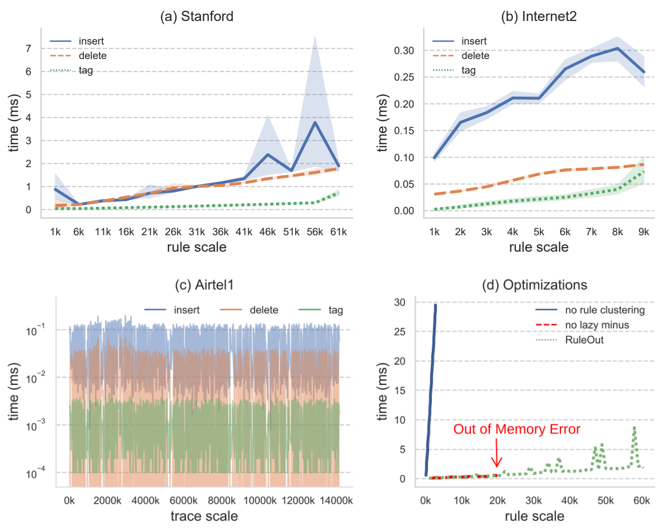


Fig. 9: Speed of graph update and tag generation in different rule sets: (a) Stanford, (b) Internet2, and (c) Airtel1. (d) Effect of optimization techniques on rule insertion speed.

D. Graph Update Time

We first evaluate RuleOut efficiency and scalability by measuring rule insertion and deletion time during incremental update of the non-overlapping rule dependency graph. Since the Stanford and Internet2 sets do not provide the rule update order, we emulate a random update order without loss of generality. We also find that inserting or deleting an individual rule is extremely fast, with a time span quite short to measure. We choose to measure the aggregate update time per 1,000 rules and then use its one thousandth as the average update time per rule. To best quantify RuleOut scalability as rule scale increases, we focus on the routers with the most rules in Stanford and Internet2 sets. Figure 9(a) and Figure 9(b) report update time for both sets. Both time for inserting and deleting a rule increases with rule scale. In comparison with deletion, insertion takes more time because it usually involves matching with more edges. When the graph contains fewer than 10k rules, almost both insertion and deletion can be finished in less than 0.5 ms. Even given more than 60k rules in the graph, update can still complete within 4 ms on average (Figure 9(a)). Some fluctuations may appear during update (e.g., cases with 46k and 56k rules). We suspect that this is mainly because of the random update order we emulate.

Different from the Stanford and Internet2 sets, the Airtel1 set provides the exact order of rule update. We can easily replay the trace therein and record the update time. Since it contains more than 14 millions of rule updates, we report the statistics upon every 10k updates in Figure 9(c). Rule update in Airtel1 is much faster than in Stanford and Internet2 because Airtel1 uses a rather simple rule format, which specifies only the IPv4 header fields. The most expensive insertion operation can complete in less than 1 ms.

E. Tag Generation Time

Along with rule update time, we report also tag generation time in Figure 9(a)∼Figure 9(c). It demonstrates a similar

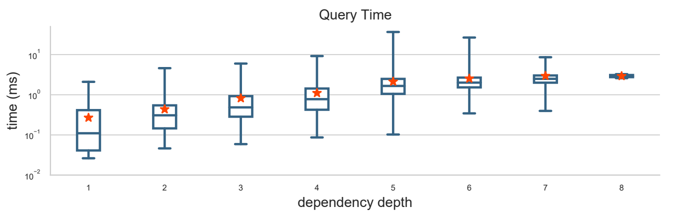


Fig. 10: Tag query time with varying dependency depth. Measurements per depth are plotted with min-max values and 25th-50th-75th percentile values and marked with average values.

trend as update time, that is, it takes a longer time to tag a rule as rule scale increases. Overall, tag generation is much faster than rule insertion and deletion. For Stanford and Internet2 sets, tagging a rule among fewer than 10k ones takes less than 0.1 ms. Airtel1 supports an even faster tagging process, taking less than 0.01 ms to tag a rule on average.

F. *Optimization Effect*

Figure 9 (d) demonstrates the effect of our optimization techniques—lazy minus and rule clustering. The results validate how our optimization guarantees scalability and efficiency. Specifically, we compare three versions of RuleOut—the original RuleOut, one without rule clustering, and one without lazy minus. First, rule clustering narrows down rule scale for update. Without this optimization, the algorithm has to waste much time in matching unrelated rules. This significantly limits scalability. As shown in Figure 9(d), when the graph has 2.8k rules, it already takes near 30 ms to insert a rule and becomes prohibitively slow. After integrating rule clustering, both of the other two versions deliver a satisfactory scalability. However, without using lazy minus, the algorithm needs to maintain more and more fragmented matching spaces as rule scale increases. This tends to quickly drain memory upon 20k rules and makes the algorithm stuck. With both rule clustering and lazy minus integrated, RuleOut can update a large graph fairly fast.

G. *Tag Query Time*

Finally, we report our measurement over tag query time. Two major factors dominate query time for the tag sequence of a specific route. One is the number of switches on the route. The other is the time for finding the matching rule of the queried packet on each switch. Typical SDN-oriented networks (e.g., the Stanford campus network [25]) usually feature with short routes within three hops [25]. We thus focus on tag query time on individual switches. Specifically, finding the highest-priority matching rule for a packet starts with the lowest-priority all-wildcard R_0. It takes a longer time as the dependency chain expands. This is exactly what the statistics in Figure 10 demonstrate. Besides dependency depth, the in-degree of a rule node for matching also affects the query time. In particular, when a rule node has more in-edges, it averagely takes more matching operations to determine whether another rule down the dependency chain matches the queried packet. Given that most dependency chains may expand only one (Figure 6(a), (c), and (d)) or several hops (Figure 6(b)), RuleOut can complete tag query on each switch within 1 ms for most queries.

VI. Discussion

A. *Reactive Rule Installation*

If the controller installs rules reactively, it generates rules for processing unknown flows upon their arrival. In this case, the RuleOut service may receive a query packet whose matching rules are yet to be generated by the controller. This can be addressed by a simple engineering choice. Specifically, upon receiving a query packet, the RuleOut service first matches the packet header with the dependency graph of the ingress switch. An unknown packet fails to match with any existing rule therein. The RuleOut service then hands it over to the controller module for rule generation. Once the desired rules are available, the RuleOut service continues to insert them into the dependency graph, augment them with tags, and issue the tag sequence to the sender of the query packet.

B. *Bit-Flip Errors*

Bit-flip errors may occur to packets in transmission or rules on switches.

For packets, bit-flip errors can occur to either packet headers or packet payloads, or both. Of more concern to forwarding-anomaly detection are bit-flip errors in packet headers. In particular, if flipped bits happen to be in header fields that match against rules, the related packet may match with no rule or with an incorrect one. Both lead to forwarding anomalies. Existing solutions can detect such anomalies by collecting packet traces. For RuleOut, since we have augmented packet headers with a sequence of tags. For a packet with bit-flit errors to match with an incorrect rule, bit-flip errors should 1) simultaneously occur in both tags and header fields, and 2) coincidentally make the faulty tags and header fields happen to match with a certain on the switch. We suspect that this related to only rare cases. Furthermore, we find that forwarding-anomaly detection does not necessarily need to pay special attention to bit-flip errors in packets. Integrity check metadata such as checksums in packet headers or message authentication codes in packet payloads are readily available to detect and filter corrupted packets.

For rules, bit-flip errors turn on-switch rules into different encodings in hardware and thus make them process packets that they are not supposed to match with. Existing solutions still need to collect packet traces for detecting the so caused forwarding anomalies. As discussed in Section II, it necessitates an unbearable overhead to use traces for verifying forwarding correctness of each packet. That enforces heavy processing workloads on both switches and the controller as well as high bandwidth consumption between them. As for our RuleOut, bit-flipped rules lead to two types of impacts on packet forwarding. First, it is highly likely that the packets that faulty rules are supposed to match have no rule to match and get dropped. Second, it is relatively rare that faulty rules turn to match with some packets and make those packets incorrectly forwarded. The second case is relatively rare because RuleOut has already augmented overlapping rules with unique tags. A faulty rule needs to coincidentally flip some bits in tags and other matching fields to match with tagged packets. In other words, if RuleOut encounters no bit-flipped rules, we do not have to bother any potential forwarding anomaly caused by rules. This would be a luxury for existing solutions because even if rules are correctly installed and taking effect, intrinsic overlappings among them are exactly a critical root cause for forwarding anomalies in SDN. In contrast, our RuleOut can fundamentally use hardware error detection [48]–[50] to detect and re-install bit-flipped rules.

VII. Conclusion

We have studied the idea of disambiguating rules against SDN forwarding anomalies. It augments the matching fields of dependent rules with unique tags such that a packet can match at most one rule on a switch. Whenever any rule becomes faulty, a packet supposed to match the faulty rule cannot incorrectly match other rules. This way, we can automatically filter forwarding anomalies without introducing probe packets or collecting packet traces. Leveraging source routing, we implement rule disambiguation through RuleOut. We develop a series of efficient algorithms and optimization techniques toward practicality and efficiency. Evaluation results over public rule sets show that RuleOut uses tags of only several bits long to disambiguate thousands to millions of rules. RuleOut can tag a rule and respond to tag query fairly fast within a few milliseconds. For future work, we plan to practice RuleOut on hardware switches and adapt RuleOut to multipath data center networks [51]. We also plan to release the source code of RuleOut upon publication.

Acknowledgment

We would like to sincerely thank the Editors and Reviewers of IEEE/ACM Transactions on Networking in advance for their review efforts and helpful feedback. We also wish you health and safety during the coronavirus pandemic.

The work is supported in part by National Natural Science Foundation of China under Grant No. 62172358, Natural Science Foundation of Zhejiang Province under Grant No. LY19F020050, National Natural Science Foundation of under Grant No. 61772236, Zhejiang Key R&D Plan under Grant No. 2019C03133, Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Research Institute of Cyberspace Governance in Zhejiang University, and Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang.

References

[1] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, “I know what your packet did last hop: Using packet histories to troubleshoot networks,” in *NSDI*, 2014, pp. 71–85.
[2] M. Kuzniar, P. Peresini, and D. Kostic, “What you need to know about sdn flow tables,” in *PAM*, 2015, pp. 347–359.
[3] P. Peresini, M. Kuzniar, and D. Kostic, “Dynamic, fine-grained data plane monitoring with monocle,” *IEEE ACM Transactions on Networking*, vol. 26, no. 1, pp. 534–547, 2018.
[4] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, “Is every flow on the right track?: Inspect sdn forwarding with rulescope,” in *INFOCOM*, 2016, pp. 1–9.
[5] X. Wen, K. Bu, B. Yang, Y. Chen, L. E. Li, X. Chen, J. Yang, and X. Leng, “Rulescope: Inspecting forwarding faults for software-defined networking,” *IEEE/ACM Transactions On Networking*, vol. 25, no. 4, pp. 2347–2360, 2017.
[6] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, “Mind the gap: Monitoring the control-data plane consistency in software defined networks,” in *CoNEXT*, 2016, pp. 19–33.
[7] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig, “Sdnsec: Forwarding accountability for the sdn data plane,” in *ICCCN*, 2016, pp. 1–10.
[8] P. Zhang, “Towards rule enforcement verification for software defined networks,” in *INFOCOM*, 2017, pp. 1–9.
[9] P. Zhang, H. Wu, D. Zhang, and Q. Li, “Verifying rule enforcement in software defined networks with rev,” *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 917–929, 2020.
[10] S. Xiong, Q. Cao, and W. Si, “Adaptive path tracing with programmable bloom filters in software-defined networks,” in *INFOCOM*, 2019, pp. 496–504.
[11] S. Lee, S. Woo, J. Kim, V. Yegneswaran, P. Porras, and S. Shin, “Audisdn: Automated detection of network policy inconsistencies in software-defined networks,” in *INFOCOM*, 2020, pp. 1788–1797.
[12] P. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, and C. Hu, “Foces: Detecting forwarding anomalies in software defined networks,” in *ICDCS*, 2018, pp. 830–840.
[13] P. Zhang, F. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, C. Shen, and C. Hu, “Network-wide forwarding anomaly detection and localization in software defined networks,” *IEEE/ACM Transactions on Networking*, vol. 29, no. 1, pp. 332–345, 2020.
[14] P. Zhang, C. Zhang, and C. Hu, “Fast testing network data plane with rulechecker,” in *ICNP*, 2017, pp. 1–10.
[15] ——, “Fast data plane testing for software-defined networks with rulechecker,” *IEEE/ACM Transactions on Networking*, vol. 27, no. 1, pp. 173–186, 2018.
[16] A. Shukla, S. J. Saidi, S. Schmid, M. Canini, T. Zinner, and A. Feldmann, “Toward consistent sdns: A case for network state fuzzing,” *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 668–681, 2019.
[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *CCR*, vol. 38, no. 2, pp. 69–74, 2008.
[18] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, “Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches,” *Computer Networks*, vol. 136, pp. 22–36, 2018.
[19] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, “Automatic test packet generation,” in *CoNEXT*, 2012, pp. 241–252.
[20] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, “Vericon: towards verifying controller programs in software-defined networks,” in *PLDI*, 2014, pp. 282–293.
[21] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, “Attacking the brain: Races in the sdn control plane,” in *USENIX Security Symposium*, 2017, pp. 451–468.
[22] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, “Automated bug removal for software-defined networks,” in *NSDI*, 2017, pp. 719–733.
[23] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, “A nice way to test openflow applications.” in *NSDI*, 2012, pp. 127–140.
[24] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, “Netsmc: A custom symbolic model checker for stateful network verification,” in *NSDI*, 2020, pp. 181–200.
[25] P. Kazemian, G. Varghese, and N. McKeown, “Header space analysis: Static checking for networks.” in *NSDI*, 2012, pp. 113–126.
[26] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, “Veriflow: Verifying network-wide invariants in real time,” in *NSDI*, 2013, pp. 15–27.
[27] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, “Plankton: Scalable network configuration verification through model checking,” in *NSDI*, 2020, pp. 953–967.
[28] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, “Apkeep: Realtime verification for real networks,” in *NSDI*, 2020, pp. 241–255.
[29] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li, “Differential network analysis,” in *NSDI*, 2022, pp. 601–615.
[30] C. Pang, Y. Jiang, and Q. Li, “Fade: Detecting forwarding anomaly in software-defined networks,” in *ICC*, 2016, pp. 1–6.
[31] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, “Compiling path queries,” in *NSDI*, 2016, pp. 207–222.
[32] P. Tammana, R. Agarwal, and M. Lee, “Cherrypick: Tracing packet trajectory in software-defined datacenter networks,” in *SOSR*, 2015, pp. 1–7.

[33] ——, "Simplifying datacenter network debugging with pathdump," in *OSDI*, 2016, pp. 233–248.
[34] ——, "Distributed network monitoring and debugging with switchpointer," in *NSDI*, 2018, pp. 453–456.
[35] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *NDSS*, vol. 15, 2015, pp. 8–11.
[36] F. Tomonori, "Introduction to ryu sdn framework," *Open Networking Summit*, pp. 1–14, 2013.
[37] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *NSDI*, 2015, pp. 117–130.
[38] A. Horn, A. Kheradmand, and M. Prasad, "Delta-net: Real-time network verification using atoms," in *NSDI*, 2017, pp. 735–749.
[39] O. S. Specification, "Version 1.5. 1, standard, open networking foundation. 2015," 2017.
[40] J. Naous, M. Walfish, A. Nicolosi, D. Mazières, M. Miller, and A. Seehra, "Verifying and enforcing network paths with icing," in *CoNEXT*, 2011, pp. 1–12.
[41] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig, "Lightweight source authentication and path validation," in *SIGCOMM*, 2014, pp. 271–282.
[42] D. Barrera, L. Chuat, A. Perrig, R. M. Reischuk, and P. Szalachowski, "The scion internet architecture," *Communications of the ACM*, vol. 60, no. 6, pp. 56–65, 2017.
[43] K. Bu, A. Laird, Y. Yang, L. Cheng, J. Luo, Y. Li, and K. Ren, "Unveiling the mystery of internet packet forwarding: A survey of network path validation," *ACM Computing Surveys*, vol. 53, no. 5, pp. 1–34, 2020.
[44] Q. Li, X. Zou, Q. Huang, J. Zheng, and P. P. Lee, "Dynamic packet forwarding verification in sdn," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 6, pp. 915–929, 2018.
[45] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *SOSR*, 2016, pp. 1–12.
[46] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis." in *NSDI*, 2013, pp. 99–111.
[47] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi *et al.*, "Flow event telemetry on programmable data plane," in *SIGCOMM*, 2020, pp. 76–89.
[48] A. Bremler-Barr, D. Hay, D. Hendler, and R. M. Roth, "Peds: a parallel error detection scheme for tcam devices," *IEEE/ACM Transactions on Networking*, vol. 18, no. 5, pp. 1665–1675, 2010.
[49] S. Pontarelli, M. Ottavi, A. Evans, and S.-J. Wen, "Error detection in ternary cams using bloom filters," in *DATE*, 2013, pp. 1474–1479.
[50] P. Reviriego, S. Pontarelli, and A. Ullah, "Error detection and correction in sram emulated tcams," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 27, no. 2, pp. 486–490, 2018.
[51] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *SIGCOMM*, 2011, pp. 266–277.

Shaoke Xi received the B.Sc degree in computer science and technology from Northeastern University, Shenyang, China, in 2019. She is currently pursing the Ph.D. degree major in cyber science and technology with Zhejiang University, Hangzhou, China. Her research interests include network security and blockchain networks.

Kai Bu received the B.Sc. and M.Sc. degrees in computer science from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2006 and 2009, respectively, and the Ph.D. degree in computer science from The Hong Kong Polytechnic University, Hong Kong, in 2013. He is currently an Associate Professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. His research interests include network security and computer architecture. He is a member of the ACM, the IEE, and the CCF. He was a recipient of the Best Paper Award of IEEE/IFIP EUC 2011 and the Best Paper Nominee of IEEE ICDCS 2016.

Wensen Mao received the B.Eng degree in computer science from Zhejiang University, Hangzhou, China, 2020. His current research interest is neuralsymbolic programming.

Xiaoyu Zhang received the B.Eng degree in computer science from Zhejiang University, Hangzhou, China, 2022. Her research interest includes network security.

Kui Ren is a professor and associate dean of College of Computer Science and Technology at Zhejiang University, where he also directs the Institute of Cyber Science and Technology. Before that, he was with State University of New York at Buffalo. He received his PhD degree in Electrical and Computer Engineering fromWorcester Polytechnic Institute. Kui's current research interests include Data Security, IoT Security, AI Security, and Privacy. He received Guohua Distinguished Scholar Award from ZJU in 2020, IEEE CISTC Technical Recognition Award in 2017, SUNY Chancellor's Research Excellence Award in 2017, Sigma Xi Research Excellence Award in 2012 and NSF CAREER Award in 2011. Kui has published extensively in peer-reviewed journals and conferences and received the Test-of-time Paper Award from IEEE INFOCOM and many Best Paper Awards from IEEE and ACM including MobiSys'20, ICDCS'20, Globecom'19, ASIACCS'18, ICDCS'17, etc. His h-index is 74, and his total publication citation exceeds 32,000 according to Google Scholar. Kui is a Fellow of ACM, a Fellow of IEEE, and a Clarivate Highly-Cited Researcher. He is a frequent reviewer for funding agencies internationally and serves on the editorial boards of many IEEE and ACM journals. He currently serves as Chair of SIGSAC of ACM China.

Xinxin Ren was with the GTTX Network Technology, China. His research interest includes network security.