RuleScope: Inspecting Forwarding Faults for Software-Defined Networking

Xitao Wen, Kai Bu, *Associate Member, IEEE*, *Member, ACM*, Bo Yang, Yan Chen, *Senior Member, IEEE*, Li Erran Li, *Fellow, IEEE*, Xiaolin Chen, Jianfeng Yang, *Member, IEEE*, and Xue Leng

***Abstract*— Software-defined networking (SDN) promises unprecedentedly flexible network management but it is susceptible to forwarding faults. Such faults originate from data-plane rules with missing faults and priority faults. Yet existing fault detection ignores priority faults, because they are not discovered on commercial switches until recently. In this paper, we present RuleScope, a more comprehensive solution for inspecting SDN forwarding. RuleScope offers a series of accurate and efficient algorithms for detecting and troubleshooting rule faults. They inspect forwarding behavior using customized probe packets to exercise data-plane rules. The detection algorithm exposes not only missing faults but also priority faults and the troubleshooting algorithm uncover actual forwarding states of data-plane flow tables. Both of them help track real-time forwarding status and benefit reliable network monitoring. Furthermore, toward fast inspection of dynamic networks, we propose incremental algorithms for rapidly evolving network policies to amortize detection and troubleshooting overhead without sacrificing accuracy. Experiments with our prototype on the Ryu SDN controller and Pica8 P-3297 switch show that the RuleScope achieves accurate fault detection on 320-entry flow tables with a cost of 1500+ probe packets within 16 s.**

***Index Terms*— Software-defined networking, forwarding fault, network troubleshooting.**

I. INTRODUCTION

RECENT measurement studies expose SDN forwarding's vulnerability to various faults [2]–[4]. When restricted to data-plane rules, such faults behave as *missing faults* and *priority faults*. A missing fault occurs when a rule is not active on a switch as expected [5]. It is mainly attributed to switch firmware or hardware glitch [5] or even rule-update message loss [2]. Current OpenFlow protocol can hardly notice missing faults because it does not acknowledge a rule update unless an expensive barrier operator is inserted [2]. Furthermore, a priority fault occurs when overlapping rules (i.e., rules with common matching packets) violate designated priority order. When implemented in ternary content-addressable memory (TCAM) in physical switches, sophisticated optimization on rule update latency [6]–[9] can potentially result in subtle priority fault, which is difficult to observe without exercising the data-plane behavior. Priority faults have already been observed on commercial switches [3]. Since SDN requires that a packet be processed by the highest-priority rule among matching ones [10], either missing faults or priority faults might lead to undesirable forwarding behavior. Directly dumping flows from the data-plane provides less reliable evidence of data-plane states as it may not reflect the actual and most up-to-date data-plane behavior. Rudimentary network debugging tools (e.g., ping, traceroute, SNMP, and tcpdump) do not either provide convenient ways to discover data-plane faults in a centralized SDN environment [11]. It is thus important to explore SDN-specific inspection schemes.

Although missing fault can be revealed with some data-plane probing tools, we find that priority fault, which is not discovered until recently [3], can still evade all existing data-plane inspecting tools. For example, typical such solutions—ATPG [11], ProboScope [5], and Monocle [12]—focus mainly on verifying rule existence on switches. We observe that without verifying rule priority order, verifying only rule existence cannot guarantee forwarding correctness. Table I exemplifies such concern. It regulates that users from the 10.10.0.0/16 subnet are not served. If a priority fault swaps the priority order of the two rules, HTTP requests from the 10.10.0.0/16 subnet will be incorrectly allowed and thus breach the access control policy even if we have verified their existence.

TABLE I
MOTIVATING RULE SET. GIVEN POSSIBLE PRIORITY SWAP, THE EXISTENCE OF BOTH RULES CANNOT GUARANTEE CORRECT PROCESSING OF THEIR COMMON MATCHING PACKETS

Priority	Matching	Action
p_{high}	ipsrc = 10.10.$*$.$*$	deny
p_{low}	protocol = http	allow

Detecting missing faults alone is already proved NP-hard [5], [11]. It is especially challenging to provably reveal both

Manuscript received January 21, 2016; revised August 3, 2016 and October 30, 2016; accepted March 10, 2017; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Ganjali. This work was supported in part by the Fundamental Research Funds for the Central Universities under Grant 2014QNA5012, in part by the National Science Foundation of China under Grant 61402404 and Grant 60963021, in part by the NSF under Grant CNS-1219116, and in part by the program for Innovative Research Team in University of Yunnan Province. A preliminary version of this manuscript has been published in the Proceedings of the 35th International Conference on Computer Communications (INFOCOM), San Francisco, CA, USA, April 10–15, 2016 [1] (doi: 10.1109/INFOCOM.2016.7524333). *(Corresponding author: Kai Bu.)*

X. Wen is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: xwe334@eecs.northwestern.edu).

K. Bu, B. Yang, and X. Leng are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: kaibu@zju.edu.cn; ybo2013@zju.edu.cn; lengsnow2015@zju.edu.cn).

Y. Chen is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: ychen@northwestern.edu).

L. E. Li is with the School of Information Science and Technology, Fudan University, Shanghai 200433, China (e-mail: erranlli@gmail.com).

X. Chen is with the Department of Computer Science, Chuxiong Normal University, Chuxiong 675000, China (e-mail: chenxl@cxtc.edu.cn).

J. Yang is with the School of Electronic Information, Wuhan University, Wuhan 430079, China (e-mail: yjf@whu.edu.cn).

Digital Object Identifier 10.1109/TNET.2017.2686443

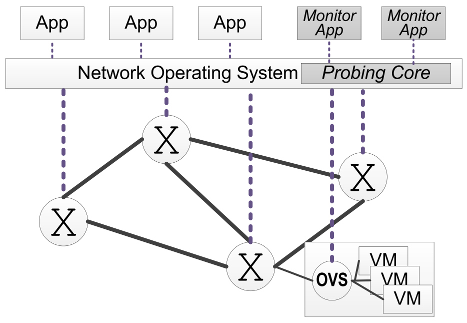


Fig. 1. RuleScope framework. X: Switch. OVS: Open vSwitch. VM: Virtual Machine.

missing and priority faults. ATPG, for example, generates a set of probing packets that exercises each rule at least once. However, the header space intersection of every rule pair must be exercised in order to completely reveal the priority faults. Considering that any of these intersections can be potentially covered by a third rule with priority fault, extending ATPG to detect priority fault will introduce prohibitive probing overhead.

In this paper, we present the RuleScope system for accurately and efficiently inspecting SDN forwarding. Beyond existing inspection solutions [5], [11], [12], RuleScope detects not only missing faults but also priority faults. It can also uncover actual data-plane forwarding states. In line with established systems [5], [11], [12], RuleScope inspects forwarding behavior through probing. As Figure 1 shows, probing functionality relies on the probing core inside the controller. The probing core injects probe packets to data plane and collects probing results for forwarding inspection. While the probing core can leverage packet tracing tools like NetSight [13], how to generate probe packets and how to process probing results for accurate and efficient inspection still remain challenging. RuleScope fulfills this mission by introducing monitoring applications atop the probing core.

The heart of monitoring applications is the algorithms we propose for detecting and troubleshooting rule faults. Our detection algorithm reveals forwarding faults using customized probe packets to exercise data-plane rules. To test rule r's activeness, feasible probe packets should match with r but not with r's higher-priority overlapping rules. Meanwhile, the probe packet should also match with r's lower-priority overlapping rules to detect r's priority fault. Whenever any such probe packet is not processed by r, the on-switch flow table encounters a rule fault. For ease of tracking overlapping rules, we model a flow table using dependency graph, where each vertex represents a rule and each edge connects a pair of overlapping rules [14]. We further explore other techniques to enhance detection efficiency without sacrificing accuracy. For example, we decompose dependency graph to enable parallel probe generation, leverage priority-fault probing results to eliminate missing-fault probing, and minimize the constraints of probe generation to speed up detection.

Beyond detecting rule faults, we explore also troubleshooting algorithms to uncover actual flow tables being executed by data-plane switch. This enables tracking real-time forwarding status and inferring how switches handle rule updates. But it is computationally challenging to generate sufficient probe packets in an offline way as the detection algorithm does. Our analysis demonstrates its potential exponential complexity with respect to flow table size. Toward effective troubleshooting, we first design an adaptive online algorithm. It generates and injects probe packets one at a time. The probing result helps calibrate subsequent probe generation such that we can minimize the number of probe packets. We also accelerate probe generation through simplifying computational constraints. To achieve higher efficiency, we further propose a semi-online troubleshooting algorithm. It adaptively generates and injects probe packets at a batch level. Albeit costing more probe packets than the online algorithm does, the semi-online algorithm promises faster troubleshooting because it mitigates redundant code re-execution and enables parallel switch-port utilization.

Forwarding inspection entails heavy complexity in both the calculation and execution of probing, rendering re-execution of inspection algorithms over frequently updated rule sets hard to favor highly dynamic networks in real time [12]. Toward real-time detection and troubleshooting of forwarding faults for dynamic networks, we further propose incremental inspection algorithms. Specifically, they incrementally monitor the forwarding states by only inspecting the changes made to the forwarding rules. Since the rule updates usually are typically small compared with the entire rule set, incremental inspection runs two to three orders of magnitude faster than the non-incremental counterparts in [1].

In summary, toward a more comprehensive solution for inspecting SDN forwarding, we make the following contributions.

- Detect both missing faults and priority faults for accurate inspection.
- Not only detect rule faults but also troubleshoot them to uncover actual data-plane flow tables.
- Present various techniques to enhance inspection efficiency without sacrificing accuracy.
- Investigate incremental forwarding inspection toward monitoring dynamic networks in real time.
- Validate accuracy and efficiency of RuleScope through theoretical analysis and experiments on a testbed with the Ryu controller [15] and Pica8 P-3297 switch [16].

The rest of the paper is organized as follows. Section II defines the forwarding inspection problem and analyzes its hardness. Section III proposes a series of algorithms for detecting and troubleshooting rule faults. Section IV proposes incremental algorithms to speed up data-plane inspection. Section V and Section VI respectively implement RuleScope prototype and report evaluation results. Finally, Section VII concludes the paper.

II. Problem

In this section, we define SDN forwarding faults and the forwarding inspection problem. We prove its NP-hardness by reduction from the satisfiability (SAT) problem.

A. Forwarding Basics

SDN enforces network policies through transforming them to switch-understandable rules. A rule should specify a *matching field* and an *action field* that respectively regulate which packets to process and how to process them. The matching field is compared against packet headers. Since SDN advocates flow-based forwarding [10], a rule aggregates multiple traditional exact-match rules using *don't care bits* or *wildcards* (denoted by $*$) that match with both bit 0 and bit 1. Thus, a packet can match with more than one rule. To avoid matching ambiguity, SDN further assigns each rule with a *priority* value. When a packet matches with multiple rules, it follows the one with the highest priority.

Based on the preceding basics, we introduce the following definitions to facilitate subsequent presentation.

Definition 1: ***Header space*** *is a union set of all possible packet headers. Given l-bit packet headers, the header space is $\{0,1\}^l$ [17].*[1]

Definition 2: ***Rule*** *r is an ordered triplet $(r^{\mathrm{P}}, r^{\mathrm{M}}, r^{\mathrm{A}})$, where r^{P}, r^{M}, and r^{A} respectively represent priority field, matching field, and action field.*[2] *Matching field r^{M} corresponds to a point in the $\{0,1,*\}^l$ space and matches with a subspace of the header space $\{0,1\}^l$.*

Definition 3: ***Matching space*** *of rule r is a set r^{MS} of all packet headers that match with r's matching field r^{M}. If r^{M} has w bits of wildcards, we have $|r^{\mathrm{MS}}| = 2^w$.*

Definition 4: ***Flow table*** *is a set FT of n rules. That is, $FT = \{r_i \mid r_i = (r_i^{\mathrm{P}}, r_i^{\mathrm{M}}, r_i^{\mathrm{A}}), 0 \le i \le n-1\}$.*

Given that flow tables undergo various checks before being pushed to switches [17], [19]–[22], we assume that FT contains neither duplicate rules nor obscured rules.

B. Forwarding Faults

SDN forwarding correctness, however, is vulnerable to rule faults on data plane. Possible rule faults manifest as missing faults and priority faults. Both are observed in recent measurement studies of commercial SDN switches [2]–[4].

Definition 5: ***Missing fault*** *happens to a rule if the rule cannot take effect on any packets in its matching space. More formally, rule $r \in FT$ is missing on a switch if for all packet $p \in r^{\mathrm{MS}}$ the switch processes p without following r^{A}.*

Missing faults arise from two scenarios. The first is when a rule is not successfully installed [5]. Current SDN, however, does not acknowledge a rule update unless an expensive barrier command is inserted [2]. The controller divides rule updates into batches and isolates them using barrier commands. A switch executes all updates prior to a barrier command and sends a barrier reply. The controller takes the barrier reply as an acknowledgement that all update commands are executed, hardly noticing missing ones therein if any. The second scenario for missing faults is when a rule becomes obscured due to priority faults.

Definition 6: ***Priority fault*** *happens to a pair of overlapping rules if their priority order is swapped. More formally, two overlapping rules r_i and r_j (i.e., $r_i^{\mathrm{MS}} \cap r_j^{\mathrm{MS}} \neq \emptyset$) with $r_i^{\mathrm{P}} > r_j^{\mathrm{P}}$ encounter a priority fault on a switch if for all packets $p \in r_i^{\mathrm{MS}} \cap r_j^{\mathrm{MS}}$ the switch processes p following r_j^{A}.*

[1]We use terms "packet" and "packet header" interchangeably.

[2]We omit rule fields that do not affect forwarding for simplicity [18].

A recent measurement study reveals priority faults on commercial SDN switches [3]. For example, HP 5406zl trims priorities before installing rules to hardware and treats rules installed later as higher-priority ones. According to the test on HP 5406zl with two rules [3], the one installed later always dominates packets that match with both rules. If the installation order does not strictly conform to the reverse priority order, it leads to priority faults and therefore incorrect forwarding.

C. Forwarding Inspection Problem

The forwarding inspection problem is to reveal inconsistency between flow table FT_{ctr} issued by the controller and flow table FT_{sw} implemented on the switch. We tackle it in two ways, *fault detection* and *fault troubleshooting*. First, fault detection aims to detect whether FT_{sw} raises rule faults. Second, fault troubleshooting aims to reproduce FT_{sw} after a fault is detected. Solving such problems needs to exercise rules in FT_{sw} using probe packets [5]. If a probe packet for rule $r \in FT_{\mathrm{ctr}}$ does not follow r's action on the switch, r is faulty. RuleScope strives for accurate forwarding inspection with limited probing overhead.

Before detailing RuleScope design, we analyze the hardness of the forwarding inspection problem. Inspired by ProboScope [5], we first prove the NP-hardness of probe packet generation via reduction from the SAT problem in Theorem 1. We then demonstrate priority-fault troubleshooting's exponential complexity in terms of the number of probe packets in Theorem 2.

Theorem 1: Generating probe packets to detect missing faults and priority faults is an NP-complete problem.

Proof: Missing Fault: We first prove the NP-completeness of missing-fault probe packet generation.

1) *Missing-fault probing is in NP.* To detect missing fault of $r_i \in FT_{\mathrm{ctr}}$, a probe packet p should match with r_i but not with higher-priority rules than r_i. Otherwise, p will be processed by another present rule $r_{i'}$ with higher priority and yield no proof of r_i's existence. Given l-bit matching field, we represent r_i's matching field as $r_i^{\mathrm{M}} = (x_{i0}, \ldots, x_{ib}, \ldots, x_{i(l-1)})$, where $x_{ib} \in \{0,1,*\}$ and $0 \le b \le l-1$. Let $(p_0, \ldots p_b, p_{l-1})$ (where $p_b \in \{0,1\}$) represent the corresponding l bits in probe packet p's packet header. Then p matching with r_i is equivalent to the following conjunction being satisfied.

$$r_i.Match = \bigwedge_{0 \le b \le l-1} S(x_{ib}, p_b), \tag{1}$$

where

$$S(x_{ib}, p_b) = \begin{cases} True, & \text{if } x_{ib} = p_b; \\ False, & \text{if } x_{ib} \neq p_b; \\ True, & \text{if } x_{ib} = *. \end{cases}$$

Furthermore, p not matching with r_i is equivalent to the following disjunction being satisfied.

$$r_i.\neg Match = \neg r_i.Match = \bigvee_{0 \le b \le l-1} \neg S(x_{ib}, p_b). \tag{2}$$

Based on Equation 1 and Equation 2, we can derive that if probe packet p for detecting missing fault of rule r_i exists, the following equation should be satisfied.

$$r_i.Match \land (\bigwedge_{\forall r_j \in FT'_{\text{sw}}} r_j.\neg Match)$$
$$= \bigwedge_{0 \le b \le l-1} S(x_{ib}, p_b), \land (\bigwedge_{\forall r_j \in FT'_{\text{sw}}} (\bigvee_{0 \le b \le l-1} \neg S(x_{jb}, p_b),), \tag{3}$$

where $FT'_{\text{sw}} = \{r_j \mid r_j \in FT_{\text{sw}} \text{ and } r_j^{\text{P}} > r_i^{\text{P}}\}$. Given a packet p, verifying whether it satisfies Equation 3 is equivalent to verifying whether given truth assignments make a CNF true. Such verification can be efficiently done in polynomial time. Therefore, missing-fault probe packet generation is in NP.

2) *An SAT problem is reducible to a missing-fault probe packet generation problem in polynomial time.* Consider an SAT instance with s CNF clauses. Each clause comprises some or all of elements in $\{x_b \mid 0 \le b \le l-1\}$. Formally speaking, the SAT instance I can be modelled as the following.

$$I = \bigwedge_{0 \le i \le s-1} (\bigvee_{0 \le b \le l-1} C_i(x_b)),$$

where $C_i(x_b)$ denotes how an element x_b contributes to the ith clause as follows.

$$C_i(x_b) = \begin{cases} x_b, & \text{if } x_b \text{ is in the } i\text{th clause;} \\ \neg x_b, & \text{if } \neg x_b \text{ is in the } i\text{th clause;} \\ False, & \text{if } x_b \text{ is NOT in the } i\text{th clause.} \end{cases}$$

We now use the SAT instance I to construct a probe packet generation instance. Based on $S(x_{ib})$ and $C_i(x_b)$, we observe that the ith clause can be mapped to a rule r_j as follows.

$$r_j.x_{jb} = \begin{cases} 0, & \text{if } C_i(x_b) = x_b; \\ 1, & \text{if } C_i(x_b) = \neg x_b; \\ *, & \text{if } C_i(x_b) = False. \end{cases}$$

Considering rules mapped from s clauses in I as rules $r_j \in FT'_{\text{sw}}$ in Equation 3, we reduce the SAT instance I to missing-fault probe packet generation as follows. Specifically, it is to generate probe packets for rule r_i containing l wildcards given the above mapped rules r_j as constraints. Since the newly introduced all-wildcard rule r_i contributes only true-value clauses to Equation 3, it does not affect the truth assignment space of $r_j.x_{jb}$ in the mapped rules and therefore of x_b in the original clauses. Now it is straightforward to show that I is satisfied if and only if the corresponding probe packet generation problem is satisfied. Since the above construction takes polynomial time and probe packet generation is in NP, missing-fault probe packet generation is NP-complete.

Priority fault: We now prove the NP-completeness of priority-fault probe packet generation. To detect priority fault of a pair of overlapping rules r_i and r_j, a probe packet p should match with both r_i and r_j. We introduce a new rule $r_{\cap ij}$ with matching space $r_{\cap ij}^{\text{MS}} = r_i^{\text{MS}} \cap r_j^{\text{MS}}$ and action $r_{\cap ij}^{\text{A}} = r_i^{\text{A}}$ or r_j^{A}. Then probing priority fault for $(r_i, r_j) \in FT_{\text{ctr}}$ is equivalent to probing missing fault for $r_{\cap ij} \in FT_{\text{ctr}} - r_i - r_j + r_{\cap ij}$, which we have already proved an NP-complete problem. □

Theorem 2: Troubleshooting priority order of a pair of overlapping rules with l-bit matching fields requires $\mathcal{O}(2^l)$ probe packets in worst cases [1].

For cases when $r_{\cap ij}^{l'\text{MS}} = \bigcup_{\forall r \in R_{ij}^{\cap}} r^{l'\text{MS}}$ and $|R_{ij}^{\cap}|$ is proportional to l', from Theorem 2 follows Corollary 1.

Corollary 1: Given a flow table with n rules, troubleshooting priority order of (r_i, r_j) on data plane requires $\mathcal{O}(2^n)$ probe packets in worst cases.

III. Inspecting Forwarding Faults

In this section, we explore solutions to the stepping stone for RuleScope—detection and troubleshooting algorithms. They generate probe packets to exercise data-plane rules and detect/troubleshoot rule faults based on probing results. The detection algorithm reveals all existing rule faults while troubleshooting algorithms uncover actual on-switch flow tables.

For ease of understanding, we in this section focus on static rule sets to inspect. In other words, if rule update happens, we may re-execute the algorithms to propose over the entire new rule set. This is obviously not a wise choice for monitoring highly dynamic networks with frequent rule updates. We defer our exploration of algorithms that require re-execution for only incremental rules (i.e., added or removed) to Section IV.

A. Probe

As Theorem 1 shows, probe-generation for priority fault has the same complexity as that for missing fault. For probing r_i's missing fault, we need solve Equation 3 to obtain an l-bit assignment of probe packet p's packet header, that is, $(p_0, \ldots p_b, p_{l-1})$ (where $p_b \in \{0, 1\}$). If p exists, it is straightforward that we have $p_b = x_{ib}$ if x_{ib} is 0 or 1. For a p_b corresponding to $x_{ib} = *$, it is complex to be individually assigned with 0 or 1; we need consider all p_b's corresponding to wildcard-x_{ib}'s at the same time by Theorem 2. For example, consider that rule r_i contains two wildcards. Then the binary assignment of the corresponding two bits in probe packet p's packet header must not be identical to or covered by that of any r_j, which p should not match. We solve Equation 3 using a high-performance SAT solver called MiniSat [23] (Theorem 1). We integrate MiniSat into our probe generation function $SampleProbe(r, R)$. Accepting rule r and a set R of rules as inputs, $SampleProbe(r, R)$ outputs a probe packet p that matches r but does not match rules in R. Furthermore, p has every bit automatically specified.

We, however, cannot efficiently and effectively probe for r_i's missing fault directly using $SampleProbe(r_i, FT'_{\text{sw}})$, where $FT'_{\text{sw}} = \{r_j \mid r_j \in FT_{\text{sw}} \text{ and } r_j^{\text{P}} > r_i^{\text{P}}\}$ (Theorem 1). Given that flow table FT_{sw} usually contains hundreds of rules [24], FT'_{sw} might impose too many constraints on $SampleProbe(r_i, FT'_{\text{sw}})$. This seriously limits the scalability of probe generation. Even worse, given that FT_{sw} may encounter missing and priority faults, rules $r_j \in FT'_{\text{sw}}$ are hardly known a priori. Any $r_j \in FT_{\text{sw}}$ with $r_j^{\text{MS}} \cap r_i^{\text{MS}} \neq \emptyset$ could have higher priority than r_i does on the switch. FT'_{sw} then should incorporate all $r_j \in FT_{\text{ctr}}$ with $r_j^{\text{MS}} \cap r_i^{\text{MS}} \neq \emptyset$. Whenever such r_j leads to $r_j^{\text{MS}} \supset r_i^{\text{MS}}$, $SampleProbe(r_i, FT'_{\text{sw}})$ is not solvable. Such undesirable

Algorithm 1 Rule Fault Detection Algorithm

```
Input  : Flow table $FT_{\text{ctr}}$
Output: Set $R_{\text{fault}}$ of faulty rules
1  $R_{\text{fault}} \leftarrow \emptyset$;
2  $G = <V, E> \leftarrow FT_{\text{ctr}}$'s dependency graph;
3  $C \leftarrow G$'s weakly connected componets;
4  Set $P$ of $<$ packet $p$, rule $v > \leftarrow \emptyset$;
5  foreach weakly connected component in $C$ do
6      Header space $H \leftarrow \emptyset$;
7      foreach $v_i$ in topological order do
8          if $v_i$ is not isolated (i.e., $v_i$.degree $!= 0$) then
9              if $\exists v_j$ that directly depends on $v_i$ then
10                 foreach $v_j$ do
11                     $p \leftarrow SampleProbe(v_i \cap v_j, H)$;
12                     if $p == \emptyset$ then
13                         $p \leftarrow SampleProbe(v_i, H)$;
14                     $P \leftarrow P \cup <p, v_i>$;
15             else
16                 $p \leftarrow SampleProbe(v_i, H)$;
17                 $P \leftarrow P \cup <p, v_i>$;
18             $H \leftarrow H \cup v_i$;
19         else
20             $p \leftarrow SampleProbe(v_i, H)$;
21             $P \leftarrow P \cup <p, v_i>$;
22 foreach $<p, v_i> \in P$ do
23     Inject $p$ to data plane;
24     if $p$ does not follow $v_i$'s action then
25         $R_{\text{fault}} \leftarrow R_{\text{fault}} \cup r_i$;
26 return $R_{\text{fault}}$;
```

cases occur quite frequently because flow tables contain many overlapping rules.

Our rule fault detection algorithm reduces the number of constraints as inputs to $SampleProbe(\cdot)$ with the help of the dependency graph of the flow table.

B. Detection

Goal: The detection algorithm aims to find faulty rules using probes on data plane. A faulty rule is the rule that fails to match actual packets that it should be able to match according to header space semantics. In this paper, we focus on rule missing fault and priority fault. Consider again the example in Table I—the toy rule set has two overlapping rules with priorities of p_{high} and p_{low}. Ideally, we expect that an HTTP request from 10.10.1.1 to be processed by p_{high}-rule. If the request is not dropped as p_{high}-rule's action specifies, we consider p_{high}-rule as faulty. Specifically, we consider the possibility that either p_{high}-rule is missing or the two rules encounter a priority-order swap.

Design: The detection algorithm consists of two key steps, probe generation and fault detection (Algorithm 1). Probe generation finds sufficient packets for verifying whether each rule is faulty (lines 2-21). Fault detection then injects probe packets to data plane and detects faulty rules based on probe feedback (lines 22-25). Between the two steps, probe generation is the core of Algorithm 1. We adopt various techniques toward efficiency while generating sufficient packets to guarantee detection accuracy. First, we reduce the scale of probe generation by dividing a flow table to independently solvable subsets of rules. Second, in each rule subset, we catch any faulty rule without exercising it twice against missing fault and priority fault. Third, we generate probe packets for a rule without necessarily involving all other rules in the same subset as constraints to $SampleProbe(\cdot)$.

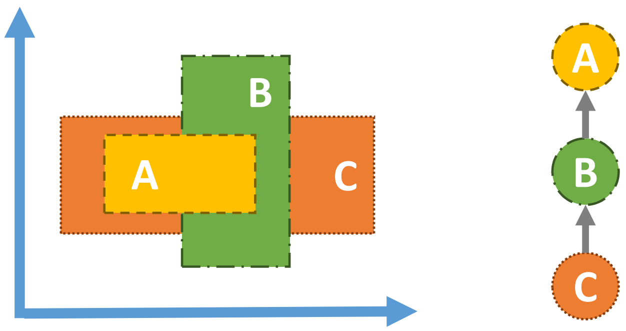


Fig. 2. An example dependency graph of a flow table.

Reduce probe generation scale using dependency graph. The dependency graph $G = < V, E >$ of flow table FT_{ctr} is the following directed acyclic graph (line 3) [14].

- For each $r_i \in FT_{\text{ctr}}$, there is a corresponding vertex v_i in V. We may use v_i and r_i interchangeably.
- For each pair of rules r_i and r_j, if r_i overlaps with r_j and has higher priority than does r_j, there is a directed edge $< v_i, v_j >$ in E.

If edge $< v_i, v_j >$ exists, we say that v_j *directly depends on* v_i. If there is a directed path from v_i to v_j, we say that v_j *depends on* v_i. Based on dependency graph, we find all maximal subgraphs each with vertices connecting with no vertex in other maximal subgraphs. Such maximal subgraphs are essentially weakly connected components of G (line 3). Since rules in different components involve no dependency, we independently generate probe packets for each component without wrestling with the entire flow table. This promises faster probe generation with smaller problem scale within each component and parallelism among different components.

Efficiently generate probe packets for each weakly connected component. Beyond reducing problem scale to independently solvable components, we further strive for efficiency in each component (lines 5-21). Specifically, we generate a probe packet to verify each directly-dependent rule pair. For example, Figure 2 depicts a flow table with three rules (A, B and C) and the dependency graph of the rules. We generate two probe packets to verify A.priority$>$ B.priority and B.priority$>$ C.priority correspondingly. Probe 1 is sampled from the header space $A \cap B$ and Probe 2 is sampled from $(B \cap C) - A$. In this process, we use several techniques to speed up the generation of a probe packet.

First, we reduce the number of probe packets by leveraging the fact that probing priority faults reveals also missing faults. Consider, for example, a pair of rules where v_j directly depends on v_i. We expect that a packet for probing their priority order match with both rules and follow v_i. One such packet is sufficient for detecting faulty v_i corresponding to v_j. Whether v_i is missing or encounters priority swap with v_j (or with another lower priority rule that also matches the probe packet), the probe packet will not be processed by v_i. Since more than one rule may directly depend on v_i, we need to enumerate all of such rules to ensure v_i's freedom of priority faults (lines 10-14). We enforce the above probing of rule pairs following topological order. When we reach a rule directly depended by no rule, we generate a probe packet for probing its missing fault only (lines 16-17 and 20-21).

Second, we speed up generating probe packets by simplifying constraints for $SampleProbe(\cdot)$. As discussed in Section III-A, $SampleProbe(\cdot)$ requires two parameters—the first (or second) regulates rules that a probe packet should (or should not) match with. When generating probe packets for v_i, the first parameter is v_i if we probe its missing fault or $v_i \cap v_j$ if we probe its priority fault. The second parameter is critical for efficient and effective probe generation. To simplify its imposing constraints on $SampleProbe(\cdot)$, we eliminate from it as many rules irrelevant to detection accuracy as possible. An easy example arises from probing a rule without dependency (lines 20-21). Since a packet matching with such rule does not match with other rules, we can use an empty set as the second parameter of $SampleProbe(\cdot)$ for acceleration.

Correctness: Having explored how Algorithm 1 efficiently generates probe packets, we now study its correctness in terms of detection accuracy in Theorem 3.

Theorem 3: Algorithm 1 can accurately detect faulty rules on data plane without false negatives or false positives.

Proof: A false negative occurs when Algorithm 1 regards a faulty rule as correct. A false positive occurs when Algorithm 1 detects a correct rule as faulty.

No false negatives: Without loss of generality, we assume that v_i is the faulty rule under study. The proof falls into two cases according to whether some rule directly depends on v_i.

When no rule directly depends on v_i, it is under missing fault while it could be isolated or directly depend on another rule. If v_i is isolated, $SampleProbe(v_i, H = \emptyset)$ must generate a probe packet (line 20), which reveals v_i's missing fault. If v_i directly depends on another rule, we generate a probe packet by $SampleProbe(v_i, \{v \mid v_i \text{ depends on } v\})$ (line 16). The probe packet does not exist if any packet matching with v_i matches with a higher priority rule v. In this case, v_i is an obscured rule, which should be eliminated from well crafted flow tables (Section II-A). $SampleProbe(v_i, \{v \mid v_i \text{ depends on } v\})$ thus can find a probe packet to reveal v_i's missing fault.

When some rule v_j directly depends on v_i, it could be under either missing fault or priority fault. If $SampleProbe(v_i \cap v_j, \{v \mid v_i \text{ depends on } v\})$ finds a probe packet (line 11), it will not follow v_i's action whether v_i is missing or priority swapped with v_j (or another lower priority rule). We thus successfully detect faulty v_i. If $SampleProbe(v_i \cap v_j, \{v \mid v_i \text{ depends on } v\})$ finds no probe packet, any packet matching with both v_i and v_j must match with a higher priority rule. In this case, we turn to $SampleProbe(v_i, \{v \mid v_i \text{ depends on } v\})$ (line 13) for generating a probe packet. The probe packet will not follow v_i's action whether v_i is missing or priority swapped with a lower priority rule, revealing faulty v_i.

Because Algorithm 1 successfully detects faulty v_i in both cases, it has no false negatives.

No false positives: By Algorithm 1, v_i is detected as faulty if its probe packet p does not follow its action. Algorithm 1 generates p in one of four cases (lines 11, 13, 16, and 20). If v_i is isolated, p follows line 20 and matches with only v_i. If p does not follow v_i's action, v_i must be missing. If v_i is not isolated, Algorithm 1 may generate p using $SampleProbe(v_i, \{v \mid v_i \text{ depends on } v\})$ (lines 13 and 16). If p follows no action (for p from lines 13 and 16) or another lower priority rule's action (for p from line 13), v_i must be missing or encounter a priority fault. Moreover, Algorithm 1 may also generate p using $SampleProbe(v_i \cap v_j, \{v \mid v_i \text{ depends on } v\})$ (line 11), where v_j directly depends on v_i. If p follows v_j's or another lower priority rule's action, v_i must be missing or encounter a priority fault. In summary, the faulty rule v_i detected by Algorithm 1 really is faulty. Algorithm 1 thus has no false positives. $\square$

C. Troubleshooting

High-level idea: The troubleshooting algorithm aims to uncover actual flow table FT_{sw} on a switch. We realize that priority value of each rule is imperceivable with data plane probes. Instead, we want to reveal the dependency of actual rules with pair-wise rule priority order, since in this way we can reconstruct the total order of rule priority that is isomorphic to the dependency graph of the actual flow table.

In this section, we reconstruct dependency graph $G_{sw} =< V, E >$ of actual effective rules on the switch. G_{sw} should satisfy the following two conditions.

- V: For each rule $r \in FT_{\text{ctr}}$, if it is not missing or obscured on the switch, its corresponding vertex v must be in V.
- E: For each pair of rules r_i and r_j in FT_{ctr}, if r_i has higher priority than r_j does on the switch, a directed edge $< v_i, v_j >$ connecting their corresponding vertices in V must be in E.

Without knowing a priori the actual rule existence and priority on the switch, we need to exhaust all possible dependency relations and accordingly generate probe packets. This, as Corollary 1 demonstrates, may cost offline troubleshooting of exponential scale probes.

As opposed to the above 'offline' approach that generates all the probe packets offline in one batch (i.e., $\mathcal{O}(2^n)$ by Corollary 1), we opt for an online approach that adaptively generates one probe packet after knowing the probe result of the previous probe packets, and a semi-online approach that generates probe packets in small batches to better leverage the probe bandwidth.

Online troubleshooting algorithm: We propose efficiently troubleshooting rule faults in an online fashion (Algorithm 2). It adaptively generates/injects probe packets, using previous probe results to reduce the number of later probe packets. For example, we could first generate and inject probe packet p for

Algorithm 2 Online Troubleshooting Algorithm

```
Input  : Flow table $FT_{\mathrm{ctr}}$
Output: Dependency graph $G_{\mathrm{sw}} =< V, E >$ of data
        plane rules
1  $V \leftarrow \{v_i \mid v_i$ corresponds to $r_i \in FT_{\mathrm{ctr}}\}$;
2  $E = \emptyset$;
3  Set $S \leftarrow$ all pairs of overlapping rules in $FT_{\mathrm{ctr}}$;
4  while $S \neq \emptyset$ do
5      $(v_i, v_j) \leftarrow$ any overlapping-rule pair in $S$;
6      $H \leftarrow \{v \mid$ if $< v, v_i >$ and $< v, v_j > \in E\}$;
7      $p \leftarrow SampleProbe(v_i \cap v_j, H)$;
8      if $\{p\} = \emptyset$ then
9          $S \leftarrow S - \{(v_i, v_j)\}$;
10     else
11         $V' \leftarrow$ set of $v \in FT_{\mathrm{ctr}}$ that matches $p$;
12         Inject $p$ to data plane;
13         if $p$ matches with no rule then
14             $V \leftarrow V - V'$;
15             $E \leftarrow E - \{$edges connecting to $v \in V'\}$;
16             $S \leftarrow S - \{$rule pairs including $v \in V'\}$;
17         else
18             $v_{\mathrm{hit}} \leftarrow$ the rule in $V'$ that processes $p$;
19             foreach $v \in V' - \{v_{\mathrm{hit}}\}$ do
20                 $E \leftarrow E \cup \{< v_{\mathrm{hit}}, v >\}$;
21                 $S \leftarrow S - \{(v_{\mathrm{hit}}, v)\}$;
22 foreach $v \in V$ and $v.outdegree = 0$ do
23     if $v.indegree = 0$ then
24         $p \leftarrow SampleProbe(v, \emptyset)$;
25     else
26         $p \leftarrow SampleProbe(v, \{v' \mid < v', v > \in E\})$;
27     if $\{p\} = \emptyset$ or $p$ does not follow $v$ upon injection then
28         $V = V - \{v\}$;
29         $E = E - \{< v', v > \mid < v', v > \in E\}$;
30 return $G_{sw} =< V, E >$;
```

Algorithm 3 Semi-Online Troubleshooting Algorithm

```
Input  : Flow table $FT_{\mathrm{ctr}}$
Output: Dependency graph $G_{\mathrm{sw}} =< V, E >$ of data
        plane rules
1  Initiate $V$, $E$, and $S$ as lines 1-3 in Algorithm 2;
2  while $S \neq \emptyset$ do
3      Set $P$ of probe packet $p \leftarrow \emptyset$;
4      foreach overlapping rule pair $(v_i, v_j)$ in $S$ do
5          Generate $p$ as lines 6-7 in Algorithm 2;
6          if $\{p\} = \emptyset$ then
7              $S \leftarrow S - \{(v_i, v_j)\}$;
8          else
9              $P \leftarrow P \cup \{p\}$;
10         foreach $p \in P$ do
11             Lines 11-21 in Alg 2;
12 Lines 22-29 in Algorithm 2;
13 return $G_{sw} =< V, E >$;
```

a pair of rules v_i and v_j (lines 5-7). On the one hand, p may be not processed by any rule upon injection. In this case, p helps reveal that not only v_i and v_j but also all other rules matching with p in FT_{ctr} are missing on the switch. We then eliminate their related vertices and edges from G_{sw} and save corresponding probes (lines 13-16). On the other hand, let v_{hit} denote the rule that processes p. Rule v_{hit} could be v_i, v_j, or another rule that also matches with p. Again, p might reveal states of more than v_i and v_j—v_{hit} has higher priority than does any other rule matching with p. We then connect v_{hit} with these rules and exclude the connected pairs from later probes (lines 18-21). The preceding cases indicate that online design yields $\mathcal{O}(|E|) = \mathcal{O}(|V|^2) = \mathcal{O}(|FT_{\mathrm{ctr}}|^2) = \mathcal{O}(n^2)$ complexity for probing rule dependency (lines 4-21).

To guarantee the correctness of Algorithm 2, we need to further probe the existence of rules on which no other rule depends (lines 22-29). For one such rule v, we first generate its probe packet p (line 23-26). If p does not follow v upon injection, v is missing on the switch. One corner case is that p might not exist if v depends on other rules (line 26). In this case, v is an obscured rule as any packet matching it matches one of higher priority rule. In this case, whether or not v exists on the switch, it will not take effect. We thus regard also obscured rules as missing. Once a missing rule is detected, we eliminate its corresponding vertex and edge(s) from G_{sw} (lines 28-29). Probing missing rules in lines 22-29 yields $\mathcal{O}(|V|) = \mathcal{O}(|FT_{\mathrm{ctr}}|) = \mathcal{O}(n)$ complexity.

Combining the above two parts, the complexity for Algorithm 2 to uncover actual on-switch flow table is $\mathcal{O}(n^2)$, way more efficient than its offline counterpart's $\mathcal{O}(2^n)$.

Semi-online troubleshooting algorithm: We further explore a faster, hybrid design to reap the benefits of both offline and online algorithms (Algorithm 3). Its major difference from the online algorithm is issuing probe packets at a batch level (lines 10-11). We can regard the online algorithm as a special case of the semi-online algorithm with batch size one. The semi-online algorithm regains a significant amount of efficiency that is otherwise lost in the fully sequential online algorithm. First, batch-level probing leaves out repeat of operations without necessarily running from scratch for each probe packet (lines 3-11). In light of this, generating x probe packets at once may be faster than invoking probe generation x times. Second, increasing the number of probe packets in flight can exploit more parallelism among switch ports.

We analyze how many probe packets Algorithm 3 costs. Each round generates a probe packet for each unverified pair of overlapping rules (lines 3-11). The number of probe packets per round is upper bounded by $\mathcal{O}(|V|^2) = \mathcal{O}(|FT_{\mathrm{ctr}}|^2) = \mathcal{O}(n^2)$. Moreover, each round reveals at least the highest-priority rule among ones unmatched in previous rounds. The number of rounds is thus upper bounded by $\mathcal{O}(|V|) = O(|FT_{\mathrm{ctr}}| = \mathcal{O}(n)$. The complexity of Algorithm 3 in terms of the number of probe packets is $\mathcal{O}(n^2) \times \mathcal{O}(n) = \mathcal{O}(n^3)$.

TABLE II
ALGORITHM COMPLEXITY

Complexity Indicator	Detection	Troubleshooting		
	Offline	Offline	Online	Semi-online
No. of Probe Packets	$\mathcal{O}(n^2)$	$\mathcal{O}(2^n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$
No. of Probe Rounds	1	1	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
No. of Probe Pkts/Rd	$\mathcal{O}(n^2)$	$\mathcal{O}(2^n)$	1	$\mathcal{O}(n^2)$

D. Complexity

Table II summarizes the complexity of detection and troubleshooting algorithms in terms of the number of probe packets. The detection algorithm (Algorithm 1) aims to quickly verify rule consistency between the controller and the switch. Algorithm 1 models rule set using dependency graph and generates a probe packet for every pair of adjacent rules. The number of probe packets generated by Algorithm 1 is thus upper bounded by $\mathcal{O}(|E|) = \mathcal{O}(|V|^2) = \mathcal{O}(|FT_{\text{ctr}}|^2) = \mathcal{O}(n^2)$. Once a faulty rule is detected, we can use troubleshooting algorithms to reveal actual rule existence and priority order on the switch. The number of probe packets generated by different troubleshooting algorithms is analyzed in Section III-C. For large and complex rule sets, we expect to rank troubleshooting algorithms in decreasing order of probe-packet number as offline algorithm, online algorithm (Algorithm 2), and semi-online algorithm (Algorithm 3).

We now discuss algorithm complexity in terms of the execution time. Thanks to the SAT solver—MiniSat—that can get an approximate solution in polynomial time, algorithms requiring a polynomial number of probe packets thus deliver detection (Algorithm 1) or troubleshooting (Algorithm 2 and Algorithm 3) in polynomial time. But for offline troubleshooting algorithm with an exponential number of probe packets, we cannot guarantee its execution in polynomial time. To rank troubleshooting algorithms in decreasing order of time complexity for large and complex rule sets, we still expect an order as offline algorithm, online algorithm (Algorithm 2), and semi-online algorithm (Algorithm 3). We will present their implementation in Section V and evaluate their accuracy and efficiency in Section VI.

IV. INSPECTING DYNAMIC NETWORKS

In dynamic networks, data-plane rules may frequently change over time. To verify the correct forwarding behavior of dynamic data-plane rules, a straightforward approach is to apply the above detection and troubleshooting algorithms on each snapshot of the data-plane rules. However, considering the computation and probing overhead, it is more favorable to have incremental algorithms that leverage the previous detection and troubleshooting results to avoid redundant probes. Because the number of changed rules is typically much smaller than the number of total data-plane rules, such optimization can bring significant speed-up over the straightforward approach. We next further investigate such incremental algorithms toward detecting and troubleshooting forwarding faults for dynamic networks in real time.

Algorithm 4 Incremental Rule Fault Detection Algorithm

```
Input : Last verified flow table $FT_{\text{prev}}$ and unverified rules $FT_{\text{new}}$
Output: Set $R_{\text{fault}}$ of faulty rules
1  $FT_{\text{ctr}} \leftarrow FT_{\text{prev}} \cup FT_{\text{new}}$;
2  $R_{\text{fault}} \leftarrow \emptyset$;
3  $G = <V, E> \leftarrow FT_{\text{ctr}}$'s dependency graph;
4  $C \leftarrow G$'s weakly connected components;
5  Set $P$ of $<$ packet $p$, rule $v> \leftarrow \emptyset$;
6  foreach weakly connected component in $C$ do
7      Header space $H \leftarrow \emptyset$;
8      foreach $v_i$ in topological order do
9          if $v_i$ is not isolated (i.e., $v_i$.degree $!= 0$) then
10             if $\exists v_j$ that directly depends on $v_i$ then
11                 foreach $v_j$ do
12                     if $v_i \in FT_{\text{prev}}$ and $v_j \in FT_{\text{prev}}$ then
13                         continue;
14                     else
15                         $p \leftarrow SampleProbe(v_i \cap v_j, H)$;
16                         if $p == \emptyset$ then
17                             if $v_i \in FT_{\text{prev}}$ then
18                                 continue;
19                             else
20                                 $p \leftarrow SampleProbe(v_i, H)$;
21                         $P \leftarrow P \cup <p, v_i>$;
22             else
23                 if $v_i \in FT_{\text{prev}}$ then
24                     continue;
25                 else
26                     $p \leftarrow SampleProbe(v_i, H)$;
27                     $P \leftarrow P \cup <p, v_i>$;
28             $H \leftarrow H \cup v_i$;
29         else
30             if $v_i \in FT_{\text{prev}}$ then
31                 continue;
32             else
33                 $p \leftarrow SampleProbe(v_i, H)$;
34                 $P \leftarrow P \cup <p, v_i>$;
35 foreach $<p, v_i> \in P$ do
36     Inject $p$ to data plane;
37     if $p$ does not follow $v_i$'s action then
38         $R_{\text{fault}} \leftarrow R_{\text{fault}} \cup r_i$;
39 return $R_{\text{fault}}$;
```

A. Incremental Detection

Goal: The incremental detection algorithm aims to reuse as much verified information as possible from the last verified flow table; it then probes for the potential faults that involve only the newly added unverified rules. Toward this goal, we consider the rule dependency relations in the last flow table as correct and avoid to probe them again. For example, consider that a rule with priority $p_{\text{higher}} > p_{\text{high}}$ is added to the flow table in Table I. The previous probes conducted to the last flow table already provide evidence for the existence and correct priority of p_{high}-rule and p_{low}-rule. Hence, the incremental detection algorithm only needs to probe the existence of p_{higher}-rule and the priority order of p_{higher}-rule and p_{high}-rule.

Design: We first propose an incremental algorithm for detecting forwarding faults (Algorithm 4). Similar with Algorithm 1, the incremental detection algorithm also consists of two key steps—probe generation and fault detection. Probe generation finds the probe packets for verifying the rule faults. Fault detection then injects probe packets to data plane and detects faulty rules based on probe feedback.

Different from the static algorithm, the incremental algorithm takes as input the last verified flow table FT_{prev} and newly added unverified rules FT_{new}. In the probe generation step, the incremental detection algorithm uses the additional information to avoid generating redundant probes. Specifically, consider a probe packet that is to verify the priority order of a pair of rules v_i and v_j (e.g., packet p at line 15). We generate the probe packet only if at least one of the rules v_i and v_j is newly added. Otherwise, the priority order should have been verified previously and the probe packet need not be generated (lines 12-13). Similarly, if a probe packet is to verify the existence of a rule v_i, we generate the probe packet only if the rule v_i is newly added. Otherwise, the existence of v_i should have been verified previously and the probe packets need not be generated (lines 17-18, 23-24 and 30-31).

The correctness of Algorithm 4 is ultimately assured by Theorem 3. Theorem 3 assures that Algorithm 1 detects faulty rules without false negatives or false positives. Furthermore, Algorithm 4 generates a set of probe packets P_{inc}, which is a subset of the set P_{full} generated by Algorithm 1. Note the fact that all the rules in the last flow table is already verified, which assures any probe packet from $P_{full} - P_{inc}$ would not be able to reveal any faults. Therefore, Algorithm 4 can also detect faulty rules without false negatives or false positives.

Complexity: The number of probe packets generated by Algorithm 4 is determined by the probe generation step (lines 8-34). Specifically, two types of probe packets can be generated: priority order probes and rule existence probes. Since the incremental algorithm only verifies the priority order between a newly added rule and another rule, the total number of priority order probe packets is $\mathcal{O}(mn)$, where m denotes the number of newly added rules and n denotes the total number of rules. Also, the incremental algorithm only verifies newly added rules for existence, thus the number of rule existence probe packets is $\mathcal{O}(m)$. Therefore, the complexity of Algorithm 4 is $\mathcal{O}(mn) + \mathcal{O}(m) = \mathcal{O}(mn)$.

B. Incremental Troubleshooting

Goal: The incremental troubleshooting algorithm aims to reveal the actual rules in an incrementally changing flow table with minimum number of probe packets. We leverage the previously verified probe results from both the fault detection algorithm and the troubleshooting algorithm. We thus avoid the redundant probes of the rule existence or priority order that is already verified by previous results. For example, consider the scenario that the rule fault detection algorithm finds one pair of rules with incorrect priority order. Since the probe results from the detection algorithm are insufficient to reconstruct the actual flow table, it further triggers the troubleshooting algorithm to uncover the actual flow table. The incremental troubleshooting algorithm now has partial probe results from the detection algorithm to narrow down the hypothesis space and therefore costs less probes.

Design: We propose efficiently troubleshooting a changing flow table by exploiting the previous probe results to reduce the number of probe packets (Algorithm 5). The incremental

Algorithm 5 Incremental Troubleshooting Algorithm

```
Input : Flow table $FT_{\text{ctr}}$ and the verified dependency
        graph $G_{\text{prior}} =< V_{\text{prior}}, E_{\text{prior}} >$
Output: Dependency graph $G_{\text{sw}} =< V, E >$ of data
        plane rules
1  $V \leftarrow V_{\text{prior}}$;
2  $E \leftarrow E_{\text{prior}}$;
3  $S \leftarrow \emptyset$;
4  $E_{TC} \leftarrow$ the transitive closure of $E$;
5  foreach overlapping rule pair $< v_i, v_j >$ in $FT_{\text{ctr}}$ do
6      if $< v_i, v_j >$ or $< v_j, v_i > \in E_{TC}$ then
7          continue;
8      else
9          $S \leftarrow S \cup < v_i, v_j >$;
10 while $S \neq \emptyset$ do
11     $(v_i, v_j) \leftarrow$ any rule pair in $S$;
12     $H \leftarrow \{v \mid$ if $< v, v_i > \in E$ or $< v, v_j > \in E\}$;
13     $p \leftarrow SampleProbe(v_i \cap v_j, H)$;
14     if $\{p\} = \emptyset$ then
15         $S \leftarrow S - \{(v_i, v_j)\}$;
16     else
17         $V' \leftarrow$ set of $v \in FT_{\text{ctr}}$ that matches $p$;
18         Inject $p$ to data plane;
19         if $p$ matches with no rule then
20             $V \leftarrow V - V'$;
21             $E \leftarrow E - \{$edges connecting to $v \in V'\}$;
22             $S \leftarrow S - \{$rule pairs including $v \in V'\}$;
23         else
24             $v_{\text{hit}} \leftarrow$ the rule in $V'$ that processes $p$;
25             foreach $v \in V' - \{v_{\text{hit}}\}$ do
26                 $E \leftarrow E \cup \{< v_{\text{hit}}, v >\}$;
27                 $S \leftarrow S - \{(v_{\text{hit}}, v)\}$;
28 foreach $v \in V$ and $v.outdegree = 0$ do
29     if $v.indegree = 0$ then
30         $p \leftarrow SampleProbe(v, \emptyset)$;
31     else
32         $p \leftarrow SampleProbe(v, \{v' \mid < v', v > \in E\})$;
33     if $\{p\} = \emptyset$ or $p$ does not follow $v$ upon injection then
34         $V = V - \{v\}$;
35         $E = E - \{< v', v > \mid < v', v > \in E\}$;
36 return $G_{sw} =< V, E >$;
```

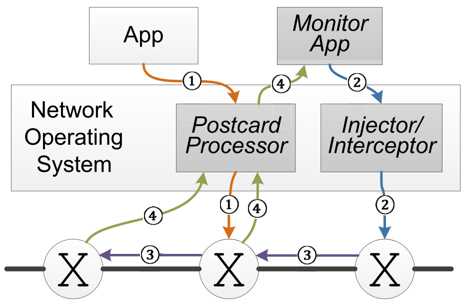


Fig. 3. RuleScope prototype architecture and work flow. X: Switch.

troubleshooting algorithm is analogous to the online troubleshooting algorithm except for the input and the initialization process. The incremental troubleshooting algorithm takes as input the current flow table FT along with its verified partial dependency graph G_{prior}. G_{prior} includes the verified existing rules in the vertex set V_{prior} and the rule set with verified priority order E_{prior}. In the initialization step, the incremental troubleshooting algorithm uses the verified dependency information to minimize the size of rule pair set S, which contains all the rule pairs with uncertain priority order.

Complexity: The number of probe packets generated by Algorithm 5 is determined by the number of iterations of the while-loop (lines 10-27), which is ultimately limited by the size of S. This is because one probe in the main loop eliminates at least one element in S. Although it is difficult to directly estimate the size of S, we can provide a loose upper bound of $|S|$ from the number of rule faults k, which we expect to be small. We define *relevant rules* v_1, v_2 *of a probe* as the rules whose priority order the probe is supposed to verify. We denote V_u as the set of relevant rules of the k faulty rules. We also denote V_r as the rest of rules. Obviously, one probe packet has two related rules, so $|V_u| \leq 2k$ and $|V_k| < n$. Since the priority order among V_k are all confirmed, S is only a subset of rule pairs within V_u or between V_u and V_k. Therefore, $|S| < 2kn + \frac{1}{2}k^2 \approx 2kn$.

V. Prototype

In this section, we present our implementation of RuleScope prototype. We first present the architecture and work flow. We then detail the experiment setup.

A. Architecture

Figure 3 demonstrates the architecture and work flow of RuleScope prototype. App transforms forwarding policies to rules, which are populated to switches (step 1). Monitor App hosts our algorithms for inspecting data-plane rule faults. They take rules constructed by App as input and generate probe packets. Injector injects probe packets to data plane (step 2). Toward forwarding inspection, we need to know how switches handle probe packets, that is, which probe packet is processed by which rule (step 3). We obtain such probing results using the postcard method by NetSight [13]. Postcard augments a rule with two additional actions. First, it tags packet headers with a unique rule ID. Second, it forwards a copy of the tagged packet to Postcard Processor. It is such instrumented rules by postcard (rather than the original rules from App) that RuleScope installs on switches (step 1). This way, we can recover packet processing history on data plane. To ease extracting probe packets from received packets on Postcard Processor, we encapsulate a unique packet ID in the payload of each probe packet. Packet IDs facilitate also correlating a probe packet with corresponding rule(s) under inspection. Finally, Postcard Processor feeds the probing results back to Monitor App, where our algorithms continue to complete the remaining inspection process (step 4).

Of particular emphasis is **multi-switch probing**. It may seem more complex as some probe packets need to traverse through several switches before reaching the one they test (Figure 3). However, we can simplify it in a straightforward way. Specifically, Postcard Processor can directly inject probe packets to any switch under test. This also enables parallel test among switches. Another concern is that a probe packet may further traverse through other switches, hit rules therein, and trigger unnecessary postcarded probe packets. We can make packet ID of probe packets to correlate to the switch it tests. Then unnecessary postcards a rule triggers on uncorrelated switches will not affect inspection accuracy. In light of these observations, we implement only the single-switch scenario for ease of evaluation and presentation.

B. Experiment Setup

Control plane: We use the Ryu OpenFlow controller [15] as the controller of RuleScope testbed. The controller runs on a server with Intel(R) Xeon(R) CPU X5560 (8M cache, 2.80 GHz, 36 GB memory), which also hosts our Monitor App, Injector, and Postcard Processor. We use ClassBench [25] as App for generating flow table dataset. Monitor App hosts our key design—Algorithms 1-3. We implement the algorithms in Python and C++ (2600+ lines of code in Python for algorithm framework and 300+ lines of code in C++ for high performance header space analysis, in addition to 2800+ lines of open-source MiniSat codes in C++ [23]). The server communicates with data plane via two 1 GE interfaces. One is for Injector to inject probe packets. The other is for Postcard Processor (in Python, 230+ lines of code) to issue instrumented rules and collect postcarded packets.

Data plane: We use a Pica8 P-3297 [16] as the SDN switch of RuleScope testbed. Per later results, 300+ rules on the switch incur about 1500+ 52-byte probe packets within 16 seconds. They cost only 0.06% of controller-switch link bandwidth and 0.0003% of switching fabric capacity (176 Gbps). Such volume of traffic can hardly affect high-performance networks. We thus omit measuring the impact of probe packets on flow rate and mount no end-hosts to data plane.

Rule-set: We generate the rule set for OpenFlow switches using network filter generation tool ClassBench [25]. Specifically, we generate prefix based forwarding and ACL filters using ClassBench and translate the filters into equivalent OpenFlow rules. We cannot simply assign priorities to these original rules. Doing so introduces various issues that cause forwarding errors. For example, assume that we

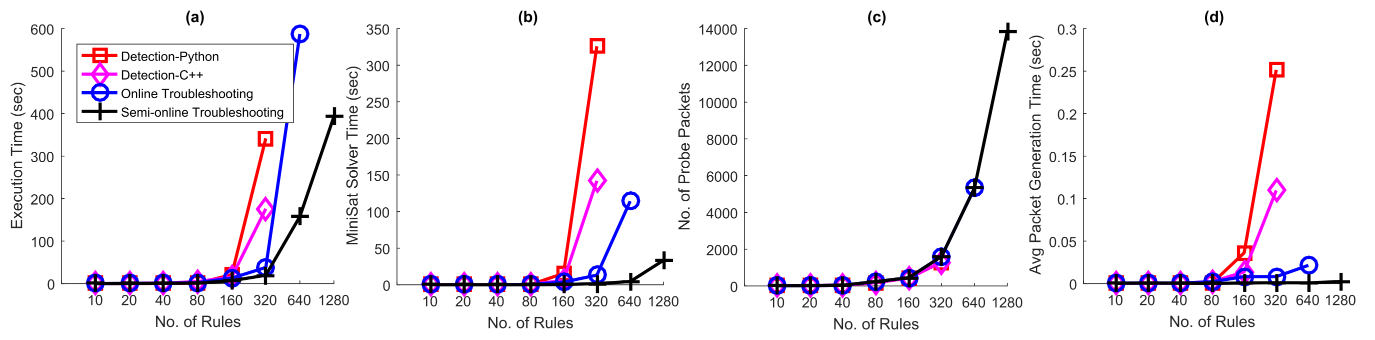


Fig. 4. Comparison of detection and troubleshooting algorithms with varying size of correct flow tables. (Same legend for all subfigures.) (a) Overall Execution Time. (b) Probe Generation Time. (c) Probe Complexity. (d) Per-probe Generation Time.

TABLE III
DEPENDENCY DEPTH OF FLOW TABLES

No. of rules	dependency depth	No. of rules	dependency depth
10	3	160	5
20	4	320	5
40	4	640	6
80	5	1280	7

TABLE IV
RATIO OF PROBE GENERATION TIME OVER OVERALL EXECUTION TIME WITH VARYING NUMBER OF CORRECT RULES

Algorithm	Flow Table Size					
	10	20	40	80	160	320
Detection-Python	4.2%	5.6%	5.1%	7.7%	74.2%	95.3%
Detection-C++	2.8%	2.0%	2.3%	8.7%	38.2%	80.7%
Online Tr	2.8%	3.6%	3.6%	22.2%	28.9%	34.7%
Semi-online Tr	3.6%	2.9%	3.1%	4.3%	4.4%	8.2%

use ClassBench to generate two rules in the following order: ipsrc=10.10.*.*, drop; ipsrc=10.10.10.10, forward via port 80. The first rule drops packets from the 10.10.0.0/16 subnet while the second one forwards packets with 10.10.10.10 as source IP address via port 80. In practice, it is normal that the second rule should be granted higher priority. Thus, simply assigning decreasing priorities to generated rules following their original order will cause forwarding errors. In this case, packets with source IP address of 10.10.10.10 will be wrongly dropped. To avoid such issues, we pre-process generated rules before assigning priorities to them. Specifically, we group overlapping rules and, for each group, sort them in decreasing order of rule granularity. Then within each sorted group, we assign descending integer priorities to rules. Priority ranges of different group do not overlap. We then generate dependency graph of the OpenFlow rule-set using the dependency generation tool in RuleTris [9]. Table III shows the depth of the dependency graphs for the rule tables used in the experiment.

VI. EVALUATION

In this section, we evaluate the efficacy and efficiency of our algorithms on the RuleScope testbed. Efficacy is measured in terms of how accurately the algorithms detect/troubleshoot rule faults on data plane. Experiments show that the detection algorithm can detect faulty rules without false negatives/positves while troubleshooting algorithms can faithfully construct the dependency graph of on-switch flow table. We focus more on reporting statistics for efficiency, which is measured in terms of execution time and the number of probe packets.

Rule fault emulation: We concern with both missing faults and priority faults. To emulate missing faults, we directly ignore issuing some rules to the switch. We conduct continuous measurements on Pica8 P-3297 and find no priority fault as in [3]. As a work-around, we swap priorities of some overlapping rules before issuing them to the switch to emulate priority faults.

Algorithm implementation: We implement the core header space analysis library in Python and C++. Except *Detection-Python*, all other algorithms use the C++ implementation for efficiency. We highlight the efficiency difference between Python and C++ implementations of the detection algorithm to exhibit the potential efficiency improvement space in code optimization.

A. Detection Versus Troubleshooting With Correct Rules

We first evaluate the algorithms under varying number of correct on-switch rules. Figure 4 reports the evaluation results.

Figure 4(a): Overall execution time: All algorithms' overall execution time increases with flow table size. Semi-online troubleshooting algorithm keeps being faster than its online counterpart. For experiment instances reported in Figure 4, semi-online costs 23.1% less time than does online given 10 rules whereas this gap increases to 48.3% given 320 rules. We were expecting that detection be faster than troubleshooting. Then we could run faster detection first and invoke slower troubleshooting only if rule faults are detected. Small flow tables do live up to such expectation. For 10 to 80 rules, the overall execution time of detection algorithm is 64.9% (83.8%) on average of that of online (semi-online) troubleshooting algorithm. However, when flow table size reaches 160, detection becomes slower than troubleshooting. When flow table size is 320, detection algorithm in C++ costs 176.0 seconds whereas online troubleshooting algorithm and semi-online troubleshooting algorithm take 37.9 seconds and 19.6 seconds, respectively. Furthermore, we observe that C++ implementation is faster than Python implementation by around 30% to 50%. On the other hand, the online and semi-online troubleshooting algorithms scale better with large rule tables.

Figures 4(b)-(d): Probe overhead: The main reason why detection is much slower than troubleshooting for large flow

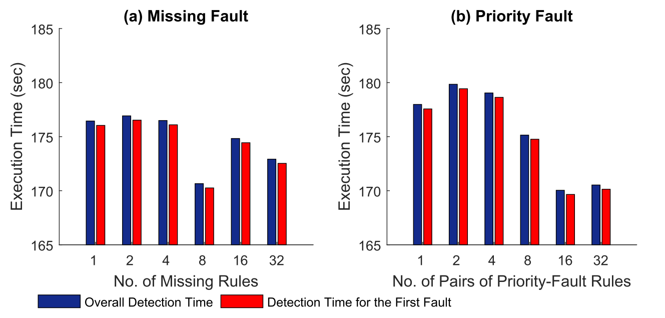


Fig. 5. Execution time of detection algorithm with 320 rules including varying number of (a) missing rules and (b) pairs of priority-fault rules.

tables is that its probe generation time leaps when flow table size exceeds 160 whereas troubleshooting algorithms' stays smoother (Figure 4(b)). All algorithms' probe generation time increases with flow table size. The ratio of probe generation time over overall execution time increases with flow table size as well (Table IV, with limited deviation for small flow tables). Although taking different time, all algorithms generate comparative number of probe packets (Figure 4(c)). This indicates that the algorithms have quite different per-probe generation time (Figure 4(d)). For 320 rules, per-probe generation time of detection-C++, online troubleshooting, and semi-online troubleshooting is 135 ms, 23 ms, and 12 ms, respectively. This detection-troubleshooting gap stems from the scale of the second input/constraint for MiniSat solver (Algorithms 1-3). When generating a probe packet for a pair of rules, detection algorithm considers all rules directly or indirectly depended by the pair as constrains whereas troubleshooting algorithms take into account only the directly-dependent ones.

B. Detection With Faulty Rules

We then evaluate time efficiency of detection algorithm with a 320-rule flow table including varying number of faulty rules. The number of faulty rules comprises the number of missing rules and the number of pairs of priority-fault rules. Figure 5(a) and Figure 5(b) respectively report the evaluation results under 1-32 randomly picked missing rules and pairs of priority-fault rules. All instances use the same flow table and therefore the same set of probe packets.

We have three observations from the results. First, detecting the first faulty rule approximates the overall detection time, regardless of the number of faulty rules. Second, varying number of faulty rules causes limited fluctuation to the overall detection time. The standard deviation of the overall detection time is around 2 seconds, which is only 0.6% of the average overall detection time. The preceding two observations are because over 95% of the overall detection time is for generating probe packets (Table IV). Only after generating all probe packets can detection algorithm start inspect rule correctness. Third, more missing rules does not necessarily shorten detection time. To what extent can a missing rule affect detection time depends on the number of its associated probe packets. The more its associated probe packets are, the more it accelerates detection because of fewer postcarded packets to process.

TABLE V

RATIO OF PROBE GENERATION TIME OVER OVERALL EXECUTION TIME WITH VARYING NUMBER OF FAULTY RULES AMONG 320 ONES

Algorithm	No. of Faulty Rules					
	1	2	4	8	16	32
Det-Python	95.2%	95.2%	95.4%	95.2%	95.4%	95.6%
Det-C++	81.8%	81.7%	82.4%	84.9%	81.7%	82.4%
Online Tr	34.2%	34.5%	34.0%	27.4%	24.7%	26.8%
Semi-on Tr	8.3%	8.3%	8.2%	7.4%	5.3%	5.6%

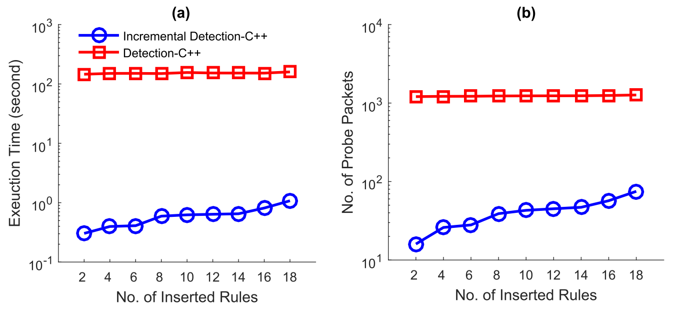


Fig. 7. Comparison of overall execution time and the number of probe packets between incremental and non-incremental detection algorithms. The flow table contains 320 existing rules. Note that the y-axis is in log scale. (a) Overall Execution Time. (b) Probe Complexity.

C. Detection Versus Troubleshooting With Faulty Rules

Next, we compare the performance of all algorithms with a 320-flow table including varying number of faulty rules. For each instance reported in Figure 6, faulty rules contain both randomly picked missing rules and priority-fault rules. Again, limited number of missing rules make the execution time of each algorithm rarely fluctuate (Figure 6(a)). The overall execution time of online troubleshooting algorithm and of semi-online troubleshooting algorithm are respectively 18.4% and 11.5% on average of that of detection-C++ algorithm. Figure 6(b) reports probe generation time while Table V reports the ratio of it over overall execution time. The ratio corresponding to detection-C++ and detection-Python keep constant as it works on the same 320 rules for each instance. For troubleshooting algorithms, more faulty rules may yield less probe generation time when detection of them helps simplify the constraints for MiniSat solver. Another major part of overall execution time is probe transmission time (Figure 6(c)), which is proportional to the number of probe packets (Figure 6(d)). Probe transmission time aggregates round-trip time of probe packets for all detection/troubleshooting rounds during one algorithm execution. For a batch of probe packets in each round, the round-trip time is from when the first probe packet leaves Injector to when the last probe packet reaches Postcard Processor. Such round-trip time for a probe packet depends on network bandwidth and status. On our RuleScope testbed, the round-trip time per probe packet is about 8 ms.

D. Incremental Detection and Troubleshooting

Finally, we evaluate how much the incremental detection and troubleshooting algorithms can improve the flow table inspecting efficiency after a full detection/troubleshooting from scratch. Figure 7 and Figure 8 report the overall execution time and probe complexity of incremental

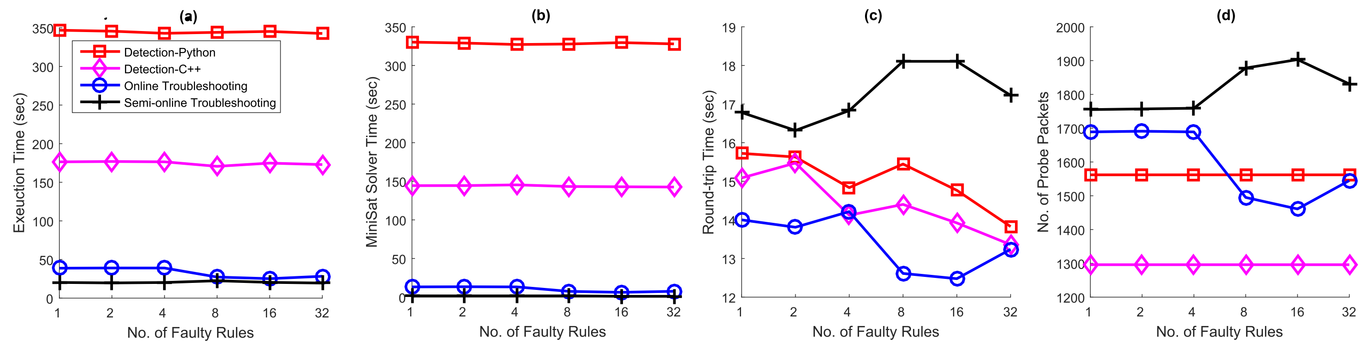


Fig. 6. Comparison of detection and troubleshooting algorithms with 320 rules including varying number of faulty rules. The number of faulty rules comprises the number of missing rules and the number of pairs of priority-fault rules. (Same legend for all subfigures.) (a) Overall Execution Time. (b) Probe Generation Time. (c) Probe Transmission Time. (d) Probe Complexity.

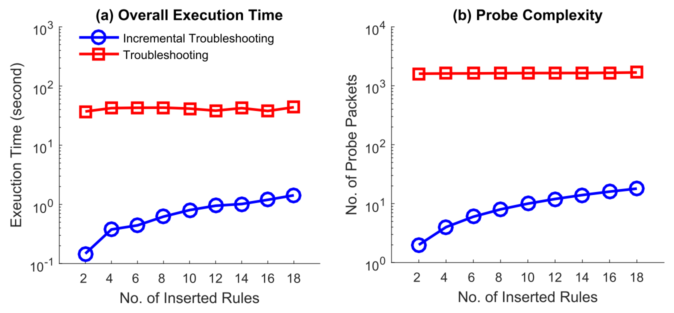


Fig. 8. Comparison between incremental and non-incremental troubleshooting algorithms. The flow table contains 320 existing rules. Note that the y-axis is in log scale. (a) Overall Execution Time. (b) Probe Complexity.

detection/troubleshooting algorithm compared with the corresponding non-incremental versions.

We have two observations from the results. First, the overall execution time of incremental detection/troubleshooting algorithms is under one second with a small rule update, which is faster than the non-incremental versions by two to three orders of magnitude. Second, the execution time and probe complexity of incremental detection/troubleshooting algorithms grows super-linearly with the size of the rule updates. These two properties ensure that the incremental algorithms can keep up with small but frequent network policy updates that are common in today's data center and enterprise networks.

VII. Conclusion

We have studied accurate yet efficient inspection of SDN forwarding and proposed RuleScope design. RuleScope provides a series of inspection algorithms to detect and troubleshoot forwarding faults on data plane. The detection algorithm exposes not only previously known missing faults but also recently discovered priority faults. Given that comprehensive network monitoring might solicit more than fault detection, we further propose troubleshooting algorithms. They uncover actual data-plane flow tables, which enable tracking real-time forwarding status and inferring how switches handle rule updates. Moreover, we extend the algorithms to be even more responsive with dynamically changing forwarding policies by exploiting the invariants in the dynamic rule updates. We believe the series of algorithms are important for building reliable SDN networks. To make our algorithms readily applicable, we explore also various techniques toward enhancing efficiency without sacrificing accuracy. We implement RuleScope with Ryu controller and Pica8 P-3297 switch. O algorithms deliver accurate and efficient inspection with limited overhead. For future work, we plan to exercise RuleScope on switches with identified priority faults [3] and arm RuleScope with efficiency enhancements [26], [27].

Acknowledgement

The authors thank IEEE INFOCOM 2016 chairs and reviewers and IEEE/ACM Transactions on Networking editors and reviewers for their helpful feedback.

References

[1] K. Bu *et al.*, "Is every flow on the right track?: Inspect SDN forwarding with rulescope," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.
[2] M. Kuzniar, P. Peresini, and D. Kostić, "Providing reliable fib update acknowledgments in SDN," in *Proc. ACM CoNEXT*, 2014, pp. 415–422.
[3] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. PAM*, 2015, pp. 347–359.
[4] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *Proc. ACM SIGCOMM*, 2014, pp. 527–538.
[5] M. Kuzniar, P. Peresini, and D. Kostic, "Proboscope: Data plane probe packet generation," EPFL, Lausanne, Switzerland, Tech. Rep. EPFL-REPORT-201824, 2014.
[6] J. V. Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 4, pp. 560–571, May 2003.
[7] H. Song and J. Turner, "Fast filter updates for packet classification using TCAM," in *Proc. IEEE GLOBECOM*, Nov. 2006, pp. 1–5.
[8] T. Mishra and S. Sahni, "DUOS—Simple dual TCAM architecture for routing tables with incremental update," in *Proc. IEEE ISCC*, Jun. 2010, pp. 503–508.
[9] X. Wen *et al.*, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *Proc. IEEE ICDCS*, Jun. 2016, pp. 179–188.
[10] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.
[11] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proc. ACM CoNEXT*, 2012, pp. 241–252.
[12] P. Perešíni, M. Kuźniar, and D. Kostić, "Monocle: Dynamic, fine-grained data plane monitoring," in *Proc. ACM CoNEXT*, 2015, p. 32.
[13] 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 *Proc. USENIX NSDI*, 2014, pp. 71–85.

[14] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," in *Proc. ACM SIGCOMM*, 2013, pp. 87–98.

[15] *Ryu SDN Framework*, accessed on Jul. 2015. [Online]. Available: http://osrg.github.io/ryu/

[16] *P-3297 Datasheet—Pica8*, accessed on Jul. 2015. [Online]. Available: http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3297.pdf

[17] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. USENIX NSDI*, 2012, pp. 113–126.

[18] *OpenFlow Switch Specification - Version 1.4.0 (Wire Protocol 0x05)*, Open Networking Foundation, Menlo Park, CA, USA, Oct. 2013.

[19] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. USENIX NSDI*, 2013, pp. 15–27.

[20] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing customizable consistency properties in software-defined networks," in *Proc. USENIX NSDI*, 2015, pp. 73–85.

[21] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "OFFICER: A general optimization framework for openflow rule allocation and endpoint policy enforcement," in *Proc. IEEE INFOCOM*, Apr. 2015, pp. 478–486.

[22] C. Prakash *et al.*, "PGA: Using graphs to express and automatically reconcile network policies," in *Proc. ACM SIGCOMM*, 2015, pp. 29–42.

[23] *The MiniSat Page*, accessed on Mar. 2015. [Online]. Available: http://minisat.se/

[24] A. R. Curtis *et al.*, "Devoflow: Scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM*, 2011, pp. 254–265.

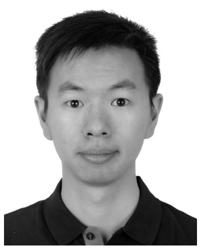
[25] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," in *Proc. IEEE INFOCOM*, Mar. 2005, pp. 2068–2079.

[26] X. Wen *et al.*, "Compiling minimum incremental update for modular SDN languages," in *Proc. ACM HotSDN*, 2014, pp. 193–198.

[27] K. Bu, "Gotta tell you switches only once: Toward bandwidth-efficient flow setup for SDN," in *Proc. IEEE SDDCS*, Apr. 2015, pp. 492–497.

Xitao Wen received the B.S. degree in computer science from Peking University, Beijing, China, in 2010, and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, USA, in 2016. His research interests span the area of networking and security in networked systems, with a current focus on software-defined network security and data center networks.

Kai Bu (A'13) 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 Assistant Professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. His research interests include wireless networking and network security. He is a member of the ACM, and the CCF. He was a recipient of the Best Paper Award of the IEEE/IFIP EUC 2011 and the Best Paper Nominees of the IEEE ICDCS 2016.



Bo Yang received the B.S. degree in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2013, and the M.S. degree in computer science from Zhejiang University, Hangzhou, China, in 2016. He is currently a Software Engineer with Microsoft, Shanghai, China. His research interests include software-defined network and network security.

Yan Chen received the Ph.D. degree in computer science from the University of California at Berkeley, Berkeley, CA, USA, in 2003. He is currently a Professor with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA. Based on Google Scholar, his papers have been cited over 7000 times and his h-index is 34. His research interests include network security, measurement, and diagnosis for large-scale networks and distributed systems. He received the Department of Energy Early CAREER Award in 2005, the Department of Defense Young Investigator Award in 2007, and the Best Paper nomination in ACM SIGCOMM 2010.

Li Erran Li (S'01–A'01–M'08–SM'10–F'13) received the Ph.D. degree in computer science from Cornell University. He was a Researcher with Bell Labs. He is currently with Uber and also an Adjunct Professor with the Computer Science Department, Columbia University. His research interests are in machine learning algorithms, artificial intelligence, and systems and wireless networking. He is an ACM Distinguished Scientist. He was an Associate Editor of the IEEE TRANSACTIONS ON NETWORKING and the IEEE TRANSACTIONS ON MOBILE COMPUTING. He co-founded several workshops in the areas of machine learning for intelligent transportation systems, big data, software defined networking, cellular networks, mobile computing, and security.

Xiaolin Chen is currently a Professor with the School of Information Science and Technology, Chuxiong Normal University, China. He is interested in enterprise and data center networks, network virtualization, and software-defined networking.

Jianfeng Yang was born in 1976. He received the bachelor's, master's, and Ph.D. degrees in communication and information system from Wuhan University in 1998, 2003, and 2009, respectively. He is currently an Associate Professor with Wuhan University. His research interests are in security and measurement for networking and large-scale distributed systems, high reliability, and real-time networking.

Xue Leng received the B.S. degree in computer science and technology from Harbin Engineering University, Harbin, China, in 2015. She is currently pursuing the Ph.D. degree major in computer science and technology with Zhejiang University, Hangzhou, China. Her research interests are software-defined network, network function virtualization, and network fault diagnosis.