Is Every Flow on The Right Track?: Inspect SDN Forwarding with RuleScope

Kai Bu*, Xitao Wen•, Bo Yang*, Yan Chen•, Li Erran Li°, Xiaolin Chen•,*
*College of Computer Science and Technology, Zhejiang University
•Northwestern University, °Fudan University, *Chuxiong Normal University

***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. Beyond simply detecting rule faults, the troubleshooting algorithms uncover actual data-plane flow tables. They help track real-time forwarding status and benefit reliable network monitoring. We explore various techniques for enhancing algorithm efficiency without sacrificing inspection accuracy. Experiments with our prototype on the Ryu SDN controller and Pica8 P-3297 switch show that RuleScope achieves accurate and efficient forwarding inspection with limited bandwidth and packet-switching overhead.**

I. Introduction

Recent measurement studies expose SDN forwarding's vulnerability to various faults [1]–[3]. When reflected 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 [4]. It is mainly attributed to switch firmware or hardware bugs [4] or even rule-update message loss [1]. Current SDN can hardly notice missing faults because it acknowledges rule update at batch level instead of desired rule level [1]. Furthermore, a priority fault occurs when overlapping rules (i.e., rules with common matching packets) violate designated priority order. Priority faults have already been observed on commercial switches [2]. Since SDN requires that a packet be processed by the highest-priority rule among matching ones [5], either missing faults or priority faults might lead to undesirable forwarding behavior. Rudimentary network debugging tools (e.g., ping, traceroute, SNMP, and tcpdump), however, do not support automatic and efficient analysis of centralized SDN [6]. It is thus important to explore SDN-specific inspection schemes.

Because priority faults are not discovered until recently [2], they evade previous solutions for inspecting SDN forwarding. For example, typical such solutions—ATPG [6], ProboScope [4], and Monocle [7]—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 occurs to rules with priorities p_{high} and p_{low}, http requests from the 10.10.0.0/16 subnet become allowed and breach the access control policy even if we may have verified the existence of both rules. Detecting missing faults alone is already proved NP-hard [4], [6]. It is more challenging to combat both missing and priority faults.

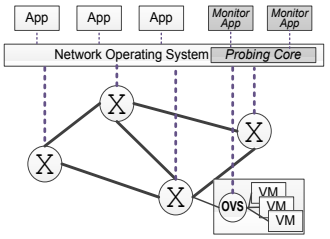

Fig. 1. RuleScope framework. X: Switch.

Priority	Matching	Action
p_{high}	src=10.10.∗.∗	deny
p_{low}	prot=http	allow

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

In this paper, we present the RuleScope system for accurately and efficiently inspecting SDN forwarding. Beyond existing inspection solutions [4], [6], [7], RuleScope detects not only missing faults but also priority faults. It can also uncover actual data-plane forwarding states. In line with established systems [4], [6], [7], 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 [8], 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 a series of 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 [9]. 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 data-plane flow tables. This enables tracking real-time forwarding status and inferring how switches handle rule updates. It is, however, computationally challenging to generate all necessary 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.

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.
- Validate accuracy and efficiency of RuleScope through theoretical analysis and experiments on a testbed with the Ryu controller [10] and Pica8 P-3297 switch [11].

Paper organization. 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 and Section V respectively implement RuleScope prototype and report evaluation results. Finally, Section VI concludes the paper and indicates future work.

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 [5], 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 $\{0,1\}^l$ [12]*[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 running to switches [12], [14], [15], we assume that FT contains neither duplicate rules nor obscured rules. An obscured rule cannot take effect due to its matching space being a subset of matching space of higher-priority 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 [1]–[3].

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 [4]. Current SDN, however, acknowledges rule updates at batch level instead of desired rule level [1]. 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}.*

A recent measurement study reveals priority faults on commercial SDN switches [2]. 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 [2], the one installed later always

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

[2] We omit rule fields that do not affect forwarding for simplicity [13].

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 [4]. If a probe packet for rule $r \in FT_{\text{ctr}}$ 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 [4], 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 $r_i \in FT_{\text{ctr}}$, 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 $r_i^{\text{M}} = (x_{i0}, ..., x_{ib}, ..., x_{i(l-1)})$, where $x_{ib} \in \{0, 1, *\}$ and $0 \le b \le l-1$. Matching with r_i is equivalent to the following conjunction being satisfied.

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

where

$$S(x_{ib}) = \begin{cases} x_{ib}, & \text{if } x_{ib} = 1; \\ \neg x_{ib}, & \text{if } x_{ib} = 0; \\ True, & \text{if } x_{ib} = *. \end{cases}$$

Then not matching with r_i is equivalent to the following disjunction being satisfied.

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

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

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

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 Formula 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.

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

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 Formula 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 a number of True to Formula 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.*

Proof. We first explore possible cases of troubleshooting priority order for a pair of overlapping rules $(r_i, r_j) \in FT_{\text{ctr}}$. Let rule $r_{\cap ij}$ capture their matching-space intersection with $r_{\cap ij}^{\text{MS}} = r_i^{\text{MS}} \cap r_j^{\text{MS}}$. Let $R_{ij}^{\cap}$ represent the set of rules that overlap with both r_i and r_j, that is, $R_{ij}^{\cap} = \{r \mid r \in FT_{\text{ctr}} - r_i - r_j \text{ and } r^{\text{MS}} \cap r_{\cap ij}^{\text{MS}} \neq \emptyset\}$. Assume that $r_{\cap ij}^{\text{M}}$ has l' bits of wildcards (i.e., $|r_{\cap ij}^{\text{MS}}| = 2^{l'}$). We derive the number of probe packets in the following two cases according to whether rules in $R_{ij}^{\cap}$ jointly exhaust all $2^{l'}$ points in $r_{\cap ij}^{\text{MS}}$.

Case 1: When rules in $R_{ij}^{\cap}$ CANNOT jointly exhaust all $2^{l'}$ points in $r_{\cap ij}^{\text{MS}}$, one probe packet is sufficient. We explain

this along with how we construct the l-bit packet header for a feasible probe packet p. For the $l - l'$ non-wildcard bits in $r^{\mathrm{M}}_{\cap ij}$, we simply copy them to corresponding bits of p. For $r \in R^{\cap}_{ij}$, let $r^{l'\mathrm{MS}}$ denote the subset of r^{MS} supported by the l' bits locating at the same positions as the l' wildcards in $r^{\mathrm{M}}_{\cap ij}$. We then select one point in $r^{l'\mathrm{MS}}_{\cap ij} - \bigcup_{\forall r \in R^{\cap}_{ij}} r^{l'\mathrm{MS}}$ for the left l' bits of p.

Case 2: When rules in $R^{\cap}_{ij}$ CAN jointly exhaust all $2^{l'}$ points in $r^{\mathrm{MS}}_{\cap ij}$, we have $r^{l'\mathrm{MS}}_{\cap ij} = \bigcup_{\forall r \in R^{\cap}_{ij}} r^{l'\mathrm{MS}}$ and require probe packets to enumerate all possible rule combinations. We next complete the proof by studying two extreme cases.

- First, $R^{\cap}_{ij}$ contains only one rule r and r^{MS} has same-position l' bits of wildcards as $r^{\mathrm{M}}_{\cap ij}$. In this case, we need only one probe packet because any point in $r^{l'\mathrm{MS}}_{\cap ij}$ simultaneously exercises r_i, r_j, and r.
- Second, $R^{\cap}_{ij}$ contains $2^{l'}$ rules each corresponding to one of the $2^{l'}$ points in $r^{\mathrm{M}}_{\cap ij}$. In this case, although all rules in $R^{\cap}_{ij}$ overlap with both r_i and r_j, they mutually non-overlap. We thus need to probe $2^{l'}$ rule triplets, that is, (r_i, r_j, r) for all r in $R^{\cap}_{ij}$.

With the second case we complete the proof. □

For cases when $r^{l'\mathrm{MS}}_{\cap ij} = \bigcup_{\forall r \in R^{\cap}_{ij}} r^{l'\mathrm{MS}}$ and $|R^{\cap}_{ij}|$ 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. Design

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.

A. Probe

As Theorem 1 shows, probe-generation for priority fault is equivalent to that for missing fault. For probing r_i's missing fault, we solve Formula 3 to obtain an assignment of $r^{\mathrm{MS}}_i = (\overline{x_{i0}}, ..., \overline{x_{ib}}, ..., \overline{x_{i(l-1)}})$. We then construct the l bits in probe packet p's packet header used for matching as follows.

$$p_b = \begin{cases} \overline{x_{ib}}, & \text{if } \overline{x_{ib}} = 0 \text{ or } 1; \\ 0 \text{ or } 1, & \text{if } \overline{x_{ib}} = True. \end{cases} \quad (4)$$

We solve Formula 3 using a high-performance SAT solver called MiniSat [16] (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 $SampleProbe(r_i, FT'_{\mathrm{sw}})$, where $FT'_{\mathrm{sw}} = \{r_j \mid r_j \in FT_{\mathrm{sw}} \text{ and } r^{\mathrm{P}}_j > r^{\mathrm{P}}_i\}$ (Theorem 1). Given that flow table FT_{sw} usually contains hundreds of rules [17], FT'_{sw} might impose too many constraints on $SampleProbe(r_i, FT'_{\mathrm{sw}})$. This seriously limits the speed of probe generation. Even worse, given that FT_{sw} may encounter missing and priority faults, rules $r_j \in FT'_{\mathrm{sw}}$ are hardly known a priori. Any $r_j \in FT_{\mathrm{sw}}$ with $r^{\mathrm{MS}}_j \cap r^{\mathrm{MS}}_i \neq \emptyset$ could have higher priority than r_i does on the switch. FT'_{sw} then should incorporate all $r_j \in FT_{\mathrm{ctr}}$ with $r^{\mathrm{MS}}_j \cap r^{\mathrm{MS}}_i \neq \emptyset$. Whenever such r_j leads to $r^{\mathrm{MS}}_j \supset r^{\mathrm{MS}}_i$, $SampleProbe(r_i, FT'_{\mathrm{sw}})$ is not solvable. Such undesirable cases occur quite frequently because flow tables contain many overlapping rules.

Our rule fault detection algorithm counterintuitively calibrates the inputs for $SampleProbe(\cdot)$ toward efficiency and efficacy without affecting detection accuracy.

Algorithm 1: Rule Fault Detection Algorithm

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

B. Detection

Goal. The detection algorithm aims to find faulty rules if any on data plane. A faulty rule does not act on some or all packets that it should process. The cause could be either missing faults or priority faults. Consider again the example in Table I—the toy rule set has two overlapping rules with priorities of p_{high} and p_{low}. When no fault exists, we expect that packets match with both rules be processed by p_{high}-rule. If one such packet does not follow p_{high}-rule's action, we consider p_{high}-rule as faulty. It is possible 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 $SampleProbe(\cdot)$.

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) [9].

- 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 enhance efficiency through reducing the number of probe packets and speeding 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. □

C. Troubleshooting

Goal and pre-thought. The troubleshooting algorithm aims to uncover actual flow table FT_{sw} on a switch. Since it is hard to recover exact rule-priority values, we build dependency graph $G_{sw} =< V, E >$ of 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.

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 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 $\mathcal{O}(|E|) = \mathcal{O}(|V|^2) = \mathcal{O}(|FT_{\text{ctr}}|^2) = \mathcal{O}(n^2)$ 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 $\mathcal{O}(|V|) = \mathcal{O}(|FT_{\text{ctr}}|) = \mathcal{O}(n)$ complexity.

Algorithm 2: Online Troubleshooting Algorithm

Input : Flow table FT_{ctr}
Output: Dependency graph $G_{\text{sw}} =< V, E >$ of data plane rules

1 $V \leftarrow \{v_i \mid v_i$ corresponds to $r_i \in FT_{\text{ctr}}\}$;
2 $E = \emptyset$;
3 Set $S \leftarrow$ all pairs of overlapping rules in FT_{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 $v \in FT_{\text{ctr}}$ 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 $v_{\text{hit}} \leftarrow$ the rule in V' that processes p;
19 **foreach** $v \in V' - \{v_{\text{hit}}\}$ **do**
20 $E \leftarrow E \cup \{< v_{\text{hit}}, v >\}$;
21 $S \leftarrow S - \{(v_{\text{hit}}, v)\}$;
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** $G_{sw} =< V, E >$;

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 $\mathcal{O}(2^n)$.

Algorithm 3: Semi-online Troubleshooting Algorithm

Input : Flow table FT_{ctr}
Output: Dependency graph $G_{\text{sw}} =< V, E >$ 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** $G_{sw} =< V, E >$;

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 $\mathcal{O}(|V|^2) = \mathcal{O}(|FT_{\text{ctr}}|^2) = \mathcal{O}(n^2)$. 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 $\mathcal{O}(|V|) = O(|FT_{\text{ctr}}| = \mathcal{O}(n)$. The complexity of Algorithm 3 in terms of the number of probe packets is $\mathcal{O}(n^2) \times \mathcal{O}(n) = \mathcal{O}(n^3)$.

IV. 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 2 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 [8]. 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 2). 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

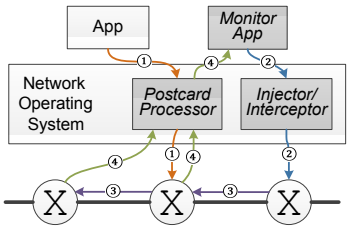

Fig. 2. RuleScope prototype architecture and work flow. X: Switch.

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 [10] 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), where co-locates App, our Monitor App, Injector, and Postcard Processor. We use ClassBench [18] as App for generating flow table dataset. Monitor App hosts our key design—Algorithms 1-3. We implement the algorithms in Python (2600+ lines of code, in addition to 2800+ lines of open-source MiniSat codes in C++ [16]). 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 [11] 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.

V. 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. For emulating 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 [2]. As a work-around, we swap priorities of some overlapping rules before issuing them to the switch.

A. Detection versus Troubleshooting with Correct Rules

We first evaluate the algorithms under varying number of correct on-switch rules. Figure 3 reports the evaluation results.

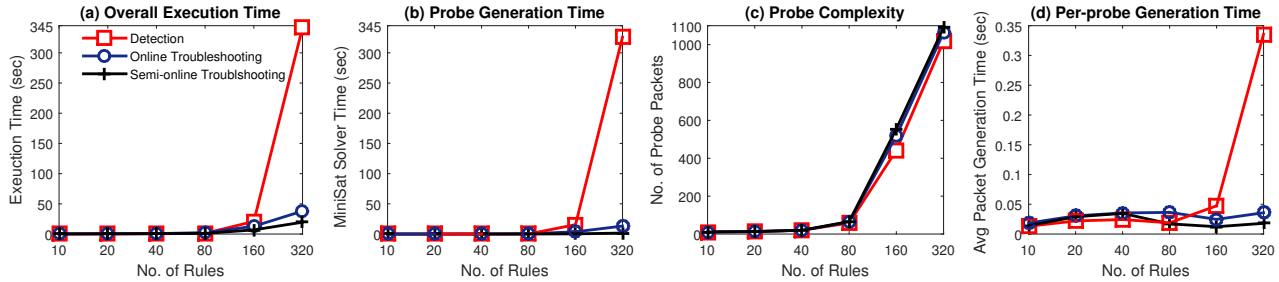

Fig. 3. Comparison of detection and troubleshooting algorithms with varying size of correct flow tables. (Same legend for all subfigures.)

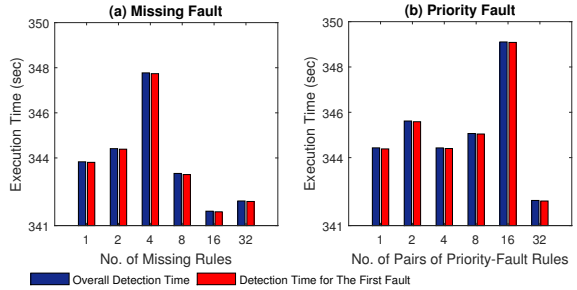

Fig. 4. Execution time of detection algorithm with 320 rules including varying number of (a) missing rules and (b) pairs of priority-fault rules.

Figure 3(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 3, 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 costs 341.6 seconds whereas online troubleshooting algorithm and semi-online troubleshooting algorithm take 37.9 seconds and 19.6 seconds, respectively.

Figures 3(b)-(d): Probe overhead. The main reason why detection is much slower than troubleshooting for large flow tables is that its probe generation time leaps when flow table size exceeds 160 whereas troubleshooting algorithms' stays smoother (Figure 3(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 II, with limited deviation for small flow tables). Although taking different time, all algorithms generate comparative number of probe packets (Figure 3(c)). This indicates that the algorithms have quite different per-probe generation time (Figure 3(d)). For 320 rules, per-probe generation time of detection, online troubleshooting, and semi-online troubleshooting is 340 ms, 40 ms, and 20 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-depended ones.

TABLE II
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	4.2%	5.6%	5.1%	7.7%	74.2%	95.3%
Online Tr	2.8%	3.6%	3.7%	22.2%	28.9%	34.5%
Semi-online Tr	3.6%	2.8%	3.1%	4.3%	4.4%	8.2%

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 4(a) and Figure 4(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 II). 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.

C. Detection versus Troubleshooting with Faulty Rules

Finally, we compare the performance of all algorithms with a 320-flow table including varying number of faulty rules. For each instance reported in Figure 5, 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 5(a)). The overall execution time of online troubleshooting algorithm and of semi-online troubleshooting algorithm are respectively 9.6% and 5.9% on average of that of detection algorithm. Figure 5(b) reports probe generation time while Table III reports the ratio

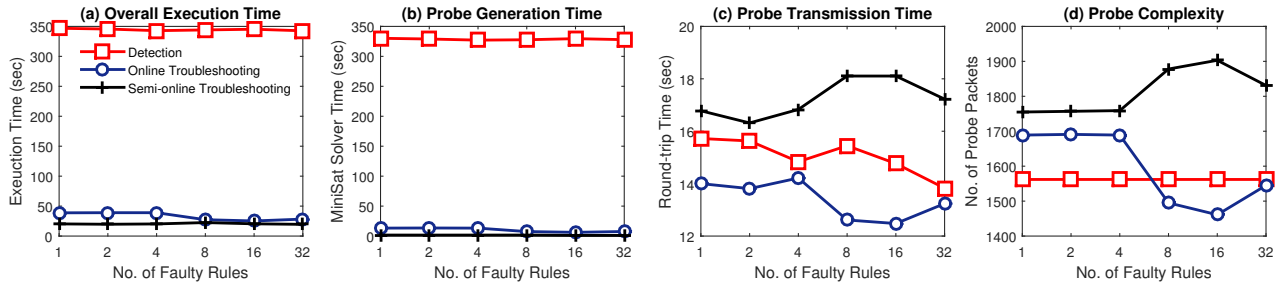

Fig. 5. Comparison of detection and troubleshooting algorithms with 320 rules including varying number of faulty rules. The number of faulty rules comprises the number of missing rules and the number of pairs of priority-fault rules. (Same legend for all subfigures.)

TABLE III
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
Detection	95.2%	95.2%	95.4%	95.2%	95.4%	95.6%
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%

of it over overall execution time. The ratio corresponding to detection algorithm keeps 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 5(c)), which is proportional to the number of probe packets (Figure 5(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.

VI. CONCLUSION

We have studied accurate yet efficient inspection of S-DN 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. Such outputs of our algorithms are important for building reliable 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. The proposed algorithms deliver accurate and efficient inspection with limited overhead. For future work, we will evaluate RuleScope on switches with identified priority faults [2], extend RuleScope to handle dynamic rule updates [7], and arm RuleScope with efficiency enhancements [19], [20].

ACKNOWLEDGMENT

This work is supported in part by the Fundamental Research Funds for the Central Universities under Grant No. 2014Q-NA5012, the National Science Foundation of China under Grant No. 61402404 and 60963021, the NSF under Grant No. CNS-1219116, and the program for Innovative Research Team in University of Yunnan Province. The authors would also like to sincerely thank IEEE INFOCOM 2016 chairs and reviewers for their helpful feedback.

REFERENCES

[1] M. Kuzniar, P. Peresini, and D. Kostić, "Providing reliable fib update acknowledgments in sdn," in *ACM CoNEXT*, 2014, pp. 415–422.
[2] M. Kuzniar, P. Peresini, and D. Kostic, "What you need to know about sdn flow tables," in *PAM*, 2015.
[3] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic engineering with forward fault correction," in *ACM SIGCOMM*, 2014.
[4] M. Kuzniar, P. Peresini, and D. Kostic, "Proboscope: Data plane probe packet generation," Tech. Rep., 2014.
[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
[6] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *ACM CoNEXT*, 2012, pp. 241–252.
[7] P. Peresini, M. Kuzniar, and D. Kostic, "Monocle: Dynamic, fine-grained data plane monitoring," in *ACM CoNEXT*, 2015.
[8] N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX NSDI*, 2014.
[9] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying sdn programming using algorithmic policies," in *ACM SIGCOMM*, 2013, pp. 87–98.
[10] Ryu SDN Framework,. [Online]. Available: http://osrg.github.io/ryu/
[11] P-3297 Datasheet - Pica8,. [Online]. Available: http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3297.pdf
[12] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks." in *USENIX NSDI*, 2012, pp. 113–126.
[13] O. S. Specification, "Version 1.4. 0, october 14, 2013."
[14] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Officer: A general optimization framework for openflow rule allocation and endpoint policy enforcement," in *IEEE INFOCOM*, 2015.
[15] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," in *SIGCOMM*, 2015.
[16] The MiniSat Page,. [Online]. Available: http://minisat.se/
[17] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *ACM SIGCOMM*, 2011, pp. 254–265.
[18] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," in *IEEE INFOCOM*, 2005, pp. 2068–2079.
[19] X. Wen, C. Diao, X. Zhao, Y. Chen, L. E. Li, B. Yang, and K. Bu, "Compiling minimum incremental update for modular sdn languages," in *ACM HotSDN*, 2014, pp. 193–198.
[20] K. Bu, "Gotta tell you switches only once: Toward bandwidth-efficient flow setup for sdn," in *IEEE SDDCS*, 2015, pp. 492–497.