Fast Testing Network Data Plane with RuleChecker

Peng Zhang, Cheng Zhang, and Chengchen Hu
Department of Computer Science and Technology
MOE Key Lab for Intelligent Networks and Network Security
Xi'an Jiaotong University

***Abstract*—A key feature of Software Defined Network is the decoupling of control pane and data plane. Although delivering huge benefits, such a decoupling also brings a new risk: the data plane states (*i.e.*, flow tables) may deviate from the control plane policies. Existing data plane testing tools like Monocle check the correctness of flow tables by injecting probes. However, they are limited in four aspects: (1) slow in generating probes due to solving SAT problems, (2) may raise false negatives when there are multiple missing rules, (3) do not support incremental probe update to work in dynamic networks, and (4) cannot test cascaded flow tables used by OpenFlow switches.**

To overcome these limitations, we present RuleChecker, a fast and complete data plane testing tool. In contrast to previous tools that generate each probe by solving an SAT problem, RuleChecker takes the flow table as whole and generates all probes through an iteration of simple set operations. By leveraging Binary Decision Diagram (BDD) to encode sets, we make RuleChecker extremely fast: around $5\times$ faster than Monocle (when detecting rule missing faults), and nearly $20\times$ faster than RuleScope (when detecting both rule missing and priority faults), and can update probes in less than 2 ms for 90% of cases, based on the Stanford backbone rule set.

***Index Terms*—Software Defined Network, Data plane faults, Probe generation, Binary Decision Diagram**

I. Introduction

Software Defined Networking (SDN) decouples control functions away from the data plane, thereby offering a centralized, flexible, and programmable network control. Such a decoupling means that the control plane, *i.e.*, controller, should be physically separated from data plane devices, *i.e.*, switches. Thus, a new risk rises: the data plane states may not agree with the control plane policies. For example, switches may fail to correctly install the rules issued by the controller [1]–[3], due to software bugs [4], [5], hardware faults [6], or attacks [7]–[9]. However, currently SDN provides no effective means to guarantee that the data plane states always correspond to the control plane policies.

Several tools have been proposed to either monitor or test the *correspondence* of the data plane. Data plane monitoring tools like VeriDP [10], [11] and REV [12] let switches tag packets with input/output ports, so as to check whether packets have been forwarded according to the rules. Both VeriDP and REV cannot handle packet rewrites, and need to modify SDN switches to add tags. Data plane testing tools like Monocle [13] and RuleScope [14] detect rule missing fault and priority fault by generating probes for rules, and checking whether the switch outputs the probes according to their corresponding rules. Compared with VeriDP and REV, Monocle and RuleScope need to send a small number of probe packets, require no switch modification and are thus a more preferable approach to check the correspondence of network data plane. However, we find both Monocle and RuleScope are fundamentally limited in the following four aspects.

(1) They are relatively slow in generating probes due to the need of solving Boolean Satisfiability (SAT) problems. For example, Monocle needs more than 42 seconds to generate probes for a production flow table of 10,958 rules, while RuleScope needs around 345 seconds to generate probes for a synthetic flow table of 320 rules.

(2) They may generate false negatives when there are multiple missing rules that are correlated. The reason is that they assume that when a rule r is missing, the probe for r will match another rule r' whose priority is lower than r, which will forward the probe to a different port. A false negative may raise if r' is also missing.

(3) They do not support incremental probe update, and are thus inefficient under dynamic network re-configurations. Specifically, when a rule is added or deleted, both Monocle and RuleScope need to recompute all affected probes, each of which corresponding to an SAT problem. Under frequent network re-configurations, they may fail to keep pace with changes at the control plane.

(4) They cannot test cascaded flow tables, a mandatory feature of OpenFlow. As the *de facto* standard for SDN, OpenFlow uses pipelined packet processing, which consists of multiple flow tables cascaded together. Cascaded flow tables make packet processing more flexible, and can greatly reduce rule numbers. However, neither Monocle nor RuleScope can be easily extended to test cascaded flow tables.

To address the above limitations, this paper presents RuleChecker, a fast and complete tool to test network data plane. Architecturally, RuleChecker is a transparent proxy sitting in-between the controller and switches. It monitors (without blocking) the rule install/remove messages, and computes/updates probes based on the rules. At the same time, it injects probes into the data plane and verifies the collected probes. RuleChecker has four key ingredients that respectively address the four limitations listed above.

(1) RuleChecker uses a new probe generation method, which does not require solving SAT problems. In this method, RuleChecker treats matching fields of rules as sets, and generates all the probes through an iteration of simple set operations. Since set operations can be efficiently performed using Binary Decision Diagrams (BDDs), RuleChecker can generate probes

978-1-5090-6501-1/17/$31.00

TABLE I
Comparison of RuleChecker and the other two flow table testing tools, *i.e.*, Monocle and RuleScope.

Features	Monocle	RuleScope	RuleChecker
Rule missing fault	√	√	√
Rule priority fault	×	√	√
No false negatives	×	×	√
Incremental update	×	×	√
Cascaded flow tables	×	×	√

much faster than Monocle and RuleScope.

(2) RuleChecker introduces a new rule dependency model, and uses multiple rounds of probing to eliminate false negatives. Specifically, when a rule is tested to be incorrect, RuleChecker repairs it and re-probes it. Thus, RuleChecker can ensure that when testing a specific rule, all other rules that it depends on have already been tested or repaired to be correct.

(3) RuleChecker only needs to re-compute a minimal number of probes when a new rule is added. Thus, RuleChecker can fast update probes to keep pace with frequent network re-configurations.

(4) RuleChecker generates a probe for each rule of each of cascaded flow table, based on a model named rule path. Thus RuleChecker can test rule missing faults in cascaded flow tables, and potentially localize the missing rules.

Table I gives a comparison of RuleChecker and the other two related tools. In sum, our contribution is three-fold:

- We propose RuleChecker, a new data plane testing tool which can generate probes much faster than previous approaches.
- We design a series of algorithms to make RuleChecker sound and complete, in the sense that it has no false negatives, supports incremental update, and can test cascaded flow tables.
- We prototype RuleChecker as a proxy between the controller and switches, and test it using both real and synthetic rule sets.

The rest of this paper proceeds as follows. Section II states problem, *i.e.*, rule faults at SDN data plane; Section III presents the baseline version of RuleChecker which is fast but has basic features, and Section IV extends it to a complete version; Section V gives the implementation of RuleChecker, and evaluates its function and performance; Section VI discusses related work, and Section VII concludes.

II. Problem Statement

This section first gives a short preliminary to SDN, and then introduces the rule faults at SDN data plane.

A. Preliminary to SDN

We consider a typical Software Defined Network (SDN) where one controller controls a set of switches, through a standard configuration protocol like OpenFlow [15]. An operator specifies her policy, like "host A should reach host B, passing firewall C", in a high-level language like Pyretic [16]. The controller then complies the policies into a set of rules, and installs these rules to flow tables of the corresponding switches. The switches are expected to forward packets according to the rules in their flow tables.

We assume a rule takes a match-action form as in OpenFlow. Specifically, a rule r is a 3-tuple $\langle p, m, a\rangle$, where $r.p$ defines the *priority* of r (larger number means higher priority), $r.m$ is the *matching fields* of r, and $r.a$ is the *action* of r [1]. Here, the matching fields include the port from which a packet is received, and header fields like TCP five-tuple. All these fields can have wildcard bits "$*$". The actions include outputting the packet to a port, dropping the packet, or rewriting some header fields.

We assume each switch has one or multiple flow tables, which are numbered sequentially. A packet will first be matched against rules in the first flow table, and then other tables, depending on the actions of the matched rule. Within a single flow table, a packet is matched against rules according to their priorities. When a packet is matched by one rule, the action of the rule is executed, and the packet can be directed to other flow tables for further matching.

B. Rule Faults at SDN Data Plane

In this paper, we consider two typical rule faults at the SDN data plane, *i.e.*, *rule missing fault*, and *rule priority fault*.

Rule missing fault. We say a rule r is experiencing a missing fault, if all packets whose headers are within the r's matching fields $r.m$ will not match r. Rule missing can take two forms: (1) the controller sends a rule installation message to the switch, but the switch fails to install the rules to its flow table, or (2) the rule that previously exists in a switch's flow table disappears without being noticed by the controller. In the following, we show possible reasons for the above two forms of rule missing.

A possible reason for (1) is that the switch software may contain bugs such that it fails to properly process the rule installation messages. Indeed, an OpenFlow controller can use `Barrier` messages to let switches acknowledge the flow installations. However, recent studies [1], [2] showed that some production SDN switches respond to `Barrier` messages even before the rules are actually installed. This means `Barrier` messages can be a bad indicator for the completion of rule installation.

As for (2), it is possible that a switch deletes a rule from its flow table due to table overflow, without reporting to the controller. Also, an attacker can compromise a switch, and tamper with the flow rules [7]–[9]. For example, a recent study showed that ONIE [17], the boot loader for many 3rd party switch OSes, is vulnerable to attacks, and by compromising ONIE, an attacker can gain persistent control over SDN switches [9].

Note that link failures can be regarded as a special case of rule missing fault. When a link fails, all rules that forward the packet to the corresponding port can be thought to be missing.

[1]Each rule may also be associated with a set of counters, while we do not include them in our model as they do not affect our method.

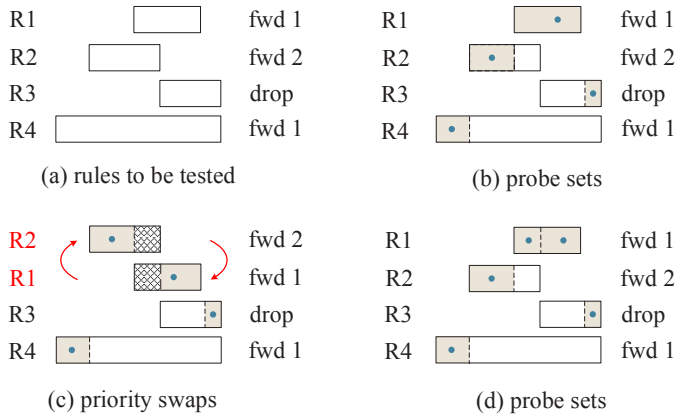


Fig. 1. An example of probe generation. (a) shows the rules to be tested; and (b) shows the probes generated that can detect missing rules; (c) shows a case of rule priority swap; (d) shows the probes generated that can detect rule priority swaps.

Rule priority fault. We say a pair of rule r_i and r_j are experiencing a priority fault, if their priorities are swapped. Priority faults can happen if the switch totally ignores the priority fields of rules. For example, it is reported that the HP ProCurve switch lacks support for rule priority [2]. In addition, priority fault can also happen due to software bugs. For example, it is reported that the Pronto-Pica8 3290 switch running PicOS 2.1.3 caches rules without respecting rule priorities [5]. Specifically, when the number of rules exceeds the size of hardware flow table, the PicOS will place all the following rules in the software flow table. Thus, it may happen that rules with higher priorities are placed in the software flow table, and packets can falsely match some lower-priority rules in the hardware flow table.

Assumption. Apart from the above two faults, there are some other faults, including matching field fault (*e.g.*, some bits of a rule's matching fields are flipped), and action fault (*e.g.*, the output port changes from port 1 to port 2). In this paper, we do not consider these faults. However, our probe-based method can potentially detect these faults without guarantee. In addition, we assume the rules to be tested should be active, meaning that there should be some packets that can match the rule.

III. RuleChecker: the Baseline Construction

This section presents the baseline construction of RuleChecker, which can detect both rule missing and priority faults. We will first present an overview, and give details on the three steps of this method.

A. Overview

Detecting rule missing fault. According Monocle, a probe that can detect the missing fault of r_i should satisfy:

1) The probe should match rule r_i, but no any other rule whose priority is higher than r_i.
2) Let r_j be the highest-priority rule satisfying $r_j.p < r_i.p$ and the probe matches r_j, then we should have $r_i.a \neq r_j.a$.

The first condition ensures that the probe will take action $r_i.a$, if r_i is not missing, and the second condition ensures that the probe will take a different action $r_j.a$, if r_i is missing. Thus, we can detect whether r_i is missing by observing the action on the probe (*e.g.*, which port the probe is output to).

Take Figure 1(a) for example, which consists of four rules, the probe generation results are shown in Figure 1(b). The shaded areas represent the probe sets, and the dots represent the sampled probes. To see why the probes work, consider $R2$ is missing. Then, $R2$'s probe will match $R4$, which will forward the probe to port 1, disagreeing with $R2$'s action (forward to port 2), and thus can be detected.

Different from Monocle and RuleScope, which generate probes by solving Boolean Satisfiability (SAT) problems, which are NP, we formulate the problem as an iteration of simple set operations:

$$\begin{aligned} &r_i.h \leftarrow r_i.m \cap \overline{\cup_{r_j.p>r_i.p} r_j.m} \\ &r_i.t \leftarrow \{override(r_i, r_j) | r_j.a \neq r_i.a,\ r_j.p < r_i.p\} \\ &override(r_i, r_j) \triangleq r_i.h \cap \overline{\cup_{r_j.p<r_k.p<r_i.p} r_k.m} \cap r_j.m \end{aligned} \quad (1)$$

Here, $r_i.h$ is a subset of $r_i.m$, which is termed as the *hitting fields* of r_i. As its name implies, $r_i.h$ is the set of packet headers that can be actually "hit" rule r_i. while $r_i.m - r_i.h$ are packet headers that will "hit" other rules with higher priorities. $override(r_i, r_j)$ encodes the set of packet headers, for which r_i and r_j are the highest-priority and the second highest-priority rules matching them, respectively. That is, r_i "overrides" r_j at headers $override(r_i, r_j)$, and if r_i is missing, these headers will match r_j instead. Each set $p \in r_i.t$ is termed as a *probe set* of rule r_i. To generate a probe for rule r_i, we can sample a probe from *any* of its probe sets in $r_i.t$.

One key feature of the above probe generation process is that the intermediate results when generating probes can be re-used for later probe generation. Thus, all probe sets can be generated with a single sweep of the flow table. This feature will manifest itself in Algorithm 1.

Detecting rule priority fault. The probes generated for detecting rule missing faults may not detect priority faults. Let us return to the previous example, and consider the priorities of $R1$ and $R2$ are swapped, as shown in Figure 1(c). We can see that all the four probes will pass the test, however, packets with headers belonging to the gridded part will be wrongly forwarded to port 2, instead of port 1. To detect such a fault, we also need to generate a probe for $override(R1, R2)$.

Generally, to detect rule priority faults, for each rule r_i, we need to sample a probe from *each* of its probe sets in $r_i.t$. In addition, the definition of $override$ in Eqs. (1) should be changed to:

$$override(r_i, r_j) \triangleq r_i.h \cap r_j.m \quad (2)$$

That is, the new definition only requires the hitting field of r_i and the matching field of r_j overlap, thereby relaxing the previous definition in Eqs. (1). Thus, to detection priority faults, RuleChecker needs to generate more probes. The following theorem states that with this new definition of $override(r_i, r_j)$, RuleChecker can detect priority faults.

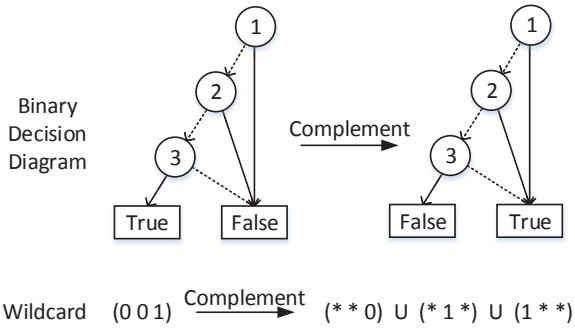


Fig. 2. Comparison of wildcard and Binary Decision Diagram (BDD) on performing set complement. In the BDD, the dashed and solid line means "0" and "1", respectively.

Theorem 1. Suppose the priorities of two rules r_i and r_j are swapped, and there exists some packet experiencing a fault, *i.e.*, falsely matching a rule whose action is different from the correct one, then RuleChecker can detect this fault.

Proof: Please refer to our technical report [18]. ■

B. Step 1: Encoding Header Fields

Before generating probes using Eqs. (1), we need a method to encode header fields, *i.e.*, $r_i.m$, $r_i.h$, $override(r_i, r_j)$. Here, we consider the following two options.

Wildcard is a good choice for representing IP prefix or suffix, and is adopted by HSA [19]. However, it is not efficient for set operations, especially set subtractions. For example, a simple subtraction $10.0.*.* \backslash 10.0.0.1$ will produce 16 wildcards. Due to this reason, HSA is relatively slow in computing packet headers that are reachable between two switches. For RuleChecker, generating probes involves intersections ($A \cap B$) and subtractions ($A \cap \bar{B}$), according to Eqs. (1). Thus, the number of wildcards will grow exponentially when generating probes.

Binary Decision Diagram (BDD [20]) is an efficient data structure for encoding Boolean expressions. Any Boolean expression can be canonically and concisely encoded with a Reduced Ordered BDD (ROBDD), a variant of BDD (we will refer to ROBDD simply as BDD in the rest of paper). A key advantage of BDD over wildcard is its efficient support for logical operations. Therefore, BDD has been used by many previous works to encode OpenFlow rules [21]–[23]. Especially, Yang and Lam [23] show that when the matching fields of a rule are expressed in prefix, suffix, and intervals, the number of nodes in the BDD encoding the rule is $2 + 2h$, where h is the number of bits in the matching fields.

Figure 2 gives an example for comparison of BDD and wildcard. We can see that using BDD, computing the complement of a set only needs flipping the two leaf nodes, while using wildcards, three wildcards will be generated. Based on the above discussion, we decide to use BDD instead of wildcard for encoding header fields.

C. Step 2: Generating Probe Sets

Algorithm 1 summarizes the probe generation process. Note here the set operations in Eqs. (1) naturally transform to logical operations, after we encode header fields with BDDs. The algorithm first sorts the rules in the order of decreasing priority (Line 1), and initializes the set of unmatched headers, denoted as H_a, as the whole set, *e.g.*, the logical "true" (Line 2). Line 3-14 generates probe sets for each rule r_i. In each loop, the algorithm first calculates the hitting fields $r_i.h$ and updates H_a (Line 4). If $r_i.h$ is not empty, then the algorithm initializes H_b as $r_i.h$, and calculates the overriding fields $override(r_i, r_j)$ for each r_j that has a lower priority than r_i (Line 5-8). If $override(r_i, r_j)$ is not empty and the actions of r_i and r_j are different, then it will be put as a probe set into $r_i.t$, and $override(r_i, r_j)$ is subtracted from H_b (Line 9-12). Note here when considering priority fault, $override(r_i, r_j)$ should not be subtracted. Finally, if there are still remaining headers in H_b (meaning that H_b will match the default rule), then H_b will be added to $r_i.t$ (Line 13-14).

Algorithm 1: `GenerateProbe`(R)

Input: $R = \{r_i,\ 1 \le i \le n\}$: the flow table be tested. For each r_i, $r_i.m$ and $r_i.a$ are the match and action field, respectively.

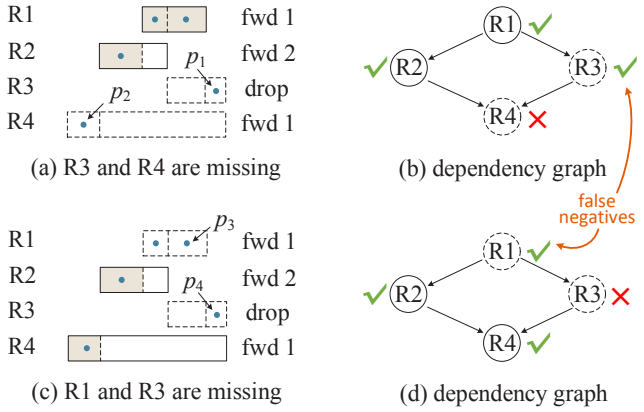
Output: $r_i.t,\ \forall r_i \in R$: the collection of probe sets for r_i ($r_i.t$ is initialized to $\emptyset$).

```
1  sort the rule set R in decreasing priority order;
2  H_a ← true;// headers not matched yet
3  foreach 1 ≤ i ≤ n do
4      r_i.h ← r_i.m ∧ H_a; H_a ← H_a − r_i.m; // matching r_i
5      if r_i.h ≠ false then
6          H_b ← r_i.h;// headers not overrided yet
7          foreach i < j ≤ n do
8              override(r_i, r_j) ← H_b ∧ r_j.m;
9              if override(r_i, r_j) ≠ false then
10                 if r_i.a ≠ r_j.a then
11                     r_i.t ← r_i.t ∪ {override(r_i, r_j)};
12                 H_b ← H_b − override(r_i, r_j);
           // matching the default rule
13         if H_b ≠ false then
14             r_i.t ← r_i.t ∪ {H_b};
```

D. Step 3: Sampling Probes

For each rule r_i, we iterate over all its probe sets in $r_i.t$, and for each probe set p, we randomly sample one probe from p. Since the probe set is represented by BDD, then we only need to find a truth assignment for the BDD, which we refer to as the AnySAT problem.

AnySAT problem is very efficient for BDD: one only needs to find a path from the root node leading to the `True` node. The running time is $\boldsymbol{O}(n)$, where n is the number of variables encoded by the BDD. If the BDD encodes a set of IPv4 addresses, then the number of variables is 32. That is, AnySAT has a linear complexity, compared with Boolean Satisfiability (SAT), which is NP-complete. This makes our RuleChecker much faster than Monocle [13] and RuleScope [14], which are both based on solving SAT.



(a) R3 and R4 are missing (b) dependency graph (c) R1 and R3 are missing (d) dependency graph

Fig. 3. An example of multiple missing rules (a)(c), and the corresponding dependency graphs (b)(d). The rules with dotted lines are missing, and the ×/✓ symbol means the probe passes/fails, respectively.

IV. RuleChecker: the Complete Construction

In the baseline construction, we showed how RuleChecker can generate probes that can detect rule missing and priority faults. However, like Monocle and RuleScope, the baseline RuleChecker still misses some important parts: (1) it may raise false negatives when there are multiple missing rules; (2) it cannot incrementally update probes when a new rule is added; (3) it does not support cascaded flow tables. This section presents a complete construction of RuleChecker, which fill the above missing parts.

A. Eliminating False negatives

The baseline RuleChecker, as well as Monocle and RuleScope, may generate false negatives when there are multiple missing rules. To see why, let us return to the example in Figure 3(a), where $R3$ and $R4$ are missing simultaneously. In this case, R_4's probe p_2 will match the default drop rule and fail the test. However, $R3$'s probe p_1 will also match the default drop rule and pass the test, resulting in a false negative. A similar case is that when $R1$ and $R3$ are missing simultaneously, as shown in Figure 3(c). In this case, $R1$'s probe p_3 will result in a false negative.

The reason for such false negatives is that when generating a probe for r_i, we assume that the existence of the lower-priority rule r_j that r_i overrides, such that when r_i is missing, the probe will hit r_j instead. That is, the validity of r_i's probe depends on the existence of r_j, and when r_j is missing, the testing result of r_i is not valid.

Rule correctness criterion. To eliminate false negatives when multiple rules are missing, we should use a new criterion for rule correctness. Before that, we first define rule dependency as follows.

Definition 1. A rule r_i is said to depend on another rule r_j, if $\exists x \in r_i.t$ such that r_j is the highest-priority rule that intersects with x. Formally, r_i depends on r_j if and only if:

- $r_j.p < r_i.p$, and
- $\exists x \in r_i.t$ such that: (i) $x \wedge r_j.m \neq \emptyset$, and (ii) $\forall r_k$, if $r_j.p < r_k.p < r_i.p$, then $x \wedge r_k.m = \emptyset$.

As shown in Figure 3(b), $R1$ depends on $R3$, and $R3$ depends on $R4$. There are two points to note here. (1) If the lower-priority rule r_j is untestable ($r_j.t = \emptyset$), we will not create the dependency relationship between r_i and r_j. (2) Dependency relationship is not transitive. As shown in Figure 3(b), we have $R1$ depends on $R3$, $R3$ depends on $R4$, but $R1$ does not depend on $R4$. Based on dependency relationship, we define the rule correctness criterion as follows.

Definition 2. A rule r_i is said to be *correct* if the probe for r_i passes the test, and either (1) r_i depends on no other rules, or (2) there exists a rule r_j that r_i depends on, and r_j has been tested to be correct.

To illustrate the above correctness criterion, let us return to Figure 3 (b), where the probe for $R4$ fails the test. Even probes for $R2$ and $R3$ pass the test, we should not mark them as correct. Similarly, since $R1$ depends on $R2$ and $R3$, both of which are not marked correct, we should not mark $R1$ as correct, either.

Dependency-aware rule probing and repairing. To eliminate false negatives due to multiple missing rules, we use multiple rounds of rule probing, and when some probes fail, we repair their corresponding rules, so as to meet the above new criterion. There are two options here, *i.e.*, the conservative approach and the aggressive approach:

(1) The conservative approach always sends probes for rules that depend on no other rules. If these probes pass the test, then we remove the rules, together with their corresponding edges, from the dependency graph. Then, we continue to send probes for rules that depend on other rules. Otherwise, if some probes fail, we repair those rules (by re-installing them) and re-send the failed probes until all probes pass the test. Take Figure 3 (a) for example. In the 1st round, only the probe for $R4$ is sent, which fails, and $R4$ is re-installed. In the 2nd round, the probe for $R4$ is sent again and passes. In the 3rd round, the probes for $R2$ and $R3$ are sent, of which the probe for $R3$ fails, and $R3$ is re-installed. In the 4th round, the probe for $R3$ is sent again, which passes. In the 5th round, the probe for $R1$ is sent, which passes. Thus, it takes 5 rounds and 6 probes in total.

(2) The aggressive approach simply sends all probes without waiting, and see if there are failed probes. Since there are no false positives, we only need to repair the rules whose probes fail, and re-send all probes again. This process continues until there are no failed probes. Let us return to the case in Figure 3 (a). In the 1st round, the probe for $R4$ fails, and $R4$ is re-installed. In the 2nd round, the probe for $R3$ fails, and $R3$ is re-installed. In the 3rd round, there are no failed probes, and the testing finishes. Thus, it takes 3 rounds and 12 probes in total.

From above, we can see that the conservative approach takes more rounds, but injects less probes. In contrast, the aggressive approach needs less rounds, but injects more probes. In our experiment, we find 3 rounds are mostly sufficient for the aggressive approach to test a flow table with multiple missing rules (see Section V-C).

Algorithm 2: `UpdateProbes`(r)

Input: r: the newly added rule; R: the set of all rules.

1 $R_h \leftarrow \{r_i \in R \mid r_i.m \wedge r.m \neq false,\ r_i.p > r.p\}$;
2 $R_l \leftarrow \{r_i \in R \mid r_i.m \wedge r.m \neq false,\ r_i.p < r.p\}$;
3 sort R_h and R_l in decreasing priority order;
4 **foreach** $r_i \in R_h$ **do**
5 **if** $r_i.probe \neq false$ *and* $r_i.override.p < r.p$ **then**
6 **if** $r_i.probe \wedge r.m \neq false$ **then**
7 $r_i.probe \leftarrow r_i.probe - r.m$;
8 **if** $r_i.probe = false$ **then**
9 generate a probe for r_i;
10 **else if** $r_i.probe = false$ *and* $r_i.a \neq r.a$ **then**
11 generate a probe for r_i;
12 **foreach** $r_i \in R_l$ **do**
13 **if** $r_i.probe \neq false$ *and* $r_i.h \wedge r.m \neq false$ **then**
14 $r_i.probe \leftarrow r_i.probe - r.m$;
15 **if** $r_i.probe = false$ **then**
16 generate a probe for r_i;
17 generate a probe for r;

B. Incremental Probe Update

Algorithm 2 summarizes the process of updating probes when a new rule is added into the flow table. For simplicity, we consider there are no rule swaps, and thus there is at most one probe set for each rule.

First, only rules that have overlapping matching fields with the newly added rule r may affect the probe generation for r, and their probes may also be affected by r. Thus, based on r, we class rules in the flow table into two groups:

- R_h: the rules whose matching fields overlap with r, and whose priority is higher than r.
- R_l: the rules whose matching fields overlap with r, and whose priority is lower than r.

For each rule $r_i \in R_h$, if it has a probe set $r_i.probe$, then $r_i.probe$ will be affected only if the rule that r_i overrides (denoted by $r_i.override$) has lower priority than r (Line 5). In this case, the overlapping fields of r_j and r are subtracted from the probe of r_i (Line 6-7). If the probe for r_i becomes empty, we re-generate probe for r_i (Line 8-9). If r_i does not have probe set yet, and r has different action as r_i, then we try to generate probe for r_i (Line 10-11).

For each rule $r_j \in R_l$, if r_j does not have probe set, it will not have probe set after r is added. In addition, even r_j has probe set, it will not be affected if the hitting fields does not overlap with the matching fields of r. Apart from the above two cases, the probe for r_i will be updated by subtracting the overlapping fields of the probe and r (Line 13-14). If the collection of probe sets becomes empty, then we try to generate a probe set for r_i (Line 15-16). Finally, we generate the probe set for r itself (Line 17).

C. Testing Cascaded Flow Tables

We use Figure 4 as an example to illustrate our method to test cascaded flow tables. In this example, there are two flow tables: Table 1 and Table 2. We assume that packets will match Table 1 and Table 2 in sequence. Each flow table has three rules, numbered in the order of decreasing priority, and at the last is a default rule that catches all unmatched headers.

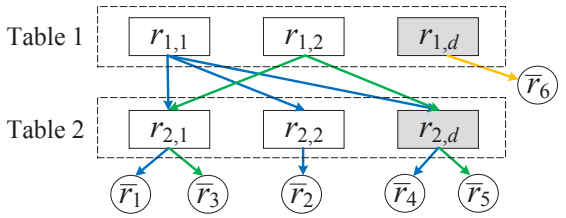


Fig. 4. An example of cascaded flow tables. The shaded are the default rules.

First, define *rule path* as an ordered list of rules which can be matched by at least one packet header in sequence. In this example, there are 6 rule paths, *e.g.*, $r_{1,1} \rightarrow r_{2,1}$, $r_{1,1} \rightarrow r_{2,2}$, etc. For each rule path, we attach the last rule of the path with an extra rule, termed *leaf rule*, to represent it. In the following, we will use leaf rules to refer to their corresponding rule paths. As shown in Figure 4, $\bar{r}_1$ through $\bar{r}_6$ are the leaf rules.

Here, leaf rules and rule paths can be constructed using an approach similar to FlowAdapter [24]. If the actions of rules do not contain "set-field" operations, the matching field of a leaf node $\bar{r}_i$ is simply an intersection of matching fields of rules along its rule path. In this example, the matching field for $\bar{r}_1$ is $\bar{r}_1.m = r_{1,1}.m \wedge r_{2,1}.m$. On the other hand, if a rule $r_{i,j}$ sets a field x, then we will not intersect the field x with of all the following rules. For example, if $r_{2,1}$ sets src_ip, then $\bar{r}_1.m = r_{1,1}.m \wedge r'_{2,1}.m$, where $r'_{2,1}.m$ is $r_{2,1}.m$ with src_ip set to wildcard.

For each rule $r_{i,j}$, let $Leaf(r_{i,j})$ denote the set of leaf rules, whose rule paths traverse $r_{i,j}$. Suppose $\bar{r}_k \in Leaf(r_{i,j})$, define $Parent(r_{i,j}, \bar{r}_k)$ as the parent of a rule $r_{i,j}$ along the rule path $\bar{r}_k$, which is the rule matched right before $r_{i,j}$ is matched along path $\bar{r}_k$. For simplicity of notations, we create a virtual rule $r_{0,0}$ as the parent of all rules that are the first rules of some paths. In this example, $Parent(r_{1,1}, \bar{r}_1) = r_{0,0}$; $Parent(r_{2,1}, \bar{r}_1) = r_{1,1}$ and $Parent(r_{2,1}, \bar{r}_3) = r_{1,2}$. Define $Relative(r_{i,j}, \bar{r}_k)$ as the set of leaf rules of $Parent(r_{i,j}, \bar{r}_k)$ but not leaf rules of $r_{i,j}$, *i.e.*,:

$$Relative(r_{i,j}, \bar{r}_k) \triangleq Leaf(Parent(r_{i,j}, \bar{r}_k)) \backslash Leaf(r_{i,j})$$

The intuitive meaning of $Relative(r_{i,j}, \bar{r}_k)$ is: when $r_{i,j}$ is missing, then the headers originally traversing path $\bar{r}_k$ will then possibly traverse paths in $Relative(r_{i,j}, \bar{r}_k)$. For example, $Relative(r_{1,1}, \bar{r}_1) = \{\bar{r}_3, \bar{r}_5, \bar{r}_6\}$, meaning that $r_{1,1}$ is missing, the headers that originally match $\bar{r}_1$ can possibly match $\bar{r}_3$, $\bar{r}_5$, or $\bar{r}_6$. For $\bar{r}_2$ and $\bar{r}_4$, the headers that originally match $\bar{r}_1$ cannot match them either, since their rule paths contain $r_{1,1}$.

Algorithm 3 summarizes the process of probe generation. The algorithm first generates the set of leaf rules $\bar{R}$ (Line 1-3), and calculates their matching fields and hitting fields (Line 4-7). Note that in the intersection performed at Line 6, $r_{i,j}.m$ should wildcard those fields that have been set by a previous rule on the rule path. Then, the algorithm generates a probe

Algorithm 3: `GenerateProbeCascade`$(R_1, \ldots, R_m)$

Input: $R_i = \{r_{i,j},\ 1 \le j \le n_i\}$: the ith flow table. For each $r_{i,j}$, $r_{i,j}.m$ and $r_{i,j}.a$ are the match and action field, respectively.

Output: $r_{i,j}.t,\ \forall r_{i,j}$: the collection of probe sets for $r_{i,j}$ ($r_{i,j}.t$ is initialized to $\emptyset$).

```
1  construct all rule paths from cascaded flow tables R_1, ..., R_m;
2  extract the set R̄ of leaf nodes from the N-tree, with |R̄| = N;
3  sort R̄ in decreasing priority order;
4  H_a ← true;// headers not matched
5  foreach 1 ≤ k ≤ N do
6      r̄_k.m ← ⋀_{r_{i,j} ∈ Path(r̄_k)} r_{i,j}.m; // get r̄_k.m
7      r̄_k.h ← r̄_k.m ∧ H_a; H_a ← H_a − r̄_k.m; // get r̄_k.h
8  foreach r_{i,j} ∈ R_i, 1 ≤ i ≤ m do
9      foreach r̄_k ∈ Leaf(r_{i,j}) do
10         if r̄_k.h ≠ false then
11             H_b ← r̄_k.h;// headers not overrided
12             foreach k < l ≤ N, r̄_l ∈ Relative(r_{i,j}, r̄_k) do
13                 override(r̄_k, r̄_l) ← H_b ∧ r̄_l.m;
14                 if override(r̄_k, r̄_l) ≠ false then
15                     if r̄_k.a ≠ r̄_l.a then
16                         r_{i,j}.t ← r_{i,j}.t ∪ {override(r̄_k, r̄_l)};
17                     H_b ← H_b − override(r̄_k, r̄_l);
18             if H_b ≠ false then
19                 r_{i,j}.t ← r_{i,j}.t ∪ {H_b};
```

set for each rule of each flow table (Line 8). For each rule $r_{i,j}$, the algorithm selects one of its leaf rules $\bar{r}_k$ (Line 9), and tries to find another rule $\bar{r}_l$ satisfying $\bar{r}_k$ overrides $\bar{r}_l$ and $\bar{r}_k.a \neq \bar{r}_l.a$ (Line 10-19). The process of finding such $\bar{r}_l$ is mostly the same with that of Algorithm 1 (Line 5-14). The difference is that $\bar{r}_l$ is chosen from $Relative(r_{i,j}, \bar{r}_k)$, rather than all rules that have priorities lower than $\bar{r}_k$, for reasons that have been explained above.

Limitations. Currently, RuleChecker doesn't support some OpenFlow features, *e.g.*, group tables. In addition, RuleChecker cannot detect if a backup rule is missing. We leave them as our future work.

V. Implementation and Evaluation

In this section, we first present the implementation of RuleChecker, and test its functions, *i.e.*, whether it can detect missing faults, priority faults, eliminate false negatives, and test cascaded flow tables. Then, we micro-benchmark the efficiency of our probe generation and update algorithms, respectively, and compare the results with Monocle and RuleScope.

A. Implementation

Our implementation of RuleChecker consists of around 7K lines of Java codes. First, we build a proxy between the controller and switches based on Netty, an asynchronous network I/O library [25]. Inside the proxy, we use the packet parsing library of Floodlight [26] to parse the OpenFlow packets. Then, we augment the proxy with the RuleChecker modules, including probe generation/update, probe injection/collection, and rule repairing. For BDD related operations, we use JDD, a Java BDD library [27].

For probe injection, let S be the switch under test, we let the controller send a `PacketOut` message encapsulating the probe to a switch S_u that is adjacent S. The choice of S_u is arbitrary, unless the probe specifies it should be injected to a specific port of S. For probe collection, we let the controller install two rules at Table 0 of each switch adjacent to S: the first is a high-priority rule that catches all probes, by matching an unused field reserved for RuleChecker, with an action of sending a `PacketIn` message to the controller; the second is a low-priority rule that forwards the rest of packets to Table 1 for further matching. Since we use Table 0 to catch probes, original rules will be placed in Table 1, instead.

B. Experiment Setup

In all experiments below, we use the topology shown in Figure 5. The controller (Floodlight [26]) and RuleChecker run on a server with a 3.1GHz dual-core Intel i5 CPU and 16GB memory. We emulate a set of software switches (Open vSwitches [28]) using Mininet [29] on another server, which has two 2.0GHz 6-core Intel E5 CPUs and 32GB memory. The switches form a star topology, where the switch to be monitored has n ports, each connected to a switch. RuleChecker acts as a proxy between the controller and switches. We let the controller install flow rules into the switch's flow table, and use RuleChecker to monitor faults in the switch' flow table.

We use both real and synthetic rules for experiments.

- Real rules. We generate the real rules by parsing the configuration files of *yoza*, one of the 16 Stanford backbone routers [19]. There are 2755 OpenFlow rules with priorities, and the matching fields include input port, TCP five-tuple, and VLAN-ID. This data set was first parsed and used by Monocle.
- Synthetic rules. We generate the synthetic rules using ClassBench [30], a public-available packet filter generator. For fair comparison, the parameters used to generate the rules are kept the same with those in RuleScope.

C. Functional Test

Detecting rule missing faults. We let the controller install 1000 rules into the flow table of a switch. The rules are chosen from the 2755 Stanford rules. Then, we randomly delete one rule from the switch's flow table, and measure the time for RuleChecker to detect the fault. Specifically, RuleChecker sends probes in sequence at a speed of 500 probes per second, and if a probe fails to return within a specific period of 0.5 seconds, RuleChecker will re-send the probe again. Each probe will be sent at most three times.

The time it takes RuleChecker to detect a single missing rule is shown as the solid blue line in Figure 6. The detection time is less than 2 seconds for most of the cases. Since we send probes from the first one in sequence, the detection time depends on where the missing rule is, and thus the line is basically linear.

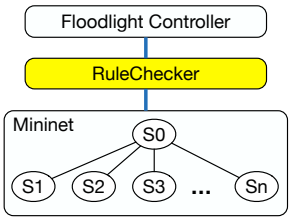


Fig. 5. The network topology used for experiments.

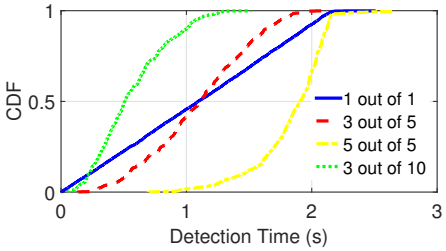


Fig. 6. The time for RuleChecker to detect missing rules.

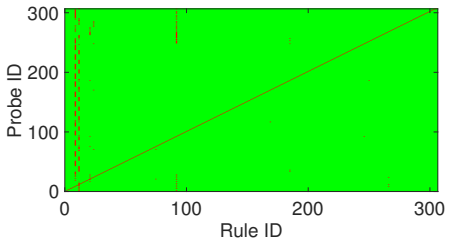


Fig. 7. The probing results of testing a cascaded flow table.

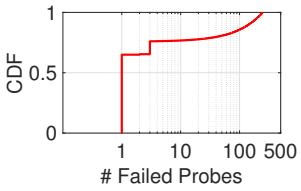


Fig. 8. The number of faults detected when priorities of two rules are swapped.

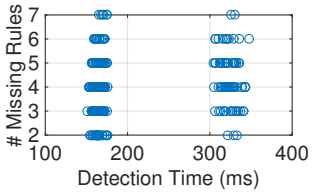


Fig. 9. The time to detect multiple missing rules with RuleChecker.

We continue to show how RuleChecker works when multiple rules are missing. We randomly delete N rules from the switch's flow table, and measure the time that it takes RuleChecker to detect a threshold number of M missing rules. As shown in Figure 6, it takes less time for RuleChecker when there are more missing rules, and when the threshold is lower. The shapes of the lines are agreeing with the results in Monocle.

Testing cascaded flow tables. For cascaded flow tables, we use the configuration files of *yoza*, a router from the Stanford network. The configuration files consists of ACL rules that filter input packets, IP forwarding rules, and ACL rules that filter output packets. These three types of rules are natural fit for three cascaded flow tables. Thus, instead of parsing the configuration files into the 2755 rules, we populate them into three flow tables, In-ACL, FWD, and Out-ACL, which consists of 8, 247, and 142 rules, respectively.

RuleChecker is able to generate probes for 306 out of the 397 rules. For each such rule, we delete it and send all the probes, and check whether these probes pass the test. Figure 7 reports the test results, where the green dots represent probes that pass the test, while red grids represent probes that fail the test. We can see that for each rule, if it is deleted, its probe will definitely fail the test. However, it is also possible that other probes also fail. This brings problems for us to pinpoint which rules are missing. We leave the localization of missing rules in cascaded flow tables as future work.

Detecting rule priority faults. We continue to test whether RuleChecker can detect priority faults. To simulate priority faults, we select a subset of Stanford rules, randomly select two rules from it, and swap their priority values. Then, we check whether the flow table remains equivalent after the swap. By equivalent, we mean that packet forwarding behaviors remain unchanged. Then, if the flow table becomes inequivalent, we use RuleChecker to check the flow table. We report the number of failed probes in Figure 8. We can see that RuleChecker can always find more than one failed probe after priority faults.

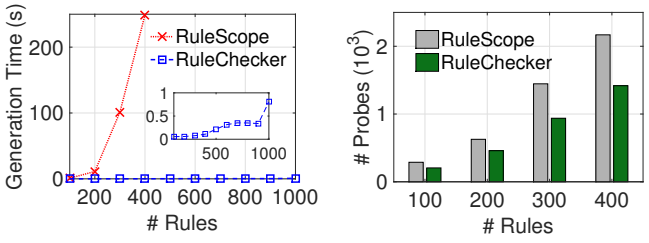
TABLE II
The probe generation time and number of generated probes for Monocle, RuleScope, and RuleChecker. The flow table consists of 2755 rules that are parsed from the Stanford yoza router configurations. RuleChecker* stands for RuleChecker without considering priority faults.

	RuleScope	Monocle	RuleChecker*	RuleChecker
time	15.58 sec	3.61 sec	0.56 sec	0.79 sec
#probes	9234	2442	2442	2742

Detecting correlated missing rules. As shown in Section IV-A, Monocle and RuleScope may generate false negatives when there are multiple missing rules that depend on one another. In the experiment, we simulate the cases where multiple rules that depend on one another are missing, and show how RuleChecker can detect all the missing rules using the aggressive approach presented in Section IV-A. We choose 8 highly correlated rules from the Stanford rule set, and delete N rules of them. We measure the time that it takes RuleChecker to detect all missing rules (the last round of rule repairing and probing is not counted). Figure 9 reports the time to detect all the missing rules for different N. We can see that the detection time falls into two classes: those in-between 150 ms and 180 ms, and those in-between 300 ms and 350 ms. The shorter ones correspond to cases that all missing rules are detected after sending all the probes once, thus only one round of rule probing is needed (around 150 ms). The longer ones correspond to cases that after one round of probing, RuleChecker repairs the missing rules, sends the probes again, and still detects missing rules. In this case, two rounds of rule probing and one round of rule repairing (around 50 ms) are needed.

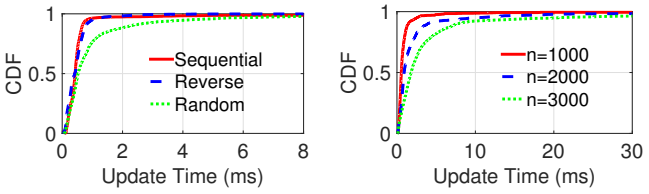
D. Probe Generation Efficiency

First, we micro-benchmark the probe generation speed of RuleChecker on the Stanford rule set, and compare it with those of RuleScope and Monocle. For benchmark, we run RuleScope and RuleChecker on a server with a 3.6GHz Intel i7 CPU. Since we do not have the codes of Monocle, we directly use the results reported in their paper, where a 2.93GHz Intel Xeon CPU is used (readers can scale the comparison results themselves). Table II reports the results, where we can see that without considering rule priority fault, RuleChecker



(a) generation time (b) # probes

Fig. 10. Comparison of RuleScope and RuleChecker on the time to generate all probes and the number of generated probes.



(a) Stanford (b) ClassBench (random)

Fig. 11. Distribution of probe update time. For the Stanford rule set (a), rules are inserted into the flow table in sequential, reverse, and random order, respectively. For the ClassBench rule set (b), rules are inserted into the flow table in random order.

(marked with *) is around $5\times$ faster than Monocle after scaling. Considering priority fault, RuleChecker is nearly $20\times$ faster than RuleScope.

The number of probes generated by RuleScope is much larger than RuleChecker and Monocle. The reason is that RuleScope generates a probe for each pair of rules whose matching fields overlap. In contrast, both RuleChecker and Monocle generate a probe only when a higher-priority rule "overrides" a lower-priority rule (see Section III). RuleChecker generates slightly more probes than Monocle, since RuleChecker can detect both priority faults and missing faults, while Monocle only detects missing faults.

We then evaluate the rule generation speed with synthetic rules generated with ClassBench [30]. The generated flow table consists of 100 to 1000 rules. Since we do not have the codes of Monocle, we only evaluate the probe generate speed of RuleChecker and RuleScope. For a fair comparison with RuleScope, we intentionally disable the RuleChecker to check whether two rules have different actions.

As shown in Figure 10(a), RuleScope generates probes for less than 200 rules in reasonably short time, while for 400 rules, the time quickly increases to 250 seconds. In contrast, RuleChecker can generate probes for 1000 rules within 1 second. As shown in Figure 10(b), RuleChecker needs less probes than RuleScope. Specifically, RuleChecker reduces the number of probes by roughly 35% when the flow table consists of 400 rules. We stop at 400 rules as RuleScope fails to generate probes within a reasonable time. This implies that RuleChecker needs to inject, collect, and verify less probes, compared with RuleScope.

E. Probe Update Efficiency

We continue to evaluate the efficiency of incremental probe update. In this experiment, we start from an empty flow table, and insert one rule each time, and measure the time RuleChecker takes to update all probes. The rules are sorted in the order of decreasing priority, and inserted in sequential, reserve, and random order, respectively. Figure 11 reports the results for both the Stanford and ClassBench rule sets.

From Figure 11(a), we can see that for the Stanford rules, RuleChecker uses less than 2ms for more than 90% of all cases, In addition, using random insertion costs more time than using sequential and reverse insertion. This indicates that the order of rule insertion can impact the update time. From Figure 11(b), we can see that the update time increases as the flow table size grows. This is because when the rule dependency will be more complicated for larger flow tables. Even so, the update time still keeps under 10 ms for more than 90% of all cases.

VI. Related Work

Recently, many tools have been developed to verify, monitor, or test the correctness of computer networks. We classify tools that are related to RuleChecker into three classes.

Data plane verification tools verify whether a network data plane satisfies some key invariants, including reachability, blackhole-freedom, loop-freedom [19], [23], [31]–[34]. Since these tools check the flow tables in a centralized manner, *i.e.*, at the controller, they may only ensure correctness at the controller side, while cannot guarantee the correctness of the data plane forwarding behaviors. As shown in previous work [2], [13], flow tables of switches may deviate from what the control plane thinks, resulting in packet forwarding behaviors that are inconsistent with controller-side rule configurations.

Data plane testing tools directly test the network data plane by sending probe or test packets. ATPG [4] generates the minimum number of test packets that can trigger all rules in the network, and verifies whether all these probe packets can be correctly received by their intended end hosts. Since ATPG only checks packet receptions, it can only verify the basic pairwise reachability, while cannot verify properties like middlebox traversals that require inspections on packet trajectories. Monocle [13] tests whether a rule is in a switch's flow table by sending a probe packet to the switch and checking which port the packet is output to. A probe packet should be constructed in such a way that it can only trigger the rule under test, while not being matching by other rules in the switch. A similar approach, RuleScope [14], also tests the flow table integrity by sending probes. The difference is that it can detect priority faults of rules as well.

As already noted in the Introduction, there are several limitations for both Monocle and RuleScope. First, the probe generation process is slow: Monocle costs around 43 seconds to generate probes for 10K real rules; RuleScope uses more than 300 seconds to generate probes for 320 synthetic rules. Second, they implicitly assume there is at most one missing rule, and thus may generate false positives when there are multiple (correlated) missing rules. In addition, they do not support incremental update, and cannot test cascaded flow tables.

Data plane monitoring tools. Different from data plane testing tools, data plane monitoring tools sample real traffic from the data plane, and verify whether their forwarding behaviors are agreeing with the control plane policies. VeriDP [10], [11] let switches imprint the forwarding behaviors into packet tags, and report the tags to the controller for verification. The controller verifies whether the tags carried by packets are the same with what the controller computes itself according to the network policies. *e.g.*, middlebox traversal. REV [12] extends VeriDP to work in adversarial settings, where tags can spoofed by compromised switches. By using message authentication codes (MACs), REV can securely verify whether the rules installed by the controller have been correctly enforced by switches. The common limitation of VeriDP and REV is that they need to modify switches to support tag/MAC generation. In addition, they do not support packet rewrites, where switches need to re-write some header fields.

Switch software debuggers. [35], [36] use symbolic execution to test the software components of SDN switches. They only carry out static testing for switch software codes, while cannot detect flow table faults that only show at runtime. However, we can use them as complementary tools to RuleChecker, in order to prevent data plane faults.

VII. Conclusion

We presented RuleChecker, which can generate probes to actively test the correctness of SDN flow tables. Different from previous probe-based testing tools like Monocle, RuleChecker is extremely fast due to a novel set-based probe generation method. Moreover, RuleChecker fills some important parts missed by previous tools, including elimination of false negatives, incremental probe update, and support of cascaded flow tables. This makes RuleChecker a more complete data plane testing tool for SDN. Our future work includes extending RuleChecker to test group tables and backup rules.

Acknowledgements. The authors would like to thank all the anonymous reviewers, and our shepherd Gabor Retvari for valuable comments and advices. This work is supported by the National Key Research and Development Program of China (No. 2017YFB0801703, 2016YFB0800100), the National Natural Science Foundation of China (No. 61772412, 61402357, 61572278), and the State Grid Corporation of China (DZ71-16-030).

References

[1] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for OpenFlow switch evaluation," in *Proceedings of Passive and Active Measurement*, 2012, pp. 85–95.

[2] P. Peresini, M. Kuzniar, and D. Kostic, "What You Need to Know About SDN Flow Tables," in *Proceedings of Passive and Active Measurement*, 2015.

[3] M. Kuzniar, P. Peresini, and D. Kostić, "Providing reliable FIB update acknowledgments in SDN," in *Proceedings of ACM CoNEXT*, 2014.

[4] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proceedings of ACM CoNEXT*, 2012.

[5] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "CacheFlow: Dependency-aware rule-caching for software-defined networks," in *Proceedings of ACM Symposium on SDN Research*, 2016.

[6] A. Bremler-Barr, D. Hay, D. Hendler, and R. M. Roth, "PEDS: a parallel error detection scheme for TCAM devices," *Proceedings of IEEE/ACM Transactions on Networking*, vol. 18, no. 5, pp. 1665–1675, 2010.

[7] M. Antikainen, T. Aura, and M. Särelä, "Spook in Your Network: Attacking an SDN with a Compromised OpenFlow Switch," in *Proceedings of Nordic Conference on Secure IT Systems*, 2014.

[8] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised sdn switch," in *Proceedings of IEEE NetSoft*, 2015.

[9] G. Pickett, "Staying persistent in software defined networks," in *Black Hat Briefings*, 2015.

[10] P. Zhang, H. Li, C. Hu, L. Hu, and L. Xiong, "Stick to the script: Monitoring the policy compliance of SDN data plane," in *Proceedings of ACM/IEEE ANCS*, 2016.

[11] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the gap: Monitoring the control-data plane consistency," in *Proceedings of ACM CoNEXT*, 2016.

[12] P. Zhang, "Towards rule enforcement verification for software defined networks," in *Proceedings of IEEE INFOCOM*, 2017.

[13] P. Peresini, M. Kuzniar, and D. Kostic, "Monocle: Dynamic, fine-grained data plane monitoring," in *Proceedings of ACM CoNEXT*, 2015.

[14] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect SDN forwarding with RuleScope," in *Proceedings of IEEE INFOCOM*, 2016.

[15] 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.

[16] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN programming with Pyretic," *Technical Reprot of USENIX*, 2013.

[17] "Open Network Install Environment (ONIE)," http://onie.org/.

[18] P. Zhang, C. Zhang, and C. Hu, "Fast testing network data plane with RuleChecker (technical report)," http://nskeylab.xjtu.edu.cn/people/pzhang/files/2017/08/rulechecker-tr.pdf.

[19] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proceedings of USENIX NSDI*, 2012.

[20] H. R. Andersen, "An introduction to binary decision diagrams," *Lecture notes, available online, IT University of Copenhagen*, 1997.

[21] E. Al-Shaer and S. Al-Haj, "Flowchecker: Configuration analysis and verification of federated openflow infrastructures," in *Proceedings of ACM workshop on Assurable and Usable Security Configuration*, 2010.

[22] V. Altukhov, V. Podymov, V. Zakharov, and E. Chemeritskiy, "Vermont-a toolset for checking sdn packet forwarding policies on-line," in *Proceedings of IEEE MoNeTeC*, 2014.

[23] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 887–900, 2016.

[24] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie, "The flowadapter: Enable flexible multi-table processing on legacy hardware," in *Proceedings of ACM HotSDN*, 2013.

[25] "Netty project," http://netty.io/.

[26] "Floodlight OpenFlow Controller," http://floodlight.openflowhub.org/.

[27] A. Vahidi, "JDD, a pure Java BDD and Z-BDD library," https://bitbucket.org/vahidi/jdd/.

[28] "Open vSwitch," http://openvswitch.org/.

[29] "Mininet," http://mininet.org/.

[30] D. E. Taylor and J. S. Turner, "ClassBench: a packet classification benchmark," in *Proceedings of IEEE INFOCOM*, 2005.

[31] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with Anteater," in *Proceedings of ACM SIGCOMM*, 2011.

[32] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proceedings of USENIX NSDI*, 2013.

[33] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of USENIX NSDI*, 2013.

[34] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proceedings of USENIX NSDI*, 2015.

[35] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A SOFT way for openflow switch interoperability testing," in *Proceedings of ACM CoNEXT*, 2012.

[36] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *Proceedings of USENIX NSDI*, 2014.