P4Tester: Efficient Runtime Rule Fault Detection for Programmable Data Planes

Yu Zhou, Jun Bi, Yunsenxiao Lin, Yangyang Wang, Dai Zhang, Zhaowei Xi, Jiamin Cao, Chen Sun
Institute for Network Sciences and Cyberspace, Tsinghua University
Department of Computer Science, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)

ABSTRACT

P4 and programmable data planes bring significant flexibility to network operation but are inevitably prone to various faults. Some faults, like P4 program bugs, can be verified statically, while some faults, like runtime rule faults, only happen to running network devices, and they are hardly possible to handle before deployment. Existing network testing systems can troubleshoot runtime rule faults via injecting probes, but are insufficient for programmable data planes due to large overheads or limited fault coverage. In this paper, we propose *P4Tester*, a new network testing system for troubleshooting runtime rule faults on programmable data planes. First, *P4Tester* proposes a new intermediate representation based on Binary Decision Diagram, which enables efficient probe generation for various P4-defined data plane functions. Second, *P4Tester* offers a new probe model that uses source routing to forward probes. This probe model largely reduces rule fault detection overheads, *i.e.* requiring only one server to generate probes for large networks and minimizing the number of probes. Moreover, this probe model can test all table rules in a network, achieving full fault coverage. Evaluation based on real-world data sets indicates that *P4Tester* can efficiently check all rules in programmable data planes, generate 59% fewer probes than ATPG and Pronto, be faster than ATPG by two orders of magnitude, and troubleshoot multiple rule faults within one second on BMv2 and Tofino.

CCS CONCEPTS

• **Networks** → Network measurement; **Error detection and error correction**.

KEYWORDS

Runtime rule fault detection, programmable data plane, test packet generation

ACM Reference Format:
Yu Zhou, Jun Bi, Yunsenxiao Lin, Yangyang Wang, Dai Zhang, Zhaowei Xi, Jiamin Cao, Chen Sun. 2019. P4Tester: Efficient Runtime Rule Fault Detection for Programmable Data Planes. In *IEEE/ACM International Symposium on Quality of Service (IWQoS '19), June 24–25, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3326285.3329040

IWQoS '19, June 24–25, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6778-3/19/06…$15.00
https://doi.org/10.1145/3326285.3329040

1 INTRODUCTION

Programmable data planes [1, 2] help network operators meet ever-increasing network operation demands. Network operators can agilely develop various functions on programmable data planes with P4 [3], such as advanced network monitoring [4], congestion-aware forwarding [5], high-performance load balancing [6], and distributed system accelerating [7–10]. Despite the programmability and flexibility, programmable data planes are inevitably prone to faults which could significantly compromise network QoS and incur revenue penalty.

Faults on programmable data plane come in various shapes and sizes. First, some faults exist in P4 programs and configurations and are referred to as static P4 faults, which can be formally debugged before deployment. Second, some faults can cause missed or abnormal table rules in running programmable switches. As these faults always appear at runtime, they can hardly be debugged before deployment and are referred to as *runtime rule faults*. In practice, these runtime rule faults are not uncommon for network devices [11–15], and can come into being due to switch software bugs [16], hardware bugs [17, 18], and abnormal operations [19]. These faults can cripple programmable data planes and degrade network reliability. Therefore, troubleshooting runtime rule faults for programmable networks is of great importance.

Troubleshooting runtime rule faults on programmable data planes is non-trivial and requires both *low overheads* and *high fault coverage*. Unfortunately, off-the-shelf solutions are far from satisfactory. *Static P4 verification* [20–23] can handle P4 program bugs before deployment but is not good at debugging runtime rule faults. *Passive network monitoring* [24–26] detects faults through tracing normal traffic but is insufficient to detect all rule faults as some faults might not be triggered by existing packets. *Proactive network testing* diagnoses faults via injecting *probes*. Comparing with P4 verification and network monitoring, network testing can proactively detect rule faults with a moderate number of probes. However, existing network testing systems either need a considerable number of servers for probe generation [14, 15], leading to unacceptable overheads, or cannot completely check all switches and table rules [11–13], leading to limited fault coverage.

In this paper, we argue for proactively testing programmable data planes for runtime rule fault detection. Given the limitations of existing solutions, we should design a new testing system dedicated to programmable networks, which has three challenging objectives.

Variability of P4 programs. Testing traditional networks only requires rigid probe generation schemes for fixed protocols and data plane functions, *e.g.* layer-3 routing. However, P4 enables network operators to develop various protocols and data plane functions, and different P4 programs process packets differently. Correspondingly,

to check runtime rule faults, different P4 programs need different patterns of probes. Thus, testing programmable data planes requires a flexible mechanism to generate probes for diverse P4 programs.

Low testing overheads. Proactive network testing mainly introduces two types of overheads, *i.e.* additional network bandwidth for forwarding probes and computation resources in control planes or endpoint servers for generating probes. First, the network testing system should generate as few probes as possible. Otherwise, too many probes might occupy too much bandwidth and impede normal traffic. Second, for the avoidance of substantial capital investment and management complexity, testing programmable data planes should introduce light workload to control planes or only occupy a small number of servers for probe generation.

High fault coverage. Making data planes free of runtime rule faults requires completely checking table rules in all switches at real time. Otherwise, some rule faults might be missed, and the effectiveness of the network testing system is compromised. However, some table rules are unreachable to the network testing system through traditional probe forwarding schemes, *e.g.* IP routing, which makes it challenging to guarantee high fault coverage.

To address the above challenges, we propose *P4Tester* as an efficient network testing system for programmable data planes. *P4Tester* is based on two key designs that make innovations on probe generation and forwarding. First, to flexibly *generate probes for various P4 programs*, we propose a intermediate representation of P4 programs and table rules based on Binary Decision Diagram (BDD) [27]. Taking BDD as the basic data structure, *P4Tester* performs automated analysis on P4 programs and table rules to generically generate probes for various data plane functions. Furthermore, *P4Tester* comes up with an efficient algorithm that can largely simplify P4 program analysis.

Second, to *achieve high rule coverage* with *low overheads*, *P4Tester* proposes a new probe model. This model forwards probes via *source routing* [28] and acquires testing results on each switch via *piggybacking rule actions in probes*. In *P4Tester*, probes have standard packet headers that can exercise table rules. After the normal packet headers, there is a source routing label stack to forward probes independently. With the label stack, *P4Tester* probes can check rules in all switches, yielding high fault coverage. On top of that, one *P4Tester* probe can check many rules, which largely reduces bandwidth overheads. *P4Tester* cautiously plans probe forwarding paths so that network operators only need one server for probe generation and can flexibly deploy the server anywhere in networks.

Our contributions are as follows.

- *P4Tester* is the first low-overhead and high-coverage network testing system to troubleshoot runtime rule faults on programmable data planes (§3).
- We provide a series of algorithms to optimize probe generation and update in *P4Tester* (§4).
- We build a prototype of *P4Tester*, deploy it on BMv2 and Tofino [29], and evaluate it with two real-world data sets. Evaluation result show that *P4Tester* can efficiently check all rules on programmable data planes, generate 59% fewer probes than ATPG and Pronto, be faster than ATPG by two orders of magnitude, and troubleshoot multiple rule faults within one second (§5).

2 BACKGROUND AND RELATED WORK

This section demonstrates the background of P4 and runtime rule faults and compares *P4Tester* with related work.

2.1 Preliminary to P4

P4 offers much flexibility to implement rich packet processing functions. We can construct compound actions with a variety of primitive actions (*e.g. modify_filed*). For a single table, P4 enables incorporating multiple match fields with various match types (*e.g. exact*, *ternary*, and *range*) and several compound actions. Furthermore, we can organize multiple tables as a consolidated directed acyclic graph with control flow logic, including if-else expressions and action-based case selections[1]. P4 also supports customization of packet formats and parsers. P4 programs have a two-phase lifecycle. At compilation time, P4 compilers (*e.g.* p4c [30]) transform P4 programs to executable code and deploy the executable code into P4 targets (*e.g.* Tofino [29]). At runtime, P4 pipelines and table entries populated by the control plane jointly determine the packet processing logic of switches.

2.2 Preliminary to Runtime Rule Faults

Before introducing runtime rule faults, we first present a definition of table rules.

Definition 1. *Table rule r is a three-tuple $\langle m, a, p\rangle$, in which the components respectively represents r's match fields, r's action, and r's priority.*

Runtime rule faults. There are mainly two types of runtime rule faults, *rule missing fault* and *rule priority fault*. First, if a rule r experiences the rule missing fault, all packets whose headers can match $r.m$ do not execute any action at all. Second, for a pair of overlapped rules r_1 and r_2 (there exist packets that can simultaneously match $r_1.m$ and $r_2.m$, and $r_1.p > r_2.p$), the priority fault happens when packets hitting $r_1.m$ and $r_2.m$ execute $r_2.a$ instead of $r_1.a$.

There are two main causes for the two runtime rule faults above. (1) Switch software may conduct incorrect or illegal rule operations which result in rule missing or rule priority swapping [16, 31]. An example is PicOS 2.1.3 that caches rules in switch software when hardware tables are full [16]. For a pair of overlapped rules r_1 and r_2, if the high-priority rule r_1 is cached in software and the low-priority rule r_2 is in hardware, packets matching $r_1.m$ and $r_2.m$ incorrectly exercise $r_2.a$. (2) Some hardware switching ASIC may be implemented incompletely or with errors. For example, HP 5406zl switching ASIC does not support rule priorities [17], and packets always exercise the later-installed rule in ignorance of rule priorities.

Runtime rule fault detection. Runtime rule faults lead to *inconsistency between data planes and control planes*. For example, the controller sends ten rule add messages to a switch, but the ASIC in the switch only has nine table rules, then the controller-switch inconsistency comes into being, and the switch encounters a rule missing fault. The main problem of rule fault detection is how to attain a full and accurate view of table rules in running network

[1]We mainly consider if-else expressions in this paper, because if-else expressions can equivalently express action-based case selections.

devices. Thus, the intuition of detecting runtime rule faults is to validate whether table rules on data planes strictly correspond to the table rule operations (*i.e.* adding or removing rules) issued by control planes. To address this problem, a general approach is to check real packet behaviors. If packets are forwarded in compliance with table rules issued by control planes, the realistic rules in network devices do not experience faults. Otherwise, rule faults may have happened.

2.3 Related Work

Static P4 verification. With the increasing maturity and popularity of P4, ensuring the reliability of programmable data planes has attracted much attention, especially in verification. Existing solutions such as ASSERT-P4 [22] and P4v [20] transform P4 programs to verification-friendly models (such as C language) and employ well-studied approaches (*e.g.* symbolic execution [32]) to find P4 program bugs. In particular, P4pktgen [23] generates test cases for P4 programs. *P4Tester* differs from P4pktgen in the following aspects. First, P4pktgen is designed to debug P4 programs and P4 toolchains statically, while *P4Tester* is devoted to checking runtime rule faults in P4 switches in real time. Second, P4pktgen conducts symbolic execution on P4 programs to generate test cases running in a behavior simulator. *P4Tester* performs analysis on P4 programs with BDD and proposes a new probe model to test table rules in distributed switches completely.

Passive network monitoring. Tracing and analyzing normal traffic, network monitoring can detect runtime rule faults via checking real packet forwarding behaviors. First, sample-based network monitoring systems, like sFlow [33], can only report faults experienced by a small portion of packets, thus have limited fault coverage. Second, NetSight [25] and EverFlow [24] provide packet-level information, but incur large bandwidth and processing overheads when checking all rules. Besides, all passive network monitoring systems can only detect rule faults that have been triggered by existing packets, and cannot check the correctness of all table rules.

Proactive network testing. Based on how to inject probes, we categorize existing network testing systems into two types. The first type is *controller-based probe injection*. Monocle [11], RuleChecker [12], and RuleScope [13] run in control planes and employ switches to inject probes via control messages. These systems could incur large control overheads, *i.e.* a large number of probes and intensive control messages. They inevitably suffer scalability bottlenecks when the network size and the rule number increase. Furthermore, they cannot check rules that forward packets from switches to servers. In brief, *controller-based probe injection yields limited testing completeness while incurring significant controller overheads*. The second one is *server-based probe injection*. ATPG [14] and Pronto [15] employ servers to inject probes. To check all rules, ATPG and Pronto require that all servers should support injecting and collecting probes. When some servers are unavailable, they can only check a small portion of rules, which makes it impossible to troubleshoot rule faults completely. In brief, *server-based probe injection has to strike a balance between fault coverage and server overheads*. Furthermore, the above systems are designed to model specific data plane functions. Only when the above systems change their inner probe generation algorithms case by case can they generate probes for various data plane functions. In summary, the off-the-shelf network testing systems either introduce large overheads or limit in fault coverage, which inevitably compromises their practicality. Table 1 presents a brief comparison between *P4Tester* and the above network testing systems.

Table 1: Comparing *P4Tester* with existing network testing systems.

	Model P4 Programs	Full Fault Coverage	Controller Overhead	Server Overhead
Monocle	No	No	High	Low
RuleScope	No	No	High	Low
RuleChecker	No	No	High	Low
ATPG	No	No	Low	High
Pronto	No	No	Low	High
P4Tester	Yes	Yes	Low	Low

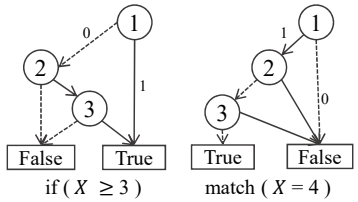


Figure 1: BDD for a three-bit field X, and each circle denotes a bit of X. The left part uses BDD to encode $if(X \leq 3)$, and the right part uses BDD to encode a table rule.

3 DESIGN OF *P4TESTER*

This section introduces key ideas and architectural overview of *P4Tester*.

3.1 Key Ideas of *P4Tester*

Testing programmable data planes requires *automatically generating probes for arbitrary P4 programs*, *high fault coverage*, *low bandwidth overheads*, and *low server overheads*. To satisfy these requirements, we propose two key ideas. The first one can overcome the complexity challenge from modeling various P4 programs, and the second one ensures completely checking rule faults with low bandwidth overheads and server overheads. Next, we will respectively illuminate the two ideas.

BDD as an intermediate representation. A P4 program can have many tables (up to 129 in Switch.p4 [34]) and if-else expressions to construct complex control flow. Furthermore, different networks possibly have various data plane functions defined by different P4 programs. Thus, we need a general method to generate probes for various P4 programs. An intuitive solution is to use symbolic execution to search all cases for P4 programs and table rules and to create probes for each case. However, this solution suffers state space explosion with the size of P4 programs increasing, incurring scalability issues as well as compromised probe generation efficiency [23].

To provide a generic probe-generation approach for different P4 programs, *P4Tester* employs Binary Decision Diagram (BDD) [27] as a new intermediate representation of P4 programs and table rules. As shown in Figure 1, BDD can uniformly represent if-else expressions and table rules. Moreover, BDD supports rich logic operations, such as conjunction ($\vee$), injection ($\wedge$), and complement

(¬). Based on the BDD representation, we can directly conduct high-performance program analysis on P4, which enables generating probes for various P4 programs and effectively improves the efficiency.

Source routing for probe forwarding. Existing network testing systems rely on layer-3 routing or layer-2 switching to forward probes, which has two drawbacks. First, it constrains the proportion of the rules that can be checked. Some table rules are not reachable to the network testing systems via the above forwarding schemes. Thus, fault coverage is fundamentally constrained. Second, it requires heavy computation on probe generation. To check all rules, network testing systems need to compute all-pair reachability (NP-Hard [15]), incurring long probe generation and update time.

To address the above drawbacks, we seek a flexible forwarding scheme, source routing [28], to forward probes in networks. We add a label stack after standard probe headers to control probe forwarding paths. Moreover, probe headers are only used to exercise table rules, while switches forward probes based on labels. Benefits of using source routing to forward probes are three-fold. (1) We could plan probe forwarding paths to let one probe header exercise many table rules cross different switches. Thus, *P4Tester* injects much fewer probes than existing testing systems. (2) It ensures full fault coverage even if there is only one available server. We can cautiously plan forwarding paths to make probes traverse in a circle and use the same server to inject and capture probes simultaneously. (3) It reduces computation complexity of probe generation by partitioning the problem into two independent parts: Computing probe headers according to table rules and computing probe forwarding paths according to the network topology. We can develop a series of efficient algorithms for rapid probe generation and update.

Furthermore, existing network testing systems rely on probe losses to infer rule faults. In particular, if probes successfully arrive at servers, operators can infer that the corresponding rules do not experience any fault. Otherwise, the rules might encounter faults. Such rule fault inference has two drawbacks. First, it cannot instantly distinguish port or link failures with rule faults and need multiple rounds of injecting probes to identify causes of probe losses, compromising rule fault detection efficiency. Second, the ultimate goal of network testing is to locate rule faults efficiently. ATPG and Pronto have to employ multiple lost probes to infer which rule experiences faults, posing long testing delay. When multiple rule faults exist, locating rule faults becomes even more complicated.

Instead of inferring faults, we use probes to piggyback actions (*i.e.* forwarding ports) of the rules under testing and to verify piggybacked rule actions directly. Benefits of rule action piggybacking are as follows. First, piggybacking rule actions distinguishes rule faults from link failures explicitly. For source routing probes, link failures lead to probe losses, and rule faults lead to wrong rule actions piggybacked in probes. Thus, we could easily detect rule faults with sound accuracy. Second, piggybacking rule actions supplies instant rule fault troubleshooting. *P4Tester* could check collected probes and directly locate the switch encountering rule faults.

3.2 Architecture of *P4Tester*

As shown in Figure 2, *P4Tester* comprises two primary components, *i.e. P4Tester agents running in switches* and *P4Tester terminal running in a server*. Before introducing these components, we first describe the workflow of *P4Tester*.

Workflow of *P4Tester*. Overall, using *P4Tester* to check table rules comprises four steps shown in Figure 2. (1) *P4 encoder* represents table rules and P4 programs with BDD. (2) Taking the BDD representations, topology, and rule update events as inputs, *probe constructor* generates and updates probes with a series of efficient algorithms. (3) *P4Tester agents* running in P4 switches forward probes and piggyback rule actions. (4) As soon as probes come back to *P4Tester* terminal, *probe checker* detects and locates rule faults.

***P4Tester* probe**. A probe of *P4Tester* comprises two parts shown in Figure 3. The first part is a standard probe header (including Ethernet, IPv4, TCP, and so on) that can exercise table rules in P4 programs. The second one is a label stack (*i.e.* L_1, L_2, ...) residing behind the probe header. In the stack, source routing labels correspond to each hop traversed by probes. For one label, there are three fields. The first 2-bit field *BOS* (bottom of the stack) identifies whether the label has been visited. If the label has been visited, the corresponding *BOS* is 1. Otherwise, *BOS* is 0. *BOS* helps P4 parser find the label corresponding to the current hop through finding the first label whose *BOS* is 0. Second, *P4Tester* records *standard_metadata.egress_spec* in *Port* (15 bits). At last, the 15-bit field *Egress* specifies the forwarding port of the probe at the current hop. Specifically, *P4Tester* agent rewrites *standard_metadata.egress_spec* with the value of *Egress* after recording the value of *standard_metadata.egress_spec* in *Port*. Overall, one *P4Tester* label has 32 bits.

***P4Tester* agent**. As shown in Figure 3, *P4Tester* agent is composed of two tables that process probes sequentially. (1) Record Table (*RT*) records egress ports specified by the original P4 programs in labels. (2) *Forward Table* (*FT*) redirects probes according to *Egress*. In total, the two tables only need two rules which are applied to all probes by default. Evaluation results in §5.3 indicate that *P4Tester* agent consumes minor data plane resources.

***P4Tester* terminal**. *P4Tester* terminal involves two aspects of functions. First, *P4Tester* terminal should generate probe headers to exercise table rules and generate *Egress* in labels to plan probe forwarding paths. Second, *P4Tester* terminal should check rule actions piggybacked in probes to verify whether corresponding rules experience faults. As shown in Figure 2, *P4Tester* terminal is composed of three elements.

(1) P4 encoder. P4 encoder relieves operators' concerns for handling various header space of different P4 programs. P4 encoder is able to automatically convert P4 programs and table rules into BDD based representations. The workflow of P4 encoder comprises the following steps. First, P4 encoder extracts global header space via merging match fields of all tables and fields referred by if-else expressions. Then, P4 encoder represents each if-else expression with BDD, and a BDD can represent one if-else expression. Next, P4 encoder represents table rules in each table with BDD, and we can get a BDD set for each table.

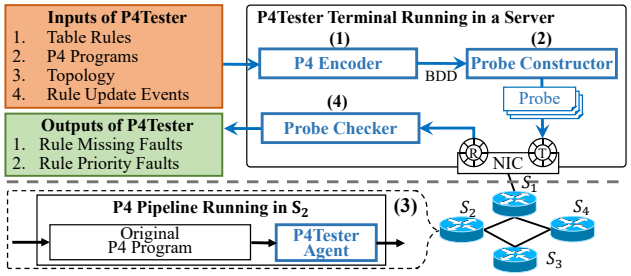


Figure 2: Architectural overview of *P4Tester*.

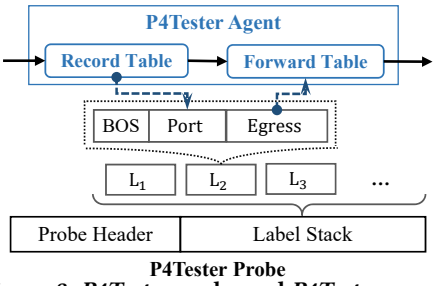


Figure 3: *P4Tester* probe and *P4Tester* agent.

(2) Probe constructor. After getting BDD representations of P4 programs and table rules, probe constructor generates probe headers and labels accordingly. Probe constructor provides two types of outputs. First, it provides a map between headers and desired actions, *i.e.* $H \mapsto \{p'_x | \ x \in [1, N_s]\}$. H denotes the probe header. We assume that probes need to traverse N_s switches, whose id x is from 1 to N_s; p'_x denotes the desired output port after being processed by switch s. Second, the constructor cautiously generates *Egress* of labels that direct probes to traverse in circles. For example, a probe emitted by *P4Tester* terminal in Figure 2 can traverse the following path: $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_1$, and the probe will finally go back to *P4Tester* terminal. Thus, *P4Tester* could use the same server to inject and collect probes. Furthermore, the constructor can easily build the label stack that enables probes to exercise all table rules. In this respect, *P4Tester* can completely check rule faults with fewer probes and only one server. Moreover, the constructor injects probes via the server NIC. Besides, the constructor also reacts to rule update events and performs quick probe update.

(3) Probe checker. Probe checker takes captured probes as inputs. Probe checker extracts rule actions as well as headers from probes, and then it verifies the correctness of piggybacked actions switch by switch. Probes can be viewed as a map between probe headers and real rule actions, *i.e.* $H \mapsto \{p''_x | \ x \in [1, N_s]\}$ (p''_x denotes an output port). Then, for a particular header H, the checker can be formulated as checking the correctness of the following proposition: $\forall x \in [1, N_s], p'_x = p''_x$. Probe checker could instantly find rule faults at the time of receiving probes. Simultaneously, probe checker locates which switches experience rule faults via finding the switch id x that breaks the above proposition.

4 PROBE GENERATION AND UPDATE IN *P4TESTER*

In this section, we describe the algorithms of generating and updating probes in *P4Tester*. Table 2 outlines the symbols used throughout the paper.

Table 2: Symbols used throughout the paper.

Symbol	Description
r	A table rule
P	A switch-level header set
$\mathbb{P}$	A network-wide header set
$\mathcal{P}$	A probe
$\mathcal{R}$	Root of a BDD tree
K	Maximum label number in a packet
N_s, N_t, N_r, N_P, $N_{\mathbb{P}}$, N_θ	Number of switches, tables per switch, rules per table, switch-level header sets, network-wide header sets, probes

Algorithm 1: Generate Switch-level Probe Headers

Input: Control flow graph of a P4 program
Output: Switch-level probe headers

$V' \leftarrow$ TopologicalSort(P4 control flow graph);
E is a Boolean expression vector whose length is $|V'|$;
for $1 \le n \le |V'|$ **do**
  if $V'[n]$ *is a table* **then**
    for $1 \le i \le N_r$ **do**
      $r'_{n,i} \leftarrow (r_{n,i}.m \wedge E[n], r_{n,i}.a, r_{n,i}.p)$;
    foreach *child* v_c *of* $V'[n]$ **do**
      $E[v_c] \leftarrow E[v_c] \vee E[n]$;
  else if $V'[n]$ *is an if-else expression* **then**
    For true child v_t, $E[v_t] \leftarrow V'[n].bdd \wedge E[n] \vee E[v_t]$;
    For false child v_f, $E[v_f] \leftarrow \neg V'[n].bdd \wedge E[n] \vee E[v_f]$;
$\overline{P} \leftarrow \{\}$; /* $\overline{P}$ stores SHSs. */
for $1 \le n \le |V'|$ **do**
  $\overline{P}' \leftarrow \{\}$; /* $\overline{P}'$ temporarily stores SHSs. */
  for P' *in* $\overline{P}$ **do**
    for $1 \le i \le N_r$ **do**
      if $r'_{n,i}.m \wedge P' \neq False$ **then**
        $\overline{P}' \leftarrow \overline{P}' \cup \{P' \wedge r'_{n,i}.m\}$;
        $P' \leftarrow \neg r'_{n,i}.m \wedge P'$;
  $\overline{P} \leftarrow \overline{P}' \cup \overline{P}$;

4.1 Probe Generation

In this part, *P4Tester* answers two questions. The first one is *how to generate headers for checking all rules*, and the second one is *how to generate labels to convey probes to switches under testing*. Probe constructor is composed of three independent steps. For the first question, Step 1 generates probe headers to test table rules at the switch level (*i.e.* one probe header for one rule in one switch), and Step 2 generates probe headers to test rules in the network wide (*i.e.* one probe header for multiple rules in different switches). For the second question, Step 3 generates probe forwarding paths and the corresponding label stack.

Solving problems of each step independently cannot guarantee global optimization on the number of probes. *P4Tester* insists on pursuing independent solutions of each step to improve efficiency, *i.e.* quick probe generation and update. More importantly, even with

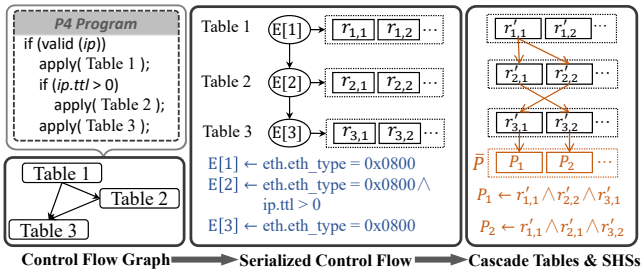


Figure 4: An example of SHS generation. $r_{i,j}$ denotes the j-th rule in the i-th table. P_i denotes i-th SHS of a switch.

the partial optimization, *P4Tester* generates the minimal number of probes among all the existing solutions when checking rule faults in the same networks.

Step 1: Generating switch-level probe headers. In this step, probe constructor needs to convert rules and P4 programs to probe headers. For one switch, we name probe headers that exercise the same rules as a *switch-level header set* (*SHS*). SHS generation requires handling complex P4 control flow. First, *P4Tester* conducts simplification on BDD representations of P4 programs and equivalently transforms the complex control flow graph to simple cascade tables. Then, we can generate SHSs based on cascade tables.

Simplifying P4 program control flow comprises two parts. (1) *Serializing control flow. P4Tester* first transforms the control flow graph to a serialized control flow graph. In the serialized graph, each table is accompanied by a generated if-else expression, and this expression globally determines whether a packet could be processed by corresponding tables. (2) *Merging if-else expressions and table rules. P4Tester* removes the if-else expressions and generates new rules through the conjunction of if-else expressions and match of original rules.

We provide Algorithm 1 to generate SHSs. Figure 4 presents an example to generate SHSs for a simple P4 program. First, the algorithm performs the topological sort on the control flow graph. Then, the algorithm iterates over nodes in the control flow graph and computes E. E identifies the global conditions, under which the P4 program executes the nodes. At last, we can generate SHSs for cascade tables (Line 12-19). Algorithm 1 generates SHSs for rule missing faults, while we can extend the algorithm to detect rule priority faults according to *GenerateProbeCascade* in RuleChecker [12]. Moreover, Algorithm 1 satisfies Lemma 1.

Lemma 1. $\forall i, j \in [1, N_P],\ P_i \wedge P_j \equiv false$ *when* $i \neq j$.

Step 2: Generating network-wide probe headers. Based on SHSs of all switches, Step 2 generates probe headers that match multiple table rules across different switches, and we call such headers as *network-wide probe headers*. We refer network-wide probe headers that exercise the same set of table rules as a *network-wide header set* (*NHS*). For example, if we want $\mathbb{P}$ to exercise rules that are respectively exercised by $P_{1,1}, P_{2,1}, ..., P_{N_s,1}$ ($P_{x,y}$ denotes the y-th SHS in the x-th switch), we can generate $\mathbb{P}$ via the conjunction of above SHSs, *i.e.* $\mathbb{P} \leftarrow \bigwedge_{1 \le i \le N_s} P_{i,1}$. We call $\mathbb{P}$ represents $P_{1,1}, ..., P_{N_s,1}$. According to Lemma 1, $\mathbb{P}$ only represents one SHS in one switch. We define the SHSs represented by $\mathbb{P}$ as follows.

$$Re(\mathbb{P}) \triangleq \{P \mid P \wedge \mathbb{P} \neq false \text{ and } \mathbb{P} - P = false\}$$

Algorithm 2: Generate Network-level Probe Headers

Input: Switch Probe Set $P_{i,j}$ of every switch
Output: NHSs $(\mathbb{P}_1, \mathbb{P}_2, ...)$

1 **Function** *QueryBDDTree*($\mathcal{N}$, *target*)
2 **if** $\mathcal{N}$ *is a leaf node* **then**
3 **return** $\mathcal{N}$;
4 $\mathcal{N}' \leftarrow$ *none*;
5 **if** $Left(\mathcal{N}) \wedge target \neq false$ **then**
6 $\mathcal{N}' \leftarrow QueryBDDTree(Left(\mathcal{N}), target)$;
7 **if** $\mathcal{N}' =$ *none and* $Right(\mathcal{N}) \wedge target \neq false$ **then**
8 $\mathcal{N}' \leftarrow QueryBDDTree(Right(\mathcal{N}), target)$;
9 **if** $\mathcal{N}' \neq$ *none* **then**
10 $\mathcal{N} \leftarrow \mathcal{N} \wedge \neg\mathcal{N}'$;
11 **return** $\mathcal{N}'$;
12 **foreach** $1 \le n \le N_s$ **do**
13 $\mathcal{R}_n \leftarrow CreateBDDTree(P_{n,1}, P_{n,2}, ..., P_{n,N_P})$;
14 $\overline{\mathbb{P}} \leftarrow \emptyset$; /* $\overline{\mathbb{P}}$ stores NHSs. */
15 $\mathbb{H} \leftarrow false$; /* Avoid repeated iterations over $P_{i,j}$. */
16 **for** $1 \le n \le N_s$ **do**
17 **foreach** $1 \le m \le N_P$ **do**
18 **if** $P_{i,j} \wedge \mathbb{H} \neq false$ **then**
19 $\mathbb{P} \leftarrow P_{i,j}$; /* $\mathbb{P}$ is a NHS. */
20 $\Phi \leftarrow \{\}$; /* Store represented SHSs. */
21 **for** $n < n' \le N_s$ **do**
22 $\mathcal{N} \leftarrow QueryBDDTree(\mathcal{R}_n, \mathbb{P})$;
23 **if** $\mathcal{N} \neq$ *none* **then**
24 $\mathbb{P} \leftarrow \mathbb{P} \wedge \mathcal{N}$;
25 $\Phi \leftarrow \Phi \cup \{\mathcal{N}\}$;
26 $Re(\mathbb{P}) \leftarrow \Phi$;
27 $\overline{\mathbb{P}} \leftarrow \overline{\mathbb{P}} \cup \{\mathbb{P}\}$;
28 $\mathbb{H} \leftarrow \mathbb{H} \vee \mathbb{P}$;

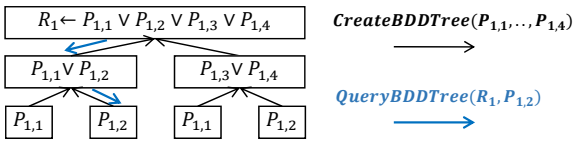


Figure 5: A BDD tree. Black arrows show the procedure of *BuildBDDTree*, and blue arrows show the the procedure of *QueryBDDTree*.

Intuitively, Step 2 tries to find NHSs that can represent all SHSs in all switches. We formulate the problem solved by Step 2 as follows.

Definition 2. *Step 2 generates* $\mathbb{P}_1$, $\mathbb{P}_2$, ..., $\mathbb{P}_{N_\mathbb{P}}$ *that satisfy* $\forall i \in [1, N_s], j \in [1, N_P], \exists x \in [1, N_\mathbb{P}], P_{i,j} \in Re(\mathbb{P}_x)$.

The optimization goal of Step 2 is to generate as few NHSs as possible. However, getting the optimal number of NHSs is NP-hard and needs long computation time. To this end, we design a greedy algorithm for NHS generation. The high-level idea of the algorithm is to make $|Re(\mathbb{P})|$ (size of $Re(\mathbb{P})$) as large as possible for each $\mathbb{P}$.

To further optimize the efficiency of NHS generation, we propose a new data structure called *BDD tree*, a complete binary tree, each

Algorithm 3: Generate Probe Forwarding Paths

Input: Topology G, Max Probe Size K, Switch S_0
Output: Probe forwarding paths

1 $\Omega \leftarrow \{\}$;
2 **Function** *TraverseST(s)*
3 **foreach** *child c of s* **do**
4 $\Omega \leftarrow \Omega \cup \{\langle s, Port(s \Rightarrow c)\rangle\}$;
5 *TraverseST*(c);
6 $\Omega \leftarrow \Omega \cup \{\langle c, Port(c \Rightarrow s)\rangle\}$;
7 *BuildSTwithBFS*(G, S_0);
8 *TraverseST*(S_0);
9 $\mathcal{S} \leftarrow \{\}$; /* $\mathcal{S}$ is a segment. */
10 $\omega \leftarrow S_0$;
11 $\mathcal{F} \leftarrow \{\}$; /* $\mathcal{F}$ stores probe forwarding paths. */
12 **foreach** $1 \le n \le |\Omega|$ **do**
13 $l \leftarrow |Path(S_0, \omega)| + |Path(\Omega[n].sid, S_0)| + |\mathcal{S}| + 1$;
14 **if** $l \ge K$ **then**
15 $\mathcal{F} \leftarrow \mathcal{F} \cup \{Path(S_0, \omega) \cup \mathcal{S} \cup Path(\Omega[n-1].sid, S_0)\}$;
16 $\omega \leftarrow \Omega[n].sid$;
17 $\mathcal{S} \leftarrow \{\}$;
18 *segment* $\leftarrow$ *segment* $\cup \{\Omega[n]\}$;
19 $\mathcal{F} \leftarrow \mathcal{F} \cup \{Path(S_0, \omega) \cup \mathcal{S} \cup Path(\Omega[n].sid, S_0)\}$;

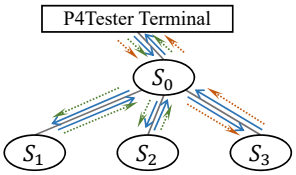
of whose nodes stores a BDD. Figure 5 shows a BDD tree example. For a BDD tree, the BDD of a node is the disjunction of its children. node $\mathcal{N}$ comprises two properties, *i.e.* $Left(\mathcal{N})$ and $Right(\mathcal{N})$, to represent the two children of $\mathcal{N}$. There are two operations for the BDD tree. (1) *CreateBDDTree* returns the root node of a newly-created BDD tree whose leaf nodes store all SHSs of a switch. (2) *QueryBDDTree* (Line 1-11 in Algorithm 2) returns a leaf node which matches a target BDD. Furthermore, *QueryBDDTree* removes the queried leaf node from the BDD tree, which guarantees once query to any SHS and can effectively improve the efficiency of generating NHSs.

Algorithm 2 shows the basic procedure of NHS generation. Furthermore, we can generate one probe header for each generated NHS via solving BDD AnySAT, which has linear complexity. The overall time complexity of Step 2 is $O(N_s N_P log(N_P))$.

Step 3: Generating source routing labels. Step 3 is to generate a label stack that forwarding probes according to network topologies. As Maximum Transmission Unit (MTU) limits probe packet sizes, a probe can only convey a certain number of labels at once. Thus, there could be multiple probes for one NHS to test corresponding rules in different switches.

Algorithm 3 shows the procedure of probe forwarding path generation, and Figure 6 shows an example of Algorithm 3 to generate paths for a four-node topology. The main idea of the algorithm is to generate a global path that includes all switches and to slice the global path into several segments. Moreover, we control path slicing to keep the segment size smaller than K, and then the probe packet size can satisfy the MTU limitation.

In Algorithm 3, S_0 is the switch connected to *P4Tester* terminal. First, we build a spanning tree (ST) via conducting a breadth-first



Probe Forwarding Paths

Segments	Paths
$\mathcal{F}$[1]	$S_0 \to S_3 \to S_0 \to S_3 \to S_0$
$\mathcal{F}$[2]	$S_0 \to S_3 \to S_0$

Global probe forwarding path Ω →

Figure 6: An example for generating probe forwarding paths by Step 3.

search (BFS) on the network topology. Then, we recursively conduct pre-order traversal on the ST to build the global path Ω with *TraverseST*. Ω is a list of two-tuples $\langle sid, port\rangle$ as *sid* denotes the switch id and *port* denotes the output port. After getting Ω, we can generate paths $\mathcal{F}$ whose lengths are smaller than K (Line 9-19). We use $Path(x, y)$ to denote the shortest path from x to y. As this algorithm will iterate over all nodes in Ω, it will continuously check the length of the circle $Path(S_0, \omega) \cup segment \cup \Omega[n] \cup Path(\Omega[n].sid, S_0)$ (Line 13). If the circle length surpasses K, the algorithm will create a new probe forwarding path in $\mathcal{F}$ (Line 14-17). Furthermore, the algorithm should also include the last segment (Line 19). For each NHS, the probes generated according to $\mathcal{F}$ can traverse all switches.

4.2 Probe Update

When rule additional, removal, and modification happen, network testing systems need to update probe headers and forwarding paths accordingly. Rule operations may happen in high frequency, requiring quick probe update. *P4Tester* proposes three efficient schemes for updating probes.

Time-least probe update. When the controller frequently updates table rules, *P4Tester* should timely check whether the rule update event is executed correctly, which is important to guarantee consistent network update. *P4Tester* proposes *time-least probe update* to pursue least time of verifying rule existence. As for adding a new rule r_{new} in switch s, time-least update directly generates the NHS $\mathbb{P}_{new}$ via $\mathbb{P}_{new} \leftarrow r_{new}.m \wedge \neg \mathcal{R}_s$ and $Tr(\mathbb{P}_{new}) \leftarrow \{s\}$ ($\mathcal{R}_s$ is the BDD tree root of switch s). Then, *P4Tester* generates a new probe for $\mathbb{P}_{new}$. As for rule removal, time-least update does not perform any operation and directly employs current probes to verify whether the rules have been successfully removed. The time complexity of time-least update is $O(1)$.

Quantity-least probe update. Time-least update inevitably suffers probe explosion as it generates one new probe for every added rule and never removes probes whose corresponding rules have expired. *P4Tester* proposes *quantity-least probe update* which optimizes the number of probes. The high-level idea of quantity-least update is to employ as many existing NHSs as possible to represent SHSs of updated rules. First, for the switch experiencing rule update, we recalculate SHSs and rebuild the BDD Tree. Then, we iterate all existing NHSs and find all SHSs that can be represented by existing NHSs. After that, we need to generate new probes for the SHSs which cannot be represented by existing NHSs. Second, quantity-least update does not need to recompute all probes from the ground up. Instead, it only recomputes a part of NHSs, which promises relatively high efficiency.

Hybrid probe update. The above two schemes pursue different metrics, but they can work jointly to embrace both high efficiency

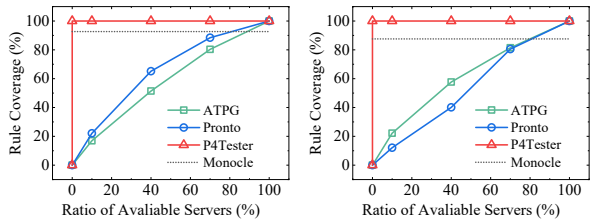


Figure 7: Fault coverage comparsion of *P4Tester*, ATPG, Pronto, and Monocle.

and fewer probes. Hybrid probe update integrating time-least update and quantity-least update comprises two steps. (1) When a rule update event happens, hybrid update instantly employs time-least update to check rules and stores the event in a buffer. (2) When the number of buffered events is larger than a threshold, *P4Tester* employs quantity-least update to execute a batch of update events and to aggregate probes generated by time-least update.

5 EVALUATION

Implementation and setup. We implement a prototype of *P4Tester* terminal with about 5000 lines of Java code and use JDD [35] to support BDD operations. Furthermore, *P4Tester* agent is implemented as a piece of P4 code which consists of two tables with two table rules in total. We deploy *P4Tester* terminal prototype on a server with 32GB RAM and 4 Intel E3-1280 3.9GHz CPU cores. We deploy *P4Tester* agent on two P4 targets. The first one is BMv2 [36], which is the widely-used P4 software target. The second one is Wedge 100BF-32X [37] equipped with Tofino ASIC and 32 100GbE ports.

Objectives. We evaluate *P4Tester* with five objectives.

- We compare *P4Tester* with three well-known countermeasures in terms of probe generation efficiency, probe numbers, and rule coverage (§5.1).
- We evaluate the scalability of *P4Tester* when the number of routers and rules increases (§5.2).
- We evaluate the additional resource usage of *P4Tester* agent on the hardware target (§5.3).
- We evaluate the efficiency of *P4Tester* when rule addition and rule removal lead to probe update (§5.4).
- We deploy *P4Tester* on two testbeds and present *P4Tester*'s overall efficiency of troubleshooting rule faults on BMv2 switches and Tofino switches (§5.5).

5.1 Comparing *P4Tester* with Countermeasures

In this part, we compare *P4Tester* with the three off-the-shelf network testing systems, ATPG, Pronto, and Monocle, with respect to *fault coverage*, *probe numbers*, and *probe generation time*. We use two real-world data sets from Stanford and Internet2. To accommodate table rules from real-world routers, we develop a P4 program which forwards packets based on prefixes of IPv4 destination addresses. The P4 program is used throughout the evaluation experiments.

Table 3: Comparing ATPG, Pronto, *P4Tester*, and Monocle in terms of Probe generation time and probe numbers.

		ATPG	Pronto	*P4Tester*	Monocle
Internet2	Time (s)	1992.5	12.3	3.2	-
	Probes	35416	32379	13235	103911
Stanford	Time (s)	2807.01	3.48	0.3	-
	Probes	3319	5540	1493	15370

Rule coverage. Rule coverage is important for fault coverage. We first evaluate rule coverage when the ratio of available endpoint servers varies, and we quantify rule coverage via the ratio of rules that can be checked by network testing systems. As shown in Figure 7(a) and Figure 7(b), ATPG and Pronto have similar rule coverage on two data sets. Moreover, their rule coverage grows linearly with the ratio of available servers. In other words, they require that all servers support injecting and collecting probes. Otherwise, they can only check a part of table rules, which significantly compromises testing completeness. Monocle's rule coverage is 92% for Internet2 and 83% for Stanford, which is agnostic to the server numbers because Monocle uses a controller to inject probes and cannot check the rules that forward packets to servers. Notably, *P4Tester* only needs one server to achieve full rule coverage, no matter how large the network is.

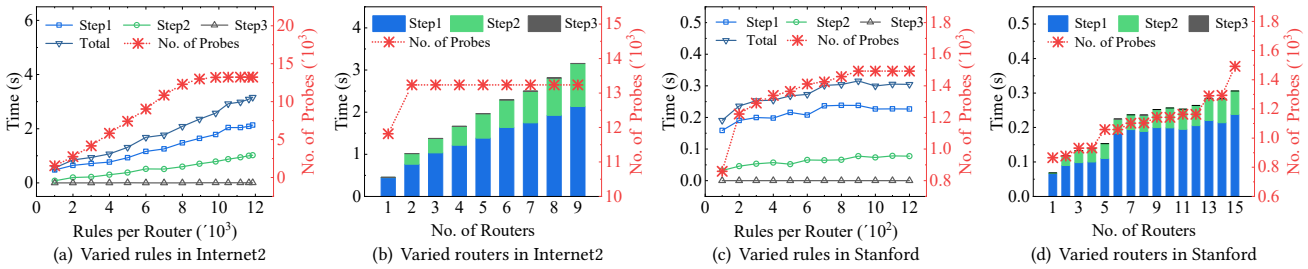
The number of probes. Table 3 summarizes comparison regarding probe numbers. *P4Tester* generates much fewer probes than ATPG and Pronto. Specifically, *P4Tester* reduces the number of probes by over 59% for Internet2, and by over 55% for Stanford. Comparing with Monocle, *P4Tester* reduces the probe number by about one order of magnitude for the two two data sets. In summary, *P4Tester* introduces much less bandwidth overhead than all the countermeasures.

Probe generation efficiency. As shown in Table 3, the probe generation time of ATPG is over 100x larger than Pronto, and over 1000x larger than *P4Tester*. The probe generation time of *P4Tester* is less than Pronto by over one order of magnitude for the Stanford data set. In summary, *P4Tester* yields much better probe generation efficiency than the existing server-based probe generation solutions which employ servers to inject probes.

5.2 Scalability of *P4Tester*

To show the scalability of *P4Tester*, we evaluate it in terms of probe generation efficiency and probe numbers when the number of table rules and routers varies.

The number of probes. Figure 8(a) and Figure 8(c) show that with the number of rules per router increasing, the number of probes grows linearly. When there are more rules per router, *P4Tester* can check more table rules with the same probes. Thus, the number of probes converges to a specific value, revealing that *P4Tester* probes can check multiple rules and are of high-usage. Figure 8(b) shows how the number of routers influences the number of probes on Internet2. Particularly, the number of probes does not grow too much with the number of routers due to two reasons. (1) Probe headers generated for existing routers can also be used to test rules in a newly-added router. (2) The probe size is big enough to accommodate rule actions in newly-added switches. Through source routing, *P4Tester* could direct probes to traverse more switches. In this respect, one probe of *P4Tester* could exercise more rules than existing solutions. For Stanford, as shown in Figure 8(d), the probe number grows linearly with the number of routers.



(a) Varied rules in Internet2 (b) Varied routers in Internet2 (c) Varied rules in Stanford (d) Varied routers in Stanford

Figure 8: Probe generation with different numbers of rules and routers.

Probe generation efficiency. As shown in Figure 8(a), Figure 8(b), Figure 8(c), and Figure 8(d), the time of Step 1 and Step 2 grows linearly with the number of rules and routers. Moreover, the delay of Step 3 is ignorant of the number of rules. When there is only one router, the delay of Step 2 and Step 3 is close to 0s. The above results indicate that the algorithms of *P4Tester* present linear scalability against the number of routers and rules.

5.3 Resource Usage of *P4Tester*

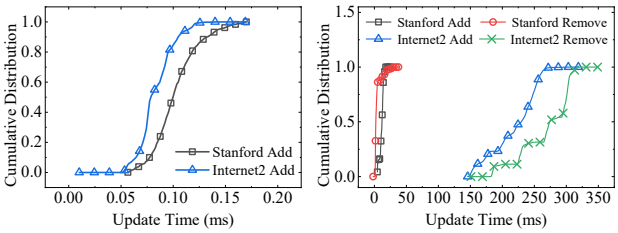
In this part, we show the hardware resource usage of *P4Tester* agents in Tofino, and we build a prototype of the agent that supports 16 source routing labels. As shown in Table 4, one *P4Tester* agent only need 5.7% very long instruction word actions to implement compound actions and 14.1% packet header vector to accommodate source routing labels, while all the other resources are not required. Furthermore, the size of packet header vector grows linearly with the number of source routing labels. In summary, *P4Tester* agent brings minor resource overheads to hardware targets and have a very small impact on the other data plane functions.

5.4 Efficiency of Probe Update

As for probe updating, we measure the delay of adding one table rule with time-least update and the delay of performing one round of quantity-least update. We repeat each experiment for 1000 times and show the cumulative delay distribution in Figure 9(a) and Figure 9(b). As for time-least update, the probe update delay of adding a table rule is less than 0.5 ms. As for quantity-least update, the delay varies from several ms to hundreds of ms, which largely depends on the data sets. The Stanford data set has fewer table rules per router than the Internet2 data set. Thus, the delay of Stanford will be smaller than that of Internet2 because quantity-least update needs to regenerate the SHSs of one router.

Table 4: Additional hardware resources consumed by *P4Tester* with 16 source routing labels. The values are normalized by the usage of Switch.p4.

Resources	Usage
Match Crossbar	0%
Static Random Access Memory	0%
Ternary Content Addressable Memory	0%
Very Long Instruction Word Actions	5.7%
Hash Bits	0%
Stateful Arithmetic and Logic Units	0%
Packet Header Vector	14.1%



(a) Time-least Probe Addition (b) Quantity-least Probe Update

Figure 9: Delay distribution of time-least and quantity-least rule update.

5.5 Efficiency of Rule Testing

To demonstrate how efficiently *P4Tester* can troubleshoot real-world P4-enabled networks, we build two testbeds composed of BMv2 and Tofino switches, which is shown in Figure 10. The BMv2 testbed is based on the public topology of Stanford backbone network. As we only have one Tofino switch, we take rules from one router in the Stanford data set and install them into the switch. In the experiment, we remove a certain number of randomly-selected rules. Then, we employ *P4Tester* running in a separate server to detect and locate missed rules. To quantify the testing efficiency, we measure the delay between the time of injecting probes and the time of locating all faults. Thus, we show the rule testing delay when the number of missed rules varies. We repeat each experiment for 1000 times.

Time of troubleshooting rule faults. As for BMv2 testbed, we can see from Figure 11(a) that checking rule faults takes less than 0.88s when the number of missed rules is less than 10. Furthermore, the delay of checking rule faults grows slowly with the number of missed rules, which indicates that *P4Tester* can detect and locate missed rules with high efficiency. As for Tofino testbed, Figure 11(a) shows that the rule testing delay of *P4Tester* also increases mildly with the number of missed rules. The above results testify that *P4Tester* can efficiently troubleshoot multiple rule missing faults on both software targets and hardware targets.

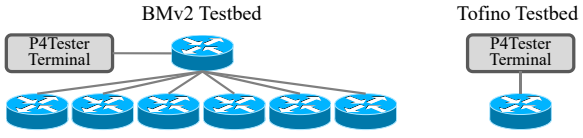


Figure 10: Testbed typologies.

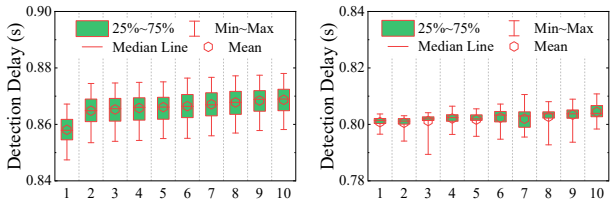


Figure 11: Delay of using *P4Tester* to detect multiple rule faults in the two testbeds.

6 DISCUSSION

With P4Tester, network operators only need one server to work as P4Tester terminal. When more servers are available, P4Tester also permits deploying multiple terminals on the servers. Meanwhile, all the table entries and table update events should be synchronized among the terminals, but each terminal only needs to generate the probes that can test a portion of table rules. For large-scale networks which may have multiple controllers, P4Tester can scale flexibly via deploying multiple terminals, which is our future work.

7 CONCLUSION

As the programmable data plane grows in popularity and maturity, ensuring its reliability and availability is of great importance but challenges the state-of-the-art network testing systems. *P4Tester* is the first low-overhead and high-fault-coverage network testing system targeting runtime rule faults on programmable data planes. *P4Tester* makes innovations on how to generate probes and how to forward probes. *P4Tester* performs analysis on P4 programs to generate probes and presents a novel probe model that uses source routing to forward probes and piggybacks rule actions in probes. *P4Tester* comes up with a series of algorithms to improve the efficiency of probe generation and update. Compared with existing solutions, *P4Tester* yields remarkable benefits, including quick probe generation and full fault coverage with only one server and much fewer probes.

ACKNOWLEDGEMENT

This research is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (61872426). Yangyang Wang is the corresponding author. We thank all anonymous reviewers for their constructive comments.

REFERENCES

[1] Pat Bosshart, Glen Gibb, Hun-seok Kim, George Varghese, Nick Mckeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of SIGCOMM*, 2013.
[2] Sharad Chole, Isaac Keslassy, Ariel Orda, Tom Edsall, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, and Shang-Tse Chuang. drmt: Disaggregated programmable switching. In *Proceedings of SIGCOMM*, 2017.
[3] Pat Bosshart, Glen Gibb, Hun-seok Kim, George Varghese, Nick Mckeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 44(3), 2014.
[4] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven network telemetry. In *Proceedings of SIGCOMM*, 2018.
[5] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of SOSR*, 2016.
[6] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of SIGCOMM*, 2017.
[7] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of SOSP*, 2017.
[8] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of NSDI*, 2018.
[9] Amedeo Sapio et al. In-network computation is a dumb idea whose time has come. In *Proceedings of HotNets*, 2017.
[10] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of SOSR*, 2015.
[11] Peter Perešíni, Maciej Kuźniar, and Dejan Kostić. Monocle: Dynamic, fine-grained data plane monitoring. In *Proceedings of CoNEXT*, 2015.
[12] Peng Zhang, Cheng Zhang, and Chengchen Hu. Fast testing network data plane with rulechecker. In *Proceedings of ICNP*, 2017.
[13] Xitao Wen, Kai Bu, Bo Yang, Yan Chen, Li Erran Li, Xiaolin Chen, Jianfeng Yang, and Xue Leng. Is every flow on the right track?: Inspect sdn forwarding with rulescope. In *Proceedings of INFOCOM*, 2016.
[14] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. *IEEE/ACM Trans. Netw.*, 22(2), 2014.
[15] Yu Zhao, Huazhe Wang, Xin Lin, Tingting Yu, and Chen Qian. Pronto: Efficient test packet generation for dynamic network data planes. In *Proceedings of ICDCS*, 2017.
[16] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of SOSR*, 2016.
[17] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about sdn flow tables. In *Proceedings of PAM*, 2015.
[18] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of SIGCOMM*, 2015.
[19] P. Zhang. Towards rule enforcement verification for software defined networks. In *Proceedings of INFOCOM*, 2017.
[20] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. p4v: Practical verification for programmable data planes. In *Proceedings of SIGCOMM*, 2018.
[21] Radu Stoenescu et al. Debugging p4 programs with vera. In *Proceedings of SIGCOMM*, 2018.
[22] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of SOSR*, 2018.
[23] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. P4pktgen: Automated test case generation for p4 programs. In *Proceedings of SOSR*, 2018.
[24] Yibo Zhu, Ben Y. Zhao, Haitao Zheng, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, and Ming Zhang. Packet-level telemetry in large datacenter networks. In *Proceedings of SIGCOMM*, 2015.
[25] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proceedings of NSDI*, 2014.
[26] Cheng Zhang, Jun Bi, Yu Zhou, Jianping Wu, Bingyang Liu, Zhaogeng Li, Abdul Basit Dgar, and Yangyang Wang. P4db: On-the-fly debugging of the programmable data plane. In *Proceedings of ICNP*, 2017.
[27] Jean Francis Michon et al. Graph based algorithms for boolean function manipulation. 2005.
[28] Carl A. Sunshine. Source routing in computer networks. *SIGCOMM Comput. Commun. Rev.*, 7(1), 1977.
[29] Barefoot Networks. Barefoot tofino switch. Website, 2019. https://barefootnetworks.com/technology/.
[30] P4 Language Consortium. P4_16 prototype compiler. Website, 2019. https://github.com/p4lang/p4c.
[31] Maciej Kuzniar, Peter Peresini, and Dejan Kostic. Proboscope: Data plane probe packet generation. Technical report, 2014.
[32] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.
[33] sFlow. sflow. Website, 2019. https://sflow.org/.
[34] The P4 Language Consortium. Consolidated switch repository. Website, 2019. https://github.com/p4lang/switch.
[35] Arash Vahidi. Jdd. Website, 2019. https://bit.ly/2SZjMjp.
[36] P4 Consortium. P4-bmv2. Website, 2019. https://github.com/p4lang/behavioral-model.
[37] Edgecore Networks. Wedge 100bf-32x. Website, 2019. https://bit.ly/2TfkI2p.