Mind the Gap: Monitoring the Control-Data Plane Consistency in Software Defined Networks

Peng Zhang, Hao Li, Chengchen Hu, Liujia Hu, Lei Xiong, Ruilong Wang, and Yuemei Zhang
Department of Computer Science and Technology
Xi'an Jiaotong University
{p-zhang,hao.li,chengchenghu}@xjtu.edu.cn; {liujia-hu,hbxgxiong,wangruilongy,zym665456}@stu.xjtu.edu.cn

ABSTRACT

How to debug large networks is always a challenging task. Software Defined Network (SDN) offers a centralized control platform where operators can *statically* verify network policies, instead of checking configuration files device-by-device. While such a static verification is useful, it is still not enough: due to data plane faults, packets may not be forwarded according to control plane policies, resulting in network faults at *runtime*. To address this issue, we present VeriDP, a tool that can continuously monitor what we call *control-data plane consistency*, defined as the consistency between control plane policies and data plane forwarding behaviors. We prototype VeriDP with small modifications of both hardware and software SDN switches, and show that it can achieve a verification speed of 3 μs per packet, with a false negative rate as low as 0.1%, for the Stanford backbone and Internet2 topologies. In addition, when verification fails, VeriDP can localize faulty switches with a probability as high as 96% for fat tree topologies.

CCS Concepts

•Networks $\rightarrow$ Network measurement; •Hardware $\rightarrow$ Error detection and error correction;

Keywords

Software Defined Network; consistency; verification

1. INTRODUCTION

Networks are prone to faults. Meanwhile, most operators still debug network faults in an ad-hoc way: checking configurations device-by-device with simple tools like ping, traceroute, SNMP, etc. As a result, it is highly-demanding and time-consuming for operators to pinpoint the root causes. A network debugging toolkit that can automatically detect, locate, and repair network faults is highly desired [57].

Software Defined Network (SDN) decouples control functions away from the data plane, so that operators can configure/program the network from a centralized point. This centralized control offers new opportunities for automating the network debugging process. Many tools have been developed to verify the correctness of network configuration in SDN [36, 37, 35, 58]. Since these tools generally debug network configurations at the controller side, we will term them as control plane verification tools.

Though the above tools are useful for reducing errors in the configuration process, they assume these configurations will be reflected in packet forwarding behaviors. As will be shown later, even if configurations are error-free at the control plane, packets forwarding can still be faulty at the data plane. Reasons for these faults include: lack of acknowledgement from switches [50, 46, 40], switch software bugs [57, 34], premature switch implementation [46], external modification [18, 47], etc (see Section 2.2).

To detect such faults, data plane verification tools have been developed. ATPG [57] sends probe packets to test the data plane, while it solely checks reception of probe packets, without inspecting the paths took by the probe packets. This prevents ATPG from verifying policies (*e.g.*, waypoint, access control) that are dependent on paths. Monocle [41] generates probe packets to test the existence of rules at switches. However, due to the slow probe generation process (at a scale of seconds), Monocle can only check a relatively steady network configuration and cannot work under frequent network updates or reconfigurations. In addition, probe packets may be treated differently from real traffic, thereby leading to inaccurate judgements (see Section 3.1).

Given the limitations of both control plane and data plane verification tools, we think a missing part in the current SDN architecture is a tool that can ensure the operator's configurations will correctly reflect at packet forwarding behaviors. To this end, this paper proposes VeriDP, a tool that can continuously monitor the control-data plane consistency, *i.e.*, whether packet forwarding behaviors agree with the network configurations. By ensuring such consis-

CoNEXT '16, December 12-15, 2016, Irvine, CA, USA
© 2016 ACM. ISBN 978-1-4503-4292-6/16/12...$15.00
DOI: http://dx.doi.org/10.1145/2999572.2999605

tency, VeriDP enables operators to focus on configuration correctness.

VeriDP uses *path table* as the abstract of control plane configurations, and tests whether packet forwarding behaviors are conforming to the path table. A path table is a data structure that records forwarding paths between any two ports in the network. Each path entry consists of (1) a forwarding path, which consists of a sequence of hops, in the form of $\langle input\_port, switch\_ID, output\_port \rangle$, (2) a set of all packets, which can be forwarded along the path, and (3) a tag, which is a compression of the path information. VeriDP generates the path table based on the network topologies and rules installed by the controller. When a packet traverses through the network, each switch generates/updates the tag of the packet the same way as the tag in the path table is computed. Before the packet leaves the network, a report encapsulating the header and tag of the packet is sent to the controller for verification. If the forwarding path is correct, the reported header will match an entry in the path table, and the reported packet tag should be the same with the one of the matched entry.

We face the following two challenges when realizing VeriDP. First, we need an efficient method to construct the path table, so that (1) it should allow fast lookups in order to scale to large networks, (2) it can be updated incrementally to accommodate frequent rule updates, and (3) it should encode packet headers with efficient methods without consuming too much memory. Second, we need a tag compression method which allows localization of faulty switches, when inconsistency is detected. Thus, simple hash-based tag compression is not possible.

To address the above challenges, we leverage Binary Decision Diagram (BDD) to encode packet headers, and use Bloom filter for tag compression. In addition, we design an incremental path table update algorithm that only updates a small portion of path table. In summary, our contribution is two-fold:

- We propose VeriDP, the first tool (to the best of our knowledge) that can monitor the control-data plane consistency in SDN. VeriDP can achieve a false negative rate as low as 0.1% for inconsistency detection, and an accuracy as high as 96% for faulty switch localization. The verification speed is as fast as 3 μs per packet for the Stanford backbone and Internet2 topologies.
- We prototype VeriDP with small modifications of both hardware and software SDN switches. We use extensive experiments to show that VeriDP incurs a minimal overhead on data-plane packet processing.

As a first step towards monitoring control-data plane consistency, VeriDP is limited in the following aspects: (1) it cannot handle packet rewrites that will change headers of packets when they are forwarded, (2) it cannot detect packet drops due to hardware failures, and (3) it requires modification (even small) of switches.

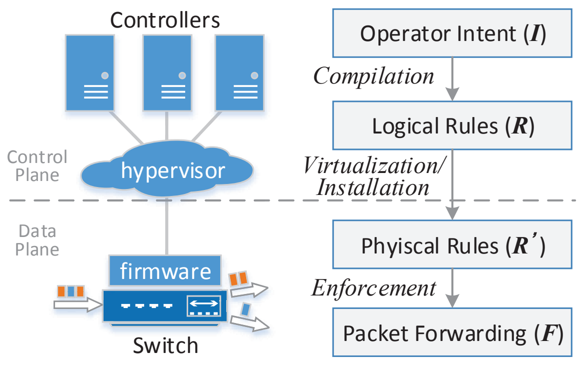


Figure 1: The translation process from operator intent to packet forwarding in SDN.

In the rest of this paper, we will first give more background on the control-data plane consistency issue (§2), and present an overview of VeriDP (§3). Then, we give details on the design and implementation of VeriDP (§4 and §5), and evaluate its performance (§6). After discussing some related work (§7), we conclude the paper (§8).

2. BACKGROUND AND MOTIVATION

In this section, we will first present some background on SDN. Specifically, we clearly define how an operator's intent is mapped to packet forwarding behaviors in SDN. Then, we show some cases under which the data plane becomes inconsistent with the control plane. Following that, we show the consequences that can be caused by such inconsistency. The reason that we separately discuss the causes and consequences of control-data inconsistency is that a single case of inconsistency can result in multiple consequences. For example, a missing rule at a switch may cause blackholes, loops, or access violation.

2.1 All the Way from Operator Intent to Forwarding Behavior

The purpose of an SDN is to translate a network operator's intent into packet forwarding behaviors [30]. Figure 1 illustrates such a translation process, which consists of four stages: operator intent (I), logical rules (R), physical rules (R'), and packet forwarding (F).

At the first stage, an operator specifies her *intent (I)* in terms of high-level policies (we will use intent and policy interchangeably in the rest of paper). For example, she may want packets from hosts inside a department directly reach a database server, while packets from hosts outside the department should go through a firewall before reaching the server. Many SDN programming languages (Frenetic [28], Pyretic [44], Merlin [53], etc.) and controllers (Ryu [9], ONOS [21], etc.) allow operators to specify such high-level intent. Though the operator's intent could be quite diverse, some intent is invariant, including pairwise reachability, backhole-freedom, loop-freedom, etc.

At the second stage, a compiler (often a built-in component of a controller) should be used to translate the high-level operator intent into device-specific *logical rules (R)* that can be understood by forwarding devices (*i.e.*, switches). By logical, we mean that these rules are complied based on a logic view of the network by the controller, and have not yet been installed at switches.

At the third stage, the controller will install the compiled rules through some southbound protocols like OpenFlow. The network may use virtualization platforms (*e.g.*, FlowVisor [52], OpenVirtex [15], etc.) to translate the rules, before they are sent to switches. We will term the rules installed by switches as *physical rules* (R'). Note that $R = R'$ may not hold, due to reasons like switch software bugs.

At the fourth stage, switches process and forward incoming packets according to the installed rules R'. At this point, the operator intent finally translates into packet *forwarding (F)*. Assuming everything goes well, the packet forwarding behaviors should exactly reflect the operator's intent, *i.e.*, $I = R = R' = F$.

Most previous tools try to verify whether $I = R$, assuming I as a set of basic invariants, *i.e.*, $I \leftarrow \{$*reachability, blackhole-freedom, loop-freedom*$\}$. We will refer to them as control plane verification tools. For other tools, ATPG verifies $I = F$ (also assuming I as set of basic invariants), and Monocle verifies whether $R = R'$. In contrast, VeriDP checks whether $R = F$.

2.2 Control-Data Plane Inconsistency: The Causes

There could be many reasons that data plane forwarding behaviors deviate from the operator intent at the control plane. The following lists four possible reasons.

Lack of data plane acknowledgement. SDN advocates a fine-grained dynamic control over switches, meaning controllers have to frequently update the data plane states. Currently, update in SDN is "open-loop" in nature, without effective data plane acknowledgement mechanisms. Then, network updates may be out of order, or even fail to take effect, without being noticed by the controller. This will result in that the controller and switches can have inconsistent views, where the controller thinks a rule has taken effect, while the switch fails to install it [41]. Note that it is challenging to design such an acknowledgement mechanism [40]. The `Barrier` command provided by OpenFlow attempts to enable controllers to confirm when switches have installed rules [8]. However, measurements show that switches may respond to `Barrier` too early before the rules are actually installed in the flow table, in order to save cost [50, 46].

Switch software bugs. According to a survey among network operators, switch/router software bug is one of the most possible reasons for network failures [57]. For example, [34] provides a concrete bug where switches wrongly manages hardware-software-hybrid flow tables. Since the size of hardware flow table is relatively small, most SDN switches use software flow tables. However, rule placement strategies are prone to faults, due to cross-rule dependency [34]. The authors showed that the Pronto-Pica8 3290 switch, which can hold up to 2000 rules in hardware flow table, simply placed all extra rules at software flow table. This simple placement strategy respects no dependency across rules, and was shown to cause forwarding behaviors that are inconsistent with controller's policies [34].

Premature switch implementation. Currently, the implementation of SDN switches is quite premature, with some mandatory features not supported by some switches. Without noting that, rules complied by the controller may not translate into the forwarding behaviors. For example, according to a recent measurement, the HP ProCurve 5406zl switch lacks support for rule priority [46]. If two rules with different priorities have overlapped matching fields but different actions, the packet may be wrongly forwarded according to the low-priority rule.

External rule modifications. Besides the SDN controller, a rule can also be modified by an external source. For example, a careless operator may install a new rule that overrides previous one through data plane configuration tools like `dpctl` [2]. An attacker can gain the access to the switch OS and modify a forwarding rule to redirect traffic elsewhere [18]. These actions may not be noticed by the controller, resulting in a faulty data plane configuration. Worse still, [47] showed that ONIE [6], the boot loader for 3rd-party switch OS, can be exploited by an attacker to gain persistent control over SDN switches, even after OS reinstallation.

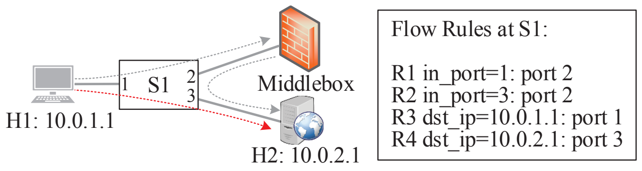
In sum, apart from misconfigurations, faults can also come from the data plane. These faults are out of the scope of many existing verification tools (*e.g.*, Veriflow [37]) that function at the control plane. Note that we are not considering inconsistency caused by network updates [49], as they only cause transient failures.

2.3 Control-Data Plane Inconsistency: The Consequences

In the following, we show how control-data plane inconsistency may break the operator's intent. We will list some common intent, including basic invariants, access control, waypoint traversal, and traffic engineering.

Basic Invariants. As mentioned earlier, all networks should satisfy some basic invariants including reachability, blackhole-freedom, loop-freedom, etc [37]. Due to control-data plane inconsistency, even these invariants are satisfied at the control layers, they may be violated when packets are forwarded. For example, when a forwarding rule is not successfully installed, or wrongly modified by an external source, the corresponding hosts may no longer be reached. These invariants can be checked by sending probe packets in the network, as in ATPG [57].

Access control. Besides basic invariants, there are also



Flow Rules at S1:

R1 in_port=1: port 2
R2 in_port=3: port 2
R3 dst_ip=10.0.1.1: port 1
R4 dst_ip=10.0.2.1: port 3

Figure 2: Middlebox traversal. The security policy of $H1 \rightarrow Middlebox \rightarrow H2$ is violated.

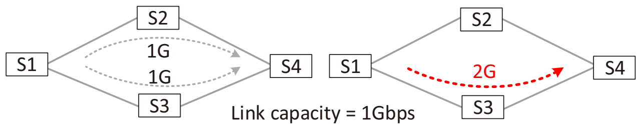


Figure 3: Traffic engineering. Path $S1 \rightarrow S3 \rightarrow S4$ can be congested if traffic engineering rules fail at $S1$.

custom policies that may vary from network to network. For example, an operator may specify some access control policies to prevent some set of hosts from talking with each other. These policies will translate into ACL rules that are installed by switches. If these ACL rules are missing in the flow table, say due to bad rule replacement strategies, the access control intent will be violated.

Waypoint traversal. An operator may also require some packets to traverse a set of middleboxes in sequence when being forwarded inside the network. Take Figure 2 as an example, where rules at switch $S1$ indicate that traffic from the client $H1$ to the server $H2$ must go through the middlebox. Now, consider the high-priority rules $R1$ and/or $R2$ fail. Then the middlebox chaining policy is violated, and the firewall is bypassed.

Traffic engineering. A traffic engineering policy may require packets to be forwarded over multiple links or tunnels according to some specific distribution. As shown in Figure 3, rules at switch $S1$ evenly distribute traffic from $S1$ to $S4$ between two paths $S1 \rightarrow S2 \rightarrow S4$ and $S1 \rightarrow S3 \rightarrow S4$. Now, consider that the rules fail at switch $S1$, resulting in that all traffic goes through the second path. Then, this traffic engineering policy is violated, resulting in sub-optimal performance.

3. DESIGN OVERVIEW

In this section, we will first discuss several design options, and then present the design of VeriDP.

3.1 Design Options

First, we consider the following two approaches for checking control-data plane consistency.

Checking flow tables. For this approach, the controller can periodically check the healthy of rules at switches' flow tables. Frequently dumping all rules from switches is clearly inefficient, and will place burden on switches. To address this problem, Monocle [41] generates probe packets to test the existence of rules. However, the probe generation is too slow (at a scale of several seconds) to keep up with frequent network updates.

Checking packet forwarding behaviors. Instead of checking flow tables, ATPG [57] directly looks at packet forwarding behaviors. Specifically, ATPG lets end hosts send probe packets into the network, and checks whether they are correctly received. Violation of reachability policies like black holes and loops can thus be detected. However, ATPG cannot detect violation of the other three types of inconsistency as listed in Section 2.3. Checking them requires we inspect not only the correct reception of packets, but also the paths traversed by packets.

Given the limitations of checking flow tables or packet receptions, we decide to check whether packet forwarding paths are consistent with control-plane configurations. With this decision made, a remaining question is should we inject probe packets and verify their forwarding paths, or should we sample real traffic from the data plane for verification? In the following, we will discuss this issue.

Probes or real traffic. First, using probe packets, we can only verify whether the forwarding paths of probe packets agree with the configurations. However, it does not necessarily mean that the real traffic will also follow the same paths. For example, consider an ACL rule $R1$ that only permits HTTP traffic from IP address 10.0.0.1:

$$\boxed{R1 \;\; src\_ip = 10.0.0.1, dst\_port = 80: \;\; \text{Allow}}$$

A probe packet with source address 10.0.0.1 and destination port 80 can trigger this rule. However, even the packet is successfully received, it may not mean the rule is correctly configured at the switch. For example, consider the above rule is prioritized by an ill-inserted rule $R2$:

$$\boxed{R2 \;\; src\_ip = 10.0.0.1, dst\_port = *: \;\; \text{Allow}}$$

The probe packet can still be received, while non-HTTP traffic, *e.g.*, SSH, from 10.0.0.1 will also be allowed, violating the controller's policy. The reason that the probe packet cannot detect such violation is that it can only verify the control-data plane consistency with respect to a single packet. To capture all possible inconsistency, we need to exhaust all possible packets, which is clearly infeasible. *Given the limitations of probe packets, we decide to sample real traffic from the data plane for verification.*

3.2 VeriDP Architecture

As shown in Figure 4, VeriDP consists of a server alongside the SDN controller, and a pipeline for each switch in the network. The server intercepts the bidirectional OpenFlow messages exchanged between the controller and switches, in order to construct the *path table*, which records all paths for each input and output port pair. The pipeline is responsible for *sampling*, *tagging*, and *reporting* packets to the server. With the path table, the server *verifies* the reported packets sent from switches, and when the verification fails, it tries to

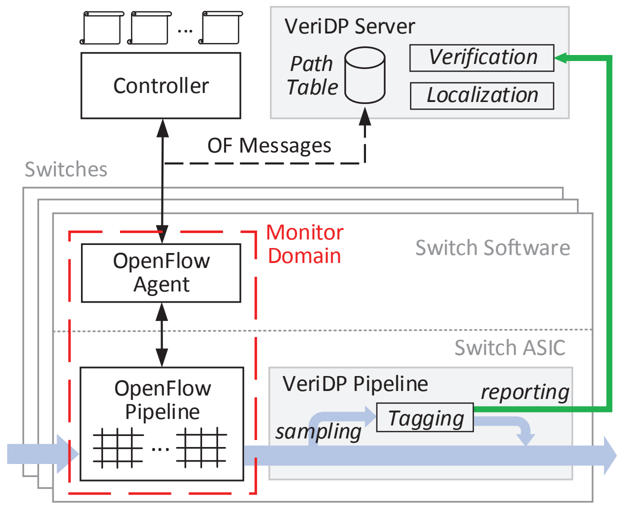


Figure 4: System architecture. The shaded components belong to VeriDP; those within the dashed rectangle are the components that VeriDP monitors.

localize the faulty switches. The dashed rectangle represents the domain that VeriDP monitors, *i.e.*, VeriDP is expected to detect the faults caused by the components inside the domain. The monitor domain includes: (1) the OpenFlow agent that terminates the OpenFlow channel, and (2) the OpenFlow pipeline that manages the hardware flow table and forwards packets based on table lookups.

3.3 VeriDP Pipeline

VeriDP distinguishes among three different types of switches: entry switches, exit switches, and internal switches. Entry and exit switches are edge switches that are directly connected with end hosts or middleboxes, and internal switches are aggregate and core switches that sit inside the network, interconnecting edge switches.

The VeriDP pipeline is responsible for generating tags for packets at entry switches, updating tags for packets at internal switches, and reporting packet headers and tags to the controller at exit switches. Entry switches' pipelines will perform sampling and all other switches are maintained stateless; exit switches' pipelines should report packets to the server (some internal switches should also conditionally report packets); all switches should tag packets that are sampled by entry switches.

The pipeline is implemented in a switch's fast path, separated from the OpenFlow pipeline. The reason for such separation is to prevent faults caused by flow tables from propagating into the tagging module. Since a typical switch can contain a cascade of flow tables, each of which may hold thousands of flow entries, flow entries used for tagging may be overridden by other rules, replaced when flow table is full, and even incorrectly modified/deleted by applications.

The pipeline processing is shown in Algorithm 1. The entry switch initializes the packet tag to zero, and the TTL

Algorithm 1: Tag(s, x, y, p)

```
Input: s: the switch ID; x/y: the local input/output port ID of packet
       p, which is received from the OpenFlow pipeline.
1 if $\langle s, x\rangle$ is an edge port then
2     $p.tag \leftarrow 0$; // initialize the tag
3     $p.ttl \leftarrow$ MAX_PATH_LENGTH; // initialize the ttl
4 $p.tag \leftarrow p.tag \sqcup \mathtt{BF}(x||s||y)$; // update the tag
5 $p.TTL \leftarrow p.TTL - 1$; // decrement the ttl
6 if $\langle s, y\rangle$ is an edge port or $y = \perp$ or $p.TTL = 0$ then
7     SendReport$(inport, \langle s, y\rangle, p.header, p.tag)$;
```

to the maximum path length (Line 1-3). Each switch updates the tag as:

$$tag \leftarrow tag \sqcup \mathtt{BF}(input\_port||switch\_ID||output\_port)$$

, where $switch\_ID$ is the identifer of the switch; $input\_port$ /$output\_port$ is the local input/output port ID of the packet; $\mathtt{BF}(x)$ is a k-bit Bloom filter holding a single element x; $\sqcup$ represents the bit-by-bit OR operation.

Initially, we were tempted to use hash-based tagging, *i.e.*, replace the BF with a hash function, and use the bit-by-bit OR instead of bit-by-bit XOR. Later, we found that this tagging method prevents us from localizing the faulty switch. On the other hand, by exploiting the information contained in Bloom filter, we can achieve a high localization probability (refer to Section 4.3 for details).

Besides tag updating, the switch also decrements the TTL by one (Line 4-5). When the packet is output to an edge port connected with an end host, or output to the dropping port $\perp$, or its TTL value hits zero, the switch sends a *tag report* to the server (Line 6-7). A tag report is a 4-tuple $\langle inport, outport, header, tag\rangle$, where $inport/outport$ is the entry/exit port of the packet; $header$ is a portion of packet header (*e.g.*, TCP 5-tuple); tag is the tag of the packet. Besides sending tag reports, the exit switch should also pop the tag from the packet, and deliver the packet to its destination host.

One thing to note is that switches should send tag reports for dropped packets. This is necessary to ensure the visibility of verification server into blackholes and loops. Here, we consider two cases of packet drops: (1) the packet does not match any forwarding entry, and (2) the packet matches some forwarding entries, but the entries do not specify any output ports. On the other hand, we do not consider packet drops due to hardware failures, which will totally prevent the failed switch from sending any tag reports.

3.4 VeriDP Server

The VeriDP server is responsible for parsing and verifying tag reports sent by switches. Central to the VeriDP server is the *path table*, which maps a pair of $\langle inport, outport\rangle$ to a list of paths that enter the network at $inport$ and exit at $outport$. Each path is again a pair of $\langle headers, tag\rangle$, where $headers$ is a set of headers allowed for the path, and tag is the tag representing the path.

Table 1: Part of the path table for Figure 5. $[\cdot]$ represents $\texttt{BF}(\cdot)$, and $\sqcup$ represents bit-by-bit OR.

inport	outport	headers	tag
$\langle S_1, 1\rangle$	$\langle S_3, 2\rangle$	$src\_ip = 10.0.1.1, dst\_ip = 10.0.2.1, dst\_port = 22$	$[1\|\|S_1\|\|3] \sqcup [1\|\|S_2\|\|3] \sqcup [3\|\|S_2\|\|2] \sqcup [1\|\|S_3\|\|2]$
		$src\_ip = 10.0.1.1, dst\_ip = 10.0.2.1, dst\_port \neq 22$	$[1\|\|S_1\|\|4] \sqcup [3\|\|S_3\|\|2]$
$\langle S_1, 2\rangle$	$\langle S_3, \perp\rangle$	$src\_ip = 10.0.1.2, dst\_ip = 10.0.2.1, dst\_port = 22$	$[2\|\|S_1\|\|3] \sqcup [1\|\|S_2\|\|3] \sqcup [2\|\|S_2\|\|2] \sqcup [1\|\|S_3\|\|\perp]$
		$src\_ip = 10.0.1.2, dst\_ip = 10.0.2.1, dst\_port \neq 22$	$[2\|\|S_1\|\|4] \sqcup [3\|\|S_3\|\|\perp]$

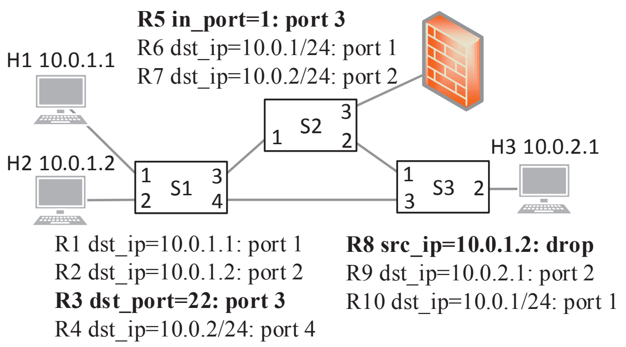

R5 in_port=1: port 3
R6 dst_ip=10.0.1/24: port 1
R7 dst_ip=10.0.2/24: port 2
H1 10.0.1.1
H2 10.0.1.2
H3 10.0.2.1
S1
S2
S3
R1 dst_ip=10.0.1.1: port 1
R2 dst_ip=10.0.1.2: port 2
R3 dst_port=22: port 3
R4 dst_ip=10.0.2/24: port 4
R8 src_ip=10.0.1.2: drop
R9 dst_ip=10.0.2.1: port 2
R10 dst_ip=10.0.1/24: port 1

Figure 5: A simple example for path table construction. The network consists of three switches and a total of 10 rules.

Algorithm 2: $\texttt{Traverse}(inport, \langle s, x\rangle, h, p, t)$

```
Input: inport: the entry port tuple; ⟨s, x⟩: the current port tuple;
  h: the current header set; p: the current path; t: the current tag;
  P_{x,y}: the predicate for admitted packets from port x to port y;
1  foreach port $y \in \{1, 2, \dots, n, \perp\}$ do
2      $h' \leftarrow h \wedge P_{x,y}$ // packets forwarded to port y
3      if $h' \neq \emptyset$ then
4          $p' \leftarrow p||\langle x, s, y\rangle$ // update the path
5          $t' \leftarrow t \sqcup$ BF($x||s||y$) // update the tag
6          if $\langle s, y\rangle$ is an edge port then
7              PathTable($inport, \langle s, y\rangle$).AddPath($h', p', t'$);
8          else
9              $\langle s', y'\rangle \leftarrow$ Link($\langle s, y\rangle$) // go to next hop
10             Traverse($inport, \langle s', y'\rangle, h', p', t'$);
```

For a concrete example, consider the toy network in Figure 5. Rule 3 redirects all SSH traffic to $S2$, and Rule 4 forwards all other packets towards 10.0.2/24 to $S3$. Rule 5 directs all traffic from port 1 to the middlebox. Rule 8 at switch $S3$ drops all traffic from $H2$. Other rules are plain forwarding rules ensuring connectivity. Table 1 shows a part of the path table for this network.

Here, we assume there are no packet rewrites, *i.e.*, packet headers remain the same when packets are forwarded in the network. In this way, when a packet is about to exit the network, the VeriDP server can verify the packet's forwarding path against its header. We acknowledge this is a limitation of the current design, and may be resolved in the future.

4. DESIGN DETAILS

4.1 Path Table Construction

A first problem for constructing path table is how to represent header sets. A straightforward way is to use wildcard expressions, just as in Header Space Analysis [36]. However, even wildcard expressions are widely used for representing suffix, they are very inefficient for representing arbitrary header sets. For example, the header set for $dst\_port \neq 22$ in the second row of Table 1 is a union of 16 wildcard expressions. In addition, wildcard expressions have a poor support of set operation like union, conjunction, and complement. For a typical network consisting of tens of switches, each of which has thousands of flow rules, a huge number of wildcard expressions will be needed to represent the whole packet sets. According to [33], characterizing the Stanford backbone network (16 switches) needs 652 million wildcard expressions.

Inspired by the previous work [56], we decide to use the Binary Decision Diagram (BDD) [22] to represent header sets. BDD is an efficient data structure for Boolean expressions, and has a better support of set operations. With BDD, we can expect to significantly reduce the size of path table.

In the following, we will show how to construct the path table from a network configuration. First, we show how to specify the configuration of a single switch in the network. A switch s with ports numbered from 1 to n can be specified by a bunch of *transfer predicates* $P_{x,y}$, where $x \in \{1, 2, \dots, n\}$ and $y \in \{1, 2, \dots, n, \perp\}$. Only those packets with headers satisfying predicate $P_{x,y}$ can transfer (*i.e.*, be forwarded) from port x to port y. Transfer predicate is a general abstraction of switch configuration, which can be parsed from both traditional and OpenFlow switches.

For now, we will assume transfer predicates of all switches are already computed, and show how to construct the path table. Later on, we will come back and explain how to generate the transfer predicates from switch configurations. Algorithm 2 summarizes the process of path table construction. From each port, denoted as $inport$, of each switch, we inject a header set h initialized to all-match (*i.e.*, a BDD of $\texttt{True}$), and a tag t initialized to zero. Then, we call $\texttt{Traverse}(inport,\ h,\ p,\ t)$, with the current path p being empty. When the header h is received at a port $\langle s, x\rangle$, we intersect h with the transfer predicate $P_{x,y}$ (Line 2). If the intersection is non-empty, we update the path p and the tag t (Line 4-5). If y is an edge port, we insert a new entry into the path table (Line 6-7); otherwise, we move on to the next-

Algorithm 3: Verify$(inport, outport, header, tag)$

```
Input: $inport/outport$: the input/output port of the packet;
       $header$: the header of the packet; $tag$: the tag of the packet.
Output: True (pass), or False (fail).
1 foreach $p \in$ PathTable$(inport, outport)$ do
2     if $header \prec p.headers$ then
3         if $tag = p.tag$ then
4             return True; // the path is correct
5         return False; // the path is wrong
6 return False; // the packet should not reach here
```

hop, and recursively call the algorithm with the new header and tag (Line 8-10).

Now we go back and show how to generate transfer predicates from switch configuration files. Specifically, we consider the Cisco configuration files for the Stanford backbone network [36]. The configuration files specify forwarding rules, in-bound ACLs, out-bound ACLs, VLAN, etc.

First, we use Algorithm 1 and 2 in [56] to transform the configuration files into *port predicates* for each switch. The port predicates consist of: (1) in-bound ACL predicate P_x^{in} for each port x: only packets satisfying P_x^{in} are allowed to be received from port x, (2) out-bound ACL predicate P_y^{out} for each port y: only packets satisfying P_y^{out} are allowed to be output to port y, and (3) forwarding predicate P_y^{fwd} for each port y: only packets satisfying P_y^{fwd} will be forwarded to port y. Then, we can compute the transfer predicates as:

$$P_{x,y} = P_x^{in} \wedge P_y^{fwd} \wedge P_y^{out},\ y \neq \perp$$

$$P_{x,\perp} = \neg P_x^{in} \vee \left(P_x^{in} \wedge P_\perp^{fwd}\right) \vee \left(P_x^{in} \wedge \vee_y \left(P_y^{fwd} \wedge \neg P_y^{out}\right)\right)$$

, where $P_\perp^{fwd} = \neg\left(\vee_y P_y^{fwd}\right)$

Intuitively, the three terms of $P_{x,\perp}$ represent three reasons that a packet is dropped: (1) filtered by the in-bound ACL, (2) not forwarded to any port, and (3) filtered by the out-bound ACL.

4.2 Tag Verification

The tag verification process is quite simple, as shown in Algorithm 3. On receiving a tag report of the form $\langle inport, outport, header, tag\rangle$, the server looks up in the path table with index $\langle inport, outport\rangle$, and possibly finds a list of paths. For each path p, it tries to match *header* with the header set of path p (Line 1-2). If matched, *tag* is compared with the tag of path p. The verification succeeds if these tags are equal (meaning that the packet followed the right path), or fails otherwise (Line 3-6). If no matched path is found (meaning that the packet should not have reached here), then the verification also fails (Line 7).

Let us turn back to Figure 5, and assume H_1 sends a packet to port 22 of H_3. The packet should take the path of $S_1 \rightarrow S_2 \rightarrow Middlebox \rightarrow S_2 \rightarrow S_3$, and the tag should be $[1||S_1||3] \sqcup [1||S_2||3] \sqcup [3||S_2||2] \sqcup [1||S_3||2]$. With

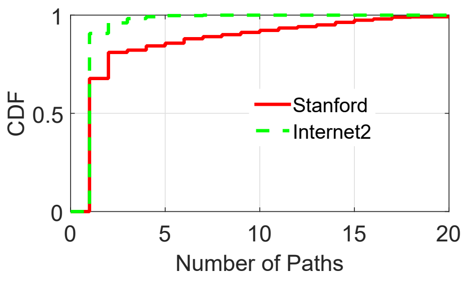


Figure 6: Distribution of the number of paths per inport-outport pair in the Stanford backbone and Internet2 networks.

$(\langle S_1, 1\rangle, \langle S_3, 2\rangle)$ as the index, the server would find two paths: one for $dst\_port = 22$ and the other for $dst\_port \neq 22$. The header of the packet would match the packet set of the first path. If the tag of the packet is the same with that of that path, the verification succeeds. Now consider that $R3$ fails, the packet will take the path of $S_1 \rightarrow S_3$. The tag would then be $[1||S_1||4] \sqcup [3||S_3||2]$, disagreeing with that of the path, and the verification fails.

Note that there may be multiple paths for an inport-outport pair, and each path corresponds to a specific header set. Thus, we should assume that the number of paths for an inport-outport pair should not be too large, so that the linear search used in Algorithm 3 is feasible. To validate this assumption, we construct the path table with the configuration files of the Stanford backbone network [4] and Internet2 [11]. As shown in Figure 6, the number of paths per inport-outport pair is relatively small, thereby validating the feasibility of linear search for these two networks.

4.3 Fault Localization

When the tag is different from the one in the path table, we know the path took by the corresponding packet is inconsistent with the control plane configuration. We are interested in which switches are to blame for such inconsistency, that is, localizing those switches that have faults in flow tables.

Take Figure 7 as an example, where Src sends a packet to Dst. Each switch has four ports numbered 1 through 4, and a "drop" port $\perp$. The correct path is $\langle 1, S1, 2\rangle$, $\langle 1, S2, 2\rangle$, $\langle 1, S4, 3\rangle$, while switch $S1$ is faulty and outputs the packet to a wrong port 4 instead of the right one 2, resulting in the actual path being $\langle 1, S1, 4\rangle, \langle 1, S3, 3\rangle, \langle 1, S6, \perp\rangle$. Even $S6$ drops the packet, we cannot simply blame $S6$ since the fault occurs at an upstream switch, *i.e.*, S_1.

To localize the faulty switches, we will first present a strawman approach, outline its limitation, and finally introduce the fault localization algorithm used by VeriDP.

Strawman Approach. Define a *hop* as a 3-tuple of the form $\langle input\_port, switch\_ID, output\_port\rangle$. That is, a hop encodes the forwarding behavior of a switch on a packet, including from which port the packet is received, and to which port the packet is forwarded. A path is then a ordered

Algorithm 4: PathInfer$(header, inport, outport, tag)$

```
Input: $inport/outport$: the input/output port of the packet;
       $header$: the header of the packet; $tag$: the tag of the packet.
Output: $pathset$: the set of all possible paths for the packet.
$pathset \leftarrow \{\}$; // all possible paths
$path \leftarrow$ GetPath$(inport, header)$; // the original path
$com\_path \leftarrow \{\}$; // common part with original path
foreach $hop \in path$ do
    $com\_path$.push_hop$(hop)$;
    if BF$(hop) \sqcap tag \neq$ BF$(hop)$ then
        break; // the path deviates from hop
while $\neg com\_path$.is_empty() do
    $dev\_hop \leftarrow com\_path$.pop_hop();
    $s \leftarrow dev\_hop.switch$;
    $x \leftarrow dev\_hop.input\_port$;
    foreach output port $y$ of switch $s$ do
        $dev\_path \leftarrow \{\langle x, s, y \rangle\}$;
        $inport' \leftarrow$ Link$(\langle s, y \rangle)$;
        $path' \leftarrow$ GetPath$(inport', header)$;
        foreach $hop \in path'$ do
            if BF$(hop) \sqcap tag \neq$ BF$(hop)$ then
                break; // dismiss this path
            $dev\_path$.push_hop$(hop)$;
            if $hop.output\_port = outport$ then
                $pathset \leftarrow pathset \cup (com\_path + dev\_path)$;
                break; // found a valid path
```

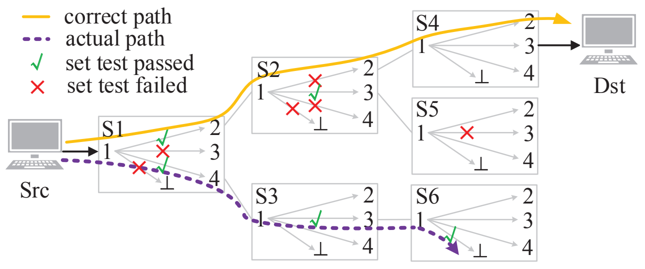


Figure 7: An example for fault localization. Each switch has four ports, and a port "⊥" representing packet drops. The path configured by the controller is $S1 \rightarrow S2 \rightarrow S4$. $S1$ falsely forwards the packet to port 4, and the path actually took by the packet is $S1 \rightarrow S3 \rightarrow S6$.

list of hops. We refer to the path computed according to network policies as the *correct path*, and the one actually traversed by the packet as the *real path*. The idea of our strawman approach is to first compute the correct path, and then for each hop of the correct path, test whether it also belongs to the real path. The test is based on the tag of the packet, which is a Bloom filter encoding all hops of the real path. If the switch has no faults, then the hop will forward the packet correctly, and this hop will pass the test; otherwise the switch should have forwarded the packet to a wrong port, and this hop will fail the test with large probability. Finally, the first hop that fails the test can be identified as a faulty switch.

One problem with the above strawman approach is that the false positives of Bloom filter may raise high localization error. Returning to the example in Figure 7, the hop $\langle 1, S1, 2 \rangle$ may falsely pass the set test due to false positives of Bloom filter, and the next hop $\langle 1, S2, 2 \rangle$ may fail the test. Then, we will falsely identify $S2$ as the faulty switch. Indeed, this problem can be mitigated if we reduce the false positive of Bloom filter by using more bits for Bloom filter. However, this would incur a large overhead on the header space.

The fault localization algorithm. To reduce the localization error without increasing header space overhead, we leverage the fact that most switches in the network are functioning well except some faulty ones. Thus, if a switch is identified as faulty, then we are expected to find a valid path from this switch towards the destination by looking up in downstream switches' flow tables. Specifically, we enumerate all possible output ports of the faulty switch, and from each of these output ports, we continue to test whether the next hops can pass the set testing.

Let us return to the example in Figure 7 and suppose we suspect that $S2$ is faulty. First, we enumerate all its output ports and know that only $\langle 1, S2, 3 \rangle$ can pass the test. Then, we continue to look up in $S5$'s flow table, and know the next hop is $\langle 1, S5, 3 \rangle$. However, this hop fails the test, meaning that $S2$ cannot be the faulty switch since there is no valid path from it to the destination. Then, we backtrace to the last hop of $S2$, *i.e.*, $S1$, and perform a similar process. This time, we can successfully find a valid path, which is just the real path of the packet.

The above method is summarized as Algorithm 4, which consists of two phases: (1) constructing the first half of the real path that is common with the correct one, and (2) constructing the second half of the real path that is different from the correct one. The first phase puts as many hops as possible into the common path com_path. The second phase backtraces to a previous hop in com_path each time, and tries to construct a valid path dev_path from one of its outports. When a valid dev_path is found, the concatenation of com_path and dev_path is put into the list of all possible paths $pathset$. This process continues until the com_path becomes empty. Section 6 will evaluate the probability that Algorithm 4 can find the real path.

4.4 Path Table Update

When the controller adds, deletes, or modifies a rule at a switch, we need to update the path table to reflect that change, in order to keep synchronized with the data plane. Rather than re-compute the whole path table, we show how to update the path table incrementally, such that only a small portion of the path table needs to be updated.

For simplicity, we only consider forwarding rules matching IP prefixes, while neglecting the in-bound and out-bound ACL rules, However, note that the incremental update can also be performed with ACL rules. Then, the transfer predicate $P_{x,y}$ can be reduced to $P_y = P_y^{fwd}$, where P_y^{fwd} is the port predicate for output port y. In addition, we only

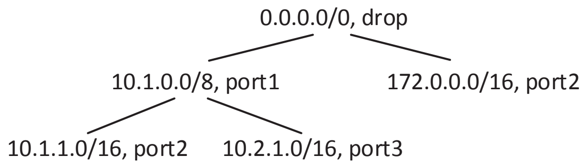


Figure 8: An example of forwarding rules organized as a tree.

consider rule addition and deletion, since rule modification can be regarded as deleting the old rule and then adding a new rule. The incremental path table update process consists of two phases: port predicate update and path entry update.

Port predicate update. This stage is an adaption of the port predicate update algorithm in [56]. It takes the newly added or deleted rule as input, and updates the port predicates accordingly.

First, define an IP forwarding rule as a 3-tuple $R = \langle prefix, match, outport \rangle$, where $prefix$ is the IP prefix to be matched, $match$ is the IP addresses that will actually be matched, and $outport$ is the output port of the matched packets. The difference between $prefix$ and $match$ will manifest in the following.

Similar to [56], we organize the forwarding rules as a forest: a rule R_j is a child of another rule R_i, if $R_i.prefix$ contains $R_j.prefix$. Figure 8 gives an example of such a forest. By longest match, the rule R with prefix $R.prefix = 10.1.0.0/8$ will match $R.match = 10.1.0.0/8 \backslash (10.1.1.0/16 \vee 10.2.1.0/16)$.

Different from [56], we add a virtual drop rule with zero prefix length, *i.e.*, $0.0.0.0/0$, and transform the forest into a tree, as shown in Figure 8. This will greatly simplify our treatment of transfer predicates for drop ports.

Then, the transfer predicate for output port x, *i.e.*, P_x, can be calculated as:

$$R_i.match = R_i.prefix \wedge (\vee_{R_j \text{ is a child of } R_i} \neg R_j.prefix)$$
$$P_x = \vee_{R_i.outport=x} R_i.match$$

Suppose a rule R_i with $R_i.outport = x$ is added, and let R_j be its parent with $R_j.outport = y$. Then, two port predicates need to be updated as:

$$P_x \leftarrow P_x \vee R_i.match$$
$$P_y \leftarrow P_y \wedge \neg R_i.match$$

Similarly, if R_i is deleted, two port predicates need to be updated as:

$$P_x \leftarrow P_x \wedge \neg R_i.match$$
$$P_y \leftarrow P_y \vee R_i.match$$

Path entry update. With port predicates updated, we are ready to update the path table. We only consider the addition of a new rule R_i, and deletion is much the same. Let $\Delta = R_i.match$, and view a transfer predicate as a set of headers. When R_i is added, P_x grows by adding Δ, and P_y shrinks by subtracting Δ. Then the path table update naturally consists of two parts:

- For each path that passes port y, we should update its header set by subtracting Δ. Let the path be p, then its header set should be updated by $p.headers \leftarrow p.headers \backslash \Delta$, If $p.headers$ becomes `False`, then we delete the path from the path table.

- This part is a little more complicated since we need to not only update existing paths, but also add new paths if there are. Let S be the switch where the rule is to be added, then we need to consider each header set that has reached the switch S in the recursive search of Algorithm 2. Let h be such a header set, and $p = (hop_1, hop_2, \ldots, hop_n)$ be the path traversed by this header set before reaching switch S. Then, we should first compute $h \leftarrow h \wedge \Delta$, then output h to port x of switch S, and continue the recursive search. If h reaches an edge port without being `False`, then we check whether the resultant path is already in the path table. If there is such a path q, then we update its header set by $q.header \leftarrow q.headers \vee h$; otherwise, we create a new path with header set h, and add it into the path table.

4.5 Traffic Sampling

Tagging and verifying every packet in the network can incur a large overhead. In this paper, we use a simple method which samples packets based on flows at entry switches. Each flow f is associated with a parameter $T_s^f > 0$, termed the *sampling interval*. The entry switch S of f maintains the last sampling instant t^f. For each packet received by S at time t, if $t - t^f > T_s^f$, S marks the packet and updates $t^f \leftarrow t$.

In order to determine the sampling interval, we introduce another per-flow parameter T_a^f, defined as the maximum inter-packet-arrival time for flow f. We aim to have a detection latency (from the time that a fault occurs and that it is detected) less than a threshold τ. To achieve this goal, T_s^f should be set to satisfy $T_s^f \le \tau - T_a^f$. To see why, consider the worst-case example in Figure 9. Packet 3 is sampled at time t_0, then a fault is experienced by the next back-to-back Packet 4 and all the following packets. Packet 7 arrives just before time $t_0 + T_s^f$, and is thus not sampled. After maximum inter-packet-arrival time T_a^f, Packet 8 arrives and is sampled, and the fault can be detected. By now, $T_s^f + T_a^f$ has elapsed since fault is first experienced by Packet 4. This is just the longest possible elapsed time to detect a fault.

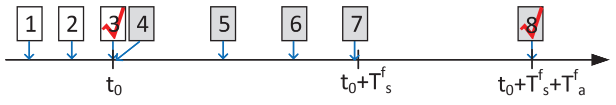


Figure 9: An example for sampling.

5. IMPLEMENTATION

Bloom filter. For Bloom filter operations, we use the approaches described by Kirsch and Mitzenmacher [38]. First, three hashes are constructed as $g_i(x) = h_1(x)+ih_2(x)$ for $i = 0, 1, 2$, where $h_1(x)$ and $h_2(x)$ are the two halves of a 32-bit Murmur3 hash of x [12]. Then, we use the first 4 bits of $g_i(x)$ to set the 16-bit Bloom filter for $i = 0, 1, 2$. A similar approach is used by Cassandra [10].

Packet format. For each data packet, VeriDP inserts three additional fields: `marker`, `tag`, and `inport`. Here, `marker` is a single bit carried in the IP TOS field, indicating whether the packet is sampled for verification; `tag` is a 16-bit Bloom filter encoding the path traversed by the packet, which is carried in the first VLAN tag; `inport` is a 14-bit identifier of the entry port (8 for switch ID and 6 for port ID), which is carried in the second VLAN tag[1]. Tag reports (sent by destination switches) are encapsulated with plain UDP packets.

VeriDP server. The VeriDP server is responsible for constructing the path table based on network configurations, verifying tag reports and localizing faulty switches based on the path table. For path table construction, we first generate transfer predicates (represented with BDDs) from OpenFlow rules. Then, we run Algorithm 2 to generate all possible paths, and calculate a tag for each path. We implement the tag verification based on Algorithm 3. When iterating over possible paths for the inport-outport pair of a packet, we need to determine whether the header of the packet belongs to the header set of the path (*i.e.*, $header \prec p.headers$, Line 2). To test this, we first generate a BDD representation for the packet header, and then intersect this BDD with the header set (which is also a BDD). The packet header belongs to the header set if the intersection is not `False`.

VeriDP pipeline (Software). We implement the VeriDP pipeline with Open vSwitch [7]. The VeriDP pipeline functions after all actions have been executed on a packet, and before the packet is sent to the output port. We currently use the TCP 5-tuple to identify a flow, and associate with each flow a sampling instant (refer to Section 4.5). We simply use a hash table to store the sampling instants of all active flows.

VeriDP pipeline (Hardware). We implement the VeriD-P pipeline with ONetSwitch [32, 31], a hardware programmable switch that we built (available online at [13]). As shown in Figure 10, we implement both the VeriDP and OpenFlow pipeline with the FPGA resource. Due to the limited resource of FPGA, we use an array to record a limited number of active flows. Besides a sampling instant, each flow entry also contains a last-hit instant, which helps us identify active flows.

6. EVALUATION

[1]Double VLAN tags are supported by 802.1ad [1]; each tag has a 2-byte Tag Control Information (TCI), which is used to carry our data.

Table 2: Path table statistics.

Setup	# entries	# paths	avg. path len.	time (s)
Stanford	26K	77K	4.85	4.32
Internet2	43K	50K	2.89	3.22
FT($k = 4$)	448	448	3.79	0.10
FT($k = 6$)	4176	4176	4.23	0.26

6.1 Experiment Setup

We use both emulated and real switches for experiments. Otherwise specified, we emulate networks with Mininet [5], consisting of Open vSwitch (OVS [7]) instances, and use Floodlight as the controller [3]. The verification server runs on a desktop with an Intel Core i7 CPU 3.6GHz and 32GB Memory. We use the following four topologies for experiments.

- Stanford backbone. The Stanford backbone network [4] consists of 16 Cisco routers and 10 layer-2 switches. A portion of the topology is shown in Figure 11. Overall, there are 757,170 forwarding rules and 1,584 ACL rules.
- Internet2. The Internet2 topology [11] consists of 9 Juniper routers. As the ACL rules are not publicly available, we only use the 126,017 IPv4 forwarding rules.
- Fat tree. We emulate fat tree topologies which represent medium-sized networks. In prior to experiments, we let the emulated hosts ping each other in order to populate the switches' flow tables with shortest-path forwarding rules.
- Single ONetSwitch. We use a single ONetSwitch that implements both OpenFlow and VeriDP to test the overhead of VeriDP on data plane.

For Stanford and Internet2, we translate the configuration files to equivalent OpenFlow rules, and install them at Open vSwitches with Floodlight. The translation algorithm is customized based on [57], which was originally written for the Beacon controller.

Since there are loops in the Stanford and Internet2 topologies, we remove all the loops in prior to generating the path table. Specifically, when searching for all paths using Algorithm 2, if a port is visited for the second time for a path, we will not continue to search that path. Thus, initially the control plane and data plane are inconsistent: the control plane is loop-free, while the data plane contains loops. The path table statistics are summarized as Table 2.

6.2 Function Test

For function test, we generate faults in the emulated Stanford backbone network, and test whether VeriDP can detect these faults and pinpoint the faulty switches.

Black hole. In Figure 11, there is a flow (red solid line) from *boza* to *coza* with destination address 172.20.10.33. Similar

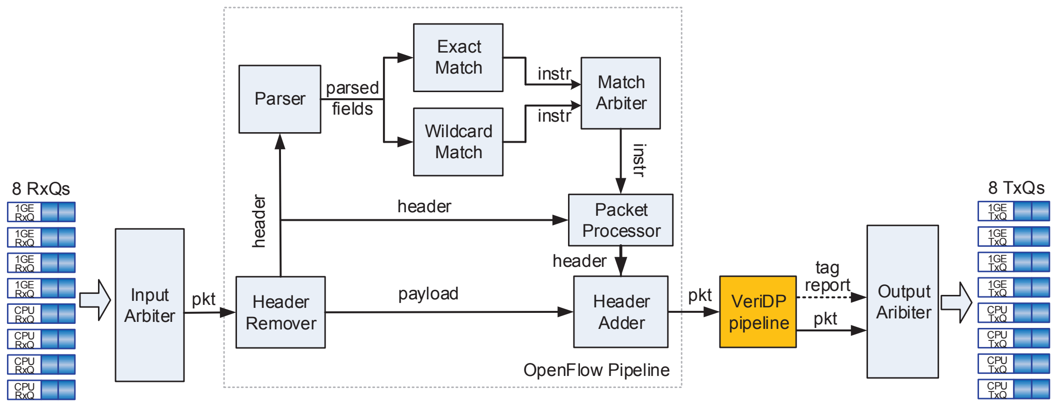
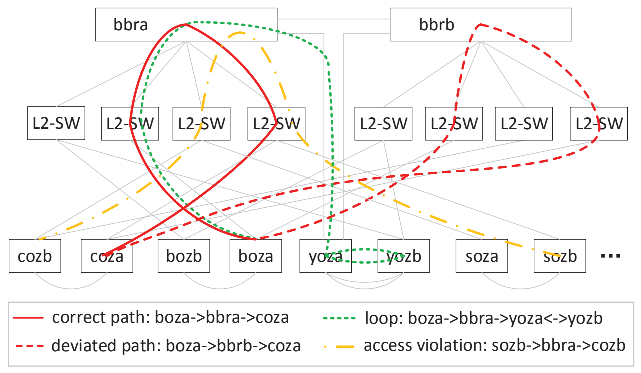


Figure 10: Hardware implementation of the VeriDP pipeline based on ONetSwitch.



bbra
bbrb
L2-SW
cozb
coza
bozb
boza
yoza
yozb
soza
sozb
correct path: boza->bbra->coza
deviated path: boza->bbrb->coza
loop: boza->bbra->yoza<->yozb
access violation: sozb->bbra->cozb

Figure 11: Function test on the Stanford backbone network (only part of the topology is shown).

to ATPG, we deliberately create a fault by modifying the action of the forwarding rule in *boza* that matched $dst\_ip =$ 172.20.10.32/27 with a drop action. Then, the flow will be dropped at *boza*. VeriDP immediately detects the fault, and localizes the faulty switch *boza*.

Path deviation. We continue to create a fault that causes the previous flow to deviate from the original path, by modifying the same rule. This time, we replace the action to forward towards *bbrb*, resulting in another path from *boza* to *coza* (red dash line). VeriDP immediately detects the fault, and recover the real path, thereby localizing the faulty switch *boza*.

Access violation. To simulate an access violation, we delete an ACL rule in *sozb* that denies all packets with destination address 10.0.0.0/8. Then, we send packets with destination address 10.63.16.0/20 from *sozb*, and they are received at *cozb* (the yellow dash dot line). VeriDP successfully detects the fault and reconstructs the path took by the flow.

Loop. Recall that we removed all loops when generating the path table at the control plane, while the data plane still uses the original configuration that contains loops. We will test whether VeriDP can detect these loops. We inject a flow with destination address 172.26.4.152 from *boza*, which loops

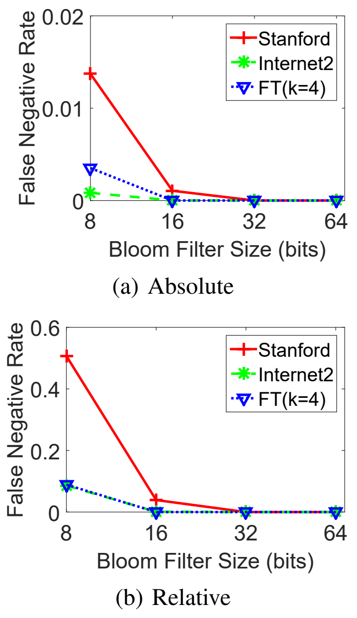


Figure 12: False positive rate vs. Bloom filter size.

between *yoza* and *yozb* (green dash line). Since the host with address 172.26.4.152 is connected to a port of *yoza*, then each time *yoza* receives the packet, it will forward the packet to that port. Since the port is an edge port (connected with non-routers), then multiple tag reports would be sent to the VeriDP server. Only the first tag report passes the verification, while all others fail.

6.3 Detection and Localization Performance

Detection accuracy. The inconsistency detection of VeriDP has no false positives. The reason is simple: if the forwarding path of a packet is the same with that in the path table, then the tag will be exactly the same, and the verification will pass. False negatives happen when a packet traverses a path different from the one in the path table, while satisfying two conditions: (1) the packet arrives at the destination port,

Table 3: Probability of successful fault localization when verification fails for fat tree $k = 4$ and $k = 6$.

Setup	# failed verif.	# recovered paths	localization prob.
FT ($k = 4$)	2,527	2,505	99.2%
FT ($k = 6$)	7,148	6,902	96.6%

and (2) the tag of the packet is the same with the one in the path table. The reason why condition (1) is necessary is that if the packet arrives at a wrong port (including drop ports), the packet header will definitely not match the header set of any path for the input-output pair, and thus the verification will fail.

To evaluate the detection accuracy, we randomly select paths in the path table, and generate a packet for each path. Then, for each packet we randomly select a switch on its forwarding path, and output the packet to a port different from the original one, in order to simulate a fault. Let n be the number of total packets we select, $n1$ be the number of packets that arrive at the destination port, and $n2$ be the number of packets that arrive at the destination port and the tag is same with that in the path table. Then, we define the *absolute false negative rate* as $n2/n$, and the *relative false negative rate* as $n2/n1$. To evaluate the impact of Bloom filter size on false negative rate, we vary the Bloom filter size from 8 bits to 64 bits, and measure the false negative rate for Stanford, Internet2, and fat trees ($k = 4$).

As shown in Figure 12, the absolute false negative rate is rather small for $k = 6$. Specifically, the absolute false negative rate is only around 0.1% for Stanford backbone network. The relative false negative rate is higher, but also decreases to zero for k larger than 32.

Localization accuracy. We continue to evaluate the localization accuracy using fat tree topologies. First, we select a random rule from a random switch, and change its output port to a different one. Then, we let all hosts ping each other, collect the tag reports, and perform tag verification. If the verification fails, we try to recover the actual path took by the ping packet. Table 3 reports the number of failed verifications, the number of recovered paths, and the localization probability. We can see that our fault localization algorithm can recover the real paths took by packets with a high probability when verification fails: 99.2% and 96.6% for fat tree $k = 4$ and $k = 6$, respectively.

6.4 Verification Time

We measure the time to verify a tag report, with the Stanford backbone and Internet2 topology. For each topology, we generate a test packet for each path in the path table. Then, we inject these test packets into the network and collect the tag reports. After that, we run the verification algorithm for each tag report for 10^4 times, and record the time. Figure 13 shows the verification time for a single tag report, which is calculated by taking an average over the 10^4 verifications. We can see that the verification time is

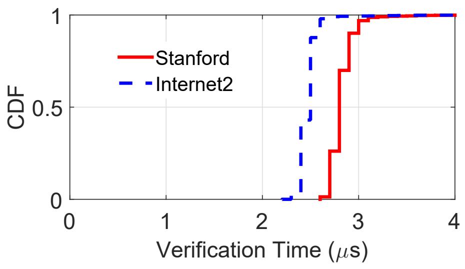


Figure 13: The time to verify a tag report on VeriDP server.

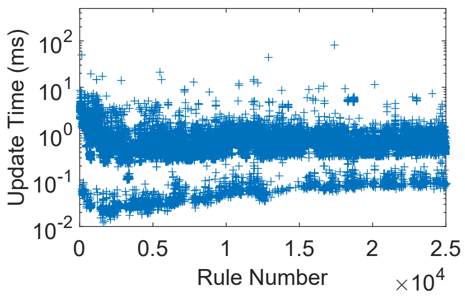


Figure 14: The time to add a new rule to a router of Internet2.

in-between 2 and 3 μs, which translates into a verification throughput of up to 5×10^5 per second. Since the verification is still single-threaded without optimization, we expect a higher throughput with multi-threading in the future.

6.5 Path Table Update Time

We continue to evaluate the incremental update algorithm using the Internet2 topology, which consists of 9 switches. First, we populate the flow tables of 8 switches with OpenFlow rules. For the remaining switch, we install rules into its flow table one-by-one, and measure the time to update the path table. There are more than 28,000 rules for this switch in total. The update time for each rule is shown in Figure 14. We can see that for most rules, the time to update the path table is less than 10ms. This update time should be sufficient, considering the time to update the data plane is at a scale of several milliseconds [34, 50].

6.6 Dataplane Overhead

Our implementation of VeriDP on the FPGA-based hardware switch (*i.e.*, ONetSwitch) can process packets at a line speed (1Gbps). To measure the processing delay, we send packets to one port of the switch, receive them from another one, and record the CPU cycles c used in this process. As the FPGA has a frequency of 125MHz, the delay can be calculated as $T = c \times 0.008\mu s$.

Table 4 reports the delay of the native OpenFlow pipeline (T_1), the delay of the VeriDP sampling module (T_2) and its overhead (T_2/T_1), the delay of the VeriDP tagging module (T_3) and its overhead (T_3/T_1). We can see that the delay of

Table 4: Processing delay of the VeriDP pipeline and native OpenFlow pipeline on the hardware SDN switch.

	Packet Size (Bytes)				
	128	256	512	1024	1500
Native (μs)	4.32	7.33	19.89	26.21	36.68
Sampling (μs)	0.15	0.14	0.14	0.14	0.15
Overhead	3.52%	1.96%	0.74%	0.55%	0.41%
Tagging (μs)	0.27	0.26	0.27	0.26	0.27
Overhead	6.29%	3.60%	1.37%	1.01%	0.74%

VeriDP sampling and tagging modules are around $0.15\mu s$ and $0.27\mu s$, respectively. For packet size of 512 Bytes, the overhead of these two modules is 0.74% and 1.37%, respectively. Note that tagging is necessary for each hop, while sampling is only performed by entry switches. Thus, non-entry switches only incur the tagging overhead.

7. RELATED WORK

Recently there have been many verification tools proposed for SDN. Some tools debug controller softwares or applications [24, 51, 55, 20, 26, 19], while others check the correctness of network policies [16, 43, 36, 35, 37, 56, 60, 58, 17, 42, 27, 48]. A limitation with these tools is that they can only ensure network correctness (*i.e.*, controller softwares or network policies) at the controller side, while cannot guarantee the correctness of switch configurations (*i.e.*, rules at flow tables), or data plane behaviors (*i.e.*, packet forwarding). As shown in previous work [46, 41], data plane configurations may deviate from those on the control plane.

Data plane testing tools [57, 41] try to check the correctness of rules at switches' flow tables. ATPG [57] generates the minimum number of probe packets that can trigger all rules in the network, and verifies whether all these probe packets can be sent and received by end hosts. Since ATPG only checks packet receptions, it can only verify that pairwise reachability are ensured by rules of switches, while cannot verify other properties like middlebox traversals that require inspections on packet trajectories. Moreover, real packets may experience different forwarding behaviors with probe packets, making the verification results less convincing. Monocle [41] 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. The probe packet should be constructed in such a way that it can only trigger the rule under test, while not being affected by other rules in the switch. A similar approach, RuleScope [23], also monitors the flow table integrity by sending probes. The difference is that it can detect priority swaps of rules as well. There are several problems with 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. As a result, they can only verify relatively steady configurations and would have issues tracking down frequent data-planes updates or reconfigurations. In addition, similar to ATPG, the probe packets may experience different forwarding behaviors with real production packets.

Packet trajectory tracers [59, 45, 54, 14] try to record packet trajectories by letting switches imprint specific information into packet headers. However, packet trajectories by themselves are not very useful unless we know whether they are correct. In contrast, VeriDP not only traces packet trajectories, but also enables the controller to reason about whether the trajectories are compliant with high-level policies. Moreover, they either require a large number of rules at switches [59, 45], or highly depend on the datacenter structures [54]. Different from the above approaches, NetSight [29] records detailed forwarding information of real packets using postcards. However, since each packet will trigger a postcard at each hop, NetSight will incur a huge volume of postcards traffic on the data plane.

Switch software debuggers [39, 25] use symbolic execution to test the software components of SDN switches. They only carry out static testing of switch software codes, while cannot detect hidden bugs that only show at runtime, *i.e.*, flow table or packet forwarding faults. In contrast, VeriDP can potentially capture these runtime bugs by constantly monitoring the policy incompliance of switches.

8. CONCLUSION AND FUTURE WORK

This paper presented VeriDP, a new tool to monitor the consistency of control plane and data plane in SDN. VeriDP used the path table abstraction at the control plane and packet tagging at the data plane, in order to detect inconsistency. The Bloom-filter-based tagging method allowed VeriDP to pinpoint the faulty switches that caused the inconsistency. We implemented VeriDP on both software and hardware switches to demonstrate its feasibility, and used real network topologies and policies to test its functions.

Our future work includes: (1) incorporating header rewrites into the current VeriDP framework, in order to support actions that need to modify packet headers, and (2) designing a method that can automatically repair the flow table of a faulty switch, in order to resolve the inconsistency with minimal human interaction.

Acknowledgements

We would like to thank our shepherd Laurent Vanbever for helping us revise the paper, and all the CoNEXT reviewers for their valuable comments. We also thank Hongkun Yang from Google for inspiring discussions with him, and Jiamin Liu, Peng Zheng, Zhimin Gong for their work at the early stage. This work is supported by the NSFC (No. 61402357, 61272459, and 61672425), the China Postdoctoral Science Foundation, the Program for New Century Excellent Talents in University (NCET-13-0450), and the Fundamental Research Funds for the Central Universities.

9. REFERENCES

[1] 802.1ad - Provider Bridges. http://www.ieee802.org/1/pages/802.1ad.html.

[2] Dpctl Documentation. https://github.com/CPqD/ofsoftswitch13/wiki/Dpctl-Documentation.

[3] Floodlight OpenFlow Controller. http://floodlight.openflowhub.org/.

[4] Hassel, the header space library. https://bitbucket.org/peymank/hassel-public.

[5] Mininet. http://mininet.org/.

[6] Open Network Install Environment (ONIE). http://onie.org/.

[7] Open vSwitch. http://openvswitch.org/.

[8] OpenFlow Switch Specification Version 1.5.1. https://www.opennetworking.org/sdn-resources/technical-library.

[9] Ryu OpenFlow Controller. http://osrg.github.io/ryu/.

[10] The Apache Cassandra Project. http://cassandra.apache.org/.

[11] The Internet2 Observatory. http://www.internet2.edu/research-solutions/research-support/observatory/.

[12] The Murmur3 hash function. https://code.google.com/p/smhasher/wiki/MurmurHash3.

[13] The ONetSwitch SDN platform. http://onetswitch.org.

[14] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN traceroute: Tracing SDN forwarding without changing network behavior. In *HotSDN*, 2014.

[15] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow. Openvirtex: Make your virtual sdns programmable. In *HotSDN*, 2014.

[16] E. Al-Shaer and S. Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *ACM workshop on Assurable and usable security configuration*, 2010.

[17] V. Altukhov, V. Podymov, V. Zakharov, and E. Chemeritskiy. Vermont-a toolset for checking sdn packet forwarding policies on-line. In *IEEE Modern Networking Technologies (MoNeTeC)*, 2014.

[18] M. Antikainen, T. Aura, and M. Särelä. Spook in Your Network: Attacking an SDN with a Compromised OpenFlow Switch. In *Secure IT Systems*. 2014.

[19] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *ACM PLDI*, 2014.

[20] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker. An assertion language for debugging sdn applications. In *HotSDN*, 2014.

[21] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. OConnor, P. Radoslavov, W. Snow, et al. ONOS: towards an open, distributed SDN OS. In *HotSDN*, 2014.

[22] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[23] 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 *IEEE INFOCOM*, 2016.

[24] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. In *USENIX NSDI*, 2012.

[25] M. Dobrescu and K. Argyraki. Software dataplane verification. In *USENIX NSDI*, 2014.

[26] R. Durairajan, J. Sommers, and P. Barford. Controller-agnostic sdn debugging. In *ACM CoNEXT*, 2014.

[27] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *USENIX NSDI*, 2015.

[28] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291, 2011.

[29] N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX NSDI*, 2014.

[30] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, et al. Leveraging SDN layering to systematically troubleshoot networks. In *HotSDN*, 2013.

[31] C. Hu, J. Yang, Z. Gong, S. Deng, and H. Zhao. Desktopdc: setting all programmable data center networking testbed on desk. *ACM SIGCOMM Computer Communication Review*, 44(4):593–594, 2015.

[32] C. Hu, J. Yang, H. Zhao, and J. Lu. Design of all programmable innovation platform for software defined networking. In *Open Networking Summit*, 2014.

[33] T. Inoue, T. Mano, K. Mizutani, S.-i. Minato, and O. Akashi. Rethinking packet classification for global network view of software-defined networking. In *IEEE ICNP*, 2014.

[34] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. CacheFlow: Dependency-aware rule-caching for software-defined networks. In *ACM SOSR*, 2016.

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

[36] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

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

[38] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[39] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT way for openflow switch interoperability testing. In *ACM CoNEXT*, 2012.

[40] M. Kuzniar, P. Peresini, and D. Kostić. Providing reliable fib update acknowledgments in sdn. In *ACM CoNEXT*, 2014.

[41] M. Kuzniar, P. Peresini, and D. Kostic. Monocle: Dynamic, fine-grained data plane monitoring. In *ACM CoNEXT*, 2015.

[42] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *USENIX NSDI*, 2015.

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

[44] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, et al. Composing software defined networks. In *USENIX NSDI*, 2013.

[45] S. Narayana, J. Rexford, and D. Walker. Compiling path queries in software-defined networks. In *HotSDN*, 2014.

[46] P. Peresini, M. Kuzniar, and D. Kostic. What You Need to Know About SDN Flow Tables. In *PAM*, 2015.

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

[48] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In *ACM POPL*, 2016.

[49] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, 2012.

[50] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An open framework for OpenFlow switch evaluation. In *Passive and Active Measurement*, pages 85–95, 2012.

[51] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, et al. Troubleshooting blackbox SDN control software with minimal causal sequences. In *ACM SIGCOMM*, 2014.

[52] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. M. Parulkar. Can the production network be the testbed? In *OSDI*, 2010.

[53] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *ACM CoNEXT*, 2014.

[54] P. Tammana, R. Agarwal, and M. Lee. CherryPick: Tracing packet trajectory in software-defined datacenter networks. In *SOSR*, 2015.

[55] A. Wundsam, D. Levin, S. Seetharaman, A. Feldmann, et al. OFRewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference*, 2011.

[56] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, 2013.

[57] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *ACM CoNEXT*, 2012.

[58] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *USENIX NSDI*, 2014.

[59] H. Zhang, C. Lumezanu, J. Rhee, N. Arora, Q. Xu, and G. Jiang. Enabling layer 2 pathlet tracing through context encoding in software-defined networking. In *HotSDN*, 2014.

[60] S. Zhang and S. Malik. Sat based verification of network data planes. In *Automated Technology for Verification and Analysis*. 2013.