# Stick to the Script: Monitoring The Policy Compliance of SDN Data Plane

Peng Zhang[†‡], Hao Li[†], Chengchen Hu[†], Liujia Hu[†], and Lei Xiong[†]
[†]Department of Computer Science and Technology, Xi'an Jiaotong University
[‡]Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory

## ABSTRACT

Software defined networks provide new opportunities for automating the process of network debugging. Many tools have been developed to verify the correctness of network configurations on the control plane. However, due to software bugs and hardware faults of switches, the correctness of control plane may not readily translate into that of data plane. To bridge this gap, we present VeriDP, which can monitor "whether actual forwarding behaviors are complying with network configurations". Given that policies are well-configured, operators can leverage VeriDP to monitor the correctness of the network data plane. In a nutshell, VeriDP lets switches tag packets that they forward, and report tags together with headers to the verification server before the packets leave the network. The verification server pre-computes all header-to-tag mappings based on the configuration, and checks whether the reported tags agree with the mappings. We prototype VeriDP with both software and hardware OpenFlow switches, and use emulation to show that VeriDP can detect common data plane fault including black holes and access violations, with a minimal impact on the data plane.

## 1. INTRODUCTION

In traditional networks, when a fault (*e.g.*, routing black hole) occurs in the network, it will be firstly noticed by some end hosts that may become unreachable. Then, customers complain and issue tickets to the network operators, who use simple tools like ping and traceroute to localize the fault and resolve it. The above process lacks automation, and inevitably incurs a long service downtime.

Since networks know every single detail of a packet's lifetime, why not let themselves raise alters to operators, instead of end hosts or customers? There are many potential benefits by letting networks take an active role in network monitoring and debugging. First, by automatically raising alters, operators can resolve the faults more efficiently, thereby reducing the network downtime. Second, some faults (*e.g.*, access violation) that may not be explicitly noticed by any end hosts can be captured by networks. Finally, networks can provide more useful information for the operators to pinpoint the fault location.

One reason that networks keep passive in monitoring and debugging may be the distributed nature of networks: no single switch can reason about the global policies in traditional networks. For example, consider a packet is dropped, the switch does not know whether it is due to faults or access control policies. As another example, if a packet is received twice by a switch, it is a fault of loop in most cases, while it also can happen with a normal policy in flexible middle-box traversal scenarios [8].

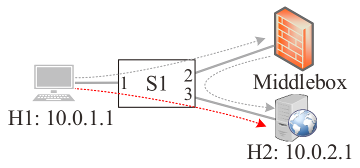
We observe that networks have the potential to take a more active role in the monitoring in the context of SDN. First, the SDN controller knows the global network policy, *i.e.*, how the data plane should behave, and there are many tools to guarantee the correctness of the network policy, either off-line [19, 15, 26] or on-the-fly [16, 14, 24]. Secondly, switches can record and report packet forwarding behaviors to the controller, through standard south bound interfaces (*e.g.*, OpenFlow [20]). By comparing the packet forwarding behaviors to the global network policy, the controller is in a good position to detect faults of the data plane.

Previous efforts on automatic network debugging are mostly focused on checking correctness of network configurations [19, 15, 16, 14, 24, 26]. However, even the controller and configurations are correct, the data plane may still experience faults due to switch software bugs [17], hardware failures [25], or malicious attacks [22]. Existing data plane verification tools either solely check reachability and thus miss path information [25], use probe packets that can be poor indicators of real traffic [25, 6], require a large number of flow rules [27, 21], depend on specific data center topology [23], or incur too much data plane traffic [9, 10].

Noted of the above limitations, we propose VeriDP, a new tool that can monitor the policy compliance of SDN data plane. In contrast to path tracers that solely rely on switches to imprint packet paths, VeriDP combines it with the network policies/configurations on the control plane. This combination delivers the following benefits. (1) With the network policy at hand, VeriDP can distinguish packet drops due to access violation from black holes, and strategic multi-traversals [8] from infinite loops. (2) It is not necessary to optimize the path encoding method so as to fit the path info into the limited header space as in [27, 21, 23], since the controller already knows the correct path, and the only task is to judge whether the path taken by packet is the same with it.

The basic idea of VeriDP is quite simple. The controller pre-compute a *path table* which records all mappings from packet headers to forwarding paths. When a packet enters the network, the entry switch decides whether to mark it according to some sampling strategies. If a packet is marked, each switch *en route* tags it with the forwarding information. Before a marked packet leaves the network, the exit switch reports its header and tag to the controller. The controller verifies whether the information encoded in the tag

*ANCS'16, March 17–18, 2016, Santa Clara, California, USA.*
© 2016 ACM. ISBN 978-1-4503-4183-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2881025.2881038



Flow Rules at S1:

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

**Figure 1: Middlebox traversal example. The security policy of** $H1 \rightarrow Middlebox \rightarrow H2$ **is violated.**

is the same with the path that the packet should take according to the path table.

Indeed, packet tagging is not a new idea for tracing packet trajectories [27, 21, 23]. VeriDP differs from them in that it is not trying to encode path information into packet headers for the receivers to decode it. Rather, VeriDP uses control-plane policies to "infer" paths, and use packet tags to "test" the correctness of real paths. The advantage is that VeriDP does not need complicated encoding methods or a large number of flow entries, in order to compress path info into limited header space.

Our contribution is two-fold:

- We propose VeriDP, a new tool to monitor the policy compliance of SDN data plane, *i.e.*, "whether packet forwarding behaviors agree with the policy configured by the controller".
- We implement VeriDP on software- and hardware-based data plane, and demonstrate that it can detect common faults like black holes, access violation, loops, while incurring minimal overhead on the data plane.

In the rest of this paper, we will first give our notion of policy compliance (§2), and then present the design of VeriDP (§3). We continue to test the function of VeriDP, and evaluate its performance (§4). After discussing some related work (§5), we conclude the paper (§6).

## 2. INTRODUCING POLICY COMPLIANCE

### 2.1 Observations

Before introducing our notion of policy compliance, we first elaborate the fact that verifying the policy compliance of data plane necessitates checking packet paths with real traffic.

**Path Check Is Important.** Pairwise reachability is a key invariant for a network. However, only checking reachability is not enough to reveal data plane faults. Consider an example of middlebox traversal. As shown in Figure 1, rules at switch $S1$ indicate that traffic from the client $H1$ to the server $H2$ must go through the middlebox. To test all these rules based on reachability, we can send two probe packets $H1 \rightarrow H2$ and $H2 \rightarrow H1$. They will follow the path of $H1(H2) \rightarrow S1 \rightarrow M \rightarrow S1 \rightarrow H2(H1)$, respectively, and thus trigger all rules in this network. Now, consider that the high-priority rules $R1$ and/or $R2$ fail, then the probe packets will take the path of $H1(H2) \rightarrow S1 \rightarrow H2(H1)$ instead. However, probe packets will still be received as normal, thus missing the faults. This example shows that to monitor the policy compliance of data plane, we need to check the paths of packets, instead of only checking pairwise reachability.

**Real Packets Are Necessary.** Verification using probe packets can only verify that the forwarding paths of probe packets agree with the rule. It does not necessarily mean the forwarding paths of real traffic do. For example, consider an ACL rule that only permits HTTP traffic from IP address 10.0.0.1:

match: $src\_ip = 10.0.0.1, dst\_port = 80,$ action="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 a ill-inserted rule:

match: $src\_ip = 10.0.0.1, dst\_port = *,$ action="ALLOW"

The probe packet can still be received. However, Non-HTTP traffic, *e.g.*, SSH, from 10.0.0.1 will also be allowed, violating the controller's policy. This is because the probe packets cannot exhaust all possibilities in the header space in order to detect such ill-inserted rules. Thus, to monitor the policy compliance of data plane, we still need to inspect the real packets.

### 2.2 Policy Compliance Model

**Notations.** A port $p$ is defined as a pair $\langle SwitchID, PortID\rangle$, where $PortID \in \{1, 2, 3, \ldots, n, \perp\}$ is the local port ID, and $\perp$ represents the dropping port. A header $h$ is defined as a point in the $\mathcal{H} = \{0, 1\}^L$ space. A header set $H$ is defined as a subset $h \subset \mathcal{H}$. A flow $f$ is defined as a pair $\langle h, p\rangle$, where $h$ is the header of the flow, and $p$ is the port where the flow enters the network. A rule $r$ is defined as a tuple $\langle p_1, H, p_2\rangle$, meaning that packets received from port $p_1$ with header $h \in H$ should be forwarded to port $p_2$. For a dropping rule, $p_2 = \langle SwitchID, \perp\rangle$. A link can be seen as a special kind of rule $\langle p_1, \mathcal{H}, p_2\rangle$, meaning that packets forwarded to port $p_1$ of one switch will be received at port $p_2$ of another switch.

**Packet Path.** When a packet $pkt$ of flow $f$ is received at port $p_1^{in}$ of $S_1$, $S_1$ looks up in its flow table. When the first rule $r = \langle p_1^{in}, H, p_1^{out}\rangle$ satisfying $h \in H$ is found, $S_1$ forwards $pkt$ to port $p_1^{out}$, and applies the link rules so that $pkt$ is received at port $p_2^{in}$ of another switch $S_2$. This process continues until $pkt$ reaches an output port $p_n^{out}$ of switch $S_n$, such that either $p_n^{out}$ is connected to an end host, or it is a dropping port. The path of flow $f$ is defined as the sequence of traversed ports, *i.e.*, $Path(\mathcal{R}, f) = \langle p_1^{in}, p_1^{out}, p_2^{in}, \ldots, p_n^{out}\rangle$.

Let $\mathcal{R}$ be the set of all rules in the network (including the link rules), and $\mathcal{R}'$ be the counterpart that is actually enforced by switches. The policy compliance of data plane is defined as follows.

DEFINITION 1. *The data plane is said to be policy compliant iff $Path(\mathcal{R}', f) = Path(\mathcal{R}, f)$ for every flow $f$ in the network, where $\mathcal{R}$ and $\mathcal{R}'$ are the set of rules configured by the controller and enforced by switches, respectively.*

VeriDP is aimed to verify the policy compliance of the SDN data plane according to the above definition. Note that there are cases that $\mathcal{R}' \neq \mathcal{R}$ but $Path(\mathcal{R}', f) = Path(\mathcal{R}, f)$ for all $f$. We do not consider it as a fault since the forwarding behaviors remain the same. Specifically, Definition 1 allows us to detect common faults on the data plane, including black holes, access violation, and loops:

**Black Holes.** In this case, there exists a flow $f$, such that $Path(\mathcal{R}', f) = \langle p_1^{in}, p_1^{out}, p_2^{in}, \ldots, p_m^{in}, \perp\rangle$, meaning that the flow is dropped by a switch that receives $f$ from port $p_m^{in}$. Suppose $f$ is destined to an host connected to port $p_n^{out}$, then we should have $Path(\mathcal{R}, f) = \langle p_1^{in}, p_1^{out}, p_2^{in}, \ldots, p_n^{in}, p_n^{out}\rangle \neq Path(\mathcal{R}, f)$.

**Access Violation.** In this case, there exists a flow $f$, such that $Path(\mathcal{R}', f) = \langle p_1^{in}, p_1^{out}, p_2^{in}, \ldots, p_m^{in}, p_m^{out}\rangle$, where $p_m^{out}$ is connected to an end host that $f$ is forbidden to reach. Then, we should have $Path(\mathcal{R}, f) = \langle p_1^{in}, p_1^{out}, p_2^{in}, \ldots, p_n^{in}, \perp\rangle$, where the switch with $p_n^{in}$ should drop $f$. Obviously, $Path(\mathcal{R}', f) \neq Path(\mathcal{R}, f)$.

**Loops.** In this case, there exists a flow $f$, such that the length of $Path(\mathcal{R}', f)$ exceeds the maximum TTL, say $MaxTTL$. On

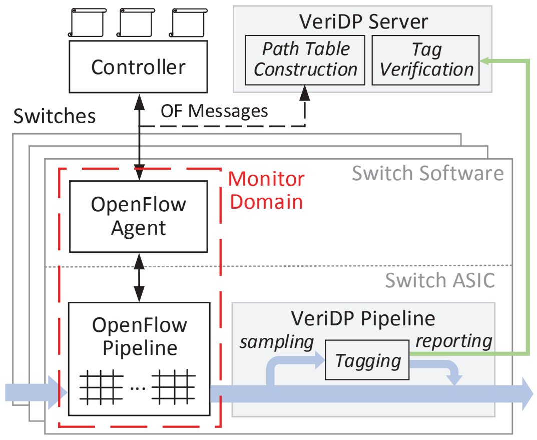


**Figure 2: System architecture. The shaded components belong to VeriDP; those within the dashed rectangle are the components that VeriDP monitors.**

the other hands, the length of $Path(\mathcal{R}, f)$ should be less than $MaxTTL$, and thus $Path(\mathcal{R}', f) \neq Path(\mathcal{R}, f)$.

## 3. DESIGN

As shown in Figure 2, VeriDP consists of two major components: the *VeriDP pipeline* on the data path, the *VeriDP server* on the control plane. The pipeline is responsible for sampling, tagging, reporting packets to the VeriDP server. The server intercepts the bidirectional OpenFlow messages exchanged between the controller and switches, in order to construct the *path table*, which records all header-to-tag mappings. With the path table, the server verifies reported packets sent from 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 through table lookups.

### 3.1 VeriDP Pipeline

The VeriDP pipeline is responsible for generating tags for packets at entry switches, updating tags for packets at core switches, and reporting packet headers and tags to the controller at exit switches. The VeriDP pipeline is implemented in a switch's fast path, separated from the OpenFlow pipeline. The reason is avoid faults caused by OF flow tables to propagate into the tagging module. Since a typical switch can contain a cascade of flow tables, each of which may hold thousands flow entries, flow entries used for tagging may be override by other rules, replaced when flow table is full, and even incorrectly modified/deleted by applications.

The VeriDP pipeline processing is shown in Algorithm 1. The entry switch initializes the packet tag to zero, and the ttl to the maximum path length (Line 1-3). Each switch updates the tag as:

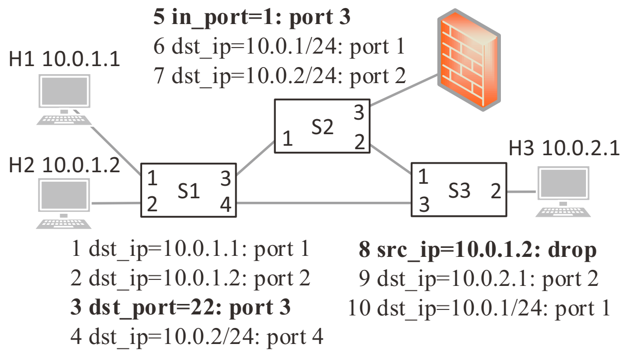
$$tag = tag \oplus \texttt{hash}(inport||switchID||output) \quad (1)$$

, and decrements the ttl value by one (Line 4-5). When the packet is output to an edge port connected with an end host, output to the dropping port $\perp$, or its ttl hits zero, the switch sends a *tag report* to the server (Line 6-7). Here, a tag report is a 4-tuple $\langle inport, outport, header, tag\rangle$, where $inport/outport$ are 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. One thing to note is that switches should send tag reports for dropped and looped packets. This is necessary to ensure the visibility of verification server into black holes and loops.

**Algorithm 1:** $\texttt{Tag}(S, x, y, p)$

**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 $p.tag \leftarrow p.tag \oplus \texttt{hash}(x||S||y)$; // 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     Report$(inport, \langle S, y\rangle, p.header, p.tag)$; // send report
```



5 in_port=1: port 3
6 dst_ip=10.0.1/24: port 1
7 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
1
2
3
4
1
2
3
1
3
2
1 dst_ip=10.0.1.1: port 1
2 dst_ip=10.0.1.2: port 2
3 dst_port=22: port 3
4 dst_ip=10.0.2/24: port 4
8 src_ip=10.0.1.2: drop
9 dst_ip=10.0.2.1: port 2
10 dst_ip=10.0.1/24: port 1

**Figure 3: A simple example for path table construction. The network consists of three switches and a total of 10 rules.**

### 3.2 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 $\langle inport, outport\rangle$ 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 represents the path.

For a concrete example, consider the toy network in Figure 3. 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 is a part of the path table for this topology.

#### 3.2.1 Representing the Header Set

A 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 [15] and ATPG [25]. However, wildcard expressions are suitable for representing suffix, while very inefficient for representing arbitrary header set. 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 of tens of switches, each of which has thousands of flow rules, a huge number of wildcard expressions are needed to represent all the possible packet sets. According to [13], characterizing the Stanford backbone network (16 switches) needs 652 million wildcard expressions.

Inspired by the previous work [24], we decide to use the Binary Decision Diagrams (BDDs) [7] to represent header sets. BDD is an efficient data structure for Boolean expressions, and has a better

**Table 1: Part of the path table for Figure 3. $[\cdot]$ represents the hash function.**

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

**Algorithm 2:** Traverse($inport, \langle S, x\rangle, H, t$)

```
Input: inport: the input port of the header; ⟨S,x⟩: the currently
       visited port; H: the header set; t: the tag
1  $\bar{H} \leftarrow H$; // headers of dropped packets
2  $H \leftarrow H \wedge P^A_x$; // ACL predicate of port x
3  foreach port y of switch S do
4      $\hat{H} = H \wedge P^F_y$ // FWD predicate of port y
5      if $\hat{H} \neq \emptyset$ then
6          $H \leftarrow H - \hat{H}$;
7          $\hat{H} = \hat{H} \wedge P^A_y$ // ACL predicate of port y
8          if $\hat{H} \neq \emptyset$ then
9              $\bar{H} \leftarrow \bar{H} - \hat{H}$;
10             $t \leftarrow t \oplus$ hash($x||S||y$) // update the tag
11             if $\langle S, y\rangle$ is an edge port then
12                 Insert($inport, \langle S, y\rangle, \hat{H}, t$);
13             else
14                 Traverse($inport$, Link($\langle S, y\rangle$), $\hat{H}, t$);
15 $t \leftarrow t \oplus$ hash($x||S||\perp$);
16 Insert($inport, \langle S, \perp\rangle, \bar{H}, t$);
```

support of set operations. With BDDs, we can expect to significantly reduce the size of path table.

### 3.2.2 Constructing the Path Table

We show how to construct the path table from a configuration similar to the Stanford backbone network configuration [3]. For simplicity, we assume the configuration files have already been transformed into a set of predicates using the method in [24]. For each input port $x$, there is an ACL predicate $P^A_x$, meaning that packets that satisfy $P^A_x$ are allowed to input from port $x$. Similarly, for each output port $y$, there is an ACL predicate $P^A_y$, meaning that packets that satisfy $P^A_y$ are allowed to output to port $y$. Finally, each outport $y$ also has a FWD (forwarding) predicate $P^F_y$ which guard which packets will be forwarded to port $y$.

Algorithm 2 summarizes the process of constructing path table from the above predicates. For each edge port connected with end hosts, we inject a header set $H$ initialized to all-headers (*i.e.*, a BDD of True), and a tag $t$ initialized to zero. When the header $H$ is received at a port $\langle S, x\rangle$, the algorithm intersect $H$ with the ACL predict of port $x$ (Line 2), and then iteratively intersect the resultant header set $\hat{H}$ with the forwarding rules of all output ports (Line 3-4). For each port $y$ that intersection is non-empty, the header set $\hat{H}$ is intersected further with the ACL rules of $y$ (Line 5-7). If $\hat{H}$ is still non-empty, the algorithm updates the tag $t$, and either inserts an path entry if $y$ is an edge port, or recursively calls the algorithm with the new header and tag (Line 8-14). If there are still headers that are not forwarded to any ports (recorded by $\bar{H}$), they would be dropped, and the algorithm updates the tag and inserts an entry (Line 15-16).

### 3.2.3 Verifying the Tags

Algorithm 3 specifies the simple process of tag verification. On

**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         else
6             return False; // the path is wrong
7 return False; // the packet should not reach here
```

receiving a tag report $\langle inport, outport, header, tag\rangle$, the server looks up in the path table with index $\langle inport, outport\rangle$, and 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 3, and assume $H_1$ sends a packet to port 22 of $H_3$. The packet should take the path of $S_1 \rightarrow S_2 \rightarrow S_2 \rightarrow S_3$, and the tag should be $[1||S_1||3] \oplus [1||S_2||3] \oplus [3||S_2||2] \oplus [1||S_3||2]$. With $(\langle S_1, 1\rangle, \langle S_3, 2\rangle)$ 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 rule $R3$ fails. Then, the packet will take the path of $S_1 \rightarrow S_3$, and the tag would be $[1||S_1||4] \oplus [3||S_3||2]$, disagreeing with that of the path.

## 3.3 Sampling

Tagging and verifying every packet in the network can incur a large overhead. This overhead can be made significantly smaller since packets of the same flow will very likely experience the same forwarding behaviors. 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 $T^f_s > 0$, 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 $t - t^f > T^f_s$, $S$ marks the packet and updates $t^f \leftarrow t$.

## 4. IMPLEMENTATION AND EVALUATION

## 4.1 Implementation

**Packet Format.** VeriDP needs each data packet to carry three additional elements: marker, tag, and inport. Here, marker is just 1 bit indicating whether the packet is sampled for verification or not; tag is the XORs of the lower 16 bits of hash output (currently we use CRC32); inport is the input port of the packet: 10 bits for switch ID, and 6 bits for local port ID. Thus, VeriDP currently can support 1024 switches, each of which can have up to 63 ports (one reserved for drop port). We put the 1-bit marker into the IP TOS field, and use two VLAN tags[1] to carry `tag` and `inport`. Finally, tag reports are sent to the verification server using UDP packets, each carrying four fields, *i.e.*, `inport`, `outport`, `header`, `tag`.

**VeriDP Server.** The server is responsible for constructing the path table based on network configuration, and searching the path table for tag verification. The path table construction is based on codes from [24], which iterates over all possible paths in the network to detect bugs (*e.g.*, black holes, loops) in the configuration files. We modify the codes by computing the tag for each path using Eq(1) to construct the path table. In addition, we add a virtual dropping port to each switch, and compute paths that end at this dropping port (*i.e.*, the header sets corresponding to headers that should be dropped by this switch). Before looking up in the path table, we first construct a BDD predicate from the $header$ field in the tag report. To determine whether $header \prec p.headers$ (Line 2 in Algorithm 3), we check whether the intersection of their BDD representation is not `False`.

**VeriDP Pipeline.** The VeriDP pipeline is responsible for sampling and marking packets, updating tags for marked packets, and sending tag reports to the VeriDP server. We implement the VeriDP pipeline with both the CPqD OpenFlow-1.3 software switch [2] and ONetSwitch [12], a hardware SDN switch we previously built. For the software switch, the VeriDP pipeline functions after all actions have been executed on a packet, and before the packet is sent out. For the hardware switch, the VeriDP and OpenFlow pipeline are both implemented using the FPGA resource. Since it requires switches to maintain the sampling instance for each physical flow, we have not yet implemented the sampling components on the hardware switch due to limit of time.

## 4.2 Correctness

We use Mininet [4] to emulate a $k = 4$ fat tree topology, and use `pingall` to establish routes between each pair of end hosts. Both the verification server and the Mininet run on the same PC, with Intel i3 3.4GHz CPU and 8GB Memory.

**Black Holes.** We initiate a UDP flow from one host $H1$ to another host $H2$, at a rate of 100 packets/sec. We set up the verification server and set the sampling interval to 0.1 second. At 15.8 seconds, we manually remove the forwarding rule for $H2$ from the flow table of a switch on the path, in order to simulate a black hole. The effect is shown in Figure 4(a).

**Access Violations.** Suppose $S$ is the access switch of a host $H2$. We manually add an ACL rule to let $S$ block all packets from another host $H1$. Then, we set up the verification server and initiate a UDP flow from $H1$ towards $H2$. The sampling interval and packet rate is still set to 0.1 second and 100 packets/sec, respectively. At 17.2 seconds, we manually remove the ACL rule from the flow table of $S$ to simulate an access violation. The effect is shown in Figure 4(b).

## 4.3 Performance

**Verification Throughput.** We saturate the verification server with tag reports, and measure how many can the server process per second. For the $k = 4$ fat tress, we observe the throughput is around $4 \times 10^6$ verifications/sec. We also use the topology of the Stanford backbone network, which consists of 16 routers and 10 switches. We observe a lower throughput of $0.7 \times 10^6$ verifications/sec. Since the verification is still single-threaded without optimization, we expect a higher throughput with multi-threading in the future.

[1]Double VLAN tags are supported by 802.1ad [1]; each tag consists of 12 bits VLAN ID, which can be used to carry our data.

**Table 2:** Processing delay of the VeriDP pipeline and native OpenFlow pipeline on the hardware switch.

| | Packet Size (bytes) | | | | |
|---|---|---|---|---|---|
| | 128 | 256 | 512 | 1024 | 1500 |
| VeriDP ($\mu s$) | 0.19 | 0.20 | 0.20 | 0.20 | 0.19 |
| Native ($\mu s$) | 5.62 | 8.63 | 14.65 | 26.69 | 37.88 |
| Overhead | 3.41% | 2.31% | 1.32% | 0.73% | 0.50% |

For the above verification, we generate the configuration with simple `pingall`. Therefore, only shortest paths are computed for each pair of hosts, *i.e.*, there is only one entry for each inport-outport pair in the path table, and Algorithm 3 only needs to check one entry. In real networks, there may be multiple paths between each pair of hosts, and packets with different headers can traverse via different paths. Thus, we continue to measure how many linear searches will Algorithm 3 perform in real networks.

**Real Network Policies.** We construct the path table with the configuration files of the Stanford backbone network [3] and Internet2 [5]. The Stanford network consists of 757,170 forwarding rules, and 1584 ACL rules; Internet2 has 126,017 forwarding rules without ACL rules. The time to construct the path table is 3830 ms for Stanford network, and 1327 ms for Internet2. We count the number of paths per inport-outport pair. The distribution is reported in Figure 5. We can see that the number of paths for each entry is relatively small, meaning that Algorithm 3 only needs a few time of searches to match the header (if the header can be matched).

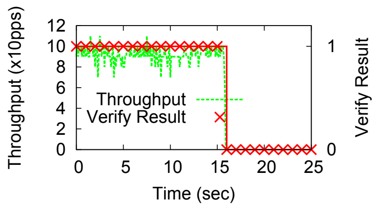
## 4.4 Overhead

Our implementation of VeriDP on the hardware switch can process packets at line speed (1Gbps). We use simulation to find that it takes 24 clock cycles to tag a packet. As the FPGA has a frequency of 125MHz, the additional delay is $24 \times \frac{1}{125 \times 10^6} = 0.192 \mu s$ per hop. Then, we send packets to one port of the switch, receive them from another one, and record the elapsed times. Let $T1$ be the elapsed time for native switch with OpenFlow pipeline only, and $T2$ be that with the modified switch with OpenFlow+VeriDP pipeline. Then, the processing delay of VeriDP pipeline is $\Delta T = T2 - T1$. Table 2 reports the value of $\Delta T$, $T1$, and the overhead $\Delta T / T2$, for packet sizes from 64 bytes to 1500 bytes. Table 2 reports the delay of VeriDP pipeline, native OpenFlow pipeline, and the relative overhead. We can see that the delay of VeriDP pipeline is around $0.20 \mu s$, agreeing with the simulation results. Besides, the overhead drops when packet size increases, and is strictly less than $5\%$.

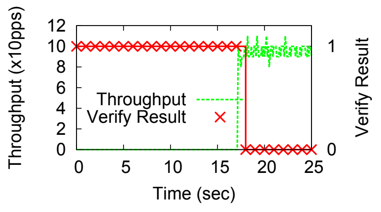
## 5. RELATED WORK

Recently there are many verification tools proposed for SDN [11]. We broadly classify them into two groups: *control plane verification* and *data plane verification*.

**Control Plane Verification.** Some tools are aimed to check the correctness of network configuration files. **Anteater** [19] models key network invariants (reachability, loop-freedom, black-hole-freedom, etc.) as SAT problems, and uses general solvers to check them. **Header Space Analysis** [15, 14] represents packet headers as points in $n$-bit space, and switches as transform functions that operate on the space. By analyzing the composite transform functions of switches, Header Space Analysis can check whether the key invariants are satisfied. **VeriFlow** [16] can incrementally check whether a new rule will violate the network invariants in real time. **NoD** [18] allows operators to check the correctness of net-



(a) Black Hole



(b) Access Violation

**Figure 4: Example detection of black hole and access violation with VeriDP.**

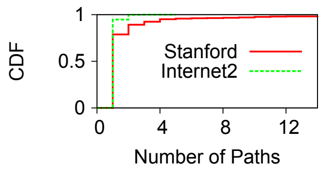


**Figure 5: CDF of number of paths per inport-outport pair in the Stanford backbone network and Internet2.**

work configuration at a higher abstraction (termed beliefs). The above tools are orthogonal to VeriDP which checks the compliance of data plane to network policies. They complement VeriDP by ensuring the network polices are correct, a premise for VeriDP to detect bugs.

**Data Plane Verification. ATPG** [25] generates a minimum number of probe packets to trigger all rules in the network. However, it only checks the reception of probe packets, without verifying their trajectories which are vital to configuration correctness. **SDN Traceroute** [6] enables the SDN controller to trace the trajectory of a flow, also based on probe packets. A limitation of them is that real packets may experience different forwarding behaviors with probe packets, making the verification results less convincing. Packet trajectory tracers like **PathletTracer** [27], **PathQuery** [21], and **CherryPick** [23] let each switch to imprint path information into packet headers, so that packet trajectories can be decoded by the receivers. 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.

## 6. CONCLUSION AND FUTURE WORK

This paper presented VeriDP, a new tool to monitor the policy compliance of SDN data plane. VeriDP checks whether packet forwarding behaviors are agreeing with the network configuration files, based on packet tagging. We implemented VeriDP on both software and hardware switches to demonstrate its feasibility, and used emulation to show it can detect common data plane faults like black holes and access violation. Our future work includes designing a fault localization method to pinpoint root causes when policy incompliance is detected.

**Acknowledgement.** The authors would like to thank all the anonymous reviewers for their comments. This work is supported by NSFC (No. 61402357), the China Postdoctoral Science Foundation (2015M570835), the Fundamental Research Funds for the Central Universities, and the open project of Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory (ITD-U14001/KX142600008).

## 7. REFERENCES

[1] 802.1ad - Provider Bridges. http://www.ieee802.org/1/pages/802.1ad.html.
[2] CPqD: The OpenFlow 1.3 compatible user-space software switch. http://cpqd.github.io/ofsoftswitch13/.
[3] Hassel, the header space library. https://bitbucket.org/peymank/hassel-public.
[4] Mininet. http://mininet.org/.
[5] The Internet2 Observatory. http://www.internet2.edu/research-solutions/research-support/observatory/.
[6] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN traceroute: Tracing SDN forwarding without changing network behavior. In *HotSDN*, 2014.
[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.
[8] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *USENIX NSDI*, 2014.
[9] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-defined network? In *HotSDN*, 2012.
[10] 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.
[11] 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.
[12] C. Hu, J. Yang, H. Zhao, and J. Lu. Design of all programmable innovation platform for software defined networking. In *Open Networking Summit*, 2014.
[13] 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.
[14] 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.
[15] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.
[16] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.
[17] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT way for openflow switch interoperability testing. In *ACM CoNEXT*, 2012.
[18] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *USENIX NSDI*, 2015.
[19] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *ACM SIGCOMM*, 2011.
[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
[21] S. Narayana, J. Rexford, and D. Walker. Compiling path queries in software-defined networks. In *HotSDN*, 2014.
[22] G. Pickett. Staying persistent in software defined networks. In *Black Hat Briefings*, 2015.
[23] P. Tammana, R. Agarwal, and M. Lee. CherryPick: Tracing packet trajectory in software-defined datacenter networks. In *SOSR*, 2015.
[24] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, 2013.
[25] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *ACM CoNEXT*, 2012.
[26] 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.
[27] 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.