Debugging SDN in HPC Environments

Mami Hayashida
Lab for Advanced Networking
University of Kentucky
Lexington, Kentucky 40506
mhaya2@netlab.uky.edu

Sergio Rivera
Lab for Advanced Networking
University of Kentucky
Lexington, Kentucky 40506
sergio@netlab.uky.edu

James Griffioen
Lab for Advanced Networking
University of Kentucky
Lexington, Kentucky 40506
griff@netlab.uky.edu

Zongming Fei
Lab for Advanced Networking
University of Kentucky
Lexington, Kentucky 40506
fei@netlab.uky.edu

Yongwook Song
Lab for Advanced Networking
University of Kentucky
Lexington, Kentucky 40506
ywsong2@netlab.uky.edu

ABSTRACT

HPC networks and campus networks are beginning to leverage various levels of network programmability ranging from programmable network configuration (*e.g.*, NETCONF/YANG, SNMP, OF-CONFIG) to software-based controllers (*e.g.*, OpenFlow Controllers) to dynamic function placement via network function virtualization (NFV). While programmable networks offer new capabilities, they also make the network more difficult to debug. When applications experience unexpected network behavior, there is no established method to investigate the cause in a programmable network and many of the conventional troubleshooting debugging tools (*e.g.*, ping and traceroute) can turn out to be completely useless. This absence of troubleshooting tools that support programmability is a serious challenge for researchers trying to understand the root cause of their networking problems.

This paper explores the challenges of debugging an all-campus science DMZ network that leverages SDN-based network paths for high-performance flows. We propose *Flow Tracer*, a light-weight, data-plane-based debugging tool for SDN-enabled networks that allows end users to dynamically discover how the network is handling their packets. In particular, we focus on solving the problem of identifying an SDN path by using actual packets from the flow being analyzed as opposed to existing expensive approaches where either probe packets are injected into the network or actual packets are duplicated for tracing purposes. Our simulation experiments show that Flow Tracer has negligible impact on the performance of monitored flows. Moreover, our tool can be extended to obtain further information about the actual switch behavior, topology, and other flow information without privileged access to the SDN control plane.

PEARC '18, July 22–26, 2018, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-6446-1/18/07…$15.00
https://doi.org/10.1145/3219104.3229277

CCS CONCEPTS

• **Networks** → **Network management**; *Programmable networks*;

KEYWORDS

network management, network debugging, software-defined networking, traceroute

ACM Reference format:
Mami Hayashida, Sergio Rivera, James Griffioen, Zongming Fei, and Yongwook Song. 2018. Debugging SDN in HPC Environments. In *Proceedings of Practice and Experience in Advanced Research Computing, Pittsburgh, PA, USA, July 22–26, 2018 (PEARC '18),* 8 pages.
https://doi.org/10.1145/3219104.3229277

1 INTRODUCTION

HPC users increasingly find themselves working with very large data sets. These *big data* sets place high demands on the computational infrastructure, storage infrastructure, and networking infrastructure. In recent years, much attention has been given to the development of new processing and storage techniques designed for big data [1, 4–7, 9, 11, 28]. The associated network challenges, on the other hand, have often been overlooked. However, recent advances in high-speed programmable and software defined networks (SDN) [17] are beginning to be used to effectively address the network challenges associated with big data. Many campus network infrastructures have been updated recently [22] with high-speed programmable switches that enable new, effective, and efficient solutions to the networking challenges caused by big data.

One such example is the *VIP (Very Important Packets) Lanes* system [12] actively being developed on the campus of the University of Kentucky that leverages SDN capabilities to provide researchers with an interface that allows them to set up high-speed, middlebox-free paths for their research traffic across the campus network. Authorized researchers (*i.e.*, scientists) on campus can now move their large data sets to various destinations, including national laboratories, local storage systems (*e.g.*, tape drives, object stores), or commercial cloud services (*e.g.*, Google Drive, AWS S3), at significantly faster speeds (approaching two orders of magnitude speedups [12]) over what is supported by the traditional campus infrastructure. More generally, programmable networks allow for custom, application-specific software to be developed and deployed that makes efficient use of the network [25], for example,

by maximizing throughput, minimizing latency, load balancing traffic, protecting/firewalling traffic, or performing network address translation (NAT), all on individual flows.

While programmable networks enable highly-customized application-specific communication, they also create new network debugging challenges. Debugging networks has never been easy – even for conventional non-programmable networks – but the problems are only magnified by SDN-enabled networks. Moreover, common tools used for network debugging such as `ping` [20] and `traceroute` [18] are of little help in the context of programmable networks. Indeed, the absence of trouble-shooting tools for SDN networks has impacted the VIP Lanes project as well, increasing the difficulty of debugging applications running across the campus SDN network infrastructure. Identifying and locating causes of unexpected network traffic behavior has often been difficult and resource-consuming. The combination of faulty equipment, continuous changes to the controller and switch state/rules by one or more controller modules/applications, an enhanced number of packet handling options (*e.g.*, forwarding, header modification, rate-limiting), and other SDN "features" only compound the network debugging process. Our own experience has shown that trouble-shooting SDN networks often requires manual box-by-box inspection by a human operator, debugging installed configurations, comparing against controller state, and working closely with the SDN software developers and end-users.

To address this problem, we propose *Flow Tracer*, a powerful data-plane packet tracing tool for programmable networks designed to trace the flow-specific forwarding rules defined by the (application-specific) control software. Unlike `traceroute` for legacy networks, Flow Tracer traces actual data streams as opposed to artificially created probe packets. The tool is able to associate packets with flows and returns, to the requesting user, the list of SDN switches that packets from the tracked flow has traveled through. Although other SDN data-plane path tracing tools have been proposed, they all follow the conventional traceroute approach based on probe packets, which are not well suited for SDN environments.

The remainder of this paper is organized as follows. Section 2 outlines the issues and challenges in debugging SDN environments. Section 3 describes the limitations and problems with conventional traceroute approaches. Section 4 then lays out the design goals and overall architecture of our Flow Tracer approach. It also describes the sequence of events that occur as part of the trace. We present experimental results of our Flow Tracer prototype in Section 5. Existing approaches for troubleshooting programmable networks and how they compare to Flow Tracer are described in Section 6. Lastly, in Section 7, we report contributions and limitations of Flow Tracer, and Section 8 concludes the paper.

2 SDN TRACING CHALLENGES

While SDN technology has matured significantly in recent years, it does not offer the same level of stability, robustness to failures, or even assurances of correct operation as the time-tested conventional router hardware/software implementations that have been in use for decades. As a result, there are far more potential sources of errors in an SDN network than a conventional network which makes SDN networks much harder to debug.

The first challenge is the SDN hardware itself. SDN switches are constantly evolving, adding new features and/or increased performance, increasing the likelihood of some sort of hardware error. Each new revision to an SDN specifications, such as OpenFlow [19], brings with it significant feature enhancements, many of which have not been thoroughly tested and are susceptible to failures or unexpected behavior from the SDN switch. In addition, the increased performance (e.g., link speeds) that come with each new generation of hardware is particularly problematic for SDN switches where programmability must also keep pace with the enhanced link speeds – creating another opportunity for errors. Example OpenFlow hardware issues we have observed in commercial OpenFlow switches include missing or partially implemented OpenFlow functionality; incoming packets not being sent to the OpenFlow rules for processing; metrics/counters not reporting correct values; switches or machines in the topology not being reported to the control software; extremely slow throughput (e.g., rules being unexpectedly pushed to software, rather than hardware, tables); silent hardware failures (e.g., unreported link failures); and incorrect failure messages (e.g., working links marked as failed). All of the above make debugging SDN networks more difficult than conventional networks, illustrating the dire need for new tools to help debug SDN networks.

SDN controller implementations are arguably even more error-prone than the SDN hardware. Like the hardware, controllers are evolving quickly, increasing the potential for errors. Information can easily become inconsistent between the switches and the controller for a variety of reasons, leading to incorrect decisions by the control software/applications. Moreover, the software must keep track of a complex and continuously changing set of OpenFlow rules that have the potential to interfere with one another. Unlike IP routing, where forwarding rules are based only on the destination IP address, OpenFlow rules can include a large number of variables that may overlap in unexpected ways with other rules causing unintended consequences. To further complicate the matter, OpenFlow rules can be assigned to various priority levels in multiple tables. While this design enables fine-grained forwarding, it makes network behavior much more complicated especially when forwarding and rewriting rules are installed into switches by multiple controller applications/modules.

3 CONVENTIONAL APPROACHES

As noted in Section 1, debugging SDN network problems often requires the expertise of a (human) network operator – possibly working in conjunction with an end system user – manually inspecting SDN configurations switch by switch in an attempt to identify the "bug" in the control software.

In conventional IP-based networks, network problems can often be identified by end system users using basic network trouble-shooting tools like `ping` [20] and `traceroute` [18] without the involvement or privileges of a network operator. Unfortunately, tools like ping and traceroute are of limited, if any, value in an SDN setting. To fill this void, a number of traceroute-like tools have been proposed [2, 10, 13, 29, 32]. Like traceroute, these tools are based on a model that uses artificially generated *probe packets* to understand network behavior and identify problems. Although probe packets

are useful in an IP-based network (where an end system's only option is to inject packets into the network – *i.e.*, it cannot control or program the network), probe packets are not the best approach for an SDN network that can be programmed.

Solutions based on probe packets suffer from a number of problems. First, the probe packet must be able to mimic the structure and content of the packets that are experiencing problems (*i.e.*, being debugged). In a conventional IP network where routers only examine the destination IP address, creating a representative probe packet that has the same IP address is straightforward. However, in an SDN network where forwarding decisions can be based on one or more fields of a packet, creating representative probe packets requires understanding the packet structure of the flow being traced and the fields used by the SDN software to make forwarding decisions. Failure to exactly replicate the flow traffic could result in different SDN processing inside the network; in other words, the probe packet could take a different path than the real packets. In addition, every packet from a flow in a conventional IP based network has the same IP destination address and can be represented by a single probe packet. In contrast, the fields used for forwarding decisions in an SDN network could change over the lifetime of a flow, which means a single probe packet cannot be constructed to represent all packets in a flow (even if the fields used for forwarding were known). Second, use of probe packets carrying a special marker that causes them to be sent to the SDN controller for analysis [2, 13] opens the door for denial-of-service attacks on the tracing mechanism itself (and indirectly on the entire network control plane). Thirdly, some SDN probe packet marking approaches [2] do not include mechanisms to restrict a flow's path information to the end systems associated with the flow – in other words, spying on other flows is possible.

In short, because software dynamically determines which bits in a packet (header) will be used to make forwarding decisions in an SDN network, a system based on general-purpose end system tools that inject representative probe packets is not an appropriate model for SDN networks. To the best of our knowledge, our model is the first data-plane based SDN path tracing tool that does not rely on probe packet generation, but instead traces "live traffic" sent by real applications.

4 FLOW TRACER

To address the problems associated with the use of probe packets in an SDN network setting, we developed a *Flow Tracer* tool capable of tracing unmodified packets that comprise real flow traffic. We implemented our Flow Tracer tool in the context of an OpenFlow network, but the general approach could be applied to any SDN network with an SDN controller. In the following we discuss the goals and overall architecture of Flow Tracer as well as the sequence of events that occur when tracing a flow.

4.1 Design Goals

In contrast to existing path tracing tools, we established the following design goals for the Flow Tracer tool:

- *Usable by End-host Users*: Most of the previously proposed SDN path tracing tools are designed to be used by network administrators with full network privileges. Our tool, like `traceroute`, is intended to be used by authorized end-users without full access privileges to the controller or any other network device in the network.
- *No Representative probe packets*: Unlike other tracing tools based on artificially generated probe packets (see Section 3), Flow Tracer should trace the path of the actual data transfer, leveraging SDN features to capture one, or a small number of, packets from the monitored flow.
- *Existing Rules Must Remain In-place and In-use*: All packets, including those in the traced flow, must be forwarded (after possibly being modified) by the existing rules currently in place. Any new rules inserted for the purpose of tracing should not be used when forwarding a traced packet toward the next-hop switch. Similarly, the controller should not bypass existing rules – say by directly sending a packet out an output port without consulting the OpenFlow rules.
- *Prevent DoS Attacks on the Trace Mechanism*: Another problem with the probe packet model is the increased risk of DoS (Denial of Service) attacks: if an attacker learns how probe packets are tagged, they can launch a DoS attack by injecting a large number of such packets into the network.
- *Keep Trace Results Private*: To prevent attackers from stealing trace result data (*e.g.*, spying on other flows), Flow Tracer trace operations should only be initiated by authorized users and should go through an SDN Tracing Service system that verifies the user's right to trace a flow.

4.2 Flow Tracing Components

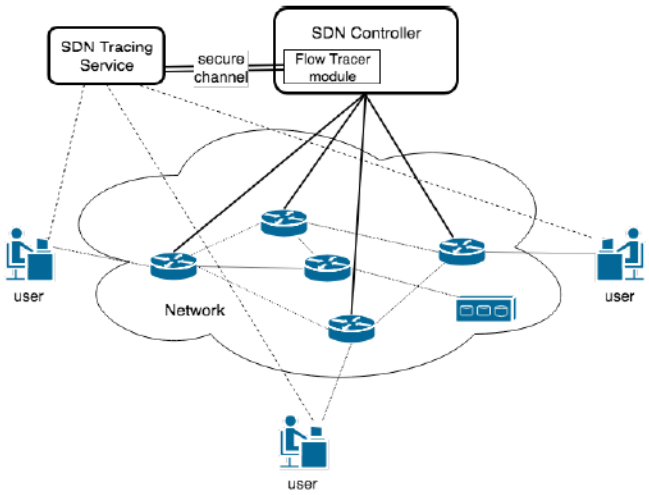

Figure 1: Overview of Flow Tracer Architecture

Figure 1 shows the global architecture of Flow Tracer which consists of four main components described below:

- **SDN Tracing Service**: A service that accepts flow trace requests from the Flow Tracer program running on a user's end system over a secure channel. The service authenticates users, checks to see if the user is allowed to trace the specified flow, and communicates with the Flow Tracer SDN controller module to install OpenFlow trace rules

in switches. When a trace is complete, the SDN Tracing Service returns the results to users.

- **Flow Tracer Module**: An internal SDN controller application that receives flow trace requests from the SDN Tracing Service and implements the Flow Tracer logic by installing OpenFlow rules in switches. It also inspects packets intercepted by switches, records the trace result, and removes the Flow Tracer rules upon completion of the flow trace operation.
- **Switches**: SDN-enabled switching/routing devices in a network where Flow Tracer rules intercept traced packets.
- **Flow Tracer Program**: The Flow Tracer program is run on end system by users to initiate a flow trace request with the SDN Tracing Service.

4.3 Tracing Steps

The following section outlines the sequence of events that compose a path-tracing session using Flow Tracer. It assumes that Flow Tracer module (internal application) has been activated in the controller.

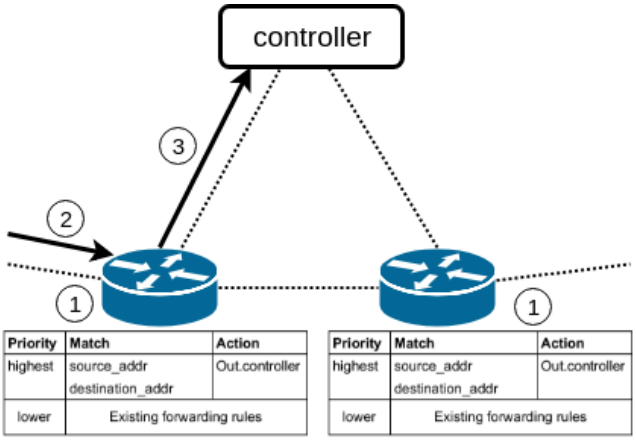

Figure 2: Sequence of events after a tracing session is initiated (Note: source and destination addresses in the match field are given simply as an example of a flow identification.)

Initiating a Tracing Session: Each Flow Tracer request begins with a user sending a request to the SDN Tracing Service. The request contains the user credentials that the SDN Tracing Service uses to authorize the tracing request. Upon authentication, or as part of the authentication process, the user must provide the flow specification of the flow the user desires to trace. Typically this will be specified using a tuple that uniquely identifies the flow (*e.g.*, source and destination IP addresses, source and destination port numbers, and the protocol number). Once the validity of a flow trace request is confirmed, the SDN Tracing Service sends a tracing session request to the Flow Tracer module within the SDN controller along with the specific flow parameters it has received from the user.

Insertion of Flow Tracer Rules: The Flow Tracer module installs a trace rule at every switch in the network based on the flow parameters provided by the user (step 1 in Figure 2). The rule, which is identical for all switches, matches on the flow specification provided by the user and instructs the switch to send packet(s) from that particular flow to the controller. The Flow Tracer rule must be assigned a higher priority level than that of all other forwarding rules in order to capture every packet that matches the criteria while the rule exists. Since we are installing rules at every switch in the network, some rules may never be hit. Therefore, to prevent Flow Tracer rules no longer in use from lingering on, our implementation sets its idle timeout value to 30 seconds, assuming the trace will begin within that amount of time.

User Begins Data Transfer: Once the network is set up for tracing (*i.e.*, the session-specific Flow Tracer rule has been installed on every switch), the SDN Tracing Service notifies the user that a test may begin. If a data flow between the source and destination hosts of the traced path is already under way, the user simply waits for the trace request results from the SDN Tracing Service. Otherwise, the user starts sending data between the two hosts.

Trace Rule Captures Packet(s): When the first packet of the traced flow – typically a TCP SYN packet – reaches the first hop, the packet is sent to the controller using an OpenFlow `PACKET_IN` message (steps 2 and 3 in Figure 2). The Flow Tracer module then examines the packet, records its arrival time and the switch DPid, and instructs the switch to remove the trace rule from the switch to prevent any subsequent packets matching the rule from being forwarded to the controller (step 4 in Figure 3). The Flow Tracer module then returns the recorded packet to the switch that intercepted it. The switch will then apply the previously existing OpenFlow rules – the ones that existed prior to the trace rules being inserted – ensuring that the packet is forwarded in exactly the same way as it would have been forwarded if tracing had not been invoked (see steps 5 and 6 in Figure 3). Any subsequent packets from the same flow will not be intercepted by trace rules at that switch, because the trace rules have been removed. This process repeats itself at each switch along the path the packets follow.

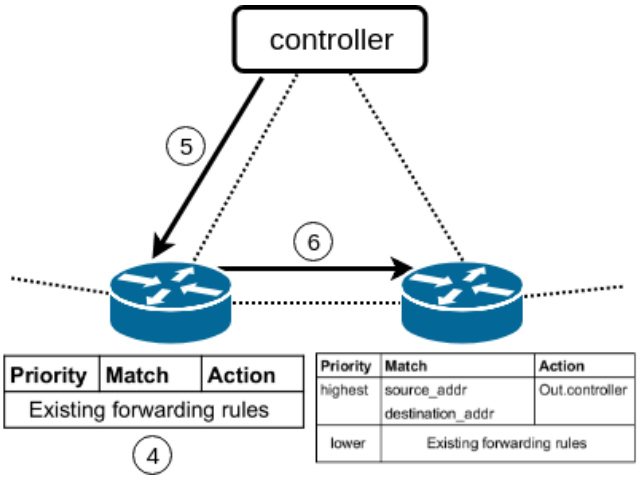

Figure 3: Removing a Trace Rule After Capturing a Traced Packet

Removal of Remaining Trace Rules: The flow trace completes when the Flow Tracer module determines that the last hop switch

has reported the traced packet to the controller. (The last hop router is identified as the last router traversed before the packet exits the "traceable" part of the network – which can be determined from inserted SDN rules.) In cases where the packets do not reach a last hop router (is dropped or does not leave a switch), the trace times-out after a specified amount of time, and the partial path is recorded. Once the trace completes, the Flow Tracer rule is removed from all switches, and thus no more packets from the traced flow will be sent to the controller from any switches.

Reporting the Trace Result: Once the flow trace is complete, the result of the trace is obtained by the SDN Tracing Service from the Flow Tracer module and is sent to the end system Flow Tracer program.

5 EVALUATION

We implemented our Flow Tracer prototype on an Aruba VAN controller [15] using version 1.3 of the OpenFlow protocol. Our preliminary tests were performed on Mininet [18] virtual network topologies. Our controller was running on an OpenStack [23] virtual instance with 2 vCPUs and 4GB of RAM running Ubuntu 14.04.4 LTS (x86_64); similarly, we created a Mininet topology on an OpenStack virtual instance with 2 vCPUs and 4GB of RAM running Ubuntu 14.04.5 LTS (x86_64) for the first two experiments (Figure 4); the second Mininet topology (Figure 8) was created on a VirtualBox [24] guest machine with 1 vCPU and 1GB of RAM running Ubuntu 14.04.4 LTS (x86_64).

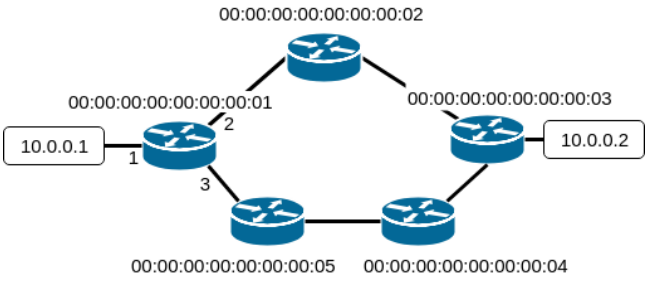

Figure 4: Mininet Test Topology

Validity: In order to verify the correctness of our Flow Tracer results, we created the topology shown in Figure 4 and installed OpenFlow rules in such a way that TCP, UDP and ICMP packets are treated differently when going from 10.0.0.1 to 10.0.0.2. Specifically, TCP packets take the longer of the two routes, *i.e.*, [01, 05, 04, 03][1]; UDP packets are forwarded through the shorter of the two routes (*i.e.*, [01, 02, 03]) whereas ICMP packets are dropped at the first switch (*i.e.*, 01). The corresponding OpenFlow rules, including one for the reverse direction, installed on DPid 01 are shown in Table 1. Table 2 shows the initial setup for a Flow Tracer session to track a flow from 10.0.0.1 to 10.0.0.2. This rule is inserted on every switch in the topology.

For our first test, we sent TCP, UDP, and ICMP packets from 10.0.0.1 to 10.0.0.2, using `hping3` [27]. We authorized Flow Tracer to trace TCP, UDP, and ICMP packets before transmitting the data packets. The results are shown in Figure 5, Figure 6, and Figure 7, respectively.

[1]Each of these numbers represents the rightmost none-zero values of the 8 two-digit DPid (Datapath ID) groups. "01", for instance, corresponds to DPid 00:00:00:00:00:00:00:01.

Priority	Match	Actions
30000	eth-type: IPV4	output:1
	ipv4-src: 10.0.0.2	
	ipv4-dst: 10.0.0.1	
30000	eth-type: IVP4	(*i.e.*drop)
	ipv4-src: 10.0.0.1	
	ipv4-dst: 10.0.0.2	
	ip-proto: icmp	
30000	eth-type: IPV4	output:2
	ipv4-src: 10.0.0.1	
	ipv4-dst: 10.0.0.2	
	ip-proto: udp	
30000	eth-type: IPV4	output:3
	ipv4-src: 10.0.0.1	
	ipv4-dst: 10.0.0.2	
	ip-proto: tcp	

Table 1: Forwarding rules installed on switch 01 (00:00:00:00:00:00:00:01)

Priority	Match	Actions
33000	eth-type: IPV4	output:Controller
	ipv4-src: 10.0.0.1	
	ipv4-dst: 10.0.0.2	

Table 2: Flow Tracer rule added for tracing flows from 10.0.0.1 to 10.0.0.2

```
00:00:00:00:00:00:00:01, 1521571956862
00:00:00:00:00:00:00:05, 1521571956871
00:00:00:00:00:00:00:04, 1521571956873
00:00:00:00:00:00:00:03, 1521571956875
```

Figure 5: TCP flow trace result (DPid, timestamp)

```
00:00:00:00:00:00:00:01, 1521572002859
00:00:00:00:00:00:00:02, 1521572002903
00:00:00:00:00:00:00:03, 1521572002904
```

Figure 6: UDP flow trace result (DPid, timestamp)

Our results show that Flow Tracer indeed reports the actual packet behavior for each particular flow. For instance, if a user realizes ICMP traffic is not reaching its destination, by starting a debug session with Flow Tracer, the user can infer that packets are not reaching any further than switch 01.

Tracing an Ongoing Flow: For this test, we used the same Mininet topology and deployed Flow Tracer after flows have started. We saturated the link capacity using `iperf` [16] with the default TCP

```
00:00:00:00:00:00:00:01, 1521572029212
```

Figure 7: ICMP flow trace result (DPid, timestamp)

window size of 85.3KBytes, host 10.0.0.2 acting as a server listening on its TCP port 5001, and 10.0.0.1 acting as a client. We conducted the test 10 times.

Despite a relatively large number of packets matching the Flow Tracer rule on every switch along the path, Flow Tracer correctly identified all the switches the flow has passed through each time, and the flooding was never severe enough to cause any issues on the controller. Unlike in the first test, however, switches cannot be listed in the order in the path (from source to destination) as the packets in the traced flow are being sent to the controller from all switches simultaneously. Which packet from which switch is recorded by the Flow Tracer module first is arbitrary. For this reason, the Flow Tracer result here should be seen as a set of switches the flow has gone through rather than an ordered list.

Effect on Throughput: To measure the effect of Flow Tracer on the throughput of data transfer, we created another Mininet topology, shown in Figure 8, in which packets from 10.0.0.1 to 10.0.0.7 passed through seven forwarding devices.

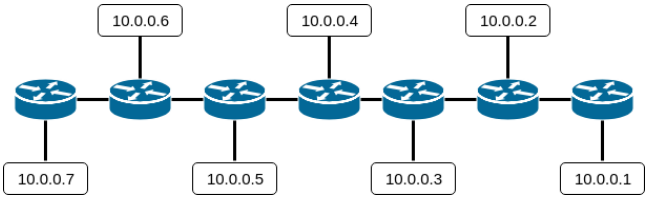

Figure 8: Mininet Topology Used for Measuring Flow Tracer Delays

For the first round of tests, we set the link bandwidth to the value of 1Gbps and, using `iperf`, sent 1GB of data from 10.0.0.1 to 10.0.0.7 under three different conditions: no Flow Tracer deployment as benchmark; Flow Tracer deployed before the start of the data transfer; Flow Tracer deployed during the data transfer. Each setting was tested 10 times.

Condition	Mean	Median	Range
Benchmark	791.4	793	784-798
Deployed Before	787.4	787.5	781-796
Deployed During	779.7	781.5	767-788

Table 3: Throughput reduction caused by Flow Tracer during a 1GB transfer over 1 Gbps links. All units in Mbps.

As shown in Table 3, the effect of Flow Tracer was minimal. When Flow Tracer was deployed before the start of the flow, the average data transfer rate of a 1GB file was 0.5% lower than the benchmark; when it was deployed during the data transfer, the average rate experienced 1.5% reduction. We repeated the same test using Mininet with high-bandwidth links, which on our test topology achieved slightly over 40 Gbps. To accommodate the much higher speeds, we transferred 54GB of data. As shown in Table 4, the larger data size and bandwidths obscured the effect of Flow Tracer even further. Those transfers for which Flow Tracer was deployed ahead of time actually gave higher measurement values than the benchmark, and those for which Flow Tracer was deployed during the data transfer did almost as well as the benchmark. Given that the difference in transfer rates among the three test cases was no more than the variations among data for each case, we consider these differences to be statistically insignificant. Also confirmed in these tests is that even with a high transfer rate of roughly 40Gbps, not only the controller was able to handle packets being sent from all seven switches through the Flow Tracer rules, but also the TCP connection recovered reliably from the effect of out-of-order packets caused by Flow Tracer.

Condition	Mean	Median	Range
Benchmark	42.04	42.35	39.2-42.9
Deployed Before	42.94	42.95	42.1-43.6
Deployed During	40.25	40.55	38.2-41.2

Table 4: Throughput reduction caused by Flow Tracer during a 54GB transfer over high-speed links. All units in Gbps.

6 RELATED WORK

The most often referenced work on data-plane-based SDN tracing tool is *SDN traceroute* [2]. Once each SDN-enabled device is assigned a color based on the graph-coloring algorithm and set up with color-specific OpenFlow debugging rules, a color-tagged probe packet is injected into the network. The probe packet is sent to the controller at every hop; yet, due to the reassigning of its color, it is forwarded to the next hop when sent back to the switch. Although this approach is relatively simple to implement, there is an obvious drawback: it assumes that the network topology information obtained through the SDN controller is correct. While this is ordinarily the case, a debug tracing tool is most needed when there is an unidentified failure in the network; if the failure prevents network administrators from obtaining an accurate map of the whole network topology, the algorithm may fail. Furthermore, this and almost all other tools rely on the production of a probe packet. As discussed earlier, creating a "clone" packet that matches all the OpenFlow rules as the actual flow packets experiencing network issues can be difficult, if not outright impossible. There are two additional unaddressed issues: First, use of the VLAN header field to tag probe packets, as proposed in this paper, could be a problem in networks where VLAN field is relevant; second, if the probe packet tagging method becomes known, it would be easy to launch a DoS attack by injecting a large number of probe packets.

There are several other works that can be considered extended versions of *SDN traceroute*. *Hybridtrace* [29] is a tool that can be used to trace paths that consist of both SDN and legacy network devices; *Track* [32] adds capabilities to handle arbitrary network functions (*i.e.*, middle boxes). As both implementations are based on SDN traceroute, they share the same advantages and disadvantages outlined above.

SDNTrace [13] presents a contrasting approach using a special type of Ethernet packet that carries the information (*e.g.*, source,

destination, protocol) about the path to be traced in its payload. Each time a probe packet is sent to the controller from a switch, the controller records its arrival and forwards it to the next hop based on the OpenFlow rules installed on the switch. The disadvantage of this approach lies in that if there is a discrepancy between the actual state of the forwarding tables of the switch and controller's knowledge of them, the probe packet would be forwarded according to the latter. In addition, while their design handles the end of the debug session cleanly, it fails to return a tracing result when a probe packet does not reach the last SDN device before reaching its path destination. The challenge of generating a probe packet applies to their work as well.

SFC Path Tracer [10], another path-tracing tool, is designed especially for an NFV/SDN environment. While it claims to take much less time for probing than *SDN traceroute*, its probe packets are not forwarded by existing forwarding rules, but by new trace rules that are extended copies of original rules. Netography [33] compares flow rules to a sequence of probe packet copies sent to a controller from switches as the probe packet traverses through the network. As in all other approaches, both of these works require generation of probe packets. In case of Netography, the VLAN PCP field is used as a flag-bit field, disallowing any other VLAN operations in the network.

Additionally, there is a number of papers that incorporate path-tracing as an integral part of examining whole network behavior. This group of works include *nbd* [14], a debugger inspired by *gdb*; *PathSeer* [3], that proposes an efficient method to trace packet trajectories in SDN-enabled datacenter networks; *RuleScope* [30] [8] that detects rule faults by sending probe packets; *VeriDP* [31], that compares the data-plane handling of packets to network policies and configurations in the control-plane; *Simon* [21], an interactive debugging tool that examines network behavior through the use of ICMP ping packets as probe packet; and *CherryPick* [26], another work on how to efficiently perform packet-tracing at data centers on their fat topologies.

7 DISCUSSION

Our novel design eliminates the need for probe packets and instead traces one or a very small number of actual data transfer packets. It is a tool designed to be deployed by users on end-hosts to obtain data on where and how far their packets have traversed over the network especially when unexpected network behavior (*e.g.*, packets getting dropped or experiencing noticeable delays) has been observed. Our implementation works especially well for TCP connections with insignificant overhead during the trace. Assuming that Flow Tracer is deployed before the beginning of a data transfer, the first SYN packet (SYN-ACK if tracing the reverse direction) is sent to the controller at every hop; by the time the subsequent packets flow through the same switches, all of the Flow Tracer OpenFlow rules have been removed. Initiating a Flow Tracer session after a flow has already begun, as well as tracing UDP flow, is slightly more problematic as multiple packets of a traced flow are sent from every switch to a controller more or less simultaneously and, as a result, the list of switches in a tracing result would not be ordered. Our test results indicate, however, that flooding is kept to minimal with our design and does not lead to any controller issues.

Even though Flow Tracer provides a useful understanding on how packets are handled over an SDN-enabled network, there is a limitation that deserves further investigation. If any of the existing forwarding rules on the flow path matches ingress ports, it would be incompatible with Flow Tracer. When a packet sent to a controller via Flow Tracer is returned to the same switch, its ingress port is now changed to the reserved "controller" port, rather than the original ingress port. This could potentially cause the packet to be misrouted if it were to be routed to the next hop based on its ingress port match. While TCP connections would likely recover from a small number of misrouted packets, the result of Flow Tracer would be invalid as the misrouted packets, an undesirable side-effect of Flow Tracer, could potentially report back paths not taken by the rest of the packets in the flow.

8 CONCLUSION

In this paper, we presented Flow Tracer, an SDN path tracing tool designed for end-users. Without modification to any of the existing forwarding rules on switches, and with addition of one rule on each switch per debug session, Flow Tracer is able to track the path of data transfer by observing one or a small number of packet(s) of the flow with negligible impact on performance. Unlike other models, Flow Tracer requires no special probe packet for tracing, which eliminates the non-trivial challenge of probe packet creation and reduces DoS attack risks to the network.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grants ACI-1541380, ACI-1541426, and ACI-1642134. We would also like to thank Lowell Pike and Jacob Chappell for their technical guidance on test environment installation and configuration.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Kanak Agarwal, Eric Rozner, Colin Dixon, and John Carter. 2014. SDN traceroute: Tracing SDN forwarding without changing network behavior. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 145–150.

[3] A. Aljaedi and C. E. Chow. 2016. Pathseer: a centralized tracer of packet trajectories in software-defined datacenter networks. In *2016 Principles, Systems and Applications of IP Telecommunications (IPTComm)*. 1–9.

[4] Apache Software Foundation. 2010–2018. Apache Hive. (2010–2018). https://hive.apache.org

[5] Apache Software Foundation. 2011–2018. Apache Hadoop. (2011–2018). https://hadoop.apache.org

[6] Apache Software Foundation. 2013–18. Impala. (2013–18). https://impala.apache.org

[7] Apache Software Foundation. 2014–2018. Apache Spark. (2014–2018). https://spark.apache.org

[8] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen. 2016. Is every flow on the right track?: Inspect SDN forwarding with RuleScope. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524333

[9] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on*

Opearting Systems Design & Implementation - Volume 6 (OSDI'04). USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264

[10] Rafael Anton Eichelberger, Tiago Ferreto, Sebastien Tandel, and Pedro Arthur PR Duarte. 2017. SFC path tracer: a troubleshooting tool for service function chaining. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*. IEEE, 568–571.

[11] Facebook. 2012–2018. Presto. (2012–2018). https://prestodb.io/

[12] J. Griffioen, K. Calvert, Z. Fei, S. Rivera, J. Chappell, M. Hayashida, C. Carpenter, S. Yongwook, and H. Nasir. 2017. VIP Lanes: High-speed Custom Communication Paths for Authorized Flows. In *2017 26th International Conference on Computer Communication and Networks (ICCCN). [to appear]*.

[13] Deniz Gurkan. 2015. SDNTrace Protocol Design and Testing. http://sdntrace-protocol.readthedocs.io/en/latest/. (2015). Online, accessed 22-October-2017.

[14] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. 2012. Where is the debugger for my software-defined network?. In *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 55–60.

[15] Hewlett Packard Enterprise. 2017. HP Virtual Applications Network SDN Controller. https://www.hpe.com/us/en/product-catalog/networking/networking-software/pip.hpe-van-sdn-controller-software.5443866.html. (2017).

[16] J.Dugan, S.Elliott, B.Mah, J.Poskanzer, and K.Prabhu. 2003–18. iPerf. https://iperf.fr/. (2003–18).

[17] D. Kreutz, F. M. V. Ramos, P. E. VerÃŋssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (Jan 2015), 14–76. https://doi.org/10.1109/JPROC.2014.2371999

[18] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 19.

[19] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communications Review* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[20] Mike Muuss. 1983. The story of the PING program. https://linux.die.net/man/8/ping. (1983).

[21] Tim Nelson, Da Yu, Yiming Li, Rodrigo Fonseca, and Shriram Krishnamurthi. 2015. Simon: Scriptable interactive monitoring for SDNs. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, 19.

[22] NSF. 2018. Campus Cyberinfrastructure (CC*). https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=504748. (2018).

[23] OpenStack Foundation. 2010–2018. OpenStack. (2010–2018). https://www.openstack.org/

[24] Oracle. 2007–2018. VirtualBox. (2007–2018). https://www.virtualbox.org/

[25] Sergio Rivera, Mami Hayashida, James Griffioen, and Zongming Fei. 2017. Dynamically Creating Custom SDN High-Speed Network Paths for Big Data Science Flows. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. ACM, 59.

[26] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2015. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, 23.

[27] Kali Tools. 2005–18. hping3. https://tools.kali.org/information-gathering/hping3. (2005–18).

[28] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[29] Shie-Yuan Wang, Chia-Cheng Wu, and Chih-Liang Chou. 2016. Hybridtrace: A traceroute tool for hybrid networks composed of SDN and legacy switches. In *Computers and Communication (ISCC), 2016 IEEE Symposium on*. IEEE, 403–408.

[30] X. Wen, K. Bu, B. Yang, Y. Chen, L. E. Li, X. Chen, J. Yang, and X. Leng. 2017. RuleScope: Inspecting Forwarding Faults for Software-Defined Networking. *IEEE/ACM Transactions on Networking* 25, 4 (Aug 2017), 2347–2360. https://doi.org/10.1109/TNET.2017.2686443

[31] P. Zhang, H. Li, C. Hu, L. Hu, and L. Xiong. 2016. Stick to the script: Monitoring the policy compliance of SDN data plane. In *2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 81–86. https://doi.org/10.1145/2881025.2881038

[32] Y. Zhang, L. Cui, F. P. Tso, and Y. Zhang. 2017. Track: Tracerouting in SDN networks with arbitrary network functions. In *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. 1–6. https://doi.org/10.1109/CloudNet.2017.8071526

[33] Y. Zhao, P. Zhang, and Y. Jin. 2016. Netography: Troubleshoot your network with packet behavior in SDN. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. 878–882. https://doi.org/10.1109/NOMS.2016.7502919