SDN traceroute: Tracing SDN Forwarding without Changing Network Behavior

Kanak Agarwal
IBM Research
Austin, TX, USA
kba@us.ibm.com

Eric Rozner
IBM Research
Austin, TX, USA
erozner@us.ibm.com

Colin Dixon[*]
Brocade
San Jose, CA, USA
colin@colindixon.com

John Carter
IBM Research
Austin, TX, USA
retrac@us.ibm.com

ABSTRACT

Software-defined networking provides flexibility in designing networks by allowing distributed network state to be managed by logically centralized control programs. However, this flexibility brings added complexity, which requires new debugging tools that can provide insights into network behavior. We propose a tool, SDN traceroute, that can query the current path taken by any packet through an SDN-enabled network. The path is traced by using the *actual* forwarding mechanisms at each SDN-enabled device *without* changing the forwarding rules being measured. This enables administrators to discover the forwarding behavior for arbitrary Ethernet packets, as well as debug problems in both switch and controller logic. Our prototype implementation requires only a few high-priority rules per device, runs on commodity hardware using only the required features of the OpenFlow 1.0 specification, and can generate traces in about one millisecond per hop.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*network management, network monitoring*

Keywords

network management; network debugging; software-defined networking; datacenter; traceroute

1. INTRODUCTION

Software-defined networking (SDN) makes it easier to modify the control plane of networks in software through open protocols. While this can enable more efficient networks and support new features, it also potentially makes the network a more complex system. In an SDN environment, multiple SDN applications, services or administrators can independently and dynamically control the distributed forwarding state of the network by directly installing rules in switches. Furthermore, SDN programs and controllers often translate high-level configurations into low-level rules. The result is that it can be difficult for network operators to predict the exact low-level rules, and thus expected network behavior, that their high-level configuration will trigger.

When troubleshooting a problem, SDN programmers and network operators must grapple with many possibilities including bugs in controller logic, switches, individual SDN applications, and their composition. This makes it imperative to have tools that can provide visibility into how different packets are handled by the network at any given time.

Perhaps the simplest tool that provides visibility into a network is `traceroute`, which helps determine the network path to reach a certain host. However, this tool is extremely limited in troubleshooting SDN-enabled networks and determining forwarding behavior based on fine-grained packet header matching. The main limitation being that the `traceroute` tool can only provide the layer-3 (IP) path information because it relies on the *time-to-live* (TTL) field in the IP header to trigger ICMP error messages from intermediate routers. Furthermore, it assumes routing and forwarding are destination-based[1]. In this paper we present *SDN traceroute,* an alternate tool for measuring paths in an SDN-enabled network. In particular, it provides the path an arbitrary packet with user-defined header fields would take if it were injected into the network at a certain point. The path is reported as a list of ports on switches, enabling it to operate across all SDN-enabled devices regardless of network layer.

Of particular note is that SDN traceroute makes minimal assumptions about the correctness of controllers and switches, but instead measures the actual forwarding behavior of the network using a small number of high-priority rules to trap and re-inject the packet at each hop. In contrast to previous approaches [9, 10], our tool does not modify the existing rules in forwarding tables thus providing a clean separation between the observing process and the observed reality.

Rather than modify the rules in the measured switch, SDN traceroute leverages the switches surrounding the measured switch to observe its behavior while still using the production rules. We accom-

[*]This work completed at IBM.

HotSDN'14, August 22, 2014, Chicago, IL, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2989-7/14/08 ...$15.00.
http://dx.doi.org/10.1145/2620728.2620756.

[1]Newer versions of `traceroute` relax the destination-based assumption by allowing for the sending of TCP and UDP traffic as well as setting the source and destination port numbers, but still do not allow for tracing arbitrary packets or observing layer-2 hops.

plish this by coloring the switch graph to assign each switch a tag which is different from its neighbors. All probes carry the same tag as the switch they are trying to measure, while production traffic carries a single network-wide default tag.

We then install rules such that switches forward any traffic marked with a non-default tag other than their own to the controller. In this way, by changing the tag at each hop, the controller can push a probe through the network using the same rules it would have followed if it were production traffic. Our current implementation uses the VLAN priority bits to carry the tag, but nearly any other header field would suffice as long as the field is not used for making forwarding decisions in the network.

We argue that injecting and tracing a real packet through the network enables a network operator to debug issues in the controller and/or switches. We demonstrate this by showing a few real-world bugs that were difficult to trace down and where SDN traceroute would have helped.

The remainder of this paper is organized as follows. Section 2 reviews the existing approaches to determine forwarding behavior in SDN-enabled networks. Section 3 and 4 present the design and evaluation of SDN traceroute, respectively. Section 5 reviews the current state-of-the-art. We discuss limitations and alternatives in Section 6 before concluding in Section 7.

2. BACKGROUND

Broadly, approaches to determine forwarding behavior in SDN-enabled networks fall into two categories. The first builds a model of the forwarding behavior at each network element and then uses this model to answer questions about the network as a whole. The second either issues active probes or monitors production traffic to observe ground truth forwarding behavior. SDN traceroute falls in the latter category, but we discuss both approaches here. Section 5 provides a more complete description of related work.

2.1 Model-driven

Model-driven approaches are appealing because they work in two steps, the first gathers enough network state to build a model and the second uses that model to answer questions. This decomposition removes the physical switches as a possible bottleneck in the second step and allows for probes and queries to be done using only the model without having to involve actual network hardware.

With a few exceptions [11, 13], network state is gathered either by scraping rules from switches directly, e.g., via the CLI or SNMP [17], or by assuming that the controller manages to maintain a correct view of the rules in the network at all times. Both of these have practical limitations. Rule scraping often has high latency as it can take seconds or more [5, 6] to read all of the rules out of a given switch. Further, finding stable snapshots of rules can be difficult [20].

Additionally, as others have noted [9, 10, 19], model-driven approaches cannot be used to debug errors in the data plane itself and are often only able to debug errors in the controller to a limited extent.

An alternative is to carefully track network state by observing all programmed rules, predicting rule expirations and triggering targeted rule scraping. While some work has pushed in this direction [11, 13], we are not aware of any system that provides a comprehensive take and we think a tool like SDN traceroute would be an excellent complement to these approaches.

2.2 Active Probes and Monitoring

Sending active probes or monitoring production traffic allows for measuring the ground truth behavior of the data plane. However, doing so requires the network infrastructure to allow for traffic to be trapped and/or logged as it traverses the network. Existing approaches to do this either change the existing rules [9, 10] to also generate notifications or assume additional infrastructure to do this logging [19].

Generally, these approaches pay for their fidelity by consuming network resources and risk perturbing the network state. As discussed in the following sections, our SDN traceroute approach goes to great length to minimize both limitations.

3. DESIGN

Similar to `traceroute`, SDN traceroute employs low-overhead probe packets to measure network paths. It works in two phases. In the first phase, it colors each switch (or vSwitch) in the network using a graph coloring algorithm. Then, using the results of this coloring, it installs a small number of high-priority rules in every switch in the network which allow them to trap probe packets coming from their neighbors.

In the second phase, SDN traceroute injects a probe packet into the network to start tracing the route. The installed rules allow the packet to progress one hop before being returned to the controller. The controller then records the hop, slightly modifies the probe packet, and re-injects it to proceed to the next hop.

The remainder of this section describes the concrete goals and requirements of SDN traceroute, its interface, and both phases of operation before discussing the assumptions it makes about other SDN applications and the network.

3.1 Goal and Requirements

Our goal is to trace the path of a given packet using the *actual* forwarding rules in the network, with as little impact on the network as possible. This goal translates into the following key requirements:

1. **Non-invasive:** The existing rules in forwarding tables should remain unchanged. This ensures existing traffic will continue to flow without any changes and that SDN traceroute will not accidentally change the forwarding behaviors it is trying to measure.

2. **Accurate:** The existing forwarding rules should be applied directly to the probes as well as production traffic when measuring the behavior of a switch. This ensures that SDN traceroute is measuring the actual forwarding behavior.

3. **Low resource consumption:** SDN traceroute should require only a small number of rules per switch and update those rules infrequently to avoid using forwarding table space and wasting other network resources.

4. **Commodity hardware:** The approach should work on current commodity hardware using existing SDN protocols, e.g., OpenFlow [15][2].

5. **Arbitrary traffic:** It should be possible to trace the path of any flow and even any given packet from within a flow.

It should be noted that simultaneously being non-invasive and accurate is particularly difficult. Traditionally, to accurately trace a probe, the existing forwarding rules used by the production traffic are modified to log each hop, but this violates the non-invasive requirement by changing the state we are trying to measure.

[2]SDN traceroute requires only OpenFlow 1.0, instead of newer OpenFlow versions that are less commonly available.

3.2 Interface

SDN traceroute runs as an application on an SDN controller so that it can push rules to the switches and listen to OpenFlow messages. Since SDN traceroute is running on the centralized controller, we assume that it has access to the topology of the network. SDN traceroute provides a simple API to measure the current forwarding behavior: it accepts an arbitrary Ethernet frame with user-specified packet header fields and an injection point in the form of a switch identifier and port. After performing the trace route, the program returns an ordered list of <switch_id, port> pairs corresponding to each hop encountered by the packet as it traversed the network. If a forwarding loop is encountered, the program returns after the first repeated pair.

3.3 Network Configuration

Before sending any probes, SDN traceroute must install rules that allow it to selectively trap probes. The rules must support two different tasks: (i) matching the incoming probe packet so the hop can be logged at the controller and (ii) *not matching* the controller-returned probe as to forward the packet downstream. In this subsection, we outline what rules are installed into the switches in order to achieve the first task. In the next subsection we show how those rules perform the second task.

The SDN traceroute application begins by applying a graph coloring algorithm to the topology. The colors will serve as tags that are an integral part of the rules required by SDN traceroute. The coloring algorithm assigns each switch a color such that no two adjacent switches (switches directly connected via a link) are assigned the same color. It further attempts to minimize the total number of colors required to color the graph. Since this is an NP-hard problem, we use a greedy algorithm to color the vertices.

SDN traceroute requires all traffic to carry a color so that switches can decide whether or not to send a probe to the controller. Our implementation uses the VLAN priority field (three bits) to carry colors. If the topology is colored using three colors, we can assign a VLAN priority tag of (001), (010), and (011) to the switches colored with the first color, second color, and the third color, respectively. Note that the default tag of (000) is reserved for the production traffic and is not used during the tag assignment process. In general, if SDN traceroute uses k-bit tags, the topology must be $(2^k - 1)$-colorable. Many current datacenter topologies use a hierarchical tree structure consisting of core, aggregation and ToR switches. These topologies (including fat-trees [3]) require only 2-bit tags as trees are 2-colorable. Even when topologies are not actually trees, if they are layered (e.g., core, aggregation and ToR layers) they remain 2-colorable.

Our goal is to have switches log all packets except packets tagged with the default tag (production traffic) and those tagged with their own color. To meet this goal, SDN traceroute installs rules in each switch to match the color of all adjacent switches. These rules are assigned the highest priority (32,767 in OpenFlow) and, upon a match, forward the packet to the controller. Note that a switch's table does not contain any rules matching on its own color. An example topology with graph coloring and the corresponding rule configuration for route tracing is shown in Figure 1. The next subsection describes how the controller sends the packet back to the switch for further routing.

The number of rules installed in a switch depends on the number of colors used by its adjacent switches. As discussed previously, in most scenarios, this requires installing one or two TCAM rules in each switch. Thus the associated network configuration and the TCAM overhead of the rules pushed by the trace route application are quite minimal.

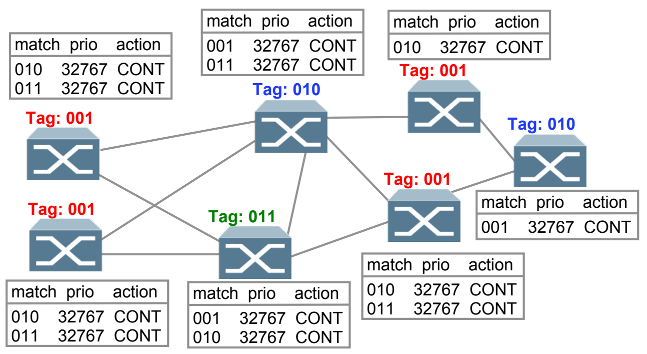

Figure 1: Example topology with graph coloring and corresponding tag-based rules. The shown topology is 3-colorable and a 3-bit header field is used for tag encoding. The figure also shows the tag matching rules installed by the trace route module in the switches. Here `CONT` refers to the send to controller action. All other match fields (not shown) are wildcarded.

Finally, note that these rules can be installed when SDN traceroute is first configured and need only be changed when the network topology changes.

3.4 Conducting the Trace Route

Once the network is configured in the manner discussed above, it is ready to accept Ethernet probe frames for route tracing. The process is best explained via an example, shown in Figure 2.

SDN traceroute begins by identifying the injection point. This is either identified in the API call or it is assumed to be the attachment point of the source host, which is looked up by source MAC or IP address. Once SDN traceroute has the injection switch identifier and port, it looks up the color of the ingress switch and inserts the color into the header tag bits of the probe frame.

SDN traceroute then sends the probe to the ingress switch as a `PACKET_OUT` message with the input port set to the injection point. The action for the `PACKET_OUT` is set to `TABLE`, indicating that the switch should treat the packet as though it had been received on the input port (step 1).

On receiving the `PACKET_OUT`, the ingress switch processes the packet in its flow table. Since the header tag bits in the packet are set to the color of the switch itself, the packet does not encounter a match on any of the high-priority rules SDN traceroute has installed. Consequently, the packet is forwarded to the next hop as though it were a regular, default-tagged packet (step 2). This ensures that the *actual* forwarding rules in the switch are used to route the packet even though it is a probe and not production traffic.

The packet arrives at the second switch while still carrying the header tag bits set to the color of the first switch. Since each switch is configured with high-priority rules that trap all packets matching the neighboring switches' colors, the packet at the second switch results in a match and is sent to the controller as a `PACKET_IN` (step 3). SDN traceroute receives the packet at the controller and logs the switch-id and port information of the switch that forwarded the packet to the controller as the next hop in the path.

Once SDN traceroute records the current hop, it modifies the received probe frame by rewriting the reserved tag field to the bits corresponding to the color of the current switch. It then sends the modified probe back as a `PACKET_OUT` to the same switch that had sent the `PACKET_IN` message. The input port in the `PACKET_OUT` is set to the input port where the packet was received at the switch. The action field is once again set to `TABLE`

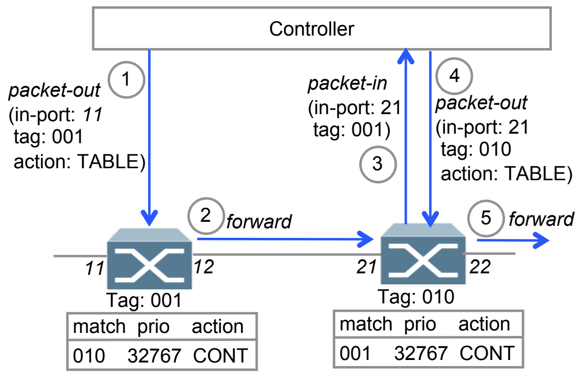
Controller
1 packet-out (in-port: 11 tag: 001 action: TABLE)
packet-in (in-port: 21 tag: 001)
3
4 packet-out (in-port: 21 tag: 010 action: TABLE)
2 forward
5 forward
11
12
21
22
Tag: 001
match prio action
010 32767 CONT
Tag: 010
match prio action
001 32767 CONT

Figure 2: Example showing `PACKET_IN` and `PACKET_OUT` exchanges between switches and the controller for route tracing. If an incoming packet at a switch does not match the tag-based matching rules configured by the trace route module, it is handled according to the production rules (not shown) configured by the other SDN modules.

(step 4). The switch receives the modified probe from the controller and applies its flow-table action on the probe. Since the reserved tag bits in the modified probe are set to the color of the switch, the tag based rules do not match and the packet is forwarded along the next hop as a regular frame (step 5).

This process (steps 3–5) repeats for each hop in the path. The process terminates when a time-out occurs between consecutive `PACKET_IN` events, indicating that the packet has left the network or been consumed by a host, or when a given <switch-id, port> is repeated in the route, indicating the presence of a loop.

Lastly, notice that in step 3 the trace route application only handles probe `PACKET_IN` messages that do not match the color of the switch sending the `PACKET_IN`. This allows `PACKET_IN` messages matching the input switch color to be forwarded to other modules in the controller for processing. This allows for scenarios where regular packet processing at a switch may itself initiate a `PACKET_IN` to the controller, such as in reactive rule installation.

3.5 Assumptions

The proposed trace route algorithm places a few restrictions on what other SDN applications are permitted to do. SDN traceroute assumes it can reserve k bits in packet headers exclusively for its use. These bits must not be used when making forwarding decisions in the network and must not be modified by any devices in the network. Furthermore, the bits must correspond to header field(s) that can be matched on using rules in the switches. In the case of OpenFlow 1.0, this means that any of the 12 matchable header fields can be used as long as they are not made available to the other SDN applications or modified by other network devices, e.g., middleboxes. A second assumption is that SDN traceroute reserves the highest priority rules. This is usually not an issue due to the large number of priorities (32,768 in OpenFlow) available for configuration and packet processing. Third, SDN traceroute assumes that the switch topology is $(2^k - 1)$-colorable.

Finally, we note that SDN traceroute explicitly *does not* assume routes are stable during the time it conducts a trace. If routes do change during a trace, it will return a route that a normal packet might have taken as the routes were changing. As with normal traffic, a trace might return a combination of the new and old routes if they are not installed in a way that provides consistent updates [16].

4. EVALUATION

We evaluate SDN traceroute by conducting two cases studies and microbenchmarks on our testbed consisting of five IBM RackSwitch G8264 OpenFlow-enabled switches connecting several commodity servers running Open vSwitch [2]. The case studies show how SDN traceroute can be useful in tracking down real-world bugs and undefined behavior in both controllers and switches.

We ran two microbenchmarks. The first repeatedly installed random routes and verified that SDN traceroute correctly discovered them. As the tool returned the correct path 100% of the time, we omit any further discussion. The second shows the latency of conducting traces on various network paths.

Our implementation of SDN traceroute is a module for the Floodlight controller [7] providing a REST API allowing a network operator to perform a trace route for an arbitrary packet. Our prototype consists of about 600 lines of code.

Undefined Switch Behavior: It is possible for a controller to not be able to accurately infer what forwarding action a switch may take. One clear example is when a bug in a switch could lead to undefined or unexpected behavior. Another, easier to reproduce, example is when a switch may contain *conflicting rules*, i.e., a set of rules that happen to match the same packet. OpenFlow provides a mechanism to install rule priorities to help resolve conflicting rules, but the forwarding behavior of a switch can be undefined when two conflicting rules have the same priority. As SDN networks may be programmed by many different programs, it is not unlikely that similar cases may occur.

We studied the behavior of our switches when conflicting rules have the same priority. First, a rule R_D that matches on destination D and outputs to port A was installed on a switch. Next, a rule R_S that matches on source S and outputs on a different port B was added. We then started a flow with source S and destination D and observed the behavior on the switch. We used port mirroring (to see the exact packets leaving A and B) and port counters to obtain the ground truth and compared that to SDN traceroute's output. Both the ground truth and SDN traceroute showed that the route corresponding to rule R_D, the first installed rule, was active. Next, we rebooted the switch and observed its behavior. After the reboot, we found that the *route changed*, and the route corresponding to rule R_S became active. Again, SDN traceroute was able to accurately infer the path.

This test highlights the benefit of using the actual forwarding mechanisms on the switch to infer routes. While conflicting rules in a switch's flow table could be detected at the controller, this also shows the ability to detect switch bugs that might manifest in ways that do not appear in flow tables.

Bugs in the Controller: Techniques to infer routes by analyzing state at the controller may be prone to errors or bugs in the controller itself. For example, consider a simple situation such as a port going down. The current stable version of the Floodlight controller [7] (v0.90 as of this writing) contains a bug that may prevent rules that forward to the downed port from being removed.[3] This in turn can prevent new routes from being installed.

Floodlight does not proactively push changes to switches, but instead relies on switches removing rules when the idle timeout of the rule fires. Then the next packet will miss, resulting in a `PACKET_IN`, which will cause the controller to either reprogram the rule or insert a changed version.

In the case of a downed port, the SDN controller waits for the `PACKET_IN` to notify it to reactively reroute traffic around the

[3]The bug is now fixed in nightly builds, but remains in the stable version.

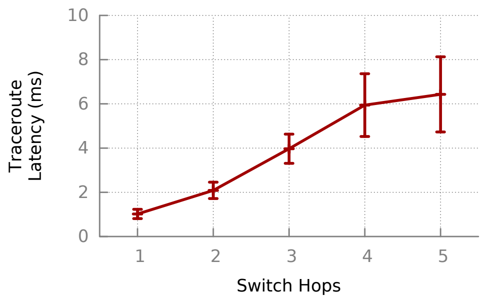

Figure 3: SDN traceroute latency to conduct a trace route, as a function of the number of hops.

downed port. However, developers found that preexisting periodic traffic that persists despite persistent loss, e.g., heartbeats, will continue to activate rules in the switches along the path to the downed port [1]. This prevents the rules from timing out, which in effect keeps the bad route active.

SDN traceroute can debug the problem by inferring the network is still routing packets to the downed port. A network operator could then identify the last hop of the path as the likely culprit, as well as verify that new traffic is routed correctly after the routes are updated.

This scenario is an example where even though the controller contains correct state, e.g., an updated view of the topology with the downed link and correct routes to avoid it, the network continues to behave in a different, and incorrect manner. The only way to find bugs where the data plane and control plane diverge is with a tool that measures the data plane directly, like SDN traceroute.

SDN traceroute Latency: Figure 3 shows the latency of conducting a SDN traceroute over paths of varying length. A *hop* is defined as a link between two switches. We created a variety of paths in our network, ranging from one to five hops and measured the latency. Our results show that SDN traceroute generally takes a little over 1 ms per hop, with the average per-hop latency taking 1.23 ms. Each point shows the average of 30 runs with error bars showing standard deviation.

5. RELATED WORK

Automatic Test Packet Generation (ATPG) [19] shares a similar goal to SDN traceroute as it looks at verifying the correctness of data plane forwarding by generating probe packets. However, it focuses on network-wide behavior rather than finding a particular path. It's worth noting that ATPG assumes the existence of test agents capable of injecting and capturing test probes. While they mention that OpenFlow could be used to accomplish the same task, they do not provide any implementation details.

NetSight [9, 10] provides a way to gather packet histories for a subset of traffic by modifying the existing rules in the network to send *postcards* for a subset of traffic that is being debugged. SDN traceroute works in a similar way, but it uses probe packets instead of monitoring the existing traffic. Our approach produces postcard-equivalents without having to modify any of the rules that handle production traffic. However, to accomplish this, it gives up the ability to determine the exact rule that matched for a packet and limits the number of probes in the network. Furthermore, NetSight assumes that switches can do line-speed truncation and mark packets with switch-specific information. SDN traceroute, on the other hand, runs on commodity hardware and requires no special support from switches other than the base OpenFlow 1.0 protocol implementation.

Anteater [14] uses SNMP [17] to gather rules from the forwarding database of each network device including firewalls, routers and switches. It then models each device as a boolean function describing what requirements must be met for a given device to forward a packet from an input interface to an output interface. When combined with a topology, this allows a SAT solver to validate certain network wide properties including loop-freedom, reachability, and consistency among replicated routers.

Header Space Analysis (HSA) [12] generalizes device forwarding behavior to a function that maps a packet header and input port to a (possibly empty) set of of (packet header, output port) pairs. With this model, HSA can check global network properties like loop-freedom and reachability for nearly arbitrary devices regardless of the protocols being used. NetPlumber [11] extends this approach to be incremental, reducing the time to check after a given rule change to about 1–10 minutes.

Veriflow [13] focuses on doing real-time invariant checking by doing incremental re-evaluation whenever new forwarding state is pushed into the network. It does this by capturing rules sent by the NOX [8] OpenFlow controller and making a minimal number of incremental changes by sorting traffic into forwarding equivalence classes and only checking the classes that have changed.

Libra [20] focuses on taking stable snapshots of forwarding state and computing correctness properties on it in very large networks, e.g., up to 10,000 switches. While the work notes that updates are bursty, these updates are in a traditional network rather than an SDN-enabled one and it is not clear if the periods of quiescence seen for 99.9% of the time would remain.

OFRewind [18] allows recording and playing back of SDN control plane traffic. Another line of work, including NICE [4], focuses on model checking controller applications rather than measuring the ground truth forwarding behavior.

6. DISCUSSION

In this section, we highlight some discussion points surrounding SDN traceroute. In particular, we contrast our probing based solution against rule scraping for inferring routes. We also discuss some practical implementation details and potential limitations of the work.

Scraping Switch State: An alternative to using SDN traceroute would be to continuously scrape rules from switches to build a complete understanding of the network's forwarding behavior and then issue logical probes against this state.

We believe using the actual switches and their forwarding tables is beneficial for many reasons. First, the default setting in some controllers, such as Floodlight, installs routes in the network reactively. Switches today have limited TCAM space which prevents upfront static installation of all forwarding rules on the switch. In this scenario, rules are actively swapped in and out of the local cache dynamically. In this reactive mode of operation, packets that do not match any rules in the switch's forwarding table are sent to the controller and only then does the controller insert forwarding rules for the packet into the switch. Therefore, an approach that simply scrapes the rules on the switch could not determine the path based on a snapshot of the existing rules in the switches.

Second, as mentioned in Section 2, constantly scraping a large number of rules from a large number of switches can be burdensome. SDN traceroute can determine the routes of packets in a lightweight manner by only communicating with the switches in the path.

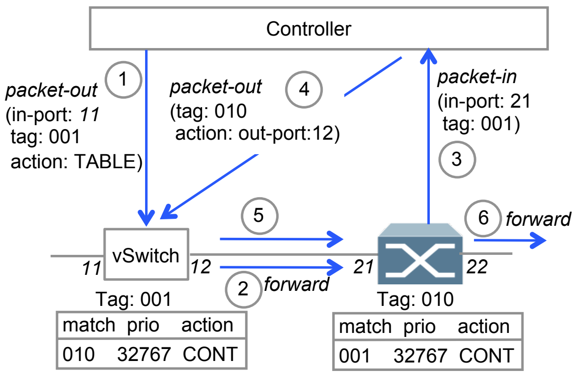

Figure 4: Example showing the trace route process when the hardware switches do not support the `TABLE` command. The process starts at vSwitch with each subsequent switch applying its forwarding rules on the controller injected packets received from its previous hop.

Third, a bug in the switch may cause inappropriate forwarding behavior or packet drops, which would not be discoverable through analyzing scraped rules.

TABLE Action Support: We noticed that some switches do not support the `TABLE` action, even though it is specified as required in OpenFlow 1.0. SDN traceroute can work with OpenFlow switches that do not support the `TABLE` command by running Open vSwitch, which supports the `TABLE` command, on the servers in the network. Alternately, a simple host agent for injecting probes could be used. SDN traceroute begins the trace route by sending the first `PACKET_OUT` to the Open vSwitch instance running on the flow source's server. Since Open vSwitch supports the `TABLE` command, SDN traceroute performs as expected with the virtual switch forwarding the packet to the next hop and the subsequent switch A sending the packet to the controller as a `PACKET_IN` with its input port information.

Instead of returning the packet labeled with A's color back to A with the `TABLE` action set, SDN traceroute uses the topology information to deduce the output port of the previous switch and sends the packet to the *previous* switch with the `PACKET_OUT`'s action set to output the packet on the port towards A. Now when the packet arrives at A, its normal flow table actions are applied thus emulating the `TABLE` action. This process continues through the network using three switches at a time to discover each hop rather than two. This only requires that the first switch in a path support the `TABLE` action.

Figure 4 shows the different steps involved in the process. The key difference is in steps 4 and 5, where instead of the sending the packet back to the switch itself with the `TABLE` action, the controller sends the packet to the previous hop switch as `PACKET_OUT` with its action set to output the packet on the port towards the intended switch.

Limiting SDN traceroute packets Ideally, SDN traceroute's probe packets would never arrive at a device that wouldn't trap them and send them to the controller. However, because SDN traceroute can only detect the end of path with a timeout, the last probe packet is likely to reach a host or leave the network. This can be prevented by installing rules in end-host firewalls and a firewall at the edge of the network to prevent packets from leaving the network.

Dealing with Middleboxes: While SDN traceroute cannot tell what middleboxes a given packet will traverse, it *can* tell what the complete path of a packet is even if it happens to traverse middleboxes as long as middleboxes do not modify the tag field. Model-based approaches using controller state or switch rules cannot determine how packets may interact with middleboxes unless middleboxes are explicitly included in the model.

7. CONCLUSION

In this paper, we present SDN traceroute, a tool for tracing the path of a network flow in SDN-enabled networks. By using the rules within the switches themselves, SDN traceroute is able to debug arbitrary flows and packets with no special support needed from switches in the network beyond basic OpenFlow 1.0. The key feature of the tool is that it can apply the same forwarding rules to the probes as those used by the production traffic without requiring any changes to the actual forwarding rules. Furthermore, SDN traceroute requires upfront installation of only a small number of rules per switch resulting in a very low resource overhead. We envision SDN traceroute as an integral part of any SDN administrator's toolkit for managing and troubleshooting the network.

References

[1] Floodlight-developers mailing list. https://groups.google.com/a/openflowhub.org/d/topic/floodlight-dev/HpB-TpASXmM/discussion.

[2] Open vSwitch. http://www.http://openvswitch.org.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[4] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[5] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *SIGCOMM*, 2011.

[6] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM*, 2010.

[7] Floodlight openflow controller. http://floodlight.openflowhub.org/.

[8] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards an operating system for networks. In *CCR*, 2008.

[9] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, 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 *NSDI*, 2014.

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

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

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

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

[15] OpenFlow-switch. https://www.opennetworking.org/standards/openflow-switch.

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

[17] Version 2 of the protocol operations for the simple network management protocol (SNMP). RFC 3416. http://www.ietf.org/rfc/rfc3416.txt.

[18] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *USENIX ATC*, 2011.

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

[20] 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 *NSDI*, 2014.