Enabling Layer 2 Pathlet Tracing through Context Encoding in Software-Defined Networking

Hui Zhang, Cristian Lumezanu, Junghwan Rhee, Nipun Arora, Qiang Xu, Guofei Jiang
NEC Laboratories America
{huizhang,lume,rhee,nipun,qiangxu,gfj}@nec-labs.com

ABSTRACT

Troubleshooting Software-Defined Networks requires a structured approach to detect mistranslations between high-level intent (policy) and low-level forwarding behavior, and a flexible on-demand packet tracing tool is highly desirable on the data plane. In this paper, we introduce a Layer 2 path tracing utility named PathletTracer. PathletTracer offers an interface for users to specify multiple Layer 2 paths to inspect. Based on the Layer 2 paths of interests, PathletTracer then accounts paths with identifiable IDs, and installs a set of flow table entries into switches to imprint path IDs on the packets going through. PathletTracer re-uses some fields in packet headers such as the `ToS` octet for recording path IDs. To efficiently carry imprints using limited bits, PathletTracer uses an encoding algorithm motivated by the calling context encoding scheme in the software engineering domain. With k bits for encoding, PathletTracer is able to trace more than 2^k paths simultaneously.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer - Communication Networks- Network Communications

Keywords

Software-Defined Networks (SDN), OpenFlow, Network Monitoring, Network Management, Troubleshooting

1. INTRODUCTION

Software-Defined Networking (SDN) enables flexible network management by separating forwarding decisions (in the control plane) from the forwarding itself (in the data plane). This allows operators to manage networks using high-level abstractions that are automatically translated into low-level functionality [12, 15]. For correct troubleshooting of SDN operations [9], it is necessary to verify whether the low-level actions performed by network devices conform to the high-level policies deployed by operators.

HotSDN'14, August 22, 2014, Chicago, IL, USA.
Copyright 2014 ACM 978-1-4503-2989-7/14/08 ...$15.00.
http://dx.doi.org/10.1145/2620728.2620742.

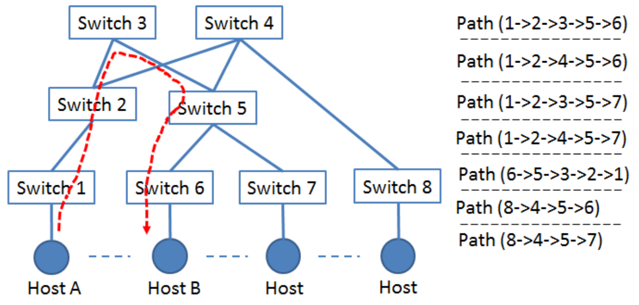


Figure 1: An example L2 network with seven paths of interest to be traced.

In this paper we focus on *path tracing*, a specific operation for SDN troubleshooting that determines the Layer 2 path taken by a given packet. Consider the network in Figure 1, where host `A` can reach host `B` via multiple paths. Path tracing allows us to verify whether a packet from host `A` has taken the desired route, such as `1->2->3->5->6`, to reach host `B`. Path tracing can help network operators to improve network performance (*e.g.*, by comparing various routing options in load balancing), to validate routing decisions (*e.g.*, by ensuring that a routing algorithm performs correctly), and to allocate resource optimally (*e.g.*, by identifying hot and cold spots in networks).

Existing approaches to determine the Layer 2 path of a packet in a software-defined network are limited to the control plane [10, 14, 7] or introduce significant overhead [7, 14]. For example, `VeriFlow` [10] analyzes the configuration pushed to network devices to infer forwarding paths and determine inconsistency. However, even if the configuration is correct, the actual forwarding in the data plane may not follow the configuration due to bugs in switch software, conflicts with existing configuration, or limited memory space to enforce the configuration [9].

`ndb` [7] is a network debugger for SDN which emits postcards from every switch that the traced packet traverses. A postcard is a logging packet that contains information about the traced packet and the flow entry it matched. The network controller collects all postcards and reconstructs the packet path. The challenge of this approach is the overhead of logging added to the control plane. `OFRewind` [14] records guest network traffic by mirroring those packets on traversed switches, therefore can trace back the paths they have taken. Such approach comes with the expense of additional in-network instrumentation.

We introduce PathletTracer, a Layer 2 path tracing utility that is both correct and scalable. PathletTracer offers an interface for users to specify multiple Layer 2 paths to trace, and get the information on which of the traced path a later packet has taken, or none of them. For correctness, PathletTracer records forwarding information in the data plane as a packet traverses a switch, rather than infer it from switch configuration. Specifically, PathletTracer associates each path to be traced with an ID and uses unused bits (*e.g.*, ToS octets) in a packet's header to carry the ID of the path the packet is traversing. Specialized forwarding rules installed by the centralized controller ensure that each switch in the network imprints path IDs to packets in transit. Once a packet has arrived at the destination, PathletTracer decodes the ID to determine the path traversed.

The scalability of PathletTracer depends on how, where, and when the centralized network controller enforces the recording of the path information by the switches. We preserve scalability by making several design decisions about how we specify and encode paths.

Pathlet tracing. PathletTracer borrows the pathlet idea from Pathlet Routing [6]. The underlying intuition is that the number of path segments (i.e., pathlets) is much less than the number of source-destination combinations, especially in a virtualized environment where we consider VM-to-VM paths. This helps reducing the number of IDs that we encode in the header of traced packets.

Calling context encoding. We use precise calling context encoding (PCCE) [13] to ensure that we can encode each path within the limited space available in the packet header. PCCE is a software engineering technique to represent the calling context, a path of function calls from the root (e.g., `main`) function, of a program in concise values at runtime. PCCE analyzes the structure of a program and inserts arithmetic operation (e.g., addition) code to update concise runtime status such that it can be uniquely decoded relative to a calling context.

There are several challenges when adapting PCCE to encode network paths. First, unlike a program which executes a single piece of code, a network infrastructure consists of a set of switches which need to be configured and coordinated in a distributed way. Second, the arithmetic operations used in PCCE to update runtime status are not supported in existing OpenFlow table action sets.

PathletTracer presents a set of algorithms to create and deploy switch forwarding rules for encoding pathlets while overcoming those problems. The contributions of the paper are as follows:

- We introduce pathlet tracing, a new and flexible mechanism for packet tracing on the SDN data plane.
- We describe a novel Layer 2 path encoding mechanism using unused octets in a packet header by adopting the software engineering approach to encode program calling context. For example, as we will show in Section 4, PathletTracer requires only 2 bits to trace the 7 paths in the example network shown in Figure 1.

2. BACKGROUND - CALLING CONTEXT ENCODING

A *calling context* is the sequence of active function invocations that lead to a program location. This information is widely used for various purposes in software engineering such as profiling, performance optimization, bug detection, etc. There are several approaches proposed to obtain this information in an efficient way. Sumner et al. proposed Precise Calling Context Encoding (PCCE) that provides a precise encoding of calling context with reliable decoding capability [13] evolved from path profiling [3]. PCCE encodes the calling context using a small number of integer identifiers. It inserts code which updates an integer ID so that the calling context at any program point can be uniquely represented by the ID along with the program counter.

The algorithm calculates the update values in the static analysis of the program call graph. Figure 2 shows that the pair of a program counter (PC) and an ID can uniquely represent a program path from the root node. For instance, the ID value is 1 at the function `E` means that the program took the path, `ACDE`, in the execution.

We see several analogies between calling context encoding and pathlet encoding in SDN: (1) the call graph of a software and the topology of a network, (2) a call site and a network switch, and (3) a calling context and a network path. This similarity inspires us to design an on-the-fly encoding approach for L2 path encoding called PathletTracer.

Context	PC:ID
A	A:0
AB	B:0
AC	C:0
ABD	D:0
ACD	D:1
ABDE	E:0
ABDF	F:0
ACDE	E:1
ABDFG	G:2
...	...

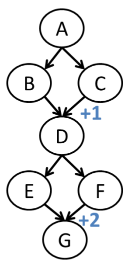


Figure 2: An example of PCCE encoding. Edge annotations (e.g., +1) represent arithmetic operations to update encoding values of calling context.

3. ARCHITECTURE

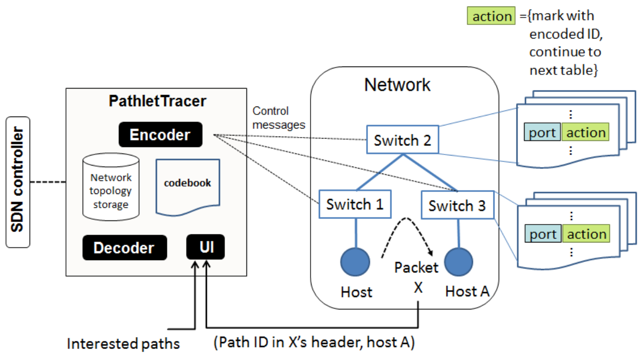
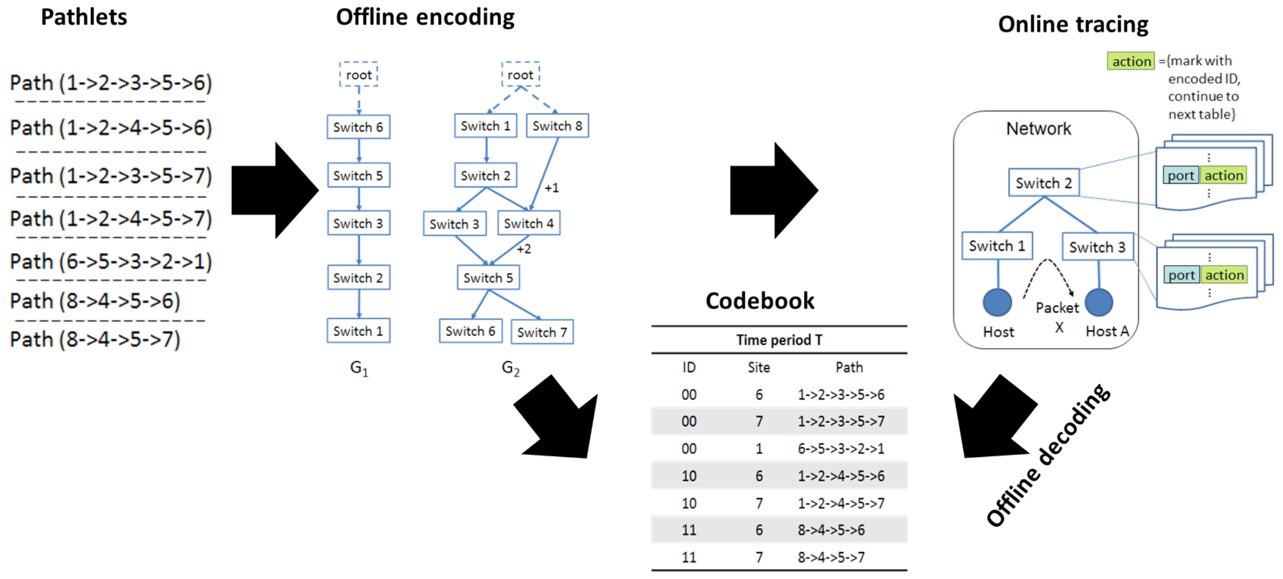


Figure 3: The architecture of PathletTracer.

Figure 3 presents the main components of PathletTracer:

- The user interface (UI) can take the following inputs: the interested paths to be traced and encoded, and the path ID from a received packet for decoding.



Time period T		
ID	Site	Path
00	6	1->2->3->5->6
00	7	1->2->3->5->7
00	1	6->5->3->2->1
10	6	1->2->4->5->6
10	7	1->2->4->5->7
11	6	8->4->5->6
11	7	8->4->5->7

Figure 4: PathletTracer workflow.

- The encoder generates the codebook and the corresponding set of forwarding rules for SDN switches based on the network topology and paths of interests to stamp the traversing packets with compact IDs for the paths.
- The decoder will use the codebook from the encoder to translate the path ID into a hop-by-hop path.

Figure 4 shows an overview of the PathletTracer workflow. Given a set of pathlets (valid Layer 2 path segments), PathletTracer compiles it with a directed acyclic graph model and applies a path encoding algorithm. This leads to a set of control messages to add flow table entries on switches for encoding packets on-the-fly. This process also generates a codebook to translate a packet ID to a Layer 2 path which the packet has taken.

In Sections 4 and 5, we describe how PathletTracer encodes paths and enforces the recording of path IDs in switches.

4. OFFLINE ENCODING

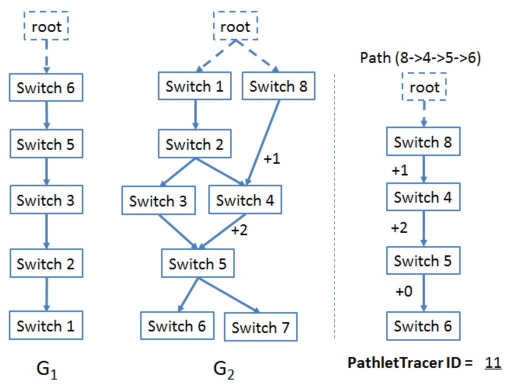


Figure 5: An encoding example with seven paths.

In this section, we describe how we encode network paths using concepts from the calling context encoding. First, the encoder builds a forest of directed acyclic graphs (DAGs) by composing the valid paths to be traced. On each DAG, the encoder creates a virtual root node, and adds a link from it to all the nodes with 0 indegree. For example, Figure 5 shows two DAGs by composing the seven paths in Figure 1. G_1 contains only the path $6 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 1$ as it does not share any link with the other 6 paths; G_2 contains the rest of the 6 paths as they share some links between each other.

Algorithm 1 Path encoding

```
function ENCODING($G = (N, E)$)
    for $n \in N$ do
        $PP[n] \leftarrow CalculatePossiblePaths(n)$
    for $e \in E$ do
        $PN[e] \leftarrow 0$
    for $n \in N$ in a topological order do
        for $e = \langle p, n \rangle$ of the incoming edges of $n$ do
            if $e$ is the first edge then
                continue
            else
                $PN[e] \leftarrow PP[p]$
    while DepthFirstSearch(G) do
        if reach an ending node $n$ of any traced path through a path $p$ then
            $ID[p] \leftarrow$ sum of each edge's $PN[e]$ on $p$
function CALCULATEPOSSIBLEPATHS($n$)
    $a \leftarrow$ the number of possible paths from the virtual root node to $n$
    return $a$
```

On each DAG, the encoder generates the IDs of the included paths. The algorithm starts by traversing the nodes in a topological order, and computes the path number (PN) for each edge. With respect to each node n, the PN value for the first incoming edge is 0; for each of the remaining edges $e : (p \rightarrow n)$ the PN value is the sum of the possible paths from the root node to p. The ID of a path with respect to a node n is the sum of PNs of all edges on the path from the virtual root node to n. Algorithm 1 describes the steps. For example, G_2 in Figure 5 has all 0-PN edges except $e : (8 \rightarrow 4)$ and $e : (4 \rightarrow 5)$. The ID of Path $8 \rightarrow 4 \rightarrow 5 \rightarrow 6$ is therefore $(1 + 2 + 0) = 3$ (11 in binary).

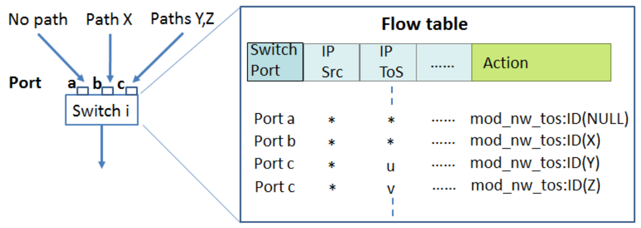
Time period T		
ID	Site	Path
00	6	1->2->3->5->6
00	7	1->2->3->5->7
00	1	6->5->3->2->1
10	6	1->2->4->5->6
10	7	1->2->4->5->7
11	6	8->4->5->6
11	7	8->4->5->7

Figure 6: A codebook example on the 7 paths in Figure 5.

After the encoding, the encoder produces a codebook for the traced paths. The codebook includes four fields: the time period T when the path IDs are valid; path identifiers (ID), the end switch on the path (Site), and the encoded path (Path). Figure 6 shows an example of the code book for the 7 paths traced in Figure 5.

The decoder uses the codebook to serve path queries. When a user sends a query on a path ID i encoded in a packet received at host x at time t, the decoder first uses the network topology information to resolve x to the switch (site) s where x is attached. The decoder then looks up the codebook with (t, i, s) and returns the full path information matching the 3-tuple value.

5. ONLINE TRACING



Switch Port	IP Src	IP ToS	......	Action
Port a	*	*	......	mod_nw_tos:ID(NULL)
Port b	*	*	......	mod_nw_tos:ID(X)
Port c	*	u	......	mod_nw_tos:ID(Y)
Port c	*	v	......	mod_nw_tos:ID(Z)

Figure 7: 3 types of switch ingress ports and the corresponding flow table entry examples for pathlet tracing.

Next we describe how tracing a path using the path IDs generated in Section 4 works. After computing the path IDs, the encoder generates control messages for all switches in the DAGs to enable online path tracing. These messages create rules in switches to imprint the encoded IDs to packets in transit. While the original PCCE work used arithmetic addition operations to record this information (e.g., ID = ID + 4), similar operations on packet header fields are not supported in existing OpenFlow table action sets; thus we need to make per-site additions in calling context encoding OpenFlow-compatible.

We solved this challenge by generating flow table rules based on the topology relationship with adjacent switches expressed in terms of ingress ports of each switch.

In general, there are three types of ingress ports on a switch in the DAGs for path tracing regarding how a switch is connected to its neighbors : (a) a port that no traced path traverses, (b) a port traversed by traced paths sharing the same ID value, and (c) a port traversed by traced paths with different IDs.

For example, switch i in Figure 7 has the link ending at its ingress port b in path X, and the link ending at port c in path Y and Z which have different encoding IDs. Therefore, the ingress port a of switch i is type (a), port b is type (b), and port c is type (c). For each type of port, PathletTracer adds a different set of flow table entries to enable switch i to imprint the path ID into the ToS bits of packets. Other fields can be in the matching rules if the traced paths are associated with specific flows.

For an ingress port of type (a), switch i sets the path ID field as the default *no-path* value $ID(NULL)$. When we choose to use upper 6 bits in the type of service (ToS) field to carry the path ID information, the encoder adds a flow table entry to switch i that assigns the value of $ID(NULL)$ to the ToS bits of all incoming packets on the port. For the first switch in a DAG (e.g., switch 6 in G_1), each of its ingress ports is type (a) unless that port's corresponding link is a traced object (e.g., the ingress port of switch 6 connecting to 5). When $ID(NULL)$ is chosen as the field default value (*e.g.*, 0 for ToS) and modifications on such fields are caused only by path tracing, the control messages and the resulting table entries for the type (a) ingress ports may be waived.

For an ingress port of type (b), switch i needs to set the path ID field as the unique path ID value $ID(X)$. The encoder adds a flow table entry to switch i that sets the ToS bits of all incoming packets on the port to $ID(X)$. As an optimization, the encoder does a depth-first search from the virtual root node, and find the first switch i in every path whose incoming link has only the paths with $ID(X)$ traverses into; only i on that path are added with the table entry for $ID(X)$ setting.

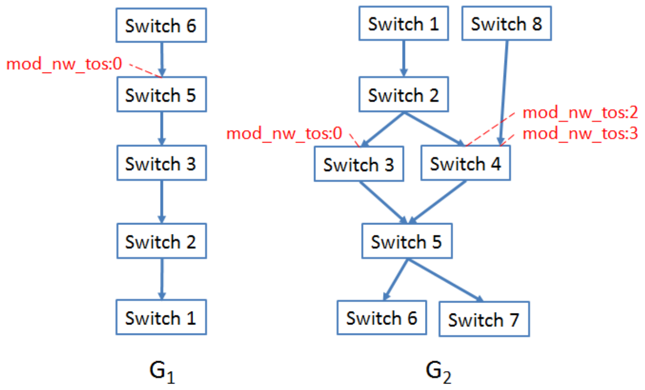


Figure 8: The 4 flow table entries for tracing 7 paths.

We use Figure 1 to illustrate the steps. The encoder will add four flow table entries for the seven paths. For path $6 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 1$, only switch 5 needs to add a table entry for the path ID 0, as its ingress port connecting to switch 6 is type (b) and is the first such ingress port along the path. Similarly, the ingress port on switch 3 (connecting to switch 2) is type (b) for two paths with ID 0; the ingress port on switch 4 (connecting to switch 2) is type (b) for two paths with ID 2; and the ingress port on switch 4 (connecting switch 8) is type (b) for 2 paths with ID 3. Switch 3 (4) is the first switch with a type (b) port having only the path with ID 0 (2 or 3). Actually, these four flow table entries are the only ones that the encoder needs to create for the seven paths if we reserve 0 for the default value $ID(NULL)$ and replace the three paths with ID 0 by the value 1.

For an ingress port of type (c), the encoder first checks whether any of the incoming paths traverses an ingress port of type (b) prior to reaching the current port. Such paths are then removed from consideration. If the path set becomes empty, no flow table entries are required. Otherwise, the encoder adds a set of entries to realize the ID addition operations if the PN value of the edge e ending at this port is not 0. The encoder searches the path from the virtual root node to the switch i for all non-0 PN values excluding $PN(e)$. For each possible combination of those PN values and value 0, we create a flow entry that matches all incoming packets with ToS value equal to the sum of the PN values in the combination and modifies the ToS value to be the sum of values in the combination plus $PN(e)$.

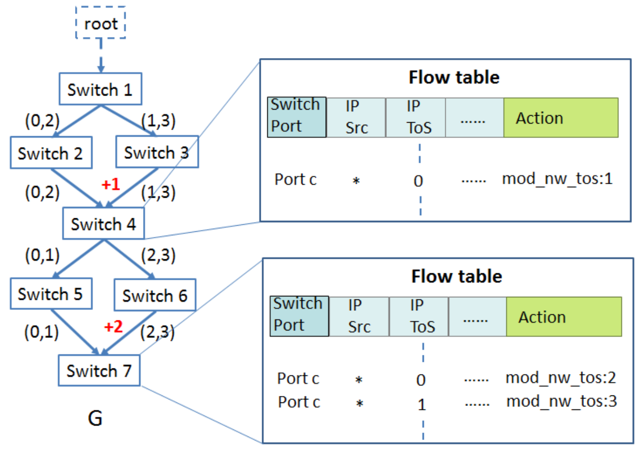


Figure 9: An example of type (c) ingress ports and the corresponding flow table entries.

For example, Figure 9 shows four paths for tracing and its DAG. Described in Section 4, the offline encoding algorithm will assign the paths IDs (0,1,2,3), and all edges with PN value 0 except the edges $3 \rightarrow 4$ ($PN = 1$) and $6 \rightarrow 7$ ($PN = 2$). For convenience, we also label each edge with the IDs of the paths traversing it. As none of the four paths traverses any type (b) ingress port, the encoder creates flow table entries in switches 4 and 7 which have incoming edges with non-zero PN. For switch 4, only one entry is created to stamp the ToS field with the PN value 1. As for switch 7, two entries are created for the PN combinations of $1, 0$, and accordingly stamp the ToS field with the value $(1+0+2) = 3$ and $(0 + 2) = 2$.

Using the forwarding rules and actions for the three types of ingress ports, PathletTracer tracks the packets traversing the encoded paths. If a packet traverses an encoded path completely, its imprint will be that path's ID. Otherwise, it is set with the value $ID(NULL)$.

6. DISCUSSION

The work presented in this paper is only a starting point toward a fully fledged realization of the PathletTracer architecture. In this section, we discuss several open issues in its design.

Path ID collection. A destination node (e.g., a VM) requires a method to collect the path IDs embedded in the headers of its interested packets. One method is running tcpdump within the node. Another method from network operator side is running tcpdump on Open vSwitch (OVS) [1], the de facto software switch running within the hypervisor. A third method is the postcard collection idea in ndb [7] but applied only onto the last switch in a traced pathlet.

ToS limitations. Using the ToS field could be a problem, both in terms of the number of bits, and because there are often QoS-related usages for that field. FlowTags [4] has suggested several ways to overcome the limitations from encoding with the ToS field. For example, one option is to extend OpenFlow to match on the 16-bit IP ID field (if fragmentation is disabled), or to use the 20-bit flow-label in IPv6.

Admission control for tracing. PathletTracer has to take admission control and reject some pathlet tracing requests when it runs out of encoding bits and/or flow-table space. A simple admission control policy can be request arrival time based, and more complicated policies may take factors such as user priority and pathlet lengths into consideration.

Multiple tables. Since V1.1, OpenFlow switches introduce a flexible pipeline with multiple tables. Packets are processed through the pipeline; they are matched and processed in the first table, and may be matched and processed in other tables afterwards. As the flow-table entries from PathletTracer is only for tracing purpose, they can be positioned in the beginning of such a pipeline, therefore will not affect the rest of actions for routing, access control lists, rate limiters, etc.

Tracing layers. While we present PathletTracer in a Layer 2 network, the pathlet tracing idea can also be applied to a $L2 + L3$ network under a single operator's administration (e.g., a data center network). For OpenFlow routers, PathletTracer can treat them the same as switches in calculating path IDs and installing new flow table entries.

Multi-path routing. There can be many possible paths between a pair of hosts in a network architecture supporting multi-path routing. For example, there are 576 equal-cost paths between any given pair of hosts in different pods of a fat tree built from 48-port GigE switches [2]. To trace all P paths between a pair of hosts in a fat-tree network, PathletTracer has to use $log(P)$ bits to encode the paths as they will compose one DAG. The edge-tier switch connecting the destination host has to use $\theta(P)$ entries as all its up links to the aggregation tier are with type (c) ports where P paths traverse. When tracing multiple sources to the destination host, the required encoding bits will grow logarithmically with the number of sources, and the flow table entries used by the edge-tier switch connecting the destination host will grow linearly with that.

7. RELATED WORK

Troubleshooting SDNs. A number of solutions, such as `Anteater` [11], `VeriFlow` [10], and `Libra` [16], have been proposed to troubleshoot SDNs. These works focus mainly on the elimination of configuration conflicts, the avoidance of routing loops and black holes, the detection of policy inconsistency, and are complementary to PathletTracer. PathletTracer can be combined with them or incorporated into such open-platform solutions as a critical component for validating the actual Layer 2 paths. PathletTracer provides a scalable real-time Layer 2 path tracing function as a essential feature in SDN troubleshooting.

Network Tracing Utilities. Carrying information in packet headers for tracing is not a new idea. For example, X-Trace [5] is a tracing framework designed to reconstruct an Internet service's task tree by propagating task ID metadata across layers and across applications. FlowTags [4] adds tags to outgoing packets for systematic policy enforcement on switches and middleboxes. Compared to them, PathletTracer does not require on-path logging of traced packets. Among the existing SDN troubleshooting solutions, PathletTracer has some overlap with systems such as `ndb` [7] and `NetSight` [8]. A unique contribution from PathletTracer is that it traces the ground-truth data-plane forwarding paths and its pathlet-oriented tracing reduces the overhead in the control plane significantly, *i.e.*, $\mathcal{O}(\texttt{P})$ with `P` referring to the number of paths subject to inspect rather than $\mathcal{O}(\texttt{p})$ or $\mathcal{O}(\texttt{f})$ where `p` and `f` denote the numbers of packets and flows respectively.

8. CONCLUSION

This paper presented the design of a new Layer 2 path tracing inspired by calling context encoding, a software engineering approach, to encode packet forwarding in the data plane using programmable SDN switches. Its pathlet-oriented design allows scalability and flexibility in troubleshooting data plane problems. With many open issues motivating our future exploration, we believe that PathletTracer complements existing SDN trouble-shooting and measurement tools by providing a simple yet effective packet tracing solution.

9. REFERENCES

[1] Open vSwitch: An Open Virtual Switch. `http://openvswitch.org/`.

[2] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 63–74.

[3] Ball, T., and Larus, J. R. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture* (1996), MICRO 29, pp. 46–57.

[4] Fayazbakhsh, S. K., Sekar, V., Yu, M., and Mogul, J. C. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (New York, NY, USA, 2013), HotSDN '13, ACM, pp. 19–24.

[5] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association, pp. 20–20.

[6] Godfrey, P. B., Ganichev, I., Shenker, S., and Stoica, I. Pathlet routing. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (2009), SIGCOMM '09, ACM, pp. 111–122.

[7] Handigol, N., Heller, B., Jeyakumar, V., Maziéres, D., and McKeown, N. Where is the debugger for my software-defined network? In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks* (2012), HotSDN '12, ACM, pp. 55–60.

[8] Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., and McKeown, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proceedings of NSDI'14* (2014), USENIX, pp. 71–85.

[9] Heller, B., Scott, C., McKeown, N., Shenker, S., Wundsam, A., Zeng, H., Whitlock, S., Jeyakumar, V., Handigol, N., McCauley, J., Zarifis, K., and Kazemian, P. Leveraging sdn layering to systematically troubleshoot networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013), HotSDN '13, ACM, pp. 37–42.

[10] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation* (2013), USENIX.

[11] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (2011), SIGCOMM '11, ACM, pp. 290–301.

[12] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing software defined networks. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation* (2013), USENIX, pp. 1–13.

[13] Sumner, W. N., Zheng, Y., Weeratunge, D., and Zhang, X. Precise calling context encoding. *IEEE Transactions on Software Engineering 38*, 5 (2012), 1160–1177.

[14] Wundsam, A., Levin, D., Seetharaman, S., and Feldmann, A. Ofrewind: Enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (2011), USENIX ATC'11, USENIX Association, pp. 29–29.

[15] Yu, M., Jose, L., and Miao, R. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), NSDI'13, USENIX Association, pp. 29–42.

[16] Zeng, H., Zhang, S., Ye, F., Jeyakumar, V., Ju, M., Liu, J., McKeown, N., and Vahdat, A. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proceedings of NSDI'14* (2014), USENIX, pp. 87–99.