FlowTracer: An Effective Flow Trajectory Detection Solution Based on Probabilistic Packet Tagging in SDN-Enabled Networks

Junxiao Wang, Heng Qi, *Member, IEEE*, Yang He, Wenxin Li, Keqiu Li, *Senior Member, IEEE*, and Xiaobo Zhou

***Abstract*—Currently, parallel data transmissions in large-scale datacenter networks are becoming increasingly crucial to application performance. Despite fine-grained control by SDN-enabled networks, some transmission errors, such as misconfigurations, will inevitably occur, resulting in high-level forwarding policies that cannot be conformed to at the data plane. Therefore, flow trajectory detection is very important for allowing datacenter network operators to troubleshoot problems and ensure that all traffic flows are running on the correct paths. However, existing solutions detect flow trajectories by recording the entire path of each packet. These methods are prone to imposing significant overheads in terms of both the number of switch entries and the amount of packet header space required. To considerably reduce this overhead, we present FlowTracer, an efficient flow trajectory detection solution, which can sample a path one link at a time instead of recording the entire path. FlowTracer consists of a method of probabilistic packet tagging and a method of trajectory reconstruction. In this paper, we first introduce the method of probabilistic packet tagging, which is performed in OpenFlow-enabled switches with very few switch entries and limited packet header space by means of double VLAN tags. Then, we explore the topological structure of datacenter networks and propose our method of trajectory reconstruction, which is performed at end hosts and achieves rapid convergence. Finally, we evaluate FlowTracer on a 48-ary fat-tree topology. The results show that FlowTracer can detect trajectories quickly while placing far smaller demands on both switch entries and packet header space than state-of-the-art techniques.**

***Index Terms*—Datacenter, SDN, trajectory detection, probabilistic packet tagging, trajectory reconstruction.**

I. Introduction

DATACENTER networks are currently essential for carrying traffic for a variety of applications, from online Internet services, e.g., Web searches, social media, and online commerce, to distributed computing services, e.g., MapReduce, Spark and Pregel. The parallel data transmissions in datacenter networks are thus becoming increasingly crucial to application service performance. Although software-defined networking (SDN) [1] has emerged as a key technology for making transmission management easier and more fine-grained [2], experience shows that transmission errors will inevitably occur [3], [4] due to two main causes. The first is unexpected switch port failure [5], [6], causing traffic flowing through the affected switch to be forwarded to another port that belongs to a non-shortest path. The second cause is control plane transaction conflict [7] or unsynchronized controller instances [8], which also cause traffic to be forwarded along incorrect paths. These transmission errors result in high-level forwarding policies (expressed at the control plane) that cannot be conformed to at the data plane. Therefore, it is very important for datacenter network operators to troubleshoot problems and ensure that all traffic is running on the correct paths.

However, trajectory detection in large-scale SDN-enabled datacenter networks poses severe challenges. Conventional tools, e.g., ping, traceroute, SNMP, configuration version control, netperf/iperf and sFlow/NetFlow, are all insufficient to provide insight into the behavior of SDN-enabled networks [9]. At this point, prior works have proposed considerable trajectory detection solutions, which can be categorized into the following two folds:

Firstly, some of the existing solutions [10]–[12] focus on the control plane. For example, SDN traceroute [10] collects the switch's packet-in message at each hop and reconstructs the trajectory at the controller. VeriFlow [11] analyzes the configurations pushed to network devices to infer forwarding paths. NetSight [12] forces all switches to send postcards when traffic passes through them and introduces a history plane for trajectory reconstruction. We argue that solutions of this kind, based on after-the-fact analysis at a non-data plane, require excessive out-of-band data collection.

Secondly, other researchers use the in-band technology at the data plane [13]–[15]. For example, Jeyakumar *et al.* [13] introduces a tiny packet program (TPP) interface to enable end hosts to collect the packet histories of each switch. But it is not compatible with standard SDN frameworks, e.g., OpenFlow [16]. There also exist OpenFlow-compatible techniques. For instance, CherryPick [17] attempts to identify

Manuscript received July 3, 2018; revised January 28, 2019 and June 19, 2019; accepted August 14, 2019. Date of publication August 21, 2019; date of current version December 10, 2019. This work was supported by the State Key Program of National Natural Science of China (Grant No. 61432002), NSFC Grant Nos. 61772112, U1701263, 61672379, and 61751203, and the Science Innovation Foundation of Dalian under Grant 2019J12GX037. The associate editor coordinating the review of this article and approving it for publication was J. Schönwälder. *(Corresponding author: Heng Qi.)*

J. Wang, H. Qi, Y. He, and K. Li are with the School of Computer Science and Technology, Dalian University of Technology, Dalian 116024, China (e-mail: hengqi@dlut.edu.cn).

W. Li is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong.

X. Zhou is with the Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University, Tianjin 300350, China.

Digital Object Identifier 10.1109/TNSM.2019.2936598

each trajectory by sampling only two links, which, however, is insufficient in the case of many detours (more than 10 hops). As for other OpenFlow-compatible techniques like PathletTracer [14] and PathQuery [15], they imprint each packet with compressed path information and can still fail in large-scale datacenter with large number of paths, because: 1) they need to reserve substantial entries in the switches for trajectory detection, while these entries are limited and expensive in TCAM-based switches [18]; 2) they may require excessive packet header space (far more than 24 bits), especially when the packets traverse non-shortest paths.

Bearing the above points in mind, we ask one question: *what happens if we try to reformulate the trajectory detection problem by sampling paths one link at a time rather then recording entire paths?* In fact, a meaningful trajectory detection method should focus on the detection of trajectories corresponding to large flows because of their higher impact on performance [19], [20]. Note that a flow can be regarded as a sequence of packets with the same 5-tuple. If, upon receipt of a packet, each switch chooses to sample an adjacent link for storage in the packet field with some probability p, then every packet received by the end host will arrive carrying the information of one of the links it traversed. After sufficient packets have been sent, the end host will have received at least one sample for every link on the flow trajectory.

FlowTracer, an in-band trajectory detection technique, is designed based on the above considerations, with the goal of reducing the overhead in terms of both switch entries and packet header space. In summary, we make the following contributions:

1) We introduce a method of probabilistic packet tagging that is performed in OpenFlow-enabled switches with very few switch entries and limited packet header space by means of double VLAN tags.
2) We explore datacenter topologies and propose a method of path reconstruction that is performed at end hosts and achieves rapid convergence. A theoretical analysis of its performance is presented.
3) We implement FlowTracer and evaluate it on a 48-ary fat-tree topology. The results show that FlowTracer can detect trajectories quickly while placing far smaller demands on both switch entries and packet header space than state-of-the-art techniques.

The remainder of this paper is organized as follows. Section II presents the motivation for and challenges facing the problem of trajectory detection in large-scale SDN-enabled datacenter networks. In Section III, we present our method of probabilistic packet tagging, which is performed in OpenFlow-enabled switches. In Section IV, we discuss our rapidly convergent method of path reconstruction, which is performed at end hosts. We discuss the current limitations of FlowTracer and relevant assumptions in Section V. In Section VI, we evaluate and analyze the performance of FlowTracer. Section VII summarizes related work, and we conclude the paper in Section VIII.

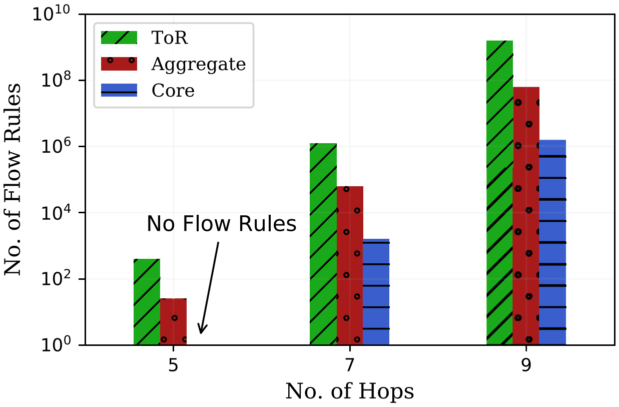


Fig. 1. The switch entry overhead of PathletTracer for different cases of packet detours in a 48-ary fat-tree topology.

II. Motivation and Challenges

Detecting trajectories by recording the entire path in each packet offers both robustness and extremely rapid convergence; however, it has several serious limitations. Principal among these is the infeasible high overhead incurred by appending the path information to packets in flight. Moreover, since the length of a path is not known a priori, it is impossible to ensure that there will be sufficient unused space in a packet for its complete trajectory.

For techniques of this kind, all the path information is compressed and stored in a codebook, in which each path has a unique identifier. Additionally, each packet needs to record the path identifier in its header space. On the one hand, the number of paths determines the size of the identifier and can further be used to determine the bit length of the packet header. On the other hand, the number of paths also determines the number of switch entries used for matching operations. More importantly, packet detours complicate the tracing task due to the vast increase in the number of possible paths in a datacenter network as the path length increases. Since the overheads in terms of both switch entries and packet header space are tightly coupled with the network scale, existing solutions that may work well in smaller networks may not necessarily scale to the case of large-scale SDN-enabled datacenter networks.

To more intuitively illustrate this point, we implemented the state-of-the-art technique PathletTracer [14] in a 48-ary fat-tree topology (the detailed topology of the datacenter network is introduced in Section VI-A). Since the technique depends on the layer in which the switch resides, we plot the number of switch flow rules for PathletTracer in each layer separately. Note that the design of PathletTracer is independent of the routing mechanism used in the network. Thus, it can be seen that the switch entry overhead is caused by trajectory detection rather than flow forwarding. As illustrated in Fig. 1, the number of required flow rules in ToR switches for tracing 7-hop paths considerably outstrips the size of TCAM. As the number of hops per path increases, the demand increases nonlinearly. The flow rules that cannot be stored in TCAM must instead be stored in SRAM or even DRAM, at the cost of supporting

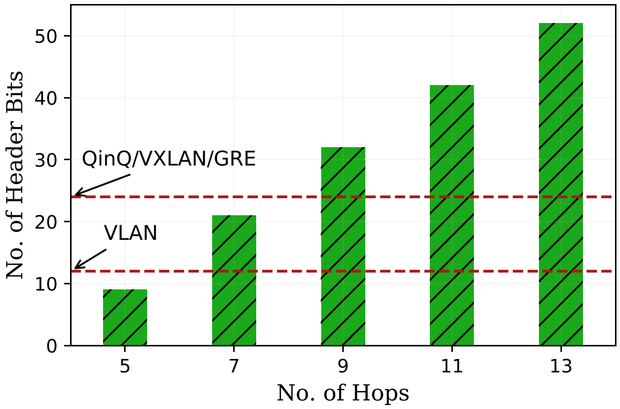


Fig. 2. The packet header space overhead of PathletTracer for different cases of packet detours in a 48-ary fat-tree topology.

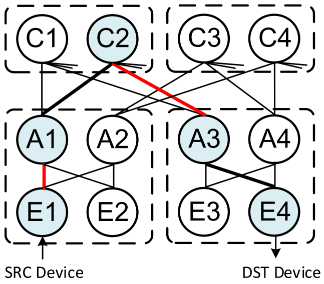


Fig. 3. Probabilistic packet tagging for link sampling. All packets of the flow traverse the path SRC-E1-A1-C2-A3-E4-DST. Each link is sampled with a fixed probability p, and each packet records at most one link.

less efficient lookups. Scaling up the network (e.g., from a 48-ary fat-tree topology to a 64-ary fat-tree topology) imposes a similar limitation in terms of flow rules since this also causes a vast increase in the number of paths. Therefore, the question of how to detect trajectories in a large-scale datacenter network while placing far fewer demands on switch entries is a major challenge.

In addition to the switch entry demand, the demand on packet header space presents another challenge. As illustrated in Fig. 2, the packet header space required for tracing a 9-hop path is incompatible with current mainstream technology, e.g., Q-in-Q, VXLAN and GRE. As the network scale and the number of hops increase, the problem of space exhaustion will become more serious since the size of the path identifiers will become much larger.

The main issue faced by PathletTracer, leading to an excessive number of flow rules and high header space consumption, is that the number of equivalent paths is considerable and will significantly increase as the network scale increases or if the packets do not traverse shortest paths (e.g., the non-5-hop cases in these figures). To uniquely identify a vast number of paths, long path identifiers are unavoidably required. To tag these identifiers, many flow rules are also unavoidably required. Thus, existing solutions such as PathletTracer are not practical due to their significant overheads in terms of switch entries and header space.

To reduce both the switch overhead and the per-packet header space requirement, we can sample paths one link at a time instead of recording entire paths. A single static "link" field can be reserved in the packet header; this field need only be large enough to hold the information for a single link (i.e., its switch pair and its distance to the receiver). Upon receiving a packet, each switch chooses to sample an adjacent link for storage in the "link" field with some probability p. After sufficient packets have been sent, the receiver (end host) will have received at least one sample of every link on the path. With this approach, the situation can be very different. On the one hand, since each packet records only the information of only one link instead of the complete path, the overhead in terms of the packet header space will remain fixed as both the network scale and the number of hops increase. On the other hand, since each link is sampled at a certain probability, only a few flow rules are needed on each switch for probability operations, thus eliminating the large number of flow rules used for matching operations. The overhead in terms of the number of switch entries will also remain fixed as the network scale and the number of hops increase. The details of the proposed method of probabilistic packet tagging (i.e., link sampling) are introduced in Section III, and the details of the proposed method of path reconstruction (i.e., leveraging link samples to form complete trajectories) are introduced in Section IV.

III. Probabilistic Packet Tagging

In this section, we describe the link sampling process in OpenFlow-enabled switches. To this end, we record the information on one link in the form of a three-tuple <*head_switch_id*, *tail_switch_id*, *distance*>. *head_switch_id* and *tail_switch_id* indicate the head switch and tail switch, respectively, of the link; thus, <*head_switch_id*, *tail_switch_id*> is the corresponding switch pair. *distance* denotes the distance between the link and its receiver. The topology of a datacenter network is typically known and permanent. Thus, *head_switch_id* and *tail_switch_id* can be uniquely determined in accordance with the datapath identifier (*DPID*) values of OpenFlow-enabled switches.

Consider the example shown in Fig. 3. The receiver (end host of *E4*) uses the link information in the marked packets to trace the path back to the sender (end host of *E1*). The path is the ordered list of links between *E1* and *E4*: <*E1*, *A1*, *4*>, <*A1*, *C2*, *3*>, <*C2*, *A3*, *2*>, <*A3*, *E4*, *1*> and <*E4*, *null*, *0*>. We constrain the sampling probability p to be identical for each link. Since the links are arranged serially, the probability that a packet will be marked with a link and then left unchanged by all downstream switches is a strictly decreasing function of the distance to the receiver. This situation can be formulated as a joint probability model. For clarity of explanation, we define the event M_0 as the case in which the receiver receives a packet marked with the link that is 0 hops away (i.e., *distance* is 0). The probability of M_0 is

$$P(M_0) = p, \tag{1}$$

which can be easily proven. Based on this probability, we can derive

$$P(M_1) = P(T_1 \cap ! T_0) = P(T_1)(1 - P(T_0)) = p(1-p), \tag{2}$$

where the events T_1 and T_0 are defined as the sampling of the link at a distance of 1 and the sampling of the link at a distance of 0, respectively. The event M_1 is defined as the case in which the receiver receives a packet marked with the link that is 1 hop away (i.e., *distance* is 1). Note that a sample can be replaced by any or all downstream switches. Therefore, the events T_1 and T_0 are mutually independent, while the events M_1 and M_0 are mutually exclusive. Similarly, it can be inferred that

$$\begin{aligned} P(M_{n-1}) &= P(T_{n-1} \cap ! T_{n-2} \cap \cdots \cap ! T_1 \cap ! T_0) \\ &= P(T_{n-1})(1 - P(T_{n-2})) \cdots (1 - P(T_1)) \\ &\quad \times (1 - P(T_0)) \\ &= p(1-p)^{n-1}, \end{aligned} \tag{3}$$

where n is the number of switches on the path. The event M_{n-1} is defined as the case in which the receiver receives a packet marked with the link that is $n - 1$ hops away (i.e., *distance* is $n - 1$). To further clarify the above mappings, let us return to the example depicted in Fig. 3, where n is 5. If the link sampling probability p is 0.2, then the probability of receiving a packet marked with the link <*E1*, *A1*, *4*> is 0.08192, the corresponding probability for the link <*A1*, *C2*, *3*> is 0.1024, the probability for the link <*C2*, *A3*, *2*> is 0.128, the probability for the link <*A3*, *E4*, *1*> is 0.16, and the probability for the link <*E4*, *DST*, *0*> is 0.2. The convergence time of link sampling is equivalent to the number of packets that the receiver must observe to reconstruct the path. For the case of $n = 5$ and $p = 0.2$, the expected number of packets needed to reconstruct the path is bounded by 19. The theoretical analysis is presented in Section IV-A. We also present a method of reducing the convergence time. This method is discussed in Section IV-B.

A. Probabilistic Packet Tagging Algorithm

The basic idea of probabilistic packet tagging is that switches will probabilistically write three-tuples of the form <*head_switch_id*, *tail_switch_id*, *distance*> into packets during forwarding. For this purpose, we need to reserve two static fields, *head_switch_id* and *tail_switch_id*, in each packet to represent the switch pair at each end of a sampled link as well as an additional small field to represent the distance of the sampled link from the receiver.

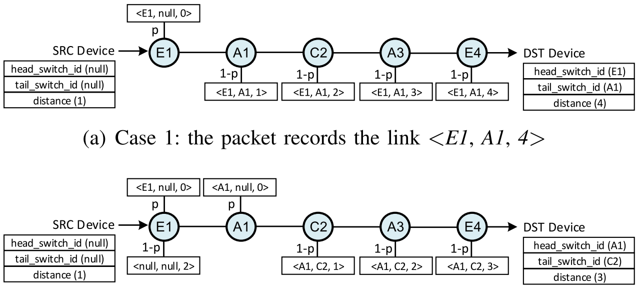
To implement the link sampling procedure on switches, we design three types of switch operations, as follows:

Op1: reset *head_switch_id* to the current switch, and reset *distance* to 0.

Op2: set *tail_switch_id* to the current switch, and increase *distance* by 1.

Op3: increase *distance* by 1.

Upon receiving a packet, a switch generates a random number in the range [0, 1]. When this random number is less than the predefined probability p, the switch chooses to mark the packet and will execute ***Op1***. Otherwise, if the value of the *distance* field is already zero, this indicates that the packet was marked by the previous switch. In this case, the switch will execute ***Op2***. By writing its own *DPID* into the *tail_switch_id* field, it represents that the link between itself and the previous switch has been sampled. Finally, if the switch chooses not to mark the packet (the random number is in the range (p, 1]), then it will execute ***Op3***.



(a) Case 1: the packet records the link <*E1*, *A1*, *4*>

(b) Case 2: the packet records the link <*A1*, *C2*, *3*>

Fig. 4. Two illustrative cases of the link sampling mechanism.

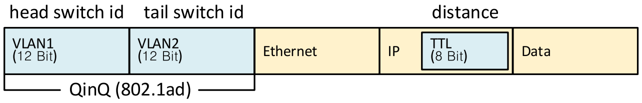


Fig. 5. Packet header fields exploited by FlowTracer.

Fig. 4 shows two illustrative cases of the link sampling mechanism. For the case shown in Fig. 4(a), switch *E1* executes ***Op1***, switch *A1* executes ***Op2***, and the remaining (downstream) switches execute ***Op3***. The receiver thus receives a packet marked with the link <*E1*, *A1*, *4*>. The receiver therefore knows that the packets of this flow traverse *E1-A1* and that this link is 4 hops away. For the case shown in Fig. 4(b), switches *E1* and *A1* both execute ***Op1***, switch *C2* executes ***Op2***, and the remaining (downstream) switches execute ***Op3***. The receiver therefore knows that the packets of the flow traverse *A1-C2* and that this link is 3 hops away. In practice, these two cases correspond to the experiences of two packets on the same flow path. It can be seen that the link information is sampled only between participating switches. When a packet arrives at the receiver, its *distance* field represents the number of hops traversed since the link was sampled. The pseudocode that is executed on the switches is shown in Algorithm 1.

B. Packet Header Space Encapsulation

FlowTracer employs the technology of double VLAN tags (Q-in-Q, 802.1ad, 0x88a8) to store the switches' *DPID* values. When a packet is sent from the sender's network protocol stack, we assume that it is encapsulated with a standard layer 3 Ethernet header, where the initial TTL value is set to 63. When the packet traverses a switch in the flow path, it will be encapsulated with an outer skin of double VLAN tags. Each VLAN tag, of 12 bits in length, is sufficient to store one switch's *DPID*.

As illustrated in Fig. 5, we let one of the VLAN tags store *head_switch_id* and the other store *tail_switch_id*. The $64 - ttl$ field is used to track the *distance* value. A 12-bit VLAN tag can theoretically be used to identify 4,096 unique switches, and the decrement operation can be executed 63 times on the TTL field. Therefore, FlowTracer offers sufficient support for

Algorithm 1: Probabilistic Packet Tagging

```
Input: packets of flow $W$;
       link sampling probability $p$;
       switch's unique identifier $id$;
Output: packets with updated header space;
foreach packet $w$ in $W$ do
    let $x$ be a random number from [0,1];
    if $x < p$ then
        write $id$ into $w.head\_switch\_id$;
        write $0$ into $w.distance$;
    else
        if $w.distance = 0$ then
            write $id$ into $w.tail\_switch\_id$;
        increment $w.distance$ by 1;
```

an 48-ary fat-tree topology and allows the packets of a flow to traverse non-shortest paths of at most 64 hops, with many detours.

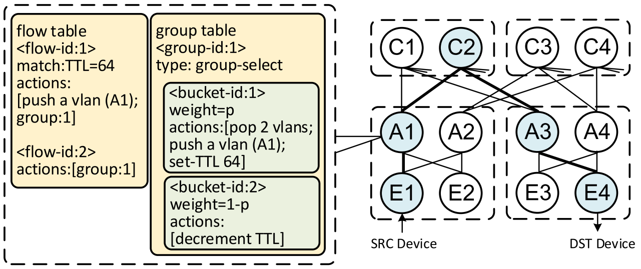
C. Standard SDN Interface Adaptation

For compatibility with the standard SDN interface OpenFlow [21], we implement Algorithm 1 with off-the-shelf SDN switches. For a commodity SDN switch, e.g., Pica8 P-3297, the ASIC typically offers the line rate for double VLAN tags, i.e., Q-in-Q. FlowTracer is implemented with only three table entries on the OpenFlow platform.

As illustrated in Fig. 6, we employ the group bucket *<bucket-id:1>* to execute ***Op1***. When this bucket is selected with probability *p*, the switch pushes its own *DPID* into the first VLAN field and writes a value of 64 into the TTL field. If the value of the TTL field is already 64, we employ the flow entry *<flow-id:1>* to execute ***Op2***. In this case, the value of the first VLAN field was set by the previous switch. The current switch will push its own *DPID* into the second VLAN field and decrement the TTL field by 1. Finally, we employ the group bucket *<bucket-id:2>* to execute ***Op3***. When this bucket is selected with probability *1-p*, the switch simply decrements the TTL field by 1. Through this mechanism, link sampling can be incrementally performed on OpenFlow switches. Certainly, this mechanism allows considerable TCAM capacity to be saved for the switches in the datacenter network. The consumption of additional packet header space is also limited to double VLAN tags.

While each marked packet carries only one link sample from the path it has traversed, by combining a modest number of such packets, a receiver can reconstruct the entire path. Since the receiver knows at which switch the sender resides, the receiver can use these marked packets to reconstruct the path back to the sender.

A general method of path reconstruction can be easily devised: the receiver continues to track marked packets until all links are identified. Then, the trajectory can be detected by sequentially connecting the links in accordance with their *distance* values. However, this general method is prone to slow convergence, especially when the packets traverse a non-shortest path with many detours. To address this shortcoming, we explore the relevant network topologies and propose a rapidly convergent method of path reconstruction based on this general method. The general method and the advanced method are both introduced in Section IV.



```
flow table
<flow-id:1>
match:TTL=64
actions:
[push a vlan (A1);
group:1]

<flow-id:2>
actions:[group:1]
```

```
group table
<group-id:1>
type: group-select
  <bucket-id:1>
  weight=p
  actions:[pop 2 vlans;
  push a vlan (A1);
  set-TTL 64]
  <bucket-id:2>
  weight=1-p
  actions:
  [decrement TTL]
```

Fig. 6. The OpenFlow implementation of probabilistic packet tagging. Each switch is installed with two flow table entries and one group table entry.

IV. Rapidly Convergent Path Reconstruction

In this section, we show how the end host reconstructs the paths based on the received marked packets. For notational purposes, we assume that the receiver has a map of its upstream switches, denoted by G. G is a directed acyclic graph (DAG) with the receiver as the root. This assumption is reasonable and practical. It is easy to obtain such a map of upstream switches for a receiver. In addition, the receiver typically knows at which switch the sender resides. This information rarely changes in a datacenter network. The end host maintains a map <[*Flow*[*Packet*]], G> for the flows in G. Before presenting the rapidly convergent method of path reconstruction, we will first discuss how the general method works.

A. General Path Reconstruction

Intuitively, for path reconstruction, the receiver will use the upstream switch map G as a roadmap and perform a breadth-first search from the root. Let the set of link fields marked with a *distance* d be denoted by $[Flow[Packet]]_d$ (not including duplicates). At *distance* 1, the receiver enumerates all switches that are one hop away from itself in G, checks which of these switches have *DPID* values that match with the link fields in $[Flow[Packet]]_d$, and stores the matching *DPID* values in set S_1. Therefore, S_1 is the set of switches that are one hop away from the receiver in the reconstructed path graph. Similarly, S_d denotes the set of switches at *distance* d from the receiver in the reconstructed path graph. Once the receiver finds the *DPID* values at which the senders reside, with the maximum *distance*, the complete path graph can be determined. The pseudocode that is executed at the end host is shown in Algorithm 2.

Performance Analysis: Since FlowTracer is based on a probabilistic method, the expected number of packets required for tracing a path should be analyzed.

As described in Section III, if we constrain the sampling probability p to be identical for each link, then the probability of receiving a packet marked with a sampled link that is d hops away is $p(1-p)^d$. This function is monotonic in the distance from the receiver. The events of receiving packets marked with different links are mutually exclusive and independent. Now, we will formulate the general problem of path reconstruction as a special case of coupon collection with different collection probabilities. The coupon collection problem is one of the most famous problems in probability theory [22]. The aim of the coupon collection problem is to answer the following question: Suppose that there are n types of coupons, that each type of coupon has the same probability of acquisition, and that the supply of coupons is unlimited. If we have acquired each type of coupon at least once, how many coupons have been collected already? In contrast to the classical version, the special case of interest in FlowTracer is as follows: Supposing that there are n types of links sampled in the packets, a link sample of the j^{th} type will be obtained with probability $p(1-p)^{n-j}$.

Let W be the number of packets required for tracing an n-hop path. The expectation value of W is expressed as follows:

$$\begin{aligned} E(W) &= \sum_k kP(W=k) \\ &= \sum_k k(P(W \geq k) - P(W \geq k+1)) \\ &= \sum_{k=1} kP(W \geq k) - \sum_{k=0} kP(W \geq k+1) \\ &= \sum_{k=1} kP(W \geq k) \\ &\quad - \sum_{k=1} (k-1)P(W \geq k) = \sum_{k=1} P(W \geq k). \end{aligned} \tag{4}$$

Since the probability of $W \geq k$ can be expressed as

$$\begin{aligned} P(W \geq k) = & \sum_{J=\{j_0,\ldots,j_i\}, J\subseteq\{0,\ldots,n-1\}, J\neq\varnothing} (-1)^{|J|+1} \\ & \times \left(1 - \sum_{j\in J} p_j\right)^{k-1}, \; k \geq 2, P(W \geq 1) = 1, \end{aligned} \tag{5}$$

if we substitute equation (5) into (4), the expectation value can be expressed as

$$\begin{aligned} E(W) &= \sum_{k=1} \sum_{J=\{j_0,\ldots,j_i\}, J\subseteq\{0,\ldots,n-1\}, J\neq\varnothing} (-1)^{|J|+1} \\ &\quad \times \left(1 - \sum_{j\in J} p_j\right)^{k-1} \\ &= 1 + \sum_{J=\{j_0,\ldots,j_i\}, J\subseteq\{0,\ldots,n-1\}, J\neq\varnothing} (-1)^{|J|+1} \\ &\quad \times \frac{1 - \sum_{j\in J} p_j}{\sum_{j\in J} p_j}. \end{aligned} \tag{6}$$

Because the probability of receiving a sample is smaller the farther away the corresponding link is from the receiver, the time to convergence is dominated by the time required to receive a sample from the farthest switch. For convenience of analysis, we consider the following special case to simplify equation (6): Suppose that the probability of obtaining the information for each link is identical and equal to $p^* = p(1-p)^{n-1}$. Since

$$p^* = p(1-p)^{n-1} \leq p_j = p(1-p)^j, \forall j \in [0, n-1], \tag{7}$$

the expected number of packets required for tracing an n-hop path satisfies

$$E(W) < E^*(W) = \sum_{i=0}^{n-1} \frac{1}{p^*(n-i)} = \frac{1}{p^*} \sum_{i=0}^{n-1} \frac{1}{n-i} = \frac{1}{p^*} H_n, \tag{8}$$

where H_n is the n-th harmonic number. From the asymptotic behavior of the harmonic numbers, the following can be obtained:

$$\begin{aligned} E(W) < E^*(W) &= \frac{1}{p^*} H_n \\ &= \frac{1}{np^*}\left(n \log n + \gamma n + \frac{1}{2} + O\left(\frac{1}{n}\right)\right), \end{aligned} \tag{9}$$

where $\gamma \approx 0.5772$ is the Euler-Mascheroni constant. Now, we can use the Markov inequality to bound the desired probability:

$$P\left(W \geq \frac{c}{p^*} H_n\right) < P\left(W^* \geq \frac{c}{p^*} H_n\right) \leq \frac{1}{c}. \tag{10}$$

We can also obtain the variance of W^*, which satisfies

$$\begin{aligned} Var(W^*) &= \sum_{i=1}^{n} \frac{1 - ip^*}{i^2 p^{*2}} < \sum_{i=1}^{n} \frac{1}{i^2 p^{*2}} = \frac{1}{p^{*2}} \sum_{i=1}^{n} \frac{1}{i^2} \\ &< \frac{\pi^2}{6p^{*2}} = \frac{\pi^2}{6p^2(1-p)^{2n-2}}. \end{aligned} \tag{11}$$

Then, we can use the Chebyshev inequality to bound the other desired probability:

$$\begin{aligned} P\left(\left|W - \frac{1}{p^*} H_n\right| \geq \frac{c}{p^*}\right) &< P\left(\left|W^* - \frac{1}{p^*} H_n\right| \geq \frac{c}{p^*}\right) \\ &\leq \frac{\pi^2}{6c^2}. \end{aligned} \tag{12}$$

Finally, we can attempt to determine a minimal upper bound on the expected number of required packets. According to equation (9), the expectation value can be expressed as

$$\begin{aligned} E(W) &< \frac{1}{np^*}\left(n \log n + \gamma n + \frac{1}{2} + O\left(\frac{1}{n}\right)\right) \\ &= \frac{1}{np(1-p)^{n-1}}\left(n \log n + \gamma n + \frac{1}{2} + O\left(\frac{1}{n}\right)\right) \\ &\approx \frac{1}{np(1-p)^{n-1}} n(\ln n + O(1)) \approx \frac{n \ln n}{np(1-p)^{n-1}} \\ &= \frac{\ln n}{p(1-p)^{n-1}}. \end{aligned} \tag{13}$$

Taking the second derivative of the denominator, we find

$$\frac{d^2}{dp}\left[p(1-p)^{n-1}\right] > 0. \tag{14}$$

Since the denominator is convex and the numerator is constant, the upper bound on the expectation value expressed

Algorithm 2: General Path Reconstruction

```
Input: map <[Flow[Packet]], G>;
       switches at which the senders reside R;
       end-host receiver v;
Output: path graph of flow packets in G;
let $G$ be a DAG with its root at v;
let $S_d$ be empty for $1 \leq d \leq$ max(distance);
foreach d in [1, max(distance)] do
    foreach DPID of switch $s \in [Flow[Packet]]_d$ do
        insert s into $S_d$;
    if DPID of switch $s \in$ R then
        $\max(distance)_{s-v} = d$;
extract the path graph from $(S_1, S_d, \ldots, R)$;
```

Algorithm 3: Accelerated Path Reconstruction

```
Input: map <[Flow[Packet]], G>;
       switches at which the senders reside R;
       end-host receiver v;
Output: path graph of flow packets in G;
let $G$ be a DAG with its root at v;
let $S_d$ be empty for $1 \leq d \leq$ max(distance);
foreach d in [1, max(distance)] do
    switch link type do
        case TOR-AGG:
        case CORE-AGG:
            foreach DPID of switch $s \in [Flow[Packet]]_d$ do
                insert s into $S_d$;
                search for the shortest path to the next-hop switch
                $s'$;
                insert $s'$ into $S_{d-1}$;
        case Others:
    if DPID of switch $s \in$ R then
        $\max(distance)_{s-v} = d$;
extract the path graph from $(S_1, S_d, \ldots, R)$;
```

in (13) should have an extremum within $p \in [0, 1]$. Then, let the first derivative of the denominator be 0:

$$\frac{d}{dp}\left[p(1-p)^{n-1}\right] = 0. \tag{15}$$

When $p = \frac{1}{n}$, the denominator is maximal for any n. The expectation value of W has the minimal upper bound under this condition.

Theorem I: For a path of n hops in a datacenter network, the link sampling probability p of FlowTracer should be set to $\frac{1}{n}$ such that the upper bound on the expected number of required packets will have the minimal value. The variance of the expectation value is less than $\pi^2/6p^2(1-p)^{2n-2}$. The expectation value will be greater than $cH_n/1(1-p)$ with a probability of less than $1/c$. The expectation value will lie outside $[c/p(1-p) - H_n/p(1-p), c/p(1-p) + H_n/p(1-p)]$ with a probability of less than $\pi^2/6c^2$.

In a fat-tree topology, the equal-cost shortest paths between any two end hosts can be up to 5 hops in length, e.g., the path E1-A1-C2-A3-E4 in Fig. 3. Theoretically, tracing a path with $n = 5$ is expected to require at most 52 packets if the probability is set to $\frac{1}{2}$, at most 20 packets if the probability is set to $\frac{1}{5}$, and at most 25 packets if the probability is set to $\frac{1}{10}$. By contrast, for a path with detours such that $n = 7$, tracing this path is expected to require at most 38 packets if the probability is set to $\frac{1}{5}$, at most 35 packets if the probability is set to $\frac{1}{7}$, and at most 37 packets if the probability is set to $\frac{1}{10}$. Although FlowTracer cannot trace a trajectory from a single packet, with the typical high-speed bandwidth of datacenters [23], [24], the required number of packets can be delivered to the destination almost instantaneously.

In practice, the determination of the sampling probability (also called the bucket weight) typically relies on the real topology and the flow distribution. For an operator, it is important to ensure that most flows will be traced with good performance (i.e., few required packets). Thus, the first thing to consider is the hop count of the flows that occupy the majority of the network. For some symmetrical topologies such as fat trees, under the assumption that all leaf servers are performing TCP transmission with each other and the rates of transmission are identical, 5-hop flows must account for the vast majority for two reasons: (1) detours do not, in fact, occur very often, and (2) the rate of interpod transmission is much greater than that of intrapod transmission. Given these considerations, it is desirable to set the probability on the basis of 5-hop paths.

Although detours rarely occur, when they do, the general method of path construction may take a long time to converge. Due to detours, packets will traverse paths containing many more links. In turn, the number of packets that the receiver must observe to reconstruct such a path will be much greater. In fact, the expected number of required packets will increase nonlinearly. For instance, for a 5-hop path ($n = 5$), it is expected that at most 20 packets will be required for path reconstruction. However, when the hop count increases to 7 (with one detour), the number of packets expected to be required increases to 35 (by almost a factor of two).

To address this situation, we explore the topological structure of datacenter networks and present a rapidly convergent version of the path reconstruction method. We find that a straightforward solution to the problem is to reduce the number of links required to reconstruct the path.

B. Datacenter Network Topology Exploration

In the widely used datacenter k-ary fat-tree topology, the number of core layer switches is $\frac{k^2}{4}$ and the total number of aggregation layer switches and access layer switches is $\frac{k^2}{2}$. Overall, the total number of switches is $\frac{5k^2}{4}$. By means of packet header encapsulation with double VLAN tags, as described in Section III-B, 4,096 switches can be uniquely identified; thus, this approach is sufficiently compatible with a 48-ary fat-tree network topology. We consider the detour model in such a topology.

In a datacenter network, although not common, some packet detours will inevitably occur for several reasons. On the one hand, misconfiguration failures at switches or links may force rerouting to alternative non-shortest paths [25], [26]. Consider the topology in Fig. 3, where the packets are routed on the shortest path E1-A1-C2-A3-E4. If the link C2-A3 fails, the route must be changed to a non-shortest path. On the other hand, recently proposed techniques may reroute packets to idle non-shortest paths to avoid link congestion [27]–[29]. Therefore, the ability to detect flow trajectories in the case of packet detours is very important. For some topologies, such as fat trees, that have symmetrical structures, we find that the flow trajectories can be uniquely determined without requiring all links to be known to the receiver.

Theorem II: In a datacenter fat-tree topology, only the links leading from the access layer to the aggregation layer and from the core layer to the aggregation layer are required to be known to the receiver. By searching for the shortest paths between these necessary links, all other links on the path can then be inferred.

Proof: First, let the flow trajectory be $< s_1, s_2, \ldots, s_i, s_{i+1}, \ldots, s_{N-1}, s_N >$. The DPIDs of switches s_1 and s_N are known, and they should be the first and last switches located in the access layer. Let the information on the i-th link known to the receiver be $L_i = (s_i, s_{i+1}, distance(s_i, s_{i+1}))$. Because $distance(s_1, s_2) = \max i_{1 \to N}(distance(s_i, s_{i+1})) = N - 1$ and the DPID of S_1 is known, the receiver can determine the value of N via L_1 and only via L_1. In other words, L_1 is essential for the receiver to identify the complete path. On the other hand, to uniquely determine the entire trajectory, it is simply necessary to ensure that all subtrajectories are uniquely determined. Given the symmetrical nature of the fat-tree topology, any subpath is composed of uplinks and downlinks, and the entire subpath is uniquely determined as long as the uplink or downlink is uniquely determined. Once the receiver has received confirmation of L_1 and L_3, there is no need for it to be informed of L_2, and so on. The number of intrapod links is $2(j + 1)$, $j \geq 0$, and the number of interpod links is $2k$, $k \geq 0$. Therefore, within a pod, all uplinks are essential, and between pods, all downlinks are essential. When both the intrapod and interpod subtrajectories can be uniquely determined, the entire trajectory will be uniquely determined. Thus, Theorem II is proven. ■

We identify five possible detour cases under the assumption of packet-level forwarding strategy consistency,[1] as follows: (1) intrapod detour, (2) source pod detour, (3) destination pod detour, (4) core layer detour, and (5) mixed detour. We consider that the number of detours should be greater than or equal to 0. In the case that the number of detours is 0, the trajectory lies on the shortest path.

The typical path with no detour is shown in Fig. 3. According to Theorem II, only the link E1-A1 and the link C2-A3 are required to be known to the receiver. As long as the receiver receives a packet tagged with the link at distance 4, E1-A1, it can be sure that the packets of the flow traversed a 5-hop path. Then, if the receiver is also informed of the link at distance 2, C2-A3, the complete path E1-A1-C2-A3-E4 can be uniquely determined. Since the remaining links A1-C2 and A3-E4 can then be inferred, the receiver is required to receive information on only two links, i.e., E1-A1 and C2-A3, instead of all 4 links.

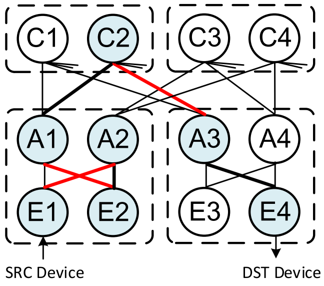


Fig. 7. A 7-hop path with a source pod detour, SRC-E1-A2-E2-A1-C2-A3-E4-DST ($n = 7$). The detour changes the subpath E1-A1 to E1-A2-E2-A1.

Consider the case in which detours occur during intrapod transmission. Intrapod detours conform to the pattern SRC-ToR-(Agg-ToR)^i-Agg-ToR-DST, where i is the number of detours within the pod. According to Theorem II, tracing a 5-hop path with an intrapod detour requires only one link at distance 4 and one link at distance 2, i.e., two links leading from the access layer to the aggregation layer, instead of all 4 links. In contrast to such intrapod detours, the other four types of detours occur during interpod transmission.

Source pod detours are detours that occur only within the source pod. Source pod detours conform to the pattern SRC-ToR-(Agg-ToR)^i-Agg-Core-Agg-ToR-DST, where i is the number of detours within the source pod. Fig. 7 depicts a 7-hop path with a source pod detour. According to Theorem II, tracking this path with a source pod detour requires the link at distance 6, E1-A2; the link at distance 4, E2-A1; and the link at distance 2, C2-A3. The Agg-Core link can be uniquely determined as long as the link C2-A3 is known to the receiver. In addition, the Agg-ToR link can be uniquely determined as long as the links E1-A2 and E2-A1 are known. Therefore, the receiver is required to receive information on only these three links, i.e., E1-A2, E2-A1 and C2-A3, instead of all 6 links.

Similarly, destination pod detours are detours that occur only within the destination pod. Destination pod detours conform to the pattern SRC-ToR-Agg-Core-Agg-(ToR-Agg)^j-ToR-DST, where j is the number of detours within the destination pod. Fig. 8 shows a 7-hop path with a destination pod detour. According to Theorem II, tracing this path with a destination pod detour requires the link at distance 6, E1-A1; the link at distance 4, C2-A3; and the link at distance 2, E3-A4. Therefore, only these three links are required to be known to the receiver, instead of all 6 links, for the complete path to be uniquely determined.

Core layer detours are detours that occur only within the core layer. Core layer detours conform to the pattern SRC-ToR-Agg-Core-(Agg-Core)^k-Agg-ToR-DST, where k is the number of detours within the core layer. A 7-hop path with a core layer detour is shown in Fig. 9. According to Theorem II, tracing this path with a core layer detour requires the link at distance 6, E1-A1; the link at distance 4, C1-A5; and the link at distance 2, C2-A3. Therefore, only these three links are required to be known to the receiver instead of all 6 links.

Finally, mixed detours are the most complex detours, simultaneously occurring within the source pod, the destination

[1] Packet-level forwarding strategy consistency is defined as follows: a switch uses the same strategy to forward each packet of a given flow. This situation is common in datacenter networks because the forwarding rules installed in a switch are typically flow entries.

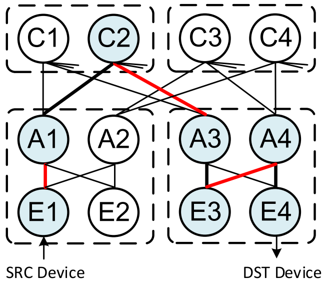


Fig. 8. A 7-hop path with a destination pod detour, SRC-E1-A1-C2-A3-E3-A4-E4-DST ($n = 7$). The detour changes the subpath A3-E4 to A3-E3-A4-E4.

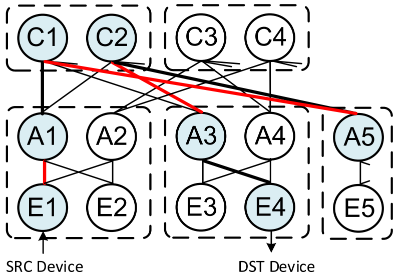


Fig. 9. A 7-hop path with a core layer detour, SRC-E1-A1-C1-A5-C2-A3-E4-DST ($n = 7$). The detour changes the subpath C1-A3 to C1-A5-C2-A3.

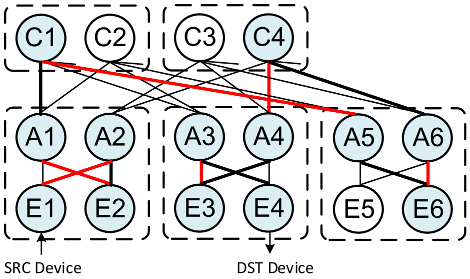


Fig. 10. A 13-hop path with mixed detours, SRC-E1-A2-E2-A1-C1-A5-E6-A6-C4-A4-E3-A3-E4-DST ($n = 13$). The mixed detours simultaneously occur in the source pod, the destination pod, the core layer and a non-source/destination pod.

pod, the core layer and/or one or more non-source/destination pods. In fact, all possible detours in a fat-tree topology can be abstracted as mixed detours. Mixed detours conform to the pattern SRC-ToR-(Agg-ToR)^i-Agg-Core-(Agg-(ToR-Agg)^j-Core)^k-Agg-(ToR-Agg)^l-ToR-DST, where i is the number of detours within the source pod, j is the number of detours within a non-source/destination pod, k is the number of detours within the core layer, and l is the number of detours within the destination pod. A 13-hop path with mixed detours is shown in Fig. 10. According to Theorem II, tracing this path with mixed detours requires the link at distance 12, E1-A2; the link at distance 10, E2-A1; the link at distance 8, C1-A5; the link at distance 6, E6-A6; the link at distance 4, C4-A4; and the link at distance 2, E3-A3. By searching for the shortest paths between these necessary links, all other links on the path can be inferred. Therefore, only these six links are required to be known to the receiver instead of all 12 links.

C. Rapidly Convergent Path Reconstruction

In summary, according to Theorem II, the number of links required to be known to the receiver is reduced from $n - 1$ to $\frac{1}{2}(n-1)$, which will significantly accelerate the convergence of path reconstruction. The rapidly convergent method of path reconstruction is summarized in Algorithm 3. Compared to the general version shown in Algorithm 2, the accelerated method requires only the links leading from the access layer to the aggregation layer and from the core layer to the aggregation layer. All other links (Agg-ToR and Agg-Core) on the path are no longer required. These links can be uniquely determined by searching for the shortest paths between the necessary links. Obviously, the expected number of packets that need to be observed at the receiver can thus be considerably reduced.

V. Discussion

Variable Probability: Since the receiver needs only to be aware of the links leading from the access layer to the aggregation layer and from the core layer to the aggregation layer, one interesting question is whether it would be better to leverage a variable sample probability instead of a fixed sample probability for tracing the flow paths. The basic idea is that we would regard the above two types of links as critical links and regard other links as noncritical links. To further accelerate the convergence of path reconstruction, the switches could be assigned a higher probability of sampling critical links and a lower probability of sampling noncritical links. In particular, an extreme manifestation of this idea would to sample only critical links and never sample other links. Theoretically, this method would ensure that each packet received at the receiver would carry a sample of a critical link. However, to distinguish critical links from noncritical links, it would be necessary to install additional flow rules on the switches. We find that the number of additional flow rules is proportional to the number of ports on the switches. Although this method is feasible in SDN-enabled networks, it may not be suitable for incremental deployment in large-scale datacenters.

Multitenancy: The method proposed in this paper relies on double VLAN tags in the data packets to encode the link samples. However, one common scenario in datacenter networks is multitenancy, which might result in conflicting usage of the VLAN tags. To allow the method to work in the case of multitenancy, we suggest that network operators should leverage VXLAN or GRE [30] for segmented encapsulation for multitenancy. Both are current mainstream technologies used for multitenancy in large-scale SDN-enabled datacenter networks and can provide 24-bit virtual network identifiers for distinguishing more than 16 million tenants. Moreover, they can coexist well with the double VLAN tags (Q-in-Q) used in FlowTracer.

Multiple paths: In this paper, we focus on the design of FlowTracer for datacenter networks. A natural question here is whether it is possible to generalize FlowTracer to the case of load balancing [31]–[33]. For flow-level load balancing schemes such as ECMP, the packets of each flow traverse the same path. However, for non-flow-level load balancing schemes such as Flowlet, the packets of a flow traverse multiple paths. For a multipath flow, there is a sub-flow in each path. We assume that the route of each sub-flow is sufficiently stable, and the number of packets in each path is sufficient for path reconstruction. When the assumption holds, the trajectory of each sub-flow can be reconstructed independently. With the trajectories of all sub-flows reconstructed by FlowTracer, the complete trajectory of the multipath flow can then be detected. Theorem II and its proof still apply in the case of multipath routing.

Path Changes: In this paper, we assume that the trajectory of each flow is constant during the path reconstruction. In practice, even a small possibility, it's still possible that the fault (e.g., switch port failure) happens to occur during the process. When this happens, the route may oscillate and the flow trajectory may change. For the same flow, the receiver may detect multiple trajectories, even including incomplete trajectories. Given this, the detected result is probably inconsistent with the actual one. However, note that the process of path reconstruction is cyclical. Although FlowTracer may reach its limits in the cycle encountering path changes, the consistent results are always detected in the next cycles, whose route has become stable.

Trajectory Length: While FlowTracer can trace a trajectory containing many detours, it should be kept in mind that there is a low probability of encountering many detours in practice. Since datacenter networks typically suffer only infrequent failures and misconfigurations [6]–[8], multiple detours will occur at the same time only rarely. Moreover, in a typical datacenter topology, the length of the shortest path between any two nodes is not too long. Thus, it is not a common task for FlowTracer to trace long trajectories. However, the ability to trace flow trajectories in the case of many detours is still critical since it can help to accurately detect network failures and misconfiguration issues. Techniques that only work well for tracing the shortest paths do not necessarily scale to the case of many detours. As the network scale increases, the number of potential detours will be greater, and the drastically increased number of paths caused by such detours will complicate the trajectory tracing task to an even greater extent.

Mice vs. Elephant Flows: We assume that the sender (source) sends sufficient packets that this sampling process can converge. Our approach relies on this property because we mark each packet with only a small piece of the path, meaning that the receiver must observe many such packets to reconstruct the complete path back to the sender. In practice, many mice flows[2] might emerge that require only a few packets to be transferred; then, this assumption may not hold. Note that this consideration is common to most flow measurement techniques, such as that presented in [34]. However, because mice flows often exert less impact on performance, it is desirable to focus instead on the measurement of elephant flows, which are more likely to play a predominant role [24]. Consequently, it might be challenging for the link sampling mechanism to identify the paths of mice flows. In other words, FlowTracer trades slightly reduced accuracy for significantly improved scalability in terms of switch entries and header space.

Flow Loss: One of the assumptions made in the design of FlowTracer is that a trajectory is traced when the flow reaches its destination. However, a flow might not reach its destination for a multitude of reasons, including drops due to network congestion, routing loops, or switch misconfigurations (e.g., race conditions [12]). It might be challenging for FlowTracer to identify the precise locations of such flow drops. Note that this problem is a common limitation for the reconstruction of trajectories at end hosts [14]. To overcome this limitation, previous work such as Pingmesh [35] could play a complementary role in combination with FlowTracer. In Pingmesh, a tracer is deployed at each end host to collect data on the end-to-end latency of the network. It can diagnose when and where a flow is lost by detecting the pattern of latency. In practice, a Pingmesh tracer could be deployed at each end host together with a FlowTracer path reconstructor. FlowTracer would be responsible for path detection in the case that the packets can reach their destination, whereas Pingmesh would serve the complementary purpose of detecting flow loss.

VI. Evaluation

In this section, we show that FlowTracer detects trajectories quickly while placing far lesser demands on both switch entries and packet header space than state-of-the-art techniques. Specifically, we evaluate FlowTracer and the state-of-the-art technique PathletTracer [14] in testbed experiments using a 48-ary scale fat-tree topology. We seek to answer the following questions: (1) What is FlowTracer's overhead in terms of the required flow rules and packet header space? (2) How many decoding entries are required at the end host for FlowTracer? (3) How does FlowTracer perform in terms of detection latency?

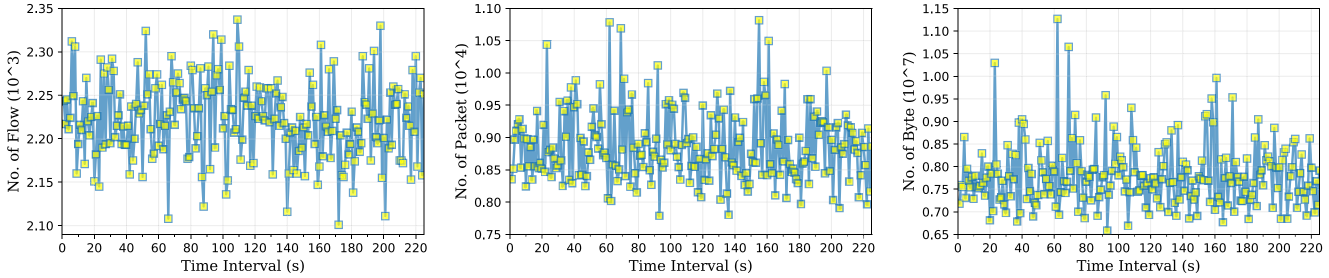
Code and Data Sharing. We have made the data snapshots and code used for FlowTracer[3] available to the research community in the hope that this will stimulate and facilitate further research.

A. Experimental Setup

Network Topology: We simulated a 48-radix fat-tree topology with 48-port OpenFlow-enabled switches capable of providing a full 1 Gbps of bandwidth to up to 27,648 hosts, which should be a common scenario in large-scale SDN-enabled datacenter networks [36], [37]. There are 576 core switches. Each core switch has one port connected to each of 48 pods. Each pod contains an edge (ToR) layer and an aggregation layer with 24 switches each. The edge (ToR) switches in every pod are assigned to 24 hosts each. There are 576 equal-cost paths between any given pair of hosts in different pods. The number of links in the topology is 110,592 in total. The simulation of this topology was implemented with Mininet [38] and Open vSwitch [5]. All switches offer support for the OpenFlow 1.3 specification.

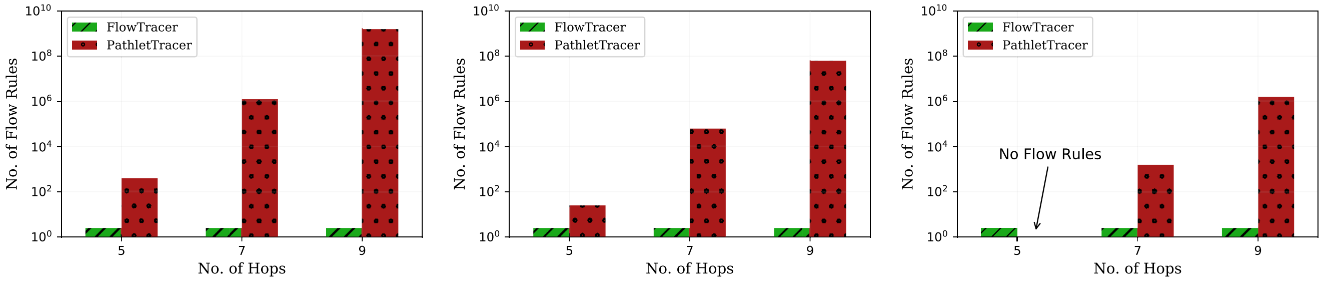
[2] A mice flow is a short (in total bytes) flow set up by a TCP (or other protocol) flow.

[3] https://github.com/FlowAnalysis/FlowTracer



(a) No. of Flows (b) No. of Packets (c) No. of Bytes

Fig. 11. Network traffic characteristics collected per second in terms of (a) flows, (b) packets and (c) bytes.



(a) ToR (b) Aggregate (c) Core

Fig. 12. FlowTracer's demands in terms of the number of switch flow rules for detecting trajectories compared to PathletTracer. For detecting non-shortest paths, i.e., cases in which packets traverse a larger number of links due to detours, the number of switch flow rules required by PathletTracer increases superlinearly. In contrast, the number of switch flow rules required by FlowTracer remains constant.

Network Traffic: We generated traffic between two end hosts in different pods, as shown in Fig. 11. The traffic was forwarded over diverse paths of lengths ranging from 5 hops (the shortest path) to 13 hops (a path with detours).

B. Switch Flow Rules

For any given switch, FlowTracer requires only 3 constant entries, two of which are flow table entries and one of which is a group table entry. In contrast, for any given switch, PathletTracer requires as many switch flow rules as the number of paths traversing the switch. Since the latter depends on the layer in which the switch resides, we plot the numbers of switch flow rules for FlowTracer and PathletTracer for each layer separately in Fig. 12.

Either for detecting shortest paths only or for detecting non-shortest paths with detours, the number of switch flow rules required by PathletTracer is far greater than that required by FlowTracer. In particular, for the detection of non-shortest paths, e.g., in the case of failures, the number of switch flow rules required by PathletTracer grows superlinearly with the path length. For detecting 7-hop paths, PathletTracer requires more than one million flow rules on a ToR switch, tens of thousands of flow rules on an aggregate switch, and thousands of flow rules on a core switch. It is clear that PathletTracer cannot scale well to datacenter networks because it is very sensitive to the number of paths in the network. In contrast, the number of switch flow rules required by FlowTracer is small and completely independent of the number of paths.

Because of the small and constant number of flow rules required, FlowTracer can be implemented in TCAM and thus can benefit from TCAM's fast, parallel lookups. In contrast, the switch flow rules of PathletTracer cannot be accommodated in TCAM. As an alternative, they must be stored in SRAM or even DRAM, at the cost of supporting less efficient lookups.

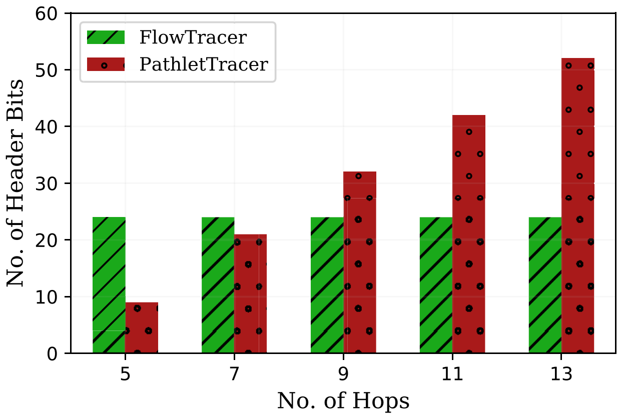


Fig. 13. FlowTracer's demands in terms of packet header space for detecting trajectories compared to PathletTracer. In particular, for detecting 7-hop paths in a 48-ary fat-tree topology, FlowTracer requires 24 bits of header space, while PathletTracer requires 21 bits. However, beyond 7 hops, FlowTracer requires a constant 24 bits, far less than PathletTracer.

C. Packet Header Space

Here, we examine the overhead in terms of packet header space. PathletTracer requires log(P) bits of header space, where P is the number of paths between any two end hosts. As shown in Fig. 13, for 7-hop paths, we find that FlowTracer

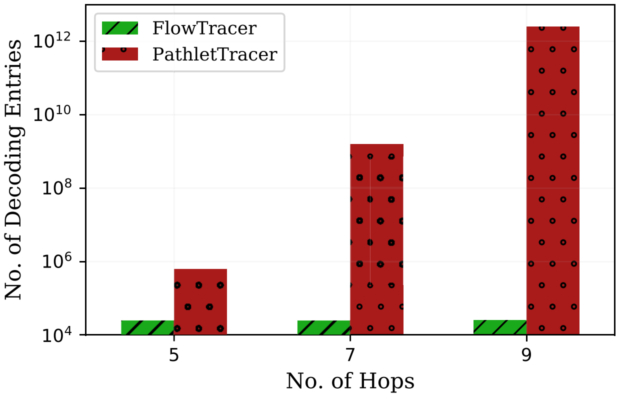


Fig. 14. FlowTracer requires significantly fewer data entries at the end hosts for detecting trajectories than PathletTracer does. In particular, for detecting 7-hop paths in a 48-ary fat-tree topology, FlowTracer requires several orders of magnitude fewer data entries than PathletTracer, which requires a data file of approximately 12 GB at each end host.

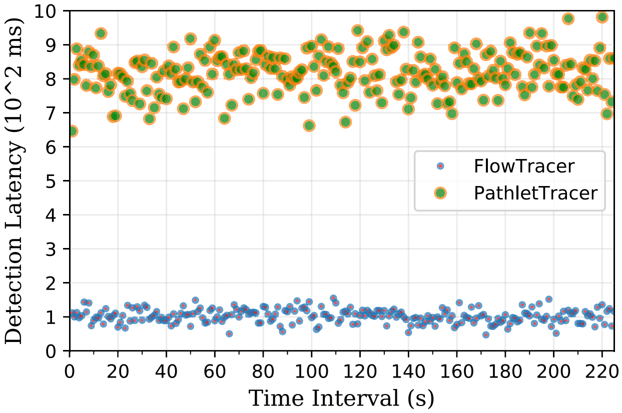


Fig. 15. FlowTracer achieves a significantly lower average latency than PathletTracer does for trajectory detection. In particular, for detecting 7-hop paths in a 48-ary fat-tree topology, FlowTracer requires only approximately 100 ms, while PathletTracer requires approximately 1000 ms.

requires a constant 24-bits of header space, slightly more than PathletTracer. However, FlowTracer trades this slightly higher overhead for short paths for a significant improvement in scalability. When the path length is greater than 7 hops, FlowTracer requires far less packet header space than PathletTracer does due to the sharp increase in the number of paths, which causes the size of the path identifiers for PathletTracer to grow rapidly. Thus, FlowTracer is better suited to large-scale datacenter networks.

Moreover, its excessive header space overhead can even make PathletTracer incompatible with current encapsulation. Current mainstream encapsulation technology, e.g., VLAN, Q-in-Q, VXLAN or GRE, provides unique identifiers of at most 24 bits for path tracing. Obviously, this is not sufficient for PathletTracer, especially with many detours. In contrast, regardless of the scale of the network topology, the header space demand of FlowTracer remains constant at 24 bits (double VLAN tags, Q-in-Q), which should be compatible with current encapsulation.

D. Decoding Entries

For path reconstruction, both FlowTracer and PathletTracer need to store a certain number of data entries at the end hosts. Here, we quantify the numbers of data entries required for both schemes. Theoretically, FlowTracer needs to store only the entire set of network links at the end hosts. In contrast, PathletTracer requires the end hosts to store a large size codebook, in which each data entry is a code assigned to a unique path. For a fair comparison, we assume that every individual end host in PathletTracer stores an equal-sized subset of the codebook that is relevant only to that end host. As shown in Fig. 14, PathletTracer needs to store a considerable number of data entries, especially in the case of many detours. For tracing 7-hop paths, PathletTracer requires more than 10^9 data entries at each end host, which translates into a data file of approximately 12 GB in size (under the assumption that each data entry is approximately 12 bytes). Because FlowTracer needs to store only the set of links, it requires far fewer data entries per host.

In practice, these data entries for decoding or tracing paths should be stored in memory for acceleration. However, for PathletTracer, it is unlikely that such a large block file (perhaps as large as 12 GB) can be completely pushed into memory. Note that the data entries should reside in the memory and that the capacity of fast memory is very tight.

E. Detection Latency

Finally, we evaluate the performance in terms of detection latency. Theoretically, the total latency consists of two time contributions: (1) the transmission time and (2) the path reconstruction time. Since current SDN-enabled switches can provide a high bisection bandwidth [39] and OpenFlow pipeline rate [40], [41], the transmission time (on the order of μs) can be ignored. However, the time needed for path reconstruction cannot be overlooked. On the one hand, while FlowTracer requires less query response time (with few decoding entries), it requires multiple queries (i.e., the receiver needs to receive multiple packets with sampled link information). On the other hand, while PathletTracer requires only a single query (i.e., the receiver needs to receive only a single packet with the complete path information), its query response time is longer (with a vast number of decoding entries).

In Fig. 15, we plot the average time needed to reconstruct a 7-hop path. FlowTracer is overwhelmingly superior to PathletTracer in terms of the average detection latency, with a significant gap of approximately a factor of 10. It is clear that FlowTracer can detect trajectories quickly in large-scale datacenter networks, even though it is a probabilistic method. The average number of packets required for trajectory detection is shown in Fig. 16. We find that the number of required packets is distributed around the upper bound of expectation as analyzed in Section IV-A.

Although FlowTracer requires multiple packets to capture a flow trajectory instead of only a single packet, FlowTracer still wins in practice in terms of the reconstruction time. We can obtain the following insights from the results: (1) the query response time takes up the largest portion of the path reconstruction time, (2) FlowTracer trades the need for a few extra packets for a significant decrease in the query response

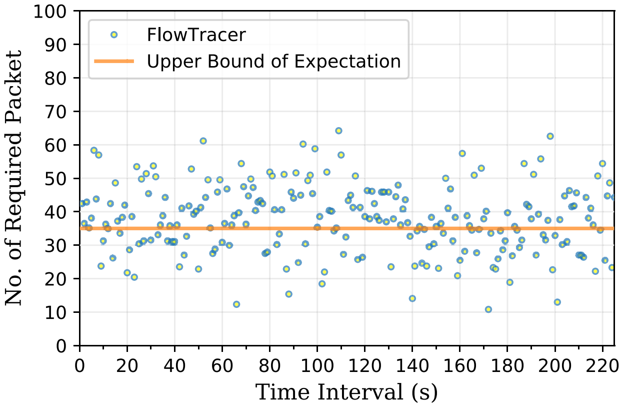


Fig. 16. FlowTracer's demand in terms of the number of packets required for path reconstruction, which is approximately as expected.

time, and (3) the availability of a high bisection bandwidth weakens the influence of these extra packets. In other words, FlowTracer has the advantage of speed simply because of its reliance on probabilistic sampling. Because of the probabilistic sampling approach, FlowTracer requires fewer decoding entries, enabling faster path reconstruction.

VII. Related Work

The challenging research topic of trajectory detection has strong implications for troubleshooting, eliminating configuration conflicts, avoiding routing errors, and detecting policy inconsistencies [42], [43] in large-scale SDN-enabled datacenter networks. A series of previous works have addressed this problem. They can be divided into two categories: control plane solutions [10]–[12] and data plane solutions [13]–[15], [44].

Control plane techniques: Unlike data plane solutions, control plane solutions always rely on after-the-fact analysis conducted at a non-data plane and incur considerable overhead in the form of out-of-band data collection. They also tend to perform worse than data plane solutions. SDN traceroute [10] collects the switch's packet-in message at each hop and reconstructs the trajectory at the controller. VeriFlow [11] analyzes the configurations pushed to network devices to infer forwarding paths. NetSight [12] forces all switches to send postcards when traffic passes through them and introduces a history plane for trajectory reconstruction. In contrast, FlowTracer traces traffic trajectories directly on the data plane without reliance on the controller.

Data plane techniques Carrying information to be used for tracing in the packet header space is an idea that has been used in [45], [46]. X-Trace [45] is a tracing framework designed to reconstruct an Internet service's task tree by propagating task ID metadata across layers and applications. FlowTags [46] adds tags to outgoing packets for systematic policy enforcement on switches and middleboxes.

In comparison with out-of-band techniques, in-band solutions typically work better at the data plane and significantly reduce the data collection overhead at the cost of supporting a narrower range of analysis. A tiny packet program (TPP) interface was introduced in [13] that enables end hosts to collect the packet histories recorded by each switch. The interface provides more visibility of network behaviors, similar to out-of-band techniques, but is not compatible with existing standard SDN frameworks, e.g., OpenFlow. OpenFlow-compatible solutions such as PathletTracer [14] and PathQuery [15] attempt to imprint each packet of a flow with compressed path information and consequently require a large number of switch flow rules and considerable packet header space because of the vast number of paths in a large-scale datacenter network and the complex matching operations conducted in switches.

CherryPick [17] is similar to FlowTracer in that it allows tracers at the end hosts to be aware of the underlying network topology. Although its objective is identical to that of FlowTracer (to reduce resource consumption), it is designed from a completely different perspective. It attempts to identify distinct trajectories in a datacenter network based on two fixed links tagged per packet. Compared to FlowTracer, it has the advantage of being able to detect the paths of mice flows precisely. However, mice flows often have little impact on performance [19], and a limitation of this approach is critical: the limited information (two fixed links) carried by each packet often cannot reveal the complete trajectory when many detours are involved. As a matter of fact, large flows with many detours in a network typically lead to severe issues. Therefore, it is essential for a solution to be able to reconstruct the complete trajectories in this case. PathDump [44] extends CherryPick, and enables network debugging functionality by leveraging detected paths at end hosts. However, PathDump traces trajectories by using the idea presented in CherryPick, which still cannot avoid the above limitation. In addition to being unable to trace long paths, CherryPick and its extension PathDump cannot adapt to the asymmetric topology. In contrast, the general path reconstruction method of FlowTracer is not limited by the topology type.

In summary, previous works that have addressed trajectory detection have not yet provided an efficient solution. In contrast, FlowTracer is compatible with OpenFlow and can significantly reduce the overhead in terms of switch flow rules and packet header space. Most importantly, FlowTracer is able to reveal the trajectories in the case of severe issues involving many detours.

SDN-enabled programmable data plane: Recent advances in data plane programmability [40], [41], [47], [48] make it easy to enable numerous network troubleshooting functionalities, including trajectory tracing. FlowTracer permits the efficient implementation of a path tracing functionality on top of these flexible platforms. Therefore, our method is complementary to them.

VIII. Conclusion

We present FlowTracer, an efficient trajectory detection technology for large-scale SDN-enabled datacenter networks, which uses link sampling to decrease the overhead in terms of the number of switch entries and the amount of packet header space required. To this end, we first introduce a method of probabilistic packet tagging that is conducted in OpenFlow-enabled switches with very few switch flow rules and limited packet header space by means of double VLAN tags. Then, we explore the topological structure of datacenter networks and propose a method of path reconstruction that is conducted at end hosts and achieves rapid convergence. Finally, we evaluate FlowTracer on a 48-ary fat-tree topology. The results show that FlowTracer works very well for path tracing even in the case of many detours. Compared to state-of-the-art techniques, FlowTracer requires far fewer switch flow rules and less packet header space.

REFERENCES

[1] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 3, pp. 1617–1634, 3rd Quart., 2014.

[2] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[3] P. C. D. R. Fonseca and E. S. Mota, "A survey on fault management in software-defined networks," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2284–2321, 4th Quart., 2017.

[4] B. Heller *et al.*, "Leveraging SDN layering to systematically troubleshoot networks," in *Proc. ACM HotSDN*, 2013, pp. 37–42.

[5] B. Pfaff *et al.*, "The design and implementation of open vSwitch," in *Proc. USENIX NSDI*, 2015, pp. 117–130.

[6] A. Capone, C. Cascone, A. Q. T. Nguyen, and B. Sansò, "Detour planning for fast and reliable failure recovery in SDN with openstate," in *Proc. Conf. Design Rel. Commun. Netw.*, 2015, pp. 25–32.

[7] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 323–334, 2012.

[8] L. Schiff, S. Schmid, and P. Kuznetsov, "In-band synchronization for distributed SDN control planes," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 1, pp. 37–43, 2016.

[9] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 554–566, Apr. 2014.

[10] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "SDN traceroute: Tracing SDN forwarding without changing network behavior," in *Proc. ACM HotSDN*, 2014, pp. 145–150.

[11] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. ACM HotSDN*, 2012, pp. 49–54.

[12] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX NSDI*, 2014, pp. 71–85.

[13] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 3–14, 2014.

[14] H. Zhang, C. Lumezanu, J. Rhee, N. Arora, Q. Xu, and G. Jiang, "Enabling layer 2 pathlet tracing through context encoding in software-defined networking," in *Proc. ACM HotSDN*, 2014, pp. 169–174.

[15] S. Narayana, J. Rexford, and D. Walker, "Compiling path queries in software-defined networks," in *Proc. ACM HotSDN*, 2014, pp. 181–186.

[16] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[17] P. Tammana, R. Agarwal, and M. Lee, "CherryPick: Tracing packet trajectory in software-defined datacenter networks," in *Proc. ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, 2015, p. 23.

[18] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *Proc. ACM HotSDN*, 2014, pp. 175–180.

[19] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, 2003.

[20] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *Proc. IEEE INFOCOM*, 2011, pp. 1629–1637.

[21] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 493–512, 1st Quart., 2014.

[22] A. Boneh and M. Hofri, "The coupon-collector problem revisited—A survey of engineering problems and computational methods," *Commun. Stat. Stochastic Models*, vol. 13, no. 1, pp. 39–66, 1997.

[23] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *Proc. ACM WREN*, 2009, pp. 65–72.

[24] T. Benson, A. Akella, and A. M. David, "Network traffic characteristics of data centers in the wild," in *Proc. ACM IMC*, 2010, pp. 267–280.

[25] S. Nelakuditi, S. Lee, Y. Yu, Z.-L. Zhang, and C.-N. Chuah, "Fast local rerouting for handling transient link failures," *IEEE/ACM Trans. Netw.*, vol. 15, no. 2, pp. 359–372, Apr. 2007.

[26] B. Yang, J. Liu, S. Shenker, J. Li, and K. Zheng, "Keep forwarding: Towards *k*-link failure resilient routing," in *Proc. IEEE INFOCOM*, 2014, pp. 1617–1625.

[27] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella, "The TCP outcast problem: Exposing unfairness in data center networks," in *Proc. USENIX NSDI*, 2012, p. 30.

[28] F. P. Tso, G. Hamilton, R. Weber, C. S. Perkins, and D. P. Pezaros, "Longer is better: Exploiting path diversity in data center networks," in *Proc. IEEE ICDCS*, 2013, pp. 430–439.

[29] K. Zarifis, R. Miao, M. Calder, E. Katz-Bassett, M. Yu, and J. Padhye, "DIBS: Just-in-time congestion mitigation for data centers," in *Proc. ACM EuroSys*, 2014, p. 6.

[30] S. Guenender, K. Barabash, Y. Ben-Itzhak, A. Levin, E. Raichstein, and L. Schour, "NoEncap: Overlay network virtualization with no encapsulation overheads," in *Proc. ACM SOSR*, 2015, p. 9.

[31] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *Proc. IEEE INFOCOM*, 2013, pp. 2130–2138.

[32] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," in *Proc. ACM SIGCOMM*, 2011, pp. 266–277.

[33] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *Proc. USENIX NSDI*, 2017, pp. 407–420.

[34] T. Yang *et al.*, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. ACM SIGCOMM*, 2018, pp. 561–575.

[35] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. ACM SIGCOMM*, 2015, pp. 139–152.

[36] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, 2008, pp. 63–74.

[37] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. USENIX NSDI*, 2010, p. 19.

[38] B. Lantz and B. O'Connor, "A mininet-based virtual testbed for distributed SDN development," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 365–366, 2015.

[39] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM*, 2009, pp. 51–62.

[40] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[41] N. Shelly, E. J. Jackson, T. Koponen, N. McKeown, and J. Rajahalme, "Flow caching for high entropy packet fields," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 151–156, 2014.

[42] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM*, 2011, pp. 290–301.

[43] H. Zeng *et al.*, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *Proc. USENIX NSDI*, 2014, pp. 87–99.

[44] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *Proc. USENIX OSDI*, 2016, pp. 233–248.

[45] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proc. USENIX NSID*, 2007, p. 20.

[46] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proc. ACM HotSDN*, 2013, pp. 19–24.

[47] S. Li *et al.*, "Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability," *IEEE Netw.*, vol. 31, no. 2, pp. 58–66, Mar./Apr. 2017.

[48] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. ACM SIGCOMM*, 2013, pp. 99–110.

Junxiao Wang received the B.S. degree from Dalian Maritime University, Dalian, China, in 2014. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Dalian University of Technology, Dalian. His current research interests include software defined networking, network function virtualization, and cloud computing.

Heng Qi (M'15) received the bachelor's degree from Hunan University in 2004, and the master's and Ph.D. degrees from the Dalian University of Technology, China, in 2006 and 2012, respectively. He is a JSPS Overseas Research Fellow with the Graduate School of Information Science, Nagoya University, Japan, from 2016 to 2017. He is currently an Associate Professor with the School of Computer Science and Technology, Dalian University of Technology. His research interests include computer network and multimedia computing.

Yang He received the bachelor's degree from the Dalian University of Technology, China, in 2013, where he is currently pursuing the master's degree with the School of Computer Science and Technology. His research interest includes software defined networking.

Wenxin Li received the B.E. and Ph.D. degrees from the School of Computer Science and Technology, Dalian University of Technology, in 2012 and 2018, respectively. He is currently a Post-Doctoral Researcher with the Hong Kong University of Science and Technology. From May 2014 to May 2015, he was a Research Assistant with the National University of Defense Technology. From October 2016 to September 2017, he was a visiting student with the Department of Electrical and Computer Engineering, University of Toronto. His research interests include datacenter networks and cloud computing.

Keqiu Li (SM'14) received the bachelor's and master's degrees from the Department of Applied Mathematics, Dalian University of Technology, China, in 1994 and 1997, respectively, and the Ph.D. degree from the Graduate School of Information Science, Japan Advanced Institute of Science and Technology in 2005. He also has two-year Post-Doctoral experience with the University of Tokyo, Japan. He is currently a Professor with the School of Computer Science and Technology, Dalian University of Technology. He has published over 200 technical papers, such as the IEEE/ACM TRANSACTIONS ON NETWORKING, the IEEE TRANSACTIONS ON MOBILE COMPUTING, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and *ACM Transactions on Internet Technology*. His research interests include Internet technology, data center networks, cloud computing, and wireless networks.

Xiaobo Zhou received the B.Sc. degree in electronic information science and technology from the University of Science and Technology of China, Hefei, China, in 2007, the M.E. degree in computer application technology from the Graduate University of Chinese Academy of Science, Beijing, China, in 2010, and the Ph.D. degree from the School of Information Science, Japan Advanced Institute of Science and Technology, Nomi, Japan, in 2013. He was a Researcher with the Centre for Wireless Communications, University of Oulu, Finland, from 2014 to 2015. He is currently an Associate Professor with the Tianjin Key Laboratory of Advanced Networking, College of Intelligence and Computing, Tianjin University. His research interests include joint source-channel coding, co-operative wireless communications, network information theory, cloud computing, and software defined networking.