CherryPick: Tracing Packet Trajectory in Software-Defined Datacenter Networks

Praveen Tammana
University of Edinburgh

Rachit Agarwal
UC Berkeley

Myungjin Lee
University of Edinburgh

ABSTRACT

SDN-enabled datacenter network management and debugging can benefit by the ability to trace packet trajectories. For example, such a functionality allows measuring traffic matrix, detecting traffic anomalies, localizing network faults, etc. Existing techniques for tracing packet trajectories require either large data collection overhead or large amount of data plane resources such as switch flow rules and packet header space. We present CherryPick, a scalable, yet simple technique for tracing packet trajectories. The core idea of our technique is to *cherry-pick* the links that are key to representing an end-to-end path of a packet, and to embed them into its header on its way to destination. Preliminary evaluation on a fat-tree topology shows that CherryPick requires minimal switch flow rules, while using header space close to state-of-the-art techniques.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations

Keywords

Network monitoring; software-defined network

1. INTRODUCTION

Software-Defined Networking (SDN) has emerged as a key technology to make datacenter network management easier and more fine-grained. SDN allows network operators to express the desired functionality using high-level abstractions at the control plane, that are automatically translated into low-level functionality at the data plane. However, debugging SDN-enabled networks is challenging. In addition to network misconfiguration errors and failures [10, 13, 16], network operators need to ensure that operations at the data plane conform to the high-level policies expressed at the control plane. Noting that traditional tools (*e.g.*, NetFlow, sFlow, SNMP, traceroute) are simply insufficient to debug SDN-enabled networks, a number of tools have been developed recently [1, 11, 12, 14, 17, 26, 27].

SOSR2015, June 17 - 18, 2015, Santa Clara, CA, USA
Copyright 2015 ACM. ISBN 978-1-4503-3451-8/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2774993.2775066.

A particularly interesting problem in SDN debugging is to be able to reason about flow of traffic (*e.g.*, tracing individual packet trajectories) through the network [11, 13, 14, 16, 17, 27]. Such a functionality enables measuring network traffic matrix [28], detecting traffic anomalies caused by congestion [9], localizing network failures [13, 14, 16], or simply ensuring that forwarding behavior at the data plane matches the policies at the control plane [11]. We discuss related work in depth in §5, but note that existing tools for tracing packet trajectories can use one of the two broad approaches. On the one hand, tools like NetSight [11] support a wide range of queries using after-the-fact analysis, but also incur large "out-of-band" data collection overhead. In contrast, "in-band" tools (*e.g.*, PathQuery [17] and PathletTracer [27]) significantly reduce data collection overhead at the cost of supporting a narrower range of queries.

We present CherryPick, a scalable, yet simple "in-band" technique for tracing packet trajectories in SDN-enabled datacenter networks. CherryPick is designed with the goal of minimizing two data plane resources: the number of switch flow rules and the packet header space. Indeed, existing approaches to tracing packet trajectories in SDN trade off one of these resources to minimize the other. At one end of the spectrum is the most naïve approach of assigning each network link a unique identifier and switches embedding the identifier into the packet header during the forwarding process. This minimizes the number of switch flow rules required, but has high packet header space overhead especially when the packets traverse along non-shortest paths (*e.g.*, due to failures along the shortest path). At the other end are techniques like PathletTracer [27] that aim to minimize the packet header space, but end up requiring a large number of switch flow rules (§3); PathQuery [17] acknowledges a similar limitation in terms of switch resources.

CherryPick minimizes the number of switch flow rules required to trace packet trajectories by building upon the naïve approach — each network link is assigned a unique identifier and switches simply embed the identifier into the packet header during the forwarding process. However, in contrast to the naïve approach, CherryPick minimizes the packet header space by *selectively picking a minimum number of essential links to represent an end-to-end path*. By exploiting the fact that datacenter network topologies are often well-structured, CherryPick requires packet header space comparable to state-of-the-art solutions [27], while retaining the minimal switch flow rule requirement of the naïve approach. For instance, Table 1 compares the number of switch flow rules and the packet header space required by CherryPick against the above two approaches for a 48-ary fat-tree topology.

Table 1: CherryPick achieves the best of the two existing techniques for tracing packet trajectories — the minimal number of switch flow rules required by the naïve approach and close to the minimal packet header space required by PathletTracer [27]. These results are for a 48-ary fat-tree topology; M and B stand for million and billion respectively. See §3 for details.

	CherryPick			PathletTracer			Naïve		
Path length	4	6	8	4	6	8	4	6	8
#Flow rules	48	48	48	576	1.2M	1.7B	48	48	48
#Header bits	11	22	33	10	21	31	24	36	48

In summary, we make three contributions:

- We design CherryPick, a simple and scalable packet trajectory tracing technique for SDN-enabled datacenter networks. The main idea in CherryPick is to exploit the structure in datacenter network topologies to minimize number of switch flow rules and packet header space required to trace packet trajectories. We apply CherryPick to a fat-tree topology in this paper to demonstrate the benefits of the technique (§2).

- We show that CherryPick can trace all 4- and 6-hop paths in an up-to 72-ary fat-tree with no hardware modification by using IEEE 802.1ad double-tagging (§2).

- We evaluate CherryPick over a 48-ary fat-tree topology. Our results show that CherryPick requires minimal number of switch flow rules while using packet header space close to state-of-the-art techniques (§3).

2. CHERRYPICK

In this section, we describe the CherryPick design in detail with a focus on enabling L2/L3 packet trajectory tracing in a fat-tree network topology. We discuss, in §4, how this design can be generalized to other network topologies.

2.1 Preliminaries

The fat-tree topology. A k-ary fat-tree topology contains three layers of k-port switches: Top-of-Rack (ToR), aggregate (Agg) and core. A 4-ary fat-tree topology is presented in Figure 1 (ToR switches are nodes with letter T, Agg with letter A and core with letter C). The topology consists of k pods, each of which has a layer of $k/2$ ToR switches and a layer of $k/2$ Agg switches. $k/2$ ports of each ToR switch are directly connected to servers and each of remaining $k/2$ ports connected to $k/2$ Agg switches. The remaining $k/2$ ports of each Agg switch are connected to $k/2$ core switches. There are a total of $k^2/4$ core switches where port i on each core switch is connected to pod i.

To ease the following discussion, we group the links present in the topology into two categories: *i) intra-pod link* and *ii) pod-core link*. An intra-pod link is one connecting a ToR and an Agg switch, whereas a pod-core link is one connecting an Agg and a core switch. Note that there are $k^2/4$ intra-pod links within a pod and a total of $k^3/4$ pod-core links. In addition, we define *affinity core segment*, or affinity segment, to be the group of core switches that can be directly reached by each Agg switch at a particular position in each pod. In Figure 1, $C1$ and $C2$ in affinity segment 1 are directly reached by Agg switches at the lefthand side in each pod (*i.e.*, $A1$, $A3$, $A5$, and $A7$).

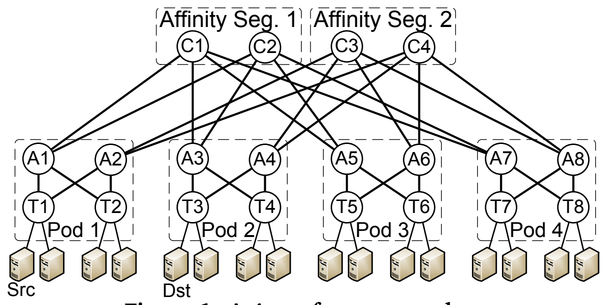

Figure 1: A 4-ary fat-tree topology.

Routing along non-shortest paths. While datacenter networks typically use shortest path routing, packets can traverse along non-shortest paths due to several reasons. First, node and/or link failures can enforce routing of packets along non-shortest paths [5, 18, 24]. Consider, for instance, the topology in Figure 1 where a packet is being routed between Src and Dst along the shortest path Src → T1 → A1 → C1 → A3 → T3 → Dst. If link C1 → A3 fails upon the packet arrival, the packet will be forced to traverse a non-shortest path. Second, recently proposed techniques reroute packets along alternate (potentially non-shortest) paths [20, 23, 25] to avoid congested links. Finally, misconfiguration may also create similar situations. We use "detour" to collectively refer to situations that force packets to traverse along a non-shortest path. The ability to be able to trace packet trajectories in case of packet detours is, thus, important since this may reveal network failures and/or misconfiguration issues.

However, packet detours complicate trajectory tracing due to the vastly increased number of possible paths even in a medium-size datacenter. For instance, given a 48-ary fat-tree topology, the number of shortest paths (*i.e.*, 4-hop paths) between a host pair in different pods is just 576. On the other hand, there exist almost 1.31 million 6-hop paths for the same host pair. As we show in §3, techniques that work well for tracing shortest paths [27] do not necessarily scale to the case of packet detours.

Detour model. As we discuss in §4, CherryPick is designed to work with arbitrary routing schemes. However, to ease the discussion in this paper, we focus on a simplified detour model where once the packet is forwarded by the Agg switch in the source pod, it does not traverse any ToR switch other than the ones in the destination pod. For instance, in Figure 1, consider a packet traversing from Src in Pod 1 to Dst in Pod 2. Under this simplification, the packet visits none of ToR switches $T5$, $T6$, $T7$ and $T8$ once it leaves $T1$. Of course, in practice, packets may visit ToR switches in non-destination pods. We leave a complete discussion of the general detour model to the full version.

2.2 Overview of CherryPick

We now give a high-level description of CherryPick design. Consider the naïve approach that embeds in the packet header an identifier (ID) for each link that the packet traverses. For a 48-port switch, it is easy to see that this approach requires $\lceil \log(48) \rceil = 6$ bits to represent each link. Indeed, the header space requirement for this naïve approach is far higher than the theoretical bound, $\log(P)$ bits, where P is the number of paths between any source-destination pair. For tracing 4-hop paths, the naïve scheme requires 24 bits whereas only 10 bits are theoretically required since P is 576 ($P = k^2/4$).

CherryPick builds upon the observation that datacenter network topologies are often well-structured and allow reconstructing the end-to-end path without actually storing each link as the packet traverses. CherryPick, thus, *cherry-picks a minimum number of links essential to represent an end-to-end path.* For instance, for the fat-tree topology, it suffices to store the ID of the pod-core link to reconstruct any 4-hop path. To handle a longer path, in addition to picking a pod-core link, CherryPick selects one extra link every additional 2 hops. Hence, tracing any n-hop path ($n \geq 4$) requires only $(n-4)/2+1$ links worth of header space[1]. However, the cherry-picking of links makes it impossible to use local port IDs as link identifiers and using global link IDs requires a large number of bits per ID due to the sheer number of links in the topology. CherryPick, thus, assigns link IDs in a manner that each link ID requires fewer bits than a global ID and that the end-to-end path between any source-destination pair can be reconstructed without any ambiguity. We discuss in §2.3 how CherryPick reconstructs end-to-end paths using cherry-picking the links along with a careful assignment of non-global link IDs.

CherryPick leverages VLAN tagging to embed chosen links in the packet header. While the OpenFlow standard [19] does not dictate how many VLAN tags can be inserted in the header, typically commodity SDN switches only support IEEE 802.1ad double-tagging. With two tags, CherryPick can keep track of all 1.31 million 6-hop paths in the 48-ary fat-tree while keeping switch flow memory overhead low. As hardware programmability in SDN switch increases [3, 12], we expect that the issue raised by the limited number of tags can be mitigated.

2.3 Design

We now discuss CherryPick design in depth. We focus on three aspects of the design: (1) selectively picking links that allow reconstructing the end-to-end path and configuring switch rules to enable link picking; (2) careful assignment of link IDs to further minimize the packet header space; and (3) the path reconstruction process using the link IDs embedded in the packet header.

Picking links. Consider a packet arriving at an input port. We call a link attached to the input port *ingress link*. For each packet, every switch has a simple link selection mechanism that can be easily converted into OpenFlow rules. If the packet matches one of the rules, the ingress link is picked; and its ID is embedded into the packet using VLAN tag.

The following describes the link selection mechanism at each switch level, and Figure 2 shows flow rules derived from the mechanisms:

- *ToR:* If a ToR switch receives the packet from an Agg switch and if the packet's source belongs to the same pod, the switch picks the ingress link connected to the Agg switch that forwarded the packet. However, if both source and destination are in the same pod, the switch ignores the ingress link (we use `write_metadata` command to implement the "do nothing" operation). For all other cases, no link is picked.
- *Aggregate:* If an Agg switch receives the packet from a ToR switch and if the packet's destination is in the same pod, the ingress link is chosen. Otherwise, no link is picked.
- *Core:* Core switch always picks the ingress link.

Using the above set of rules, CherryPick selects the minimum number of links required to reconstruct the end-to-end path of any packet. For ease of exposition, we present four examples in Figure 3. First, Figure 3(a) illustrates the baseline 4-hop scenario. In this scenario, core switch $C2$ only picks an ingress link and other switches (*e.g.*, $A1$, $A3$ and $T3$) do nothing. In case of

[1]A 2-hop path is the shortest path between servers in the same pod, for which CherryPick simply picks one intra-pod link at Agg.

Port	IP Src	IP Dst	Action
3	10.pod.0.0/16	10.pod.0.0/16	`write_metadata:` 0x0/0x0
4	10.pod.0.0/16	10.pod.0.0/16	`write_metadata:` 0x0/0x0
3	10.pod.0.0/16	*	`push_vlan_id:` linkID(3)
4	10.pod.0.0/16	*	`push_vlan_id:` linkID(4)

(a) ToR switch

Port	IP Src	IP Dst	Action
1	*	10.pod.0.0/16	`push_vlan_id:` linkID(1)
2	*	10.pod.0.0/16	`push_vlan_id:` linkID(2)

(b) Aggregate switch

Port	IP Src	IP Dst	Action
1	*	*	`push_vlan_id:` linkID(1)
2	*	*	`push_vlan_id:` linkID(2)
3	*	*	`push_vlan_id:` linkID(3)
4	*	*	`push_vlan_id:` linkID(4)

(c) Core switch

Figure 2: OpenFlow table entries at each switch layer for the 4-ary fat-tree. In this example, the address follows the form of 10.*pod.switch.host*, where *pod* denotes pod number (where switch is), *switch* denotes position of switch in the pod, *host* denotes the sequential ID of each host. These entries are stored in a separate table which will be placed at the beginning of a table pipeline. In (a), 3 and 4 are port numbers connected to Agg layer. In (b), 1 and 2 are port numbers connected to ToR layer.

one detour at source pod (Figure 3(b)), $T2$ and $C2$ will choose each ingress link of a packet while others not. A similar picking process is undertaken in case of one detour at destination pod (Figure 3(c)). When one detour occurs between aggregate and core switch (Figure 3(d)), only core switches which see the detoured packet pick the ingress links.

Link ID assignment. CherryPick assigns IDs for intra-pod links and pod-core links separately.

i) Intra-pod links: Since pods are separated by core switches, the same set of links IDs is used across all pods. Since each pod has $(k/2)^2$ intra-pod links, we need $(k/2)^2$ IDs. Links at the same position across pods have the same link ID.

ii) Pod-core links: Assigning IDs to pod-core links such that the correct end-to-end path can be reconstructed while minimizing the number of bits required to represent the ID is non-trivial. Indeed, one way to assign IDs is to consider all pod-core links, and assign each link a unique ID. However, there are $k^3/4$ pod-core links and such an approach would require $\lceil \log(k^3/4) \rceil$ many bits per link ID. We show that this can be significantly reduced by viewing the problem as an edge coloring problem of a complete bipartite graph. In this problem, the goal is to assign colors to the edges of the graph such that adjacent edges of a vertex have different colors. Edge-coloring a complete bipartite graph requires α different colors where α is the graph's maximum degree.

To view a fat-tree as a complete bipartite graph with two disjoint sets, we treat pod as a vertex in the first set and an affinity core segment as a vertex in the second set (compare Figures 4(a) and 4(b)). We group edges (links) from an Agg switch to an affinity core segment and supersede them by one edge that we call *cluster edge* (Figure 4(b)). Since the maximum degree is k (i.e., the number of cluster edges at an affinity core segment), we need k different colors to edge-color this bipartite graph. Note that one cluster edge is a collection of all $k/2$ links. Therefore, we need k different color sets such that each color set has $k/2$

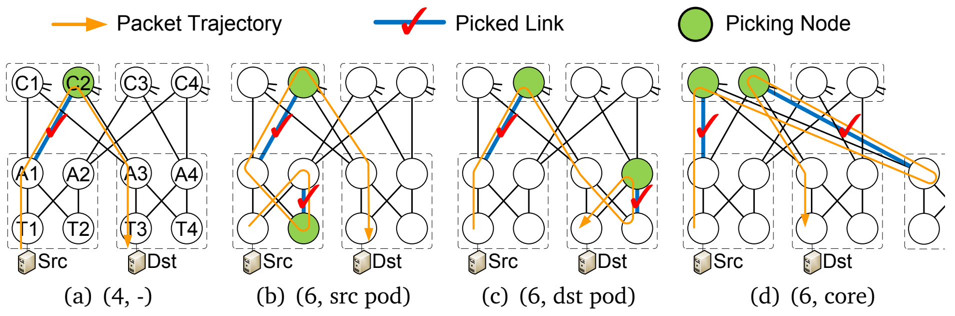

Figure 3: An illustration of link *cherry-picking* in CherryPick. (x, y) means x number of hops and detour at location y.

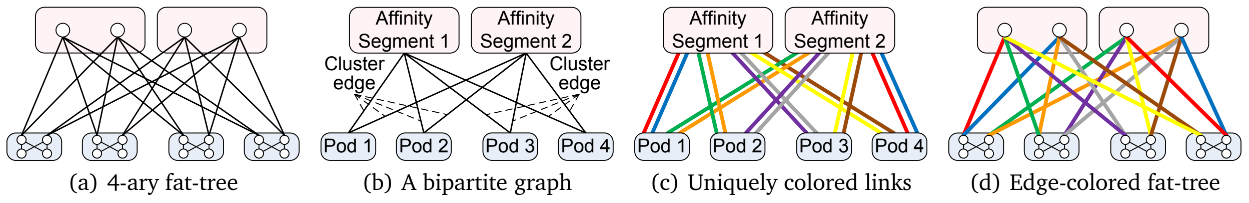

Figure 4: Edge-coloring pod-core links in a 4-ary fat-tree.

different colors and any two color sets are disjoint (Figure 4(c)). Thus in total $k(k/2)$ different colors are required. The actual color assignment is done by applying a near-linear time algorithm [6]. Figure 4(d) shows an accurate color allocation.

Putting it together, the number of unique IDs required is $3k^2/4$ ($(k/2)^2$ for the intra-pod links and $k(k/2)$ for the pod-core links). Thus, CherryPick requires a total of $\lceil \log(3k^2/4) \rceil$ bits to represent each link. For 48-ary and 72-ary fat-tree, CherryPick requires just 11 and 12 bits respectively to represent each link. CherryPick can thus support an up-to 72-ary fat-tree topology using the 12 bits available in VLAN tag.

Path reconstruction. In our scheme, when a packet reaches its destination, it contains a minimum set of link IDs necessary to reconstruct a complete path. To keep it simple, suppose that those link IDs in the header are extracted and stored in the order that they were selected. At the destination, a network topology graph is given and each link in the graph is annotated with one ID. Path reconstruction process begins from source ToR switch in the graph. Initially, a list S contains the source ToR switch. Until all the link IDs are consumed, the following steps are executed: i) take one link ID (say, l) from the ID list and find, from the topology graph, a link whose ID is l (if l is in the pod ID space, search for the link in either source or destination pod depending on whether pod-core link is consumed; otherwise, search for it in the current affinity segment); ii) identify two switches (say, s_a and s_b) that form the link; iii) out of the two, choose one (say, s_a) closer to the switch (say, s_r) that was most recently added to S; iv) find a shortest path (which is simple because it is either 1-hop or 2-hop path) between s_a and s_r and add all intermediate nodes (those closer to s_r first) and s_a later to S; v) add the remaining switch s_b to S. After all link IDs are consumed, we add to S the switches that form a shortest path from the switch included last in S to the destination ToR switch. Finally, we obtain a complete path by enumerating switches in S.

3. EVALUATION

In this section, we present preliminary evaluation results for CherryPick, and compare it against PathletTracer [27] over a 48-ary fat-tree topology. We evaluate the two schemes in terms of number of switch flow rules (§3.1), packet header space (§3.2) and end-host resources (§3.3) required for tracing packet trajectories. While preliminary, our evaluation suggests that:

- CherryPick requires minimal number of switch flow rules to trace packet trajectories. In particular, CherryPick requires as many rules as the *number of ports per switch*. In contrast, PathletTracer requires number of switch flow rules linear in *the number of paths* that the switch belongs to. For tracing 6-hop paths in a 48-ary fat-tree topology, for instance, CherryPick requires three orders of magnitude fewer switch rules than PathletTracer while supporting similar functionality.

- CherryPick requires packet header space close to state-of-the-art techniques. Compared to PathletTracer, CherryPick trades off slightly higher packet header space requirements for significantly improved scalability in terms of number of switch flow rules required to trace packet trajectories.

- CherryPick requires minimal resources at the end hosts for tracing packet trajectories. In particular, CherryPick requires as much as three orders of magnitude fewer entries at the destination when compared to PathletTracer for tracing 6-hop paths on a 48-ary fat-tree topology.

3.1 Switch flow rules

CherryPick, for any given switch, requires as many flow rules as the number of ports at that switch. In contrast, for any given switch, PathletTracer requires as many switch flow rules as the number of paths that contain that switch. Since the latter depends on the layer at which the switch resides, we plot the number of switch flow rules for CherryPick and PathletTracer across each layer separately (see Figure 5).

We observe that for tracing shortest paths only, the number of switch flow rules required by PathletTracer is comparable to those required by CherryPick. However, if one desires to trace non-shortest paths (*e.g.*, in case of failures), the number of switch flow requirement of PathletTracer grows super-linearly with the number of hops constituting the paths[2]. For tracing 6-hop paths,

[2]The number of switch flow rules required by PathletTracer could be reduced by tracing "pathlets" (a sub-path of an end-to-end path) at the cost of coarser tracing compared to CherryPick.

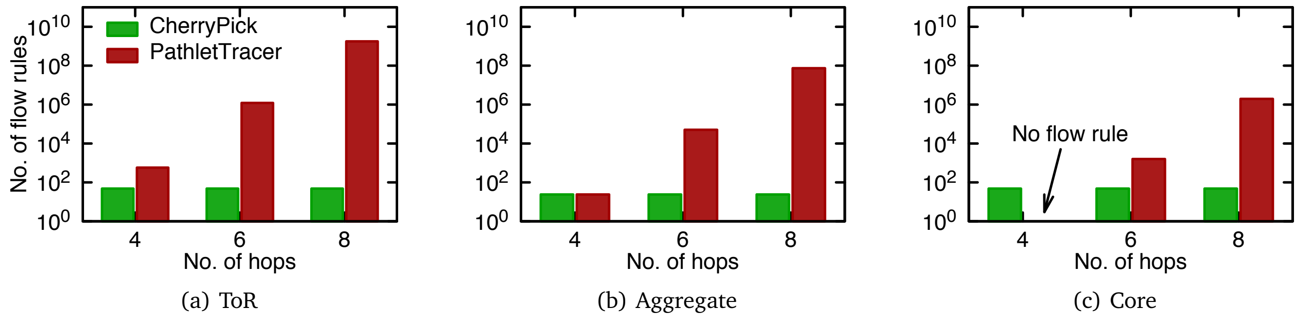

Figure 5: CherryPick requires number of switch flow rules comparable to PathletTracer for tracing shortest paths. However, for tracing non-shortest paths (*e.g.*, packets may traverse such paths in case of failures), the number of switch flow rules required by PathletTracer increases super-linearly. In contrast, the number of switch flow rules required by CherryPick remains constant.

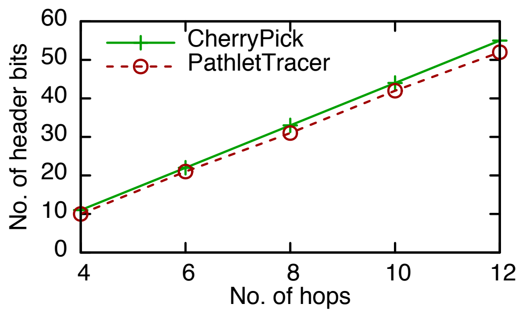

Figure 6: CherryPick requires packet header space comparable to PathletTracer for tracing packet trajectories. In particular, for tracing 6-hop paths in a 48-ary fat-tree topology, CherryPick requires 22 bits while PathletTracer requires 21 bits worth of header space.

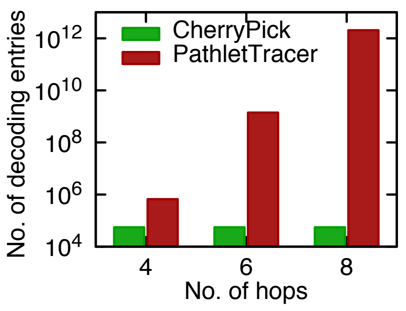

Figure 7: CherryPick requires significantly fewer entries than PathletTracer at each end host compared to trace packet trajectories. In particular, for tracing 6-hop paths in a 48-ary fat-tree topology, CherryPick requires less than 1MB worth of entries while PathletTracer requires approximately 12GB worth of entries at each end host.

for instance, PathletTracer requires over a million rules on ToR switch, tens of thousands of rules on Aggregate switch, and thousands of rules on Core switch. This means that PathletTracer would not scale well for path tracing at L3 layer because packets may follow non-shortest paths. In contrast, CherryPick requires a small number of switch flow rules independent of path length.

3.2 Packet header space

We now evaluate the number of bits in the packet header required to trace packet trajectories (see Figure 6). Recall from §2 that to enable tracing of any n-hop path in a fat-tree topology, CherryPick requires embedding $(n-4)/2+1$ links in the packet header. The number of bits required to uniquely represent each link increases logarithmically with number of ports per switch; for a 48-ary fat-tree topology, each link requires 11 bits worth of space. PathletTracer requires $\log(P)$ bits worth of header space, where P is the number of paths between the source and the destination. We observe that CherryPick requires slightly higher packet header space than PathletTracer (especially for longer paths); however, as discussed earlier, CherryPick trades off slightly higher header space requirement with significantly improved scalability in terms of switch flow rules.

3.3 End host resources

Finally, we compare the performance of CherryPick against PathletTracer in terms of the resource requirements at the end host. Each of the two schemes stores certain entries at the end host to trace the packet trajectory using the information carried in the packet header. Processing the packet header requires relatively simple lookups into the stored entries for both schemes, which is a lightweight operation. We hence only quantify the number of entries required for both schemes.

CherryPick stores, at each destination, the entire set of network links, each annotated with a unique identifier. PathletTracer requires storing a "codebook", where each entry is a code assigned to each of the unique path. For fair comparison, we assume that individual hosts in PathletTracer store an equal-sized subset of the codebook that is only relevant to them.

Figure 7 shows that PathletTracer needs to store non-trivial number of entries if non-shortest paths need to be traced. For instance, PathletTracer requires more than 10^9 entries *at each end host* to trace 6-hop paths, which translates to approximately 12GB worth of entries as each entry is about 12 bytes long. As discussed earlier, this overhead for PathletTracer could be reduced at the cost of tracing at coarser granularities. In contrast, since CherryPick stores only the set of links constituting the network, it requires a small, fixed number of entries (~56K entries per host for a 48-ary fat-tree topology).

4. DISCUSSION

Preliminary evaluation (§3) suggests that CherryPick can enable packet trajectory tracing functionality in SDN-enabled datacenter networks while efficiently utilizing data plane resources. In this section, we discuss a number of research challenges for CherryPick that constitute the ongoing work.

Generalization to other network topologies. In this paper, we focused on the CherryPick design for fat-tree topology. A natural question here is whether it is possible to generalize CherryPick to other datacenter network topologies (*e.g.*, VL2, DCell, BCube, HyperX, etc.). While these topologies differ, we observe that they share a common structure: these topologies can be broken down into bipartite-like graphs. Intuitively, CherryPick may be able to exploit this property to cherry-pick a subset of links in the path. Generalizing CherryPick to these topologies and devising techniques to automatically produce corresponding flow rules constitutes our main ongoing work.

Impact of routing schemes and link failures. The current design of CherryPick is agnostic to the routing protocol used in the network. In particular, the design does *not* assume that the packets are routed along shortest paths (*e.g.*, 6-hop paths in evaluation results from §3). In fact, since each packet carries all the information required to trace its own trajectory, CherryPick works even in case where packets within a single flow traverse along different paths [8, 21]. The design also works when packets encounter link failures on their way to the destination — such packets may traverse along a non-shortest path; CherryPick will still pick the necessary links and be able to reconstruct the end-to-end trajectory.

Impact of packet drops. One of the assumptions made in design of CherryPick is that the packet trajectory is traced when the packet reaches the destination. However, a packet may not reach the destination for a multitude of reasons, including packet drops due to network congestion, routing loops[3], or switch misconfiguration (*e.g.*, race condition [11]). Note that this problem is fundamental to most techniques that reconstruct packet trajectory at the destination [27]. Prior work suggests to partially mitigate this issue by forwarding the packet header of the dropped packet to the controller and/or the designated server for analysis. The challenge for CherryPick here is that due to selectively picking links that represent an end-to-end path, it may be challenging to identify the precise location of packet drop. We are currently investigating this issue.

Impact of controller and switch misconfiguration. At a minimum, the flow rules of CherryPick must be correctly pre-installed at switches and the switches must insert correct link identifiers into packet headers. However, if the SDN controller and switches behave erratically with respect to inserting correct link identifiers into packet headers, it may not be possible for CherryPick to trace the correct packet trajectory. Note that both of these issues are related to the correctness of the SDN control plane. Indeed, there has been significant work recently [2, 4, 15] in ensuring correctness of control plane behavior in SDN-enabled networks. In contrast, CherryPick focuses on debugging the SDN data plane by enabling packet trajectory tracing as the packets traverse through the network data plane.

Packet marking. Similar to other schemes that rely on packet marking, CherryPick faces the challenge of having limited packet header space. CherryPick currently relies on multiple VLAN tagging capability specified in the SDN standard [19] to partially mitigate this issue. Indeed, adding more than two tags is possible in commodity OpenFlow switches (e.g., Pica8 switch), but their ASIC understands two VLAN tags only. In the current implementation of the commodity OpenFlow switches, more than two tags disrupt other rules that match against fields in IP and TCP layers. We believe that this limitation may become less severe as SDN switch hardware becomes more reconfigurable, as proposed in several other research works [3, 12].

[3]In OpenFlow switch, "Decrement IP TTL" action is supported, but not enabled by default. If enabled, routing loops may result in packet drops.

5. RELATED WORK

We now compare and contrast CherryPick against the key related works [1, 11, 13, 14, 16, 17, 26, 27].

PathQuery [17] and PathletTracer [27], similar to CherryPick, enable tracing of packet trajectories by embedding the required information into the packet header. While these approaches are agnostic to the underlying network topology, they require large number of switch flow rules even in moderate size datacenter networks. In contrast, CherryPick aggressively exploits the structure in datacenter network topologies to enable similar functionality while requiring significantly fewer switch flow rules and comparable packet header space.

Several other techniques attempt to minimize data plane resources to trace packet trajectories. In particular, approaches like [7, 22] propose to keep track of trajectories on a per-flow basis by partially marking path information across multiple packets belonging to the same flow. However, these techniques do *not* work for short (mice) flows or when packets in a flow are split over multiple paths. Data plane resources for tracing packet trajectories can also be minimized by injecting test packets [1, 26] and using the paths taken by these packets as a proxy for the path taken by the user traffic. However, short-term network dynamics may result in test packets and user traffic being routed along different paths and these techniques may result in imprecise inferences.

NetSight [11] is an "out-of-band" approach in that it traces packet trajectories by collecting logs at switches along the packet trajectory. In contrast, CherryPick is an "in-band" approach as packets themselves carry path information. CherryPick is also different from approaches like VeriFlow [14], Anteater [16] and Header Space Analysis [13]. In particular, these approaches rely either on flow rules installed at switches [14] or on data plane configuration information [13, 16] to perform data plane debugging (*e.g.*, loops, reachability). In contrast, CherryPick traces trajectories of real traffic directly on data plane with no reliance on such information.

Recent advances in data plane programmability [3, 12] make it easy to enable numerous network debugging functionalities including packet trajectory tracing. CherryPick can efficiently implement the path tracing functionality on top of those flexible platforms. Thus, our approach is complementary to them.

6. CONCLUSION

We have presented CherryPick, a simple yet scalable technique for tracing packet trajectories in SDN-enabled datacenter networks. The core idea in CherryPick is that structure in datacenter network topologies enable reconstructing end-to-end paths using a few essential links. To that end, CherryPick "cherry-picks" a subset of links along the packet trajectory and embeds them into the packet header. We applied CherryPick to a fattree topology in this paper and showed that it requires minimal switch flow rules to enable tracing packet trajectories, while requiring packet header space close to state-of-the-art techniques.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported by EPSRC grant EP/L02277X/1.

7. REFERENCES

[1] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior. In *ACM HotSDN*, 2014.

[2] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *ACM PLDI*, 2014.

[3] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[4] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *USENIX NSDI*, 2012.

[5] M. Chiesa, I. Nikolaevskiy, A. Panda, A. Gurtov, M. Schapira, and S. Shenker. Exploring the Limits of Static Failover Routing. *CoRR*, abs/1409.0034, 2014.

[6] R. Cole, K. Ost, and S. Schirra. Edge-Coloring Bipartite Multigraphs in O(E log D) Time. *Combinatorica*, 21(1), 2001.

[7] D. Dean, M. Franklin, and A. Stubblefield. An algebraic approach to IP traceback. *ACM Transactions on Information and System Security*, 5(2):119–137, 2002.

[8] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the Impact of Packet Spraying in Data Center Networks. In *IEEE INFOCOM*, 2013.

[9] N. G. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM ToN*, 9(3), 2001.

[10] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *ACM SIGCOMM*, 2011.

[11] 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 *USENIX NSDI*, 2014.

[12] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *ACM SIGCOMM*, 2014.

[13] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*, 2012.

[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX NSDI*, 2013.

[15] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for Openflow Switch Interoperability Testing. In *ACM CoNEXT*, 2012.

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

[17] S. Narayana, J. Rexford, and D. Walker. Compiling Path Queries in Software-defined Networks. In *ACM HotSDN*, 2014.

[18] S. Nelakuditi, S. Lee, Y. Yu, Z.-L. Zhang, and C.-N. Chuah. Fast local rerouting for handling transient link failures. In *IEEE/ACM ToN*, 2007.

[19] Open Networking Foundation. OpenFlow Switch Specification Version 1.4.0. http://tinyurl.com/kh6ef6s, 2013.

[20] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In *USENIX NSDI*, 2012.

[21] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM*, 2011.

[22] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *ACM SIGCOMM*, 2000.

[23] 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 *IEEE ICDCS*, 2013.

[24] B. Yang, J. Liu, S. Shenker, J. Li, and K. Zheng. Keep forwarding: Towards k-link failure resilient routing. In *IEEE INFOCOM*, 2014.

[25] K. Zarifis, R. Miao, M. Calder, E. Katz-Bassett, M. Yu, and J. Padhye. DIBS: Just-in-time Congestion Mitigation for Data Centers. In *ACM EuroSys*, 2014.

[26] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. *IEEE/ACM ToN*, 22(2):554–566, 2014.

[27] H. Zhang, C. Lumezanu, J. Rhee, N. Arora, Q. Xu, and G. Jiang. Enabling Layer 2 Pathlet Tracing Through Context Encoding in Software-defined Networking. In *ACM HotSDN*, 2014.

[28] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg. Fast Accurate Computation of Large-scale IP Traffic Matrices from Link Loads. In *ACM SIGMETRICS*, 2003.