Canary: Detecting and Localizing Faults in Data Center Networks with Partial Traffic Monitoring

Wei Chen, Ye Tian, *Member, IEEE,* Zhaohui Wang, Cenman Wang, and Xinming Zhang, *Senior Member, IEEE*

***Abstract*—Silent packet drops and packet corruptions, which are caused by faulty network elements and hurt performances of cloud applications, are common in data centers but hard to detect and localize. Existing solutions based on active probes introduce additional probe traffic and are constrained by probe rate, while solutions based on passive traffic monitoring measure the entire network traffic, and are generally unable to pinpoint the locations where packet losses happen. In this paper, we present *Canary*, a system for detecting and localizing network faults with partial traffic monitoring. Canary employs a lightweight and adaptive mechanism to detect packet losses by monitoring a small set of large-sized network flows, and it ensures that on each network path, a sufficient number of packets are monitored by upstream and downstream switches. In addition, Canary encodes information of the path that a packet travels along within its header, and by leveraging path information of the lost packets, Canary is capable to localize network faults with high accuracy. We theoretically prove the effectiveness of our proposed method, and prototype Canary with P4 on commodity hardware programmable switches. Results from extensive experiments driven by real-world traffic show that Canary is lightweight regarding measurement overhead, robust under traffic dynamics, and is accurate in detecting and localizing faulty network links. In particular, comparing with the state-of-the-art solutions, Canary reduces the memory overhead by over 97% under 10^{-2} link loss rate, and increases the F1-score in localizing the faulty links by over 20% on a $k = 8$ fat-tree data center network.**

***Index Terms*—Loss detection, fault localization, partial traffic monitoring, data center network, programmable switch**

I. Introduction

CLOUD providers have built large data centers for a wide range of applications including giant AI models, massive online gaming and social networking, high-volume e-commerce transactions, etc. However, packet losses hurt performances of these applications. For example, even with a 10^{-4} loss rate, the throughput of a RDMA flow would drop over 75% [1], and a loss rate of 10^{-4} can cause a TCP CUBIC flow to drop its throughput by 50% [2].

Congestions and network faults are two major sources of packet losses. To avoid packet drops due to congestions, several congestion control algorithms have been developed in recent years [3]–[5]. Nevertheless, many packet losses are caused by faulty network elements. For example, gray failures [6] such as software bugs or malfunctional hardware cause switches to drop packet probabilistically without reporting (i.e., silent packet drops) [1], and link-related issues such as connector contamination, damaged or bent fiber, decaying transmitters, and bad transceivers cause packet corruptions [7]. Silent packet drops and packet corruptions are common in data centers but time-consuming to localize, as the upstream switch usually cannot detect the loss, while the downstream switch can not recover any information from lost or corrupted packets. For example, Microsoft reports that 12.67% links with loss rates above 0.001 are caused by corruptions [7]; and according to Alibaba, 18% network performance anomalies within their data centers are caused by silent packet drops and packet corruptions, which constitute 50% of the network faults that take over 180 minutes to localize [8].

One approach for detecting and localizing network faults are based on active probe [9]–[15]. In such a system, a number of end-hosts are selected for sending and receiving probe packets. To ensure that a probe packet travels along a pre-planned path, IP-in-IP source routing is adopted [12], [14]. Based on the probe results, an inference algorithm is generally employed to localize the network faults [9], [12]–[14], [16]. One drawback of the active probe based approaches is the additional traffic imposed on the network, and to reduce this overhead, end-hosts are only allowed to send probes at a low rate. For instance, in NetBouncer [14], an end-host is configured to send only 100 packets per 5 minutes. Clearly, with fewer probe packets, it is more difficult to detect packet losses that happen at small probabilities.

Another representative approach is based on passive traffic monitoring [17]–[19]. In such a system, flow or packet-level information is collected by switches, and packet losses are detected by comparing the measurement data collected at different positions. To reduce the memory and bandwidth overhead, sketch data structures such as Invertible Bloom filter Lookup Table (IBLT) [17] and FermatSketch [19] are employed. Although by monitoring the entire network traffic, all the lost packets can be detected, however, unless deployed hop-by-hop, these methods are unable to pinpoint the locations where packet losses happen.

In this paper, we present *Canary,* a system for detecting and localizing network faults with partial traffic monitoring. Comparing with the solutions based on active probe and passive traffic monitoring, Canary has the following merits.

First, Canary is *effective*. Unlike the traffic monitoring based solutions that are unable to localize network faults. Canary encodes information of the path that a packet travels along within its header, therefore is capable to localize the positions where packet losses happen.

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61672486 and Grant 62072425. *(Corresponding author: Ye Tian.)*

The authors are with Anhui Key Laboratory of High Performance Computing, School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anui, 230026, China (e-mail: szcw33@mail.ustc.edu.cn; yetian@ustc.edu.cn; wang_zh@mail.ustc.edu.cn; wangcenman@mail.ustc.edu.cn; xinming@ustc.edu.cn).

Second, Canary is *lightweight*. Unlike the active probe based approaches, Canary is based on passive traffic monitoring, thus does not introduce additional traffic. In addition, rather than monitoring the entire network traffic, Canary is based on partial traffic monitoring, and traces only a small number of selected network flows, as long as a sufficient number of packets are counted on each path segment for loss detecting.

Third, Canary is *adaptive*. Canary guarantees to count enough packets on each path segment, and when packets monitored on a path segment are insufficient due to traffic dynamics, Canary adaptively selects more large-size network flows on that path segment for traffic monitoring.

Fourth, Canary is *accurate*. Comparing with the solutions that send a limited number of probe packets, Canary uses much more packets collected from on-going network flows, thus reduces the errors in detecting packet losses caused by imperfect measurement. In addition, by improving a state-of-the-art inference algorithm [14], Canary can localize the faulty links that used to be overlooked by the original algorithm.

By designing and implementing the Canary system, we make the following contributions in this paper.

- *An effective way to encode path information in packet header*. We propose a novel method based on Bloom filter to encode the path segment that a packet travels along in its header, and exploit characteristic of Clos-structured data center network to ensure that each path segment can be uniquely decoded at the controller. We prove in theory that path segments are decodable.
- *A lightweight and adaptive mechanism to detect packet losses*. We design a compact pipeline that allows upstream and downstream switches to count packets and collect path information for a number of selected network flows, and detect packet losses on each path segment by comparing the measurement data. In addition, we incorporate a heavy-hitter detecting algorithm [20] to select large-sized network flows, and make sure that a sufficient number of packets are counted on each path segment under traffic dynamics.
- *An improved methodology for localizing faulty network links*. We improve the state-of-the-art algorithm in NetBouncer [14] by proposing a methodology that iteratively runs the algorithm in multiple stages. Our proposed method leverages the path information of the lost packets, preserves the high accuracy of NetBouncer's algorithm in localizing the faulty links with substantial loss rates, and overcomes the limitation of the algorithm that overlooks the links with low loss rates.
- *A prototype based on commodity hardware programmable switch*. We have implemented a Canary system prototype. In particular, we compose the switch pipeline with the $P4_{16}$ programming language [21], and realize it on the Intel Tofino based hardware programmable switch [22]. We make the prototype publicly available[1].

We have evaluated Canary on both simulated networks and a real-world testbed. With packet-level simulation, we show that Canary is inexpensive regarding the memery overhead, and provides flexibility in allocating memory resources. For example, given a link loss rate of 10^{-2}, Canary reduces the in-switch memory usage by 97.7%, comparing with FermatSketch [19], and its memory usage can be further reduced as the loss rate increases. We also show that with the assistance of the heavy-hitter detection, Canary is robust against traffic dynamics, and can reduce the controller overhead by counting enough packets without monitoring too many network flows. The experiment on a large-scale data center network indicates that Canary is capable to infer faulty network links with few false positive and false negative errors, and outperforms the state-of-the-art algorithms. For example, on a $k = 8$ fat-tree network, Canary increases the F1 score in localizing faulty network links by 36.7% and 22.2%, comparing with NetBouncer [14] and Flock [16] respectively. Finally, evaluation on the real-world testbed shows that Canary is practical for real-world deployment without impacting a switch's packet forwarding speed.

The remainder part of this paper is organized as follows. Sec. II discusses the related work; We present the system design in Sec. III; Sec. IV analyzes the experiment results, and we conclude this paper in Sec. V.

II. Related Work

A. Approaches based on Active Probe

Modern data centers nowadays deploy "always-on" active probe services such as Pingmesh [10], NetNORAD [11], and RD-Probe [15] in their networks. In such a service, active probe packets are sent from end-hosts for detecting and localizing network faults. One critical issue of the active probe based methods is the uncertainty of the paths traveled by probe packets, as multi-path routing (e.g., ECMP [23]) is widely adopted in data centers. To overcome this problem, 007 [13] employs a traceroute-like scheme to discover the paths on which packet drops happen. deTector [12] and NetBouncer [14] leverage the fact that only one path exists from a server to a top-layer switch in a Clos-structured network, and control the path that probe packets travel along with source routing based on IP-in-IP encapsulation.

To detect and localize network faults from the probe results, tomography-based techniques [24]–[27] for ISP networks are consulted. However, as data centers require continuous and near real-time diagnosis, efforts are made to speed up the inference. In particular, 007 [13] ranks links based on their likelihoods to drop packets; [9] solves a system of equations for obtaining links' packet loss rates; deTector [12] localizes faulty network links with a score-based greedy algorithm; NetBouncer [14] and Flock [16] model the problem as a minimization optimization, and apply coordinate descend (CD) and maximum likelihood estimation (MLE) algorithms respectively to infer faulty network elements.

Different from these works, Canary does not send active probes, but encodes the path segment that a packet travels along in its header, thus provides location information without introducing additional probe traffic. In addition, Canary improves the algorithm in NetBouncer [14] by developing a

[1]The prototype source code is available at https://github.com/HPCC724/Canary_FaultDiag

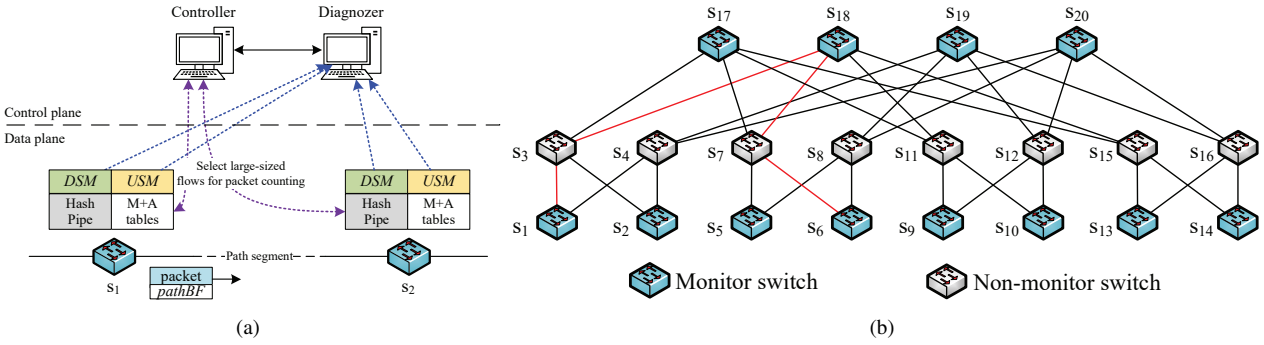


Fig. 1. (a) An overview of the Canary system. (b) A $k = 4$ fat-tree data center network, where top and bottom-layer switches are used as monitor switches.

multi-stage methodology, thus is capable to localize faulty network links with a wide rang of packet loss rates.

B. Approaches based on Passive Traffic Monitoring

For decades, NetFlow [28] has been used for collecting traffic statistics on routers, and to reduce the processing overhead, NetFlow samples packets at certain rate (e.g., 1:100). However, by sampling packets, NetFlow is unable to detect packet losses, especially when packets are silently dropped or corrupted by faulty network elements.

One feasible solution for detecting packet losses is to monitor the entire network traffic, and to reduce the memory overhead, sketch-based methods are adopted [29]–[31]. In particular, LossRadar [18] employs a Bloom filter to capture individual packet losses; FlowRadar [17] tracks packets of all the network flows with an Invertible Bloom filter Lookup Table (IBLT), and detects packet losses by comparing a pair of upstream and downstream IBLTs; ChameleMon [19] replaces IBLT with a novel sketch data structure named FermatSketch for overcoming IBLT's inherent limitation.

Canary differs from these works in two aspects. First, unlike the solutions that can provide location information only when the sketch data structures are deployed hop-by-hop, Canary is capable to provide location information even though only a subset of the switches monitor the network traffic. Second, unlike the works that monitor the entire network traffic, Canary traces only a small number of selected network flows, therefore substantially reduces the measurement overhead.

III. The Canary System

In this section, we present the Canary system. We first give an overview of the system design, then we describe the components that constitute the system in details.

A. Overview

Fig. 1(a) presents an overview of the Canary system, which works on a data center network with a Clos-structured topology like fat-tree [32] as demonstrated in Fig. 1(b). In Canary, each packet carries a Bloom filter [33], named *pathBF*, for encoding the IDs of the switches it traverses. We assume that each switch in the network has a unique ID. For example, a switch can use the MAC address that is numerically largest (or smallest) among its interfaces as the unique ID.

A subset of the switches in a data center network are selected to host *upstream and downstream meters*. A meter is an in-switch data structure for monitoring traffics of network flows, and in particular, the downstream meter, denoted as *DSM*, counts packets of a set of selected flows that the switch receives, and the upstream meter *USM* counts the selected flows' packets that are about to send out by the switch. We refer to a meter-hosting switch as a *monitor switch*, and illustrate its pipeline in Fig. 2. In particular, to have a full coverage of all the inter-switch links, we require all the bottom-layer switches in the data center network to be monitor switches, while the other switches could also serve as monitor switches. For example, in Fig. 1(b), the switches at the top and bottom layers are used as monitor switches. Note that although we focus on inter-switch links in this paper, however, Canary can be easily extended to cover host-switch links by placing meters at end-hosts.

Given a network path, we refer to a monitor switch's *upstream switch* as the closest monitor switch preceding it on the path, and this switch is referred to as its upstream switch's *downstream switch*. A pair of upstream and downstream switches cover a *path segment*. For example, the path $s_1 \rightarrow s_3 \rightarrow s_{18} \rightarrow s_7 \rightarrow s_6$ in Fig. 1(b) contains two segments: on the path segment of $s_1 \rightarrow s_3 \rightarrow s_{18}$, s_1 is s_{18}'s upstream switch, and s_{18} is s_1's downstream switch; and for the segment of $s_{18} \rightarrow s_7 \rightarrow s_6$, s_{18} and s_6 are each other's upstream and downstream switches respectively. Note that path segments have directions.

Unlike previous works that aim to measure the entire network traffic [17]–[19], Canary monitors only a number of selected network flows. As shown in Fig. 1(a), we employ the data plane algorithm of *HashPipe* [20] to select large-sized network flows for switches to monitor. By leveraging the pathBFs carried by the flows' packets, we ensure that a sufficient number of packets are counted on each path segment.

To detect packet losses, at the *diagnozer server*, we compare a monitor switch's DSM against the USMs in all its upstream switches. With the assistance of the location information encoded in the pathBFs, we localize the faulty links that have packet losses with a multi-stage inference methodology.

In summary, as shown in Fig. 1(a), the Canary system is composed of both data plane and control plane components. On data plane, the switches host USM and DSM meters to count packets, and encode pathBFs to log information of

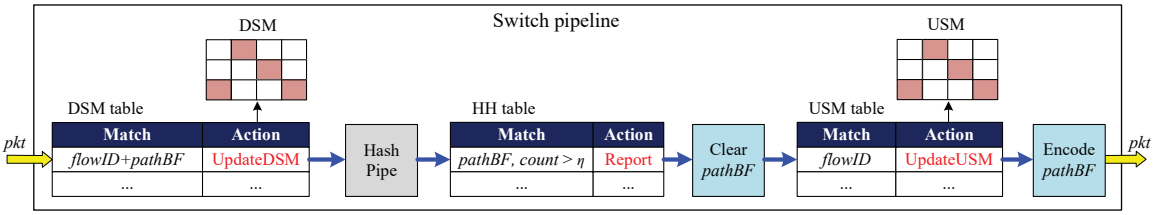


Fig. 2. Pipeline within a monitor switch.

path segments on packets; on control plane, the diagnozer collects meters to detect packet losses and localize network faults; in addition, the data plane HashPipe algorithm assists the controller to select large-sized network flows for packet counting on demand.

B. Encoding and Decoding pathBF

In this subsection, we describe the Bloom filter based method for encoding and decoding the path segment that a packet travels along. Bloom filter [33] has a history of being applied for packet traceback in both wide-area networks [34]–[36] and under the context of software-define networking [37], [38]. However, unlike the previous works that have zero knowledge on network topology, Canary exploits the characteristic of Clos-structured data center network, thus is simpler and more effective.

1) Encoding: In Canary, each packet carries a b-bit Bloom filter [33], namely *pathBF*, within its packet header, for carrying information of the path segment that it travels along. More specifically, to encode a switch's ID to pathBF, the switch s employs m pairwise independent hash functions $h_i(\cdot) : \mathbf{U} \rightarrow \{1, \cdots, b\}$, $i = 1, \cdots, m$, where $\mathbf{U}$ is the switch ID universe, to compute $h_i(s.ID)$, and sets the bit $pkt.pathBF[h_i(s.ID)]$ as 1, for $i = 1, \cdots, m$.

Different switches process packets differently. A non-monitor switch simply encodes its ID to the pathBF of any traversing packet. But for a monitor switch, as shown in Fig. 2, after updating the DSM, the switch clears all the bits in the packet's pathBF, and encodes its own ID before sending the packet out. For example, consider the path $s_1 \rightarrow s_3 \rightarrow s_{18} \rightarrow s_7 \rightarrow s_6$ in Fig. 1(b), each packet received by the top-layer monitor switch s_{18} has 2 IDs of s_1 and s_3 encoded in its pathBF, and after s_{18} clears the pathBF, the packet travels along the second segment of the path, and has the IDs of s_{18} and s_7 encoded when arriving to s_6.

2) Decoding: As we will describe in Sec. III-C, when monitoring a network flow, both the upstream and downstream switches record the flow's ID as well as its packet count in the meters, and the downstream switch also records the pathBF carried by the flow's packets in its DSM. When the switches report the meters to the diagnozer server, the server decodes the path segment traversed by the flow from the pathBF.

The basic idea in decoding a path segment is simple: Suppose a network flow f is reported by an upstream monitor switch s and a downstream switch s', and s' also reports a pathBF associated with f. If there is only one path segment from s to s' on the network, then it must be the segment traversed by f. But if there exist multiple path segments, we can use the pathBF to find out which path segment f travels along. For example, consider the network in Fig. 1(b) and suppose that only the bottom-layer switches are used as monitor switches, then there are up to 4 path segments from an upstream bottom-layer switch s to a downstream bottom-layer switch s' in a different pod, and ideally, we expect that only one path segment passes the Bloom filter test.

However, since Bloom filter has inherent false positive errors [39], there are chances that an erroneous path segment from s to s' also passes the Bloom filter test, and in this case, we fail to decode the pathBF. Fortunately, on a Clos-structured network, an erroneous path segment from s to s' overlaps with the ground-truth segment only at the first and last hops, but differs in all the intermediate hops [14], [40]. From example, in Fig. 1(b), all the four path segments from s_1 to s_6 have 5 hops, and they differs in the 2^{nd}, 3^{rd}, and 4^{th} hops. Based on such a characteristic of Clos-structured network, we have the following result.

Theorem 1. *Given a Clos-structured network, suppose a path segment from an upstream switch to a downstream switch is encoded in a pathBF and both monitor switches are at the bottom layer, then the probability that we fail to decode a unique path segment from the pathBF is*

$$P_{fail} \approx \left(1 - e^{-\frac{(h-1)m}{b}}\right)^{(h-2)m} \tag{1}$$

where h is the length of the bottom-to-bottom path segment, m is the number of the hash functions, and b is the Bloom filter size.

We provide the proof of Theorem 1 in Appendix A-A. Suppose that pathBF is carried in a 32-bit field such as the VLAN tag within a packet header (i.e., $b = 32$), $m = 3$ hash functions are applied, and a bottom-to-bottom path segment has $h = 5$ hops, then according to (1), the decoding failure probability P_{fail} is as small as 2.85×10^{-5}.

Theorem 1 indicates that there exists a trivial but non-zero failure probability that a pathBF is undecodable. Fortunately, since switch ID is static, we can enumerate all the path segments in an offline way in advance, and examine whether there exist path segments with their pathBFs undecodable. If such segments exist, we can alter the IDs of the switches that cause the decoding failures (e.g., by using the second largest/smallest MAC address as the switch ID), and make sure that each path segment has a decodable pathBF.

C. Monitoring Network Flows with Meters

As illustrated in Fig. 2, a monitor switch hosts both the DSM and the USM meters in its pipeline, and we place match+action tables to monitor a small number of selected network flows with them. More specifically, for each network flow selected to be monitored by the DSM, it has a rule in the *DSM table* that matches packets with flow ID and pathBF, and updates the DSM with the matched packets. Similarly, each flow monitored by the USM has a rule in the *USM table*, which matches packets with flow ID and updates the USM.

Both USM and DSM are composed of d rows, and each row has w buckets. A DSM bucket has the following fields:

- $dirtyFlag$: a 1-bit flag indicating usability of the data in this bucket;
- $flowID$: ID of the flow (e.g., the 5-tuple) that is recorded by this bucket;
- $pathBF$: the b-bit Bloom filter encoding the path segment that the flow travels along;
- $count$: number of the flow's packets received by the DSM-hosting switch.

A USM bucket has three fields of $dirtyFlag$, $flowID$, and $count$ that are of the same meanings as in DSM.

For counting a packet pkt in DSM or USM, a monitor switch applies d hash functions, i.e., $g_i(\cdot) : \mathbf{F} \rightarrow \{1, \cdots, w\}$, $i = 1, \cdots, d$, where $\mathbf{F}$ is the flow ID universe, to compute the positions $g_i(pkt.flowID)$, and updates the corresponding buckets. More specifically, when a monitor switch receives a packet that matches a rule in the DSM table, the `UpdateDSM` action is executed as in Algorithm 1, and when it encounters a packet that matches a rule in the USM table, the `UpdateUSM` action is executed as in Algorithm 2.

Algorithm 1: `UpdateDSM`

```
Input : A packet pkt
for i = 1 ··· d do
    j = g_i(pkt.flowID);
    if DSM[i][j].dirtyFlag ≠ 1 then
        if DSM[i][j] is empty then
            DSM[i][j].flowID = pkt.flowID;
            DSM[i][j].count = 1;
            DSM[i][j].pathBF = pkt.pathBF;
        else
            if (DSM[i][j].flowID ≠ pkt.flowID) or
              (DSM[i][j].pathBF ≠ pkt.pathBF) then
                DSM[i][j].dirtyFlag = 1;
            DSM[i][j].count++;
```

From Algorithm 1 and Algorithm 2, we can see that a DSM/USM bucket has its $dirtyFlag$ set as '1' when two or more flows collide at this bucket. In addition, a DSM bucket could also have its $dirtyFlag$ set as '1' if the flow changes its path segment. We refer to a bucket with $dirtyFlag = 0$ as a *clean bucket*, and only use data from clean buckets to detect packet losses and localize network faults.

As we will see in Sec. III-G, the probability that a flow does not have a clean bucket is small, and to further increase the chance that clean buckets can be found on a pair of upstream and downstream switches covering a path segment, we use a same set of hash functions across the network, so that the flows colliding in the USM will also collide in the DSM.

Algorithm 2: `UpdateUSM`

```
Input : A packet pkt
for i = 1 ··· d do
    j = g_i(pkt.flowID);
    if USM[i][j].dirtyFlag ≠ 1 then
        if USM[i][j] is empty then
            USM[i][j].flowID = pkt.flowID;
            USM[i][j].count = 1;
        else
            if USM[i][j].flowID ≠ pkt.flowID then
                USM[i][j].dirtyFlag = 1;
            USM[i][j].count++;
```

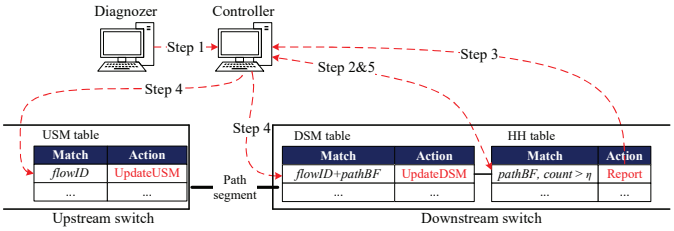


Fig. 3. Procedure to select large-sized network flows.

D. Selecting Large-Sized Network Flows

In Canary, time is divided into epochs. After each epoch, the diagnozer server collects DSMs and USMs from the monitor switches, detects packet losses by comparing the DSMs against the upstream USMs, and localizes the faulty links with the path segments encoded in the pathBFs. Intuitively, to detect packet losses on a path segment that has a low loss rate, a large number of packets should be counted. In Canary, when the diagnozer finds that on a path segment, the switches do not count enough packets, the network controller will instruct the upstream and downstream switches to monitor more network flows on that segment in the next epoch.

Since network flows in data center networks have their sizes highly skewed [41], to reduce the measurement overhead, it is in our favor to monitor large-sized network flows. To this end, we employ HashPipe [20], a data plane algorithm for detecting heavy hitters, within the pipeline. HashPipe is composed of multiple stages of hash tables that are sequentially passed by packets. When traversing the stages, each packet seeks to have the ID and packet count of its flow recorded in some hash table entry, and evicts a lighter flow that has a smaller packet count if necessary. As a consequence, using a limited memory size, HashPipe maintains only the network flows with the largest packet counts (i.e., heavy hitters). Moreover, after passing the stages, a packet can carry a state in its metadata indicating whether its flow is a heavy hitter, as well as the flow's packet count retained by the HashPipe.

As shown in Fig. 2, in a monitor switch's pipeline, we place a HashPipe after the DSM stage to select large-sized

network flows. However, for a network flow that is already monitored by the DSM, we mark its packets in metadata, so that the packets will not be counted by the HashPipe. Fig. 3 demonstrates the steps for selecting network flows on a path segment. When the diagnozer finds that the packets being counted on a path segment are fewer than a threshold α, or the flows being monitored on the segment are fewer than a threshold β, it instructs the controller to select δ additional large-sized network flows on that path segment (Step 1).

We use a match+action table named *HH table* after the HashPipe stage for selecting large-sized network flows. More specifically, to select flows on a specific path segment, the controller first inserts a rule in the HH table, which matches a packet that 1) carries a pathBF encoding the path segment; 2) has not been monitored by the DSM; and 3) belongs to a heavy-hitter flow with its packet count in the HashPipe exceeding a threshold η (Step 2). When such a packet pkt is matched, it is reported to the controller (Step 3).

On receiving pkt, the controller installs a rule matching $pkt.flowID$ and $pkt.pathBF$ to the DSM table at the downstream switch of the path segment, and inserts a rule matching $pkt.flowID$ to the USM table at the segment's upstream switch (Step 4). The controller removes the rule in the HH table after δ network flows have been selected (Step 5).

E. Detecting Packet Losses

After each epoch, the diagnozer server collects USMs and DSMs from the monitor switches, and detects packet losses by comparing the measurement data in the meters. We divide the problem of detecting and localizing packet losses in the entire network into disjoint *sub-problems*, where each sub-problem covers a part of the network without overlapping with the other sub-problems, and all the sub-problems constitute the entire network. For example, for the fat-tree network as in Fig. 1(b), each pod as well as the links between the pod's upper-layer switches and all the top-layer switches form a sub-problem, and the network consists of 4 sub-problems.

Within each sub-problem, the diagnozer server compares each DSM against its upstream USMs, and records the result in a hash table named *minus meter*, denoted as *MM*. Each MM table entry has the flowing fields.

- $flowID$: ID of the flow being monitored;
- $pathBF$: Bloom filter encoding the path segment that the flow travels along;
- $sent$: number of the flow's packets being sent out along the path segment during the epoch;
- $lost$: number of the flow's packets lost on the path segment during the epoch.

In each DSM, we first find all the flows that have at least one clean bucket. For such a flow, we hash its ID with a hash function $H(\cdot)$ to a table entry in the MM, write the flow's $flowID$ and $pathBF$ in its clean bucket to the table entry, and set $sent$ as the bucket's $count$ value. Then for this DSM, we find all its upstream USMs. In each USM, we look up the clean buckets of each flow recorded in the MM, and update the $lost$ and $sent$ fields. Finally, we remove the entries in the MM that do not have valid $lost$ values, as none of these flows has a clean USM bucket. Algorithm 3 presents the algorithm.

Algorithm 3: Algorithm for obtaining MM.

Input : A DSM and its upstream USMs

for each flow f in DSM **do**
  if there exists $DSM[i][g_i(f)].dirtyFlag == 0$ **then**
    $MM[H(f)].flowID = DSM[i][j].flowID$;
    $MM[H(f)].pathBF = DSM[i][j].pathBF$;
    $MM[H(f)].sent = DSM[i][j].count$;
for each flow f in MM **do**
  if there exists $USM[i][g_i(f)].dirtyFlag == 0$ and $USM[i][g_i(f)].flowID == f.ID$ **then**
    $MM[H(f)].lost = USM[i][g_i(f)].count - MM[H(f)].sent$;
    $MM[H(f)].sent = USM[i][g_i(f)].count$;
Remove MM entries that do not have valid $lost$ values;

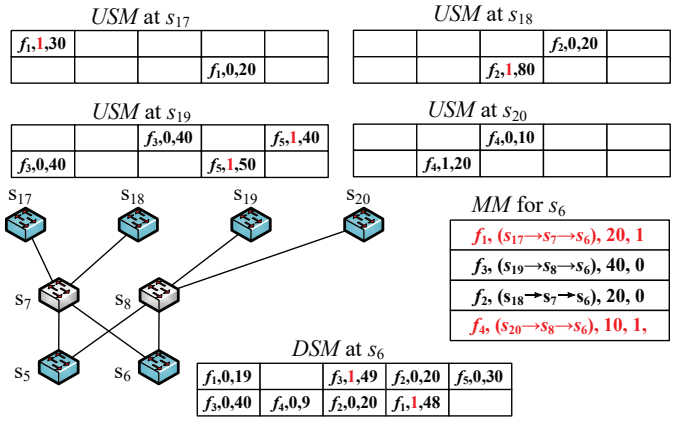


Fig. 4. An example for detecting packet losses. Each DSM/USM bucket is presented as a 3-tuple of ($flowID$, $dirtyFlag$, $count$), and each MM entry is presented as a 4-tuple of ($flowID$, $pathBF$, $sent$, $lost$).

As an example, Fig. 4 presents a sub-problem of the fat-tree network in Fig. 1(b). In Fig. 4, the DSM at s_6 is compared against the USMs at s_{17}, s_{18}, s_{19}, and s_{20}. We first find that the flows f_1, f_2, f_3, f_4, and f_5 can be retrieved from the DSM at s_6, as each flow has at least one clean bucket. We then query the flows in the USMs at s_{17}, s_{18}, s_{19}, and s_{20}, and find that for the flows f_1, f_2, f_3, and f_4, we can find at least one clean bucket, and calculate their lost packets, but for f_5, we can not obtain its lost packets due to hash collisions, therefore remove it from the MM. As a result, we find two flows, f_1 and f_4 ending at s_6 that have packet losses, in addition, f_1 traverses the path segment $s_{17} \rightarrow s_7 \rightarrow s_6$ and f_4 travels along the segment $s_{20} \rightarrow s_8 \rightarrow s_6$. Note that since we only use measurement data from clean buckets, the loss detection is error-free.

F. Localizing Faulty Network Links

For each path segment p that appears in an MM, we compute its *success probability* y_p as

$$y_p = \frac{\sum_{p \in \mathbf{F}_p} (f.sent - f.lost)}{\sum_{p \in \mathbf{F}_p} f.sent} \tag{2}$$

where $\mathbf{F}_p$ is the set of the flows on the segment p in the MM. For example, in Fig. 4, the segment $s_{17} \rightarrow s_7 \rightarrow s_6$ has a success probability of 0.95, and the segment $s_{20} \rightarrow s_8 \rightarrow s_6$ has a success probability of 0.90. To filter out noises, in practice, MMs from n consecutive epochs are aggregated, and we only compute success probabilities for the path segments that have packet losses in no less than n' out of the n epochs.

Given success probabilities of the path segments, our objective is to infer each link's success probability, which is defined as the ratio between the packets successfully transported through the link and the packets transmitted on the link.

Our initial approach is to apply the method proposed in [14], which localizes faulty network links by solving the following optimization problem.

$$\begin{aligned} \text{minimize} \quad & \sum_p \left(y_p - \prod_{i \in p} x_i\right)^2 + \lambda \sum_i x_i (1 - x_i) \\ \text{s. t.} \quad & 0 \leq x_i \leq 1 \end{aligned} \tag{3}$$

where x_i is link i's success probability, and λ is a constant with a default value of 1.

The first term of the objective function in (3) is the square error, which aims to contribute the observed path segment success probabilities to the success probabilities of the links on the segments, and the second term is a regularization term, which seeks to push x_i to the values either close to 0 or 1 for avoiding false positive errors caused by imperfect measurement.

A coordinate descent algorithm, which we refer to as the *CD algorithm*, is proposed in [14] to solve the problem. Combined with active probes, the algorithm produces relatively fewer false positive and false negative errors comparing with the other active probe based methods [9], [12], [26].

However, the CD algorithm aims to detect faulty links with substantial loss rates. For example, in [14], the algorithm is applied to detect faulty links with loss rates between 0.02 and 1.0. When a link has a very low loss rate, e.g., 10^{-3}, the CD algorithm tends to ignore it. This is because when solving the optimization problem in (3), the second term of the objective function forces the algorithm to consider the links with low loss rates as false positives and assigns a success probability of 1 to them.

Motivated by the above observation, in this paper, we propose to run the CD algorithm iteratively in multiple stages, and in each stage, we focus only on the path segments with their loss rates falling in a specific range. Moreover, since the CD algorithm tends to ignore very low loss rates, when a path segment has a very high success rate such as 0.999, we map it to a moderate-high value (e.g., 0.8), and use the mapped success rates of the path segments to drive the CD algorithm.

Our proposed method works as the following.

- **Step 1**: Divide the path segments with non-zero loss rates in groups, where each group covers a range of loss rates. For example, suppose we have detected a number of path segments with their loss rates between 0.001 and 1, we can divide them into 5 groups with the ranges as $[0.2, 1]$, $[0.1, 0.2]$, $[0.05, 0.1]$, $[0.01, 0.05]$, and $[0.001, 0.01]$ respectively.

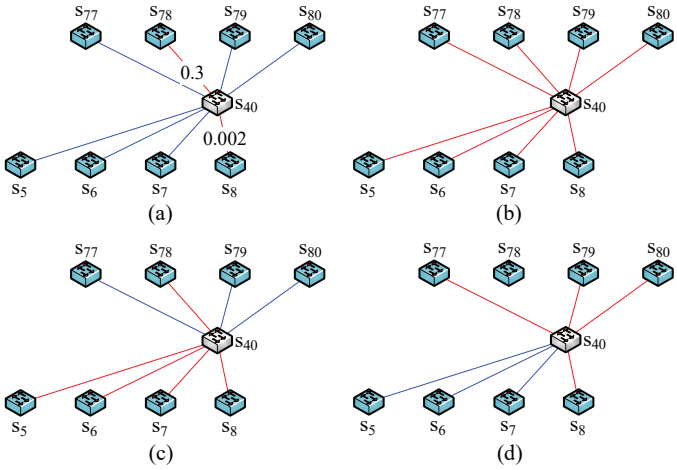


Fig. 5. An example demonstrating the multi-stage fault localization.

- **Step 2**: Sort the groups in a descending order according to loss rate, and for each group in the order:
  - **Step 2-1**: Temporarily map the success probability of a path segment in the current group to a value in $[0, 0.8]$. For example, one may use a logarithm-based function
  $$y'_j = e^{\ln(0.8) \times \frac{\ln(y_j - a)}{\ln(b - a)}} \tag{4}$$
  to map a segment's success probability y_j in the current group's range $[a, b]$ to y'_j in $[0, 0.8]$. Here we choose 0.8 as the right edge of the range for mapping a very high success rate to a moderate-high value, as previously explained.
  - **Step 2-2**: Set y'_j of the path segments in the groups of higher loss rates as 0.
  - **Step 2-3**: Set y'_j of the path segments in the groups of lower loss rates as 1.
  - **Step 2-4**: Apply the CD algorithm (w. $\lambda = 1$) to solve the problem
  $$\begin{aligned} \text{minimize} \quad & \sum_p \left(y'_p - \prod_{i \in p} x_i\right)^2 + \lambda \sum_i x_i (1 - x_i) \\ \text{s. t.} \quad & 0 \leq x_i \leq 1 \end{aligned} \tag{5}$$
  label each link detected by the algorithm that has a success probability lower than 1 as "bad", and remove it from the sub-problem.
- **Step 3**: Repeat **Step 2-1** to **Step 2-4** for the next path segment group in the order.

Fig. 5 demonstrates an example, which is taken from a sub-problem in our experiment in Sec. IV-D. Fig. 5(a) shows the ground truth, in which the link $s_{78} \rightarrow s_{40}$ has a loss rate of 0.3 and the link $s_{40} \rightarrow s_8$ has a loss rate of 0.002. Fig. 5(b) presents the seven path segments that are detected to have packet losses. In particular, by executing **Step 1**, the path segments of $s_{78} \rightarrow s_{40} \rightarrow s_5$, $s_{78} \rightarrow s_{40} \rightarrow s_6$, $s_{78} \rightarrow s_{40} \rightarrow s_7$, and $s_{78} \rightarrow s_{40} \rightarrow s_8$, which have loss rates around 0.3, are placed in a group with the range of $[0.2, 1]$; and the other three segments of $s_{77} \rightarrow s_{40} \rightarrow s_8$, $s_{79} \rightarrow s_{40} \rightarrow s_8$, and $s_{80} \rightarrow s_{40} \rightarrow s_8$, which have loss rates around 0.002, are placed in the range $[0.001, 0.01]$. As

shown in Fig. 5(c), in the first iteration of **Step 2**, we assign the success probabilities of the segments $s_{77} \rightarrow s_{40} \rightarrow s_8$, $s_{79} \rightarrow s_{40} \rightarrow s_8$, and $s_{80} \rightarrow s_{40} \rightarrow s_8$ as 1, and after applying the CD algorithm, we successfully localize the faulty link $s_{78} \rightarrow s_{40}$, which explains the packet losses on the segments of $s_{78} \rightarrow s_{40} \rightarrow s_5$, $s_{78} \rightarrow s_{40} \rightarrow s_6$, $s_{78} \rightarrow s_{40} \rightarrow s_7$, and $s_{78} \rightarrow s_{40} \rightarrow s_8$, and remove the link from the sub-problem. In the second iteration, we use the logarithm-based function to map the success probability of $y_j = 0.998$ to $y'_j = 0.795$ for the three lossy segments, apply the CD algorithm on the remaining sub-problem as shown in Fig. 5(d), and localized the faulty link $s_{40} \rightarrow s_8$ for explaining the packet losses on the path segments of $s_{77} \rightarrow s_{40} \rightarrow s_8$, $s_{79} \rightarrow s_{40} \rightarrow s_8$, and $s_{80} \rightarrow s_{40} \rightarrow s_8$.

On the other hand, if we directly apply the CD algorithm, only the faulty link $s_{78} - s_{40}$ is inferred, while the link $s_{40} - s_8$ is overlooked because of its low loss rate.

G. Discussion

1) Coordinating packet counting: To synchronize epochs across the switches network wide, we consider OmniMon's synchronization mechanism [42], which does not require a global clock, ensures strong consistency in most time, and is downgraded to weak consistency for only a small bounded time period.

To make sure that a pair of upstream and downstream switches covering a path segment count a same set of packets during an epoch. In the Canary system, epochs are labeled as odd and even alternatively, and each monitor switch maintains two instances of the USM and DSM meters corresponding to the odd and even epochs. During an odd/even epoch, the monitor switches employ the odd/even meter instances to count packets, and after the epoch, they upload the instances to the diagnozer server, and switch to the even/odd instances for the next epoch. Moreover, the upstream switch marks each packet it has counted, which indicates whether its current epoch is odd or even. On receiving a marked packet, the downstream switch counts the packet using the corresponding DSM instance. We use the `DSCP` field for packet marking.

2) Probability of having clean bucket: As described in Sec. III-C, a USM or DSM bucket for monitoring a large-sized network flow is clean only when 1) there are no other flows hashed to this bucket (i.e., no hash collision), and 2) the flow does not change its path during the epoch. Since in multi-path routing, a switch chooses the next hop for a packet by hashing the packet's static flow ID, a flow rarely changes its path during an epoch. In the following, we analyze the probability that a large-sized network flow does not have a clean bucket due to hash collisions, and have the following result.

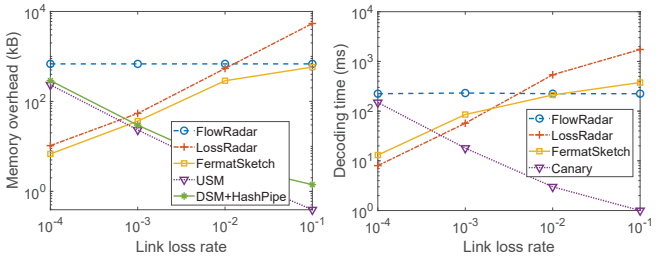
Theorem 2. *For a large-sized network flow f, the probability that it has at least one clean bucket in a USM/DSM meter is*

$$P_{clean} = 1 - \left(1 - e^{-\frac{F-1}{w}}\right)^d \tag{6}$$

where d and w are the rows and columns of the buckets in the meter, and F is the total number of the large-sized network flows monitored by the meter.

TABLE I
CONFIGURATIONS OF SOLUTIONS IN OVERHEAD COMPARISON

Solution	Data structure	Configuration
FlowRadar	Counting table	3 hash functions, bucket size = 144 bit.
	Flow filter	$\frac{1}{9}$ of counting table size.
LossRadar	Meter	3 rows of buckets, bucket size = 152 bit.
ChameleMon	FermatSketch	3 rows of buckets, bucket size = 136 bit.
Canary	USM	3 rows of buckets, bucket size = 137 bit.
	DSM	3 rows of buckets, bucket size = 169 bit.
	HashPipe	3 stages of hash tables, each has 32 entries, table entry size = 136 bit.



(a) Memory (b) Decoding time

Fig. 6. (a) Memory overhead and (b) decoding time of different solutions under various link loss rates.

We present the detailed proof of Theorem 2 in Appendix A-B. Suppose a USM/DSM meter contains $d = 3$ rows, and each row has $w = 2^{10}$ buckets, according to the theorem, for monitoring 50 large-sized network flows, P_{clean} is as high as 0.999. Even monitoring 400 flows, P_{clean} is 0.966. The analysis shows that with moderate-sized meters, Canary is capable to provide usable measurement data.

IV. EVALUATION

In this section, we evaluate Canary and compare it with the state-of-the-art solutions. We carry out the experiments with packet-level simulation and on a real-world testbed.

A. Experiment Setup

We simulate two network topologies. The first topology is composed of two switches directly connected with a link of 100 Gbit/s, and we also simulate large-scale data center networks with fat-tree topology. The simulations are conducted on a PC equipped with Intel Core i5-12600K running Ubuntu 16.04 LTS. The real-world testbed is composed of two hardware programmable switches based on the Intel Tofino chip [22]. The switches are inter-connected with a 40GbE link, and each switch is connected by a server equipped with an Intel XL710 40GbE dual-port Ethernet adapter.

If not otherwise specified, for each simulated monitor switch, we configure the DSM, USM, and HashPipe as the following. The DSM and USM have $d = 3$ rows, and each row has $w = 2^{10}$ buckets. Each DSM bucket has a size of 169 bits, which include a 1-bit $dirtyFlag$, a 104-bit $flowID$, a 32-bit $pathBF$, and a 32-bit $count$. A USM bucket does not have the $pathBF$ field, and has a size of 137 bits. Unlike the original 6-stage HashPipe in [20], we employ a lightweight HashPipe containing only 3 stages. Each HashPipe stage is

TABLE II
NETWORK-WIDE MEMORY OVERHEADS ON $k = 8$ FAT-TREE NETWORK UNDER VARIOUS LINK LOSS RATES (KB)

Link loss rate	10^{-3}	10^{-2}	10^{-1}
FlowRadar	54, 608	54, 608	54, 608
LossRadar	4, 328	43, 232	432, 304
FermatSketch	2, 912	23, 048	46, 408
Canary (bottom)	1, 694	213	58
Canary (top+bottom)	2, 542	319	86

a hash table with 32 entries, and each table entry contains a 104-bit flow ID and a 32-bit counter.

B. Loss Detection Overhead

We first examine the overhead for detecting packet losses incurred by Canary, and compare with the existing solutions of FlowRadar [17], LossRadar [18], and ChameleMon [19]. The overhead incudes the memory consumption and the decoding time of the measurement data structures. We realize these solutions on the 2-switch simulated network. When evaluating Canary, we place a USM at one switch, and place a DSM and a HashPipe at the other switch. For FlowRadar, we place a counting table and a flow filter at each switch. A bucket in the counting table has a 104-bit FlowXOR field, a 8-bit FlowCount field, and a 32-bit PacketCount field. The number of the hash functions used for the counting table is 3. The flow filter of FlowRadar is a standard Bloom filter, which incurs $\frac{1}{9}$ of the counting table's memory usage. For LossRadar, we place a meter composed of 3 bucket rows at each switch, and a meter bucket has a 32-bit count and a 120-bit xorSum fields. For ChameleMon, each switch hosts a FermatSketch, which uses 3 hash functions to map network flows to 3 bucket rows, and a FermatSketch bucket contains a 32-bit count and a 104-bit ID fields. Table I summarizes configurations of the solutions under the comparison.

In our experiment, we vary the loss rate of the inter-switch link from 10^{-4} to 10^{-1}, and send $10k$ network flows through the link for 100 ms. For each solution under the comparison, we vary number of the buckets in a row of the measurement data structures, and refer its *memory overhead* as the minimum memory usage that achieves 99% success rate. The experiment results are presented in Fig. 6(a). From the figure we can make the following observations. First, FlowRadar has a constant memory overhead, as it aims to monitor all the network flows regardless of the loss rate. LossRadar seeks to record all the lost packets, so its memory overhead is proportional to the number of the lost packets, and under the loss rate of 10^{-1}, LossRadar has the highest memory overhead. FermatSketch's memory overhead is proportional to the network flows suffering packet losses, which increases with the loss rate. Note that when the link's loss rate is 10^{-1}, nearly all the flows have packet losses, and FermatSketch's overhead is close to FlowRadar, as it actually monitors all the network flows.

Unlike these solutions, Canary's memory overhead decreases with the increase of the packet loss rate. This is because Canary only monitors a subset of network flows as long as packet losses can be detected. When the loss rate is low, more network flows should be traced by the meters, but when the loss rate is high, the system can detect packet losses even though it traces only a small number of network flows. For example, at a link loss rate of 10^{-2}, Canary consumes 2.3% of the in-switch memory consumed by FermatSketch.

Moreover, for localizing packet losses in a data center network, FlowRadar, LossRadar, or FermatSketch needs to be deployed hop-by-hop on all the switches, while Canary does not require each switch to host the meters. In Table II, we present the network-wide memory usages of the various solutions on a $k = 8$ fat-tree network that has a link loss rate varying from 10^{-3} to 10^{-1}. For Canary, we consider two deployment strategies: 1) only the bottom-layer switches host USMs and DSMs; and 2) both the top and bottom-layer switches host the meters. From the table, we can see that Canary can further reduce the memory overhead network wide, as it does not need to be deployed hop-by-hop as in the other solutions under the comparison.

Fig. 6(b) presents the time required to decode the data structures of FlowRadar, LossRadar, FermatSketch, and the meters of the Canary system. We can see that for FlowRadar and FermatSketch that seek to decode all the network flows and LossRadar that aims to recover all the lost packets, their decoding times increase with the link loss rate. On the other hand, Canary only needs to retrieve the records of a small number of selected network flows, therefore its decoding time is reduced as the link loss rate increases.

Our observation from Fig. 6 suggests that the overhead incurred by Canary decreases as the loss rate increases. We believe that this is a desired property, as a network administrator can allocate resources according to his objective, i.e., the highest loss rate the network can tolerate, rather than allocating resources blindly when they are related to the traffic characteristics that change dynamically.

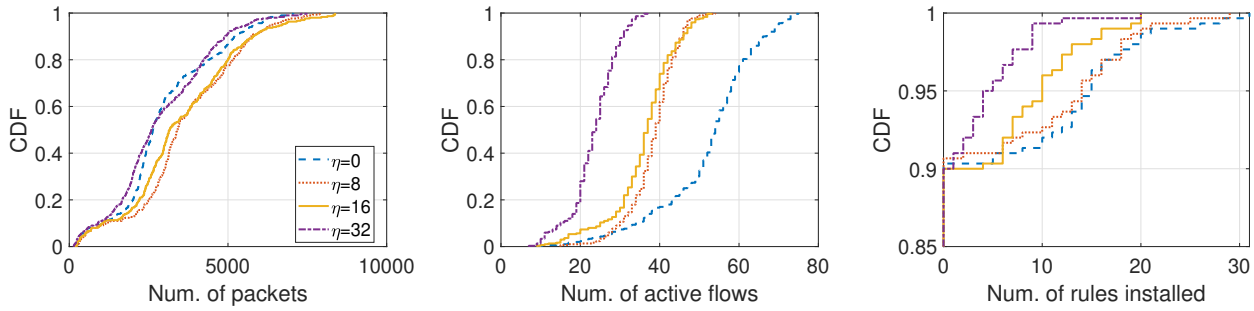
C. Counting Enough Packets

As described in Sec. III-D, in Canary, when the diagnozer server finds that the USM and DSM covering a path segment do not count enough packets, the controller will add new network flows selected by the HashPipe. In the following, we examine the effectiveness of this mechanism under the dynamics of real-world traffic.

We employ the 2-switch simulated network, and drive the experiment with the packet traces captured on ISP backbone links from MAWI [43]. The trace contains 8, 892, 143 distinct flows. We set the thresholds as $\alpha = 1,000$ and $\beta = 20$, and each time up to $\delta = 20$ flows selected by the HashPipe are added to the USM and DSM tables. When selecting large-sized network flows, we vary the threshold value of η as 8, 16, and 32. Note that a larger η suggests that the system has a higher standard in selecting large-sized network flows. We also consider the case that random flows are selected by setting η as 0.

Our experiment lasts 300 epochs, and an epoch has a duration of 100 ms. After each epoch, we evaluate the following metrics:

- Number of packets counted by the switches during the epoch.

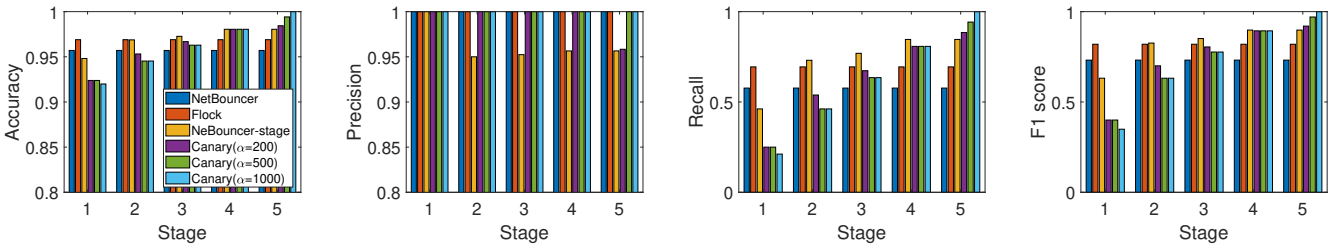


(a) Packets counted per epoch (b) Flows monitored per epoch (c) Rules installed per epoch

Fig. 7. Distributions of (a) numbers of packets counted, (b) numbers of active flows monitored, and (c) numbers of rules installed per epoch under various flow selection schemes.

TABLE III
CONFIGURATIONS OF INFERENCE ALGORITHMS.

Algorithm	NetBouncer		Flock			Canary					
Parameter	# probe pkt.	λ	p_b	p_g	ρ	λ	α	β	δ	η	# groups
Value	100	1	$1.0 - 10^{-3}$	2.5×10^{-4}	2.0×10^{-2}	1	200, 500, 1,000	20	20	16	5



(a) Accuracy (b) Precision (c) Recall (d) F1-score

Fig. 8. (a) Accuracy, (b) precision, (c) recall, and (d) F1-score of different fault inference methods over a $k = 8$ fat-tree network with 10% faulty links.

- Number of active flows monitored by the switches during the epoch.
- Number of match+action rules newly installed to the switches in this epoch.

Fig. 7 presents distributions of the metrics under different flow selection schemes. From Fig. 7(a), we can see that in most epochs, all the schemes manage to count over $\alpha = 1,000$ packets, however, in about 10% epochs, less than 1,000 packets are counted, due to the traffic dynamics when flows stop sending packets or leave the network. Fig. 7(b) shows that when flows are selected randomly (i.e., $\eta = 0$), the switches averagely monitors as many as 52.1 active flows per epoch, but when $\eta = 32$, only 23.5 active flows are traced. From Fig. 7(c), one can see that under all the schemes, in nearly 90% epochs, no match+action rules are installed, suggesting that Canary does not require frequent interventions from the controller. In addition, with the increase of the η threshold, fewer rules are installed to the switches.

Our observation suggests that with our proposed flow selection mechanism, Canary can effectively handle the traffic dynamics and ensure that a sufficient number of packets are counted. In addition, by selecting large-sized network flows, we reduce the controller overhead by installing fewer rules, and lower the chances of hash collisions in the USM/DSM by monitoring fewer network flows.

D. Faulty Link Inference

In this experiment, we examine how Canary localizes faulty links in a large-scale data center network with a $k = 8$ fat-tree topology. In the simulated network, the switches at the top and bottom layers serve as the monitor switches, and the entire network is divided into 8 sub-problems, with each sub-problem containing 24 switches and 64 inter-switch links. We apply our proposed inference method with multithreading, where each thread solves one sub-problem.

To introduce faults, we randomly select 10% links from the network, and configure them to drop packets with rates between 0.001 and 1.0. More specifically, we select 11 links to have loss rates in the range of $[0.2, 1.0]$, 13 links are selected with loss rates in $[0.1, 0.2]$, 9 links in $[0.05, 0.1]$, 9 links in $[0.01, 0.05]$, and 10 links in $[0.001, 0.01]$.

For comparison, we evaluate the following approaches: 1) NetBouncer [14], which applies the CD algorithm only once for localizing all the faulty links; 2) Flock [16], which employs a maximum likelihood estimation (MLE) algorithm to infer the faulty links; 3) We also evaluate a multi-stage version of NetBouncer (referred to as *NetBouncer-stage*), which runs the CD algorithm in 5 rounds, and in each round, the faulty links detected in the previous round are removed from the network.

To localize faulty links with Canary, we need the USMs and DSMs to count sufficient numbers of packets. For this purpose,

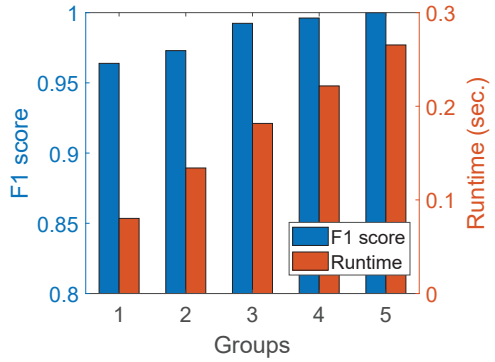
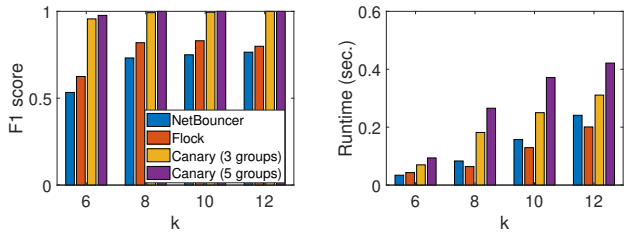


Fig. 9. Tradeoff between Canary's fault localization accuracy and inference algorithm's running time when dividing lossy path segments into various numbers of groups.



(a) F1 score (b) Runtime

Fig. 10. (a) F1 score and (b) inference algorithm's running time of NetBouncer, Flock, and two Canary systems that run in 3 and 5 stages, on fat-tree networks with k varying from 6 to 12.

we experiment with various values of the α threshold, i.e., the minimum number of packets supposed to be counted on a path segment per epoch, as 200, 500, and 1,000. For NetBouncer and NetBouncer-stage, we actively probe each path with 100 packets as in [14]. For Flock, we feed the inference algorithm with the path loss rates detected by NetBouncer with 100 probe packets. Table III presents the parameters of the inference algorithms under the comparison.

In Fig. 8, we present performances of the approaches in localizing the faulty network links in terms of accuracy, precision, recall, and F1-score. Note that for the approaches containing multiple stages, i.e., NetBouncer-stage and Canary, we present the performances after each stage in the figures. From Fig. 8 we can make the following observations. First, the single-stage algorithms have relatively poor performances. For example, NetBouncer and Flock localize only 15 and 18 out of the 26 faulty links[2]. The reason is that comparing with the links of high loss rates, the links with low loss rates contribute little to the objective function as in (3), or the likelihood function $LL(H)$ in Flock, thus are likely to be overlooked by the inference algorithms. Second, by applying the CD algorithm in multiple stages, NetBouncer-stage has its performance improved after each stage, and eventually detects 22 of the 26 faulty links, at a cost of one false positive error (i.e., one healthy link is mistakenly categorized as faulty). We explain such an improvement with the fact that by removing the inferred faulty links from the sub-problems after each stage, the CD algorithm is able to detect more faulty links with relatively lower loss rates, which used to be concealed by the links with higher loss rates detected in the previous rounds.

Our last observation is that Canary can localize the faulty links more accurately than NetBouncer, Flock, and NetBouncer-stage, and in particular, with $\alpha = 1,000$, after the 5 stages, Canary has localized all the 52 faulty links without false positive and false negative errors, therefore increases the F1 score by 36.7% and 22.2%, comparing with NetBouncer and Flock respectively. Canary's better performance can be explained in two folds. First, unlike NetBouncer and Flock that suffer from imperfect measurement, in Canary, more packets are used to detect pack losses, therefore a path segment that has a low loss rate can be detected, and its success probability can be measured more accurately. Second, by uniformly mapping all the path segments' success probabilities to the range of $[0, 0.8]$ in each stage, the CD algorithm can work in its "comfort zone", and localize the faulty links with higher accuracies.

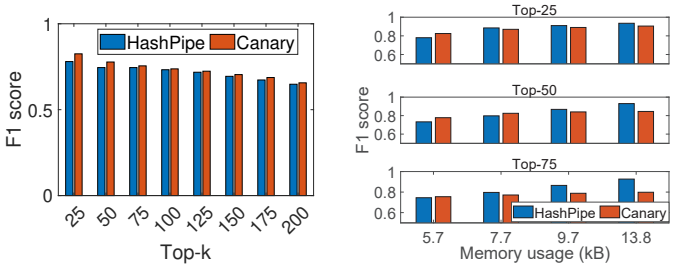
[2]NetBouncer does not consider the link direction, so we configure 26 links to drop packets in both directions.

E. Algorithm Running Time and Scalability

In Canary, we divide all the path segments with non-zero loss rates in groups, and iteratively apply the CD algorithm of NetBouncer to localize the faulty links that explain the losses in each group. Intuitively, the more groups we divide, the higher accuracy Canary can achieve. On the other hand, by executing in more stages, the inference will take a longer time. Clearly, there is a tradeoff between the number of the path segment groups and the running time of the inference algorithm.

To explore such a tradeoff, we repeat the experiment on Canary as in Sec. IV-D, but each time we evenly divide the lossy path segments into 1, 2, 3, 4, or 5 groups, and correspondingly, the CD algorithm is executed in 1, 2, 3, 4, or 5 stages. For each experiment, we examine how accurately the faulty links are detected in terms of the F1 score, as well as the time that the inference takes. Fig. 9 presents the result, from which we can see that even with 1 group, Canary can achieve an F1 score above 0.95, thanks to the over 1,000 probe packets on each path segment. In addition, by dividing the path segments into more groups, the F1 score is further increased, at a cost of a longer algorithm running time.

We also examine Canary's scalability by running the system on fat-tree networks of different sizes with k = 6, 8, 10, and 12, and compare with NetBouncer and Flock. As in Sec. IV-D, on each network, we randomly select 10% links to have packet losses, and evenly divide the links in 3 or 5 groups. Fig. 10 presents the F1 scores and the algorithms' running times of the solutions. From the figure we can make the following observations. First, all the algorithms are capable to maintain their F1 scores on large-sized networks, especially when k is no smaller than 8, and Canary has the F1 scores above 0.95 all the time. Second, as the network size increases, all the algorithms have their running time significantly prolonged, however, the increase of Canary's running time is relatively



(a) (b)

Fig. 11. (a) F1-scores in detecting top-k heavy hitters; (b) F1-scores under various memory budgets, achieved by original 6-stage HashPipe and Canary.

slower than the ones of NetBouncer and Flock, as Canary divides the problem into k sub-problems, which are solved in parallel with multithreading. The experiment result suggests that with the state-of-the-art parallel computing techniques, Canary is scalable on large-sized networks.

F. Detecting Heavy-Hitter Flows

Although Canary is not dedicatedly designed for detecting heavy-hitter flows, however, since we select large-sized network flows and count their packets in meters, by combining the flows monitored by the DSM and the lightweight HashPipe, Canary can also be used to detect heavy hitters. In the following, we employ the 2-switch simulated network and the MAWI traces as in Sec. IV-C to compare Canary with the original 6-stage HashPipe [20] in detecting heavy hitters. Note that for fairness, the original HashPipe consumes same memory as the downstream monitor switch. In particular, we configure the DSM to have 3×2^6 buckets, the lightweight HashPipe is configured as in Sec. IV-A, and the original HashPipe contains 56 entries per stage.

Fig. 11(a) presents the F1-scores in detecting the top-k largest network flows by Canary and the original HashPipe. To our surprise, Canary is more accurate in identifying the heavy hitters. Further investigation reveals that in the original HashPipe, a heavy-hitter flow may be evicted if it is inactive for some time before the end of the 30-second experiment. But in Canary, large-sized network flows are always counted by the DSM.

We then increase the memory usages of the two solutions and repeat the experiment. In particular, we enlarge the original HashPipe by having each stage to contain up to 176 hash table entries, and proportionally enlarge the DSM by having each row to contain up to 2^8 buckets. Fig. 11(b) presents the F1-scores in detecting the top-25, top-50, and top-75 largest flows under various memory budgets. From the figures, we can see that when more memory is used, the original HashPipe outperforms Canary, especially in the top-75 task. This is because in Canary, the DSM does not monitor more than $\beta = 20$ network flows as long as no less than $\alpha = 1,000$ packets are counted, but HashPipe aggressively traces as many network flows as its memory allows. The experiment results suggest that although not dedicatedly designed for heavy hitter detection, Canary achieves a decent accuracy, especially under a limited memory resource budget.

TABLE IV
RESOURCE CONSUMPTION OF CANARY ON P4-PROGRAMMABLE SWITCH

Resource	Usage	Percentage
Match crossbar	130	5.0%
Gateway	18	9.4%
Hash bit	274	5.5%
Stateful ALU	8	8.3%
SRAM	29	3.0%
TCAM	4	1.4%
Logical Table IDs	20	11.44%

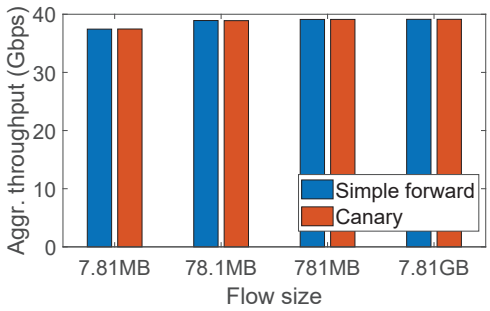


Fig. 12. Packet forwarding speeds of Canary monitor switch and simple-forwarding switch.

G. Hardware Prototype

We have implemented a prototype of the Canary system, and in particular, we compose the pipeline as illustrated in Fig. 2 with the $P4_{16}$ programming language [44], and realize monitor switch on the Edgecore Wedge 100BF-32X Tofino-based hardware programmable switch. Table IV presents usages of various resources in the programmable switch reported by the P4 compiler, from which we can see that Canary does not have high demands on hardware resources.

We use the real-world testbed to evaluate packet forwarding performance of the Canary system. In our experiment, we use two Tofino-based switches as the monitor switches, and send 4 TCP flows from one server to the other using iPerf [45]. For comparison, we also program the switches to simply forward packets. We vary the flow size and measure the aggregated throughput of the flows. Fig. 12 presents the results, from which we can see that when the flows are large enough, the aggregated throughput is close to 40 Gbit/s, which is indeed the line speed of the switch, and there is no difference between the monitor switch and the switch that simply forwards packets. As the operations introduced by Canary do not impact a programmable switch's packet forwarding speed, we conclude that Canary is practical for deployment on production networks.

V. CONCLUSION

In this paper, we present Canary, a system for detecting and localizing packet losses based on partial traffic monitoring. Canary employs in-switch meters to monitor network flows. However, unlike the existing traffic monitoring based solutions that are location agnostic, Canary encodes information of the path that a packet travels along in its header, thus can detect packet losses on each path segment. Moreover, different from the solutions that measure the entire network traffic, Canary

monitors only a small number of selected large-sized network flows, as long as a sufficient number of packets are counted for loss detecting. By improving a state-of-the-art inference algorithm and leveraging path information of the lost packets, Canary is able to infer the faulty network links, especially the ones with low loss rates, with high accuracy.

We have implemented a Canary prototype with P4 on Tofino-based hardware programmable switches, and evaluated Canary with both simulation-based experiments and a real-world testbed. The experiment results suggest that Canary is lightweight regarding measurement overhead, and provides flexibility for network administrators to allocate resources. By adaptively selecting large-sized network flows, Canary manages to count a sufficient number of packets on each path segment under traffic dynamics. Experiment on a large scale fat-tree data center network shows that Canary is capable to detect faulty network links with few false positive and false negative errors. In addition, Canary achieves a decent accuracy in detecting heavy-hitters, despite that it is not dedicatedly designed for that objective. Finally, performance evaluation on the hardware testbed shows that Canary does not impact a switch's packet forwarding speed, thus is ready for practical deployment.

Appendix A
Proofs

A. Proof of Theorem 1

Theorem 1. *Given a Clos-structured network, suppose a path segment from an upstream switch to a downstream switch is encoded in a pathBF and both monitor switches are at the bottom layer, then the probability that we fail to decode a unique path segment from the pathBF is*

$$P_{fail} \approx \left(1 - e^{-\frac{(h-1)m}{b}}\right)^{(h-2)m} \tag{1}$$

where h is the length of the bottom-to-bottom path segment, m is the number of the hash functions, and b is the Bloom filter size.

Proof. Given a b-bit Bloom filter that has $(h-1)$ switch IDs encoded when arriving to the last-hop monitor switch, according to the Bloom filter theory [39], the probability that an ID of a switch off the path segment passes the Bloom filter test (i.e., a false positive error) is approximately $(1-e^{-\frac{(h-1)m}{b}})^m$. Since an erroneous path segment differs from the ground-truth path segment in $(h-2)$ hops, therefore the probability that an erroneous path segment can be mistakenly decoded from the pathBF is

$$P_{fail} \approx \left(\left(1 - e^{-\frac{(h-1)m}{b}}\right)^m\right)^{h-2} = \left(1 - e^{-\frac{(h-1)m}{b}}\right)^{(h-2)m} \tag{7}$$

□

B. Proof of Theorem 2

Theorem 2. *For a large-sized network flow f, the probability that it has at least one clean bucket in a USM/DSM meter is*

$$P_{clean} = 1 - \left(1 - e^{-\frac{F-1}{w}}\right)^d \tag{6}$$

where d and w are the rows and columns of the buckets in the meter, and F is the total number of the large-sized network flows monitored by the meter.

Proof. Suppose that the flow f is mapped to an arbitrary bucket in a row of the DSM or USM meter. Since there are $(F-1)$ other large-sized network flows randomly mapped to the w buckets in this row, the probability that no other flows are mapped to this bucket is $(1-\frac{1}{w})^{F-1} \approx e^{-\frac{F-1}{w}}$. Since there are d rows in a meter, among the d buckets that f is mapped to, the probability that at least one collision-free bucket exists is

$$P_{clean} = 1 - \left(1 - e^{-\frac{F-1}{w}}\right)^d \tag{8}$$

□

References

[1] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *Proc. of SIGCOMM'15*, London, UK, Aug. 2015, pp. 479–491.

[2] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *ACM Queue*, vol. 14, no. 5, pp. 20–53, 2016.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. of SIGCOMM'10*, New Delhi, India, Aug. 2010, pp. 63–74.

[4] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *Proc. of SIGCOMM'15*, London, UK, Aug. 2015, pp. 523–536.

[5] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "HPCC: High precision congestion control," in *Proc. of SIGCOMM'19*, Beijing, China, Aug. 2019, pp. 44–58.

[6] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray failure: The Achilles' heel of cloud-scale systems," in *Proc. of HotOS'17*, Whistler, BC, Canada, May 2017, pp. 150–155.

[7] D. Zhuo, M. Ghobadi, R. Mahajan, K.-T. Förster, A. Krishnamurthy, and T. Anderson, "Understanding and mitigating packet corruption in data center networks," in *Proc. of SIGCOMM'17*, Los Angeles, CA, USA, Aug. 2017, pp. 362–375.

[8] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu, "Flow event telemetry on programmable data plane," in *Proc. of SIGCOMM'20*, Virtual Event, NY, USA, Aug. 2020, pp. 76–89.

[9] H. Herodotou, B. Ding, S. Balakrishnan, G. Outhred, and P. Fitter, "Scalable near real-time failure localization of data center networks," in *Proc. of KDD'14*, New York, NY, USA, Aug. 2014, pp. 1689–1698.

[10] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. of SIGCOMM'15*, London, UK, Aug. 2015, pp. 139–152.

[11] P. Lapukhov, "Move fast, unbreak things! network debugging at scale," 2016. [Online]. Available: https://archive.nanog.org/sites/default/files/Lapukhov_Move_Fast_Unbreak.pdf

[12] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, and Z. Li, "deTector: a topology-aware monitoring system for data center networks," in *Proc. of ATC'17*, Santa Clara, CA, USA, Jul. 2017, pp. 55–68.

[13] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically finding the cause of packet drops," in *Proc. of NSDI'18*, Renton, WA, USA, Apr. 2018, pp. 419–435.

[14] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, "NetBouncer: Active device and link failure localization in data center networks," in *Proc. of NSDI'19*, Boston, MA, USA, Feb. 2019, pp. 599–613.

[15] R. Ding, X. Liu, S. Yang, Q. Huang, B. Xie, R. Sun, Z. Zhang, and B. Cui, "RD-Probe: Scalable monitoring with sufficient coverage in complex datacenter networks," in *Proc. of SIGCOMM'24*, Sydney, Australia, Aug. 2024, pp. 258–273.

[16] V. Harsh, T. Meng, K. Agrawal, and P. B. Godfrey, "Flock: Accurate network fault localization at scale," *Proceedings of the ACM on Networking*, vol. 1, no. CoNEXT1, Article 3, 2023.

[17] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better netflow for data centers," in *Proc. of NSDI'16*, Santa Clara, CA, USA, Mar. 2016, pp. 311–324.

[18] ——, "LossRadar: Fast detection of lost packets in data center networks," in *Proc. of CoNEXT'16*, Irvine, CA, USA, Dec. 2016, pp. 481–495.

[19] K. Yang, Y. Wu, R. Miao, T. Yang, Z. Liu, Z. Xu, R. Qiu, Y. Zhao, H. Lv, Z. Ji, and G. Xie, "ChameleMon: Shifting measurement attention as network state changes," in *Proc. of SIGCOMM'23*, New York, NY, USA, Aug. 2023, pp. 881–903.

[20] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. of ACM SIGCOMM Symposium on SDN Research (SOSR'17)*, Santa Clara, CA, USA, Apr. 2017, pp. 164–176.

[21] "$P4_{16}$ portable switch architecture (psa)," The P4.org Architecture Working Group, Tech. Rep., 2021.

[22] "Intel tofino series," accessed on Mar. 22, 2024. [Online]. Available: https://intel.com/content/www/us/en/products/details/network-io/programmable-ethernet-switch/tofino-series.html

[23] D. Thaler and C. Hopps, "Multipath issues in unicast and multicast next-hop selection," IETF, RFC 2991, Nov. 2000. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2991.txt

[24] Y. Chen, D. Bindel, and R. H. Katz, "Tomography-based overlay network monitoring," in *Proc. of IMC'03*, Miami Beach, FL, USA, Oct. 2003, pp. 216–231.

[25] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot, "Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data," in *Proc. of CoNEXT'07*, New York, NY, USA, Dec. 2007, pp. 1–12.

[26] D. Ghita, H. Nguyen, M. Kurant, K. Argyraki, and P. Thiran, "Netscope: Practical network loss tomography," in *Proc. of IEEE INFOCOM'10*, San Diego, CA, USA, Mar. 2010, pp. 1–9.

[27] L. Ma, T. He, A. Swami, D. Towsley, K. K. Leung, and J. Lowe, "Node failure localization via network tomography," in *Proc. of IMC'14*, Vancouver, BC, Canada, Nov. 2014, pp. 195–208.

[28] B. Claise, "Cisco Systems NetFlow services export version 9," IETF, RFC 3954, Oct. 2004. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3954.txt

[29] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[30] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and fast network-wide measurements," in *Proc. of SIGCOMM'18*, Budapest, Hungary, Aug. 2018, pp. 561–575.

[31] L. Gu, Y. Tian, W. Chen, Z. Wei, C. Wang, and X. Zhang, "Per-flow network measurement with distributed sketch," *IEEE/ACM Trans. Netw.*, vol. 32, no. 1, pp. 411–426, 2024.

[32] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. of SIGCOMM'08*, Seattle, WA, USA, Aug. 2008, pp. 63–74.

[33] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[34] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer, "Hash-based IP traceback," in *Proc. of SIGCOMM'01*, San Diego, CA, USA, Aug. 2001, pp. 3–14.

[35] R. P. Laufer, P. B. Velloso, D. de O. Cunha, I. M. Moraes, M. D. Bicudo, M. D. Moreira, and O. C. M. Duarte, "Towards stateless single-packet IP traceback," in *Proc. of IEEE Conference on Local Computer Networks (LCN'07)*, Dublin, Ireland, Oct. 2007, pp. 548–555.

[36] H. Takurou, K. Matsuura, and H. Imai, "IP traceback by packet marking method with Bloom filters," in *Proc. of IEEE International Carnahan Conference on Security Technology (CCST'07)*, Ottawa, ON, Canada, Oct. 2007, pp. 255–263.

[37] S. Xiong, Q. Cao, and W. Si, "Adaptive path tracing with programmable Bloom filters in software-defined networks," in *Proc. of IEEE INFOCOM'19*, Paris, France, Jun. 2019, pp. 496–504.

[38] A. I. Basuki, D. Rosiyadi, and I. Setiawan, "Preserving network privacy on fine-grain path-tracking using P4-based SDN," in *Proc. of International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET'20)*, Tangerang, Indonesia, Nov. 2020, pp. 129–134.

[39] A. Z. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2003.

[40] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive realtime datacenter fault detection and localization," in *Proc. of NSDI'17*, Boston, MA, USA, Mar. 2017, pp. 595–612.

[41] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. of IMC'10*, Melbourne, Australia, Nov. 2010, pp. 267–280.

[42] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, , and Y. Bao, "OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy," in *Proc. of SIGCOMM'20*, Virtual Event, NY, USA, Aug. 2020, pp. 404–421.

[43] "Mawi working group traffic archive," accessed on Mar. 22, 2024. [Online]. Available: https://mawi.wide.ad.jp/mawi/

[44] "$P4_{16}$ language specification," The P4 Language Consortium, 2024. [Online]. Available: https://p4.org/wp-content/uploads/2024/10/P4-16-spec.pdf

[45] "iPerf - the ultimate speed test tool for TCP, UDP and SCTP," accessed on Mar. 22, 2024. [Online]. Available: https://iperf.fr/

Wei Chen received the bachelor's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2020. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, USTC. His research interests include data center networks and network measurement.

Ye Tian received the bachelor's degree in electronic engineering and the master's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2001 and 2004, respectively, and the Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China, in 2007. He joined USTC in 2008 and is currently an Associate Professor with the School of Computer Science and Technology, USTC. His research interests include data center networks, network measurement, and network security. He has published over 90 papers and co-authored a research monograph published by Springer. He is the winner of the Wilkes Best Paper Award of Oxford The Computer Journal in 2016. He is a member of the IEEE.

Zhaohui Wang received the bachelor's degree in computer science from University of Electronic Science and Technology of China, Chengdu, China, in 2023. She is currently pursuing the master's degree with the School of Computer Science and Technology, University of Science and Technology of China. Her research interest is focused on network security.

Cenman Wang received the bachelor's degree in computer science from University of Science and Technology of China (USTC), Hefei, China, in 2022. She is currently pursuing the master's degree with the School of Computer Science and Technology, USTC. Her research interest is focused on data center networks.

Xinming Zhang received the BE and ME degrees in electrical engineering from China University of Mining and Technology, Xuzhou, China, in 1985 and 1988, respectively, and the PhD degree in computer science and technology from the University of Science and Technology of China (USTC), Hefei, China, in 2001. Since 2002, he has been with the faculty of USTC, where he is currently a Professor with the School of Computer Science and Technology. From September 2005 to August 2006, he was a visiting Professor with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea. His research interest includes wireless networks, deep learning, and intelligent transportation. He has published more than 100 papers. He won the second prize of Science and Technology Award of Anhui Province of China in Natural Sciences in 2017. He won the awards of Top reviewers (1%) in Computer Science & Cross Field by Publons in 2019. He is a senior member of the IEEE.