LossRadar: Fast Detection of Lost Packets in Data Center Networks

Yuliang Li* Rui Miao† Changhoon Kim‡ Minlan Yu*
*Yale University †University of Southern California ‡Barefoot Networks

ABSTRACT

Packet losses are common in data center networks, may be caused by a variety of reasons (e.g., congestion, blackhole), and have significant impacts on application performance and network operations. Thus, it is important to provide fast detection of packet losses independent of their root causes. We also need to capture both the locations and packet header information of the lost packets to help diagnose and mitigate these losses. Unfortunately, existing monitoring tools that are generic in capturing all types of network events often fall short in capturing losses fast with enough details and low overhead. Due to the importance of loss in data centers, we propose a specific monitoring system designed for loss detection. We propose LossRadar, a system that can capture individual lost packets and their detailed information in the entire network on a fine time scale. Our extensive evaluation on prototypes and simulations demonstrates that LossRadar is easy to implement in hardware switches, achieves low memory and bandwidth overhead, while providing detailed information about individual lost packets. We also build a loss analysis tool that demonstrates the usefulness of LossRadar with a few example applications.

1. INTRODUCTION

Packet losses are common in data center networks and can happen for a variety of reasons. For example, Jeff Dean's keynote [7] indicates on average in a production data center for one year, up to 40-80 machines can experience packet losses, 4 network maintenance jobs can cause 30-minute random connectivity losses, and 3 router failures can cause immediate traffic blackholing for an hour. Such losses can significantly affect application tail latency and throughput especially when applications often view losses as a signal of congestion. It also takes operators tens of hours to diagnose the root causes of losses and recover from them.

It is important to detect losses *fast* and *independent of the loss types* (e.g., congestion, blackholes). With fast loss detection, operators have enough time to diagnose their root causes and mitigate the impact of losses. Being independent of loss types, we can capture all kinds of losses, especially unexpected ones such as those caused by hardware and software errors. Moreover, to diagnose the root causes, operators often need the detailed location on where losses happen, and the packet header information on what kind of packets get lost, the timing of losses, and other loss patterns.

Unfortunately, existing monitoring tools that are generic in capturing all types of network events often fall short in capturing losses fast with enough details and low overhead. End-host based solutions [37, 25, 12] cannot provide the exact location information on where losses happen. Packet mirroring [38, 13] either incurs large bandwidth overhead of mirroring all the traffic, or selectively mirrors some packets and thus miss capturing all types of losses. Some counters at switches (e.g., SNMP counters) can report a few loss types (e.g., ACL drops) but often miss the other losses that are harder to capture (e.g., caused by hardware and software errors). Flow-based monitoring tools (e.g., FlowRadar [21], NetFlow [26]) keep counters for individual flows and compare them across hops to identify losses, which do not have the timing and sequence patterns of losses and have an overhead associated with the number of flows (which can be large in large data centers [33]). One common theme of these approaches is to detect loss by keeping records of ongoing flows and identify the missing ones.

Due to the importance of loss in data centers, we propose a specific monitoring system designed for loss detection. Instead of keeping records whose overhead is proportional to the ongoing traffic or flows, we propose to keep records whose overhead is proportional to the number of losses.

We propose LOSSRADAR, a lightweight packet loss detection service that quickly reports the locations and 5-tuple flow information of individual lost packets in 10s of milliseconds. The very basic component we introduce for LOSSRADAR is a meter, a light-weight traffic digest generation logic. We place several meters at various vantage points in a network. Then, each meter periodically generates small digests summarizing all the traffic passing through itself and exports the digests

CoNEXT '16, December 12-15, 2016, Irvine, CA, USA
 ISBN 978-1-4503-4292-6/16/12…$15.00
DOI: http://dx.doi.org/10.1145/2999572.2999609

to a remote collector. The collector then decodes the digests exported by the meters. The way collector performs such analysis is inspired by the flow conservation rule in graph theory (and the current conservation rule in electrical-circuit theory as well). Given any segments (a link, switch, or group of switches) in a network and all uni-directional flows passing through the segment, the collector compares between the traffic digests generated by all the meters surrounding the segment. Any mismatch across the digests indicates packet losses, and the collector decodes the digest mismatch further to restore the identifiers of each lost packet. We also introduce an analyzer that can identify several (but not all) types of root causes of packet losses.

We design a Bloom-filter based data structure to collect traffic digests at each meter in real time with three key benefits: (1) Its memory requirement only grows linearly with the number of lost packets instead of all the transferred packets. (2) It keeps all the details of the lost packets such as 5 tuples, timing, and sequences of packets. (3) These traffic digests are generic to capture all types of losses and can be easily implemented in today's commodity switches.

Our extensive testbed evaluations with open vSwitch based prototype and large-scale packet-level simulations show that LossRadar uses only 1.4% of memory usage compared to the state-of-the-art approach when the loss rate is lower than 0.1%, and 0.5% of the bandwidth overhead of full mirroring, while providing detailed information for all the lost packets.

2. PACKET LOSS IN DATA CENTERS

Packet losses not only significantly affect application performance but also make network management tasks such as traffic engineering and diagnosis hard [12]. In this section, we first discuss the key requirements of loss detection: fast detection to minimize the impact of losses, generic detection of all types of losses, and capturing the locations and the headers of lost packets to help diagnosis. Next, we discuss existing tools in data centers which fall short in supporting these requirements of loss detection with low overhead.

2.1 Requirements on loss detection

We summarize three key requirements on loss detection: (1) We need to detect packet losses *fast* to minimize their impact. (2) We need a generic approach to capture all types of packet losses ranging from congestion losses to random losses. (3) We need to know the locations and all the details of lost packets (e.g., header fields) to infer their root causes.

Fast detection. Packet losses are common in data center networks. Studies have shown that, on average in a production data center for one year, up to 40-80 machines can experience up to 50% packet loss rates, four network maintenance jobs can cause 30-minute random connectivity losses, and three router failures can cause immediate traffic blackholing for an hour [7]. Microsoft also reports one to ten new blackholes every day in a data center [12].

Packet losses often have significant impacts on application performance and network operations. Just a few non-congestion-related losses can cause significant tail latency increase and throughput drop (especially because TCP treats them as congestion signals) [23, 30], leading to violations of service level agreements (SLAs) and revenue drops of online services [8, 35]. Moreover, it often takes operators tens of hours to identify whether a failure or performance problem is caused by packet losses, where losses happen, which flows are affected, and finally identify the root causes of losses and figure out fixes [1, 2].

Instead, we need a loss detection solution that can identify losses and narrow down their root causes fast (e.g., in tens of milliseconds). In this way, we can give operators enough time to identify their root causes or take actions to eliminate these losses, in order to minimize their impact on applications and network operations.

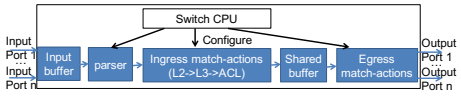


Figure 1: Switch packet processing pipeline

Generic detection of all loss types. Losses can be related to different components in the network (Figure 1) and be caused by different reasons. It is impossible to enumerate all types of losses, so we highlight a few common ones normally reported in data centers to illustrate the diversity of losses.

Congestion: Congestion loss happens at the output buffer and is caused by multiple flows competing for the same output port, and the total rate exceeds the capacity of that port.

Persistent blackholes: The persistent blackhole is a kind of problem that drops all packets matching a certain "pattern" at a switch [12, 38]. Persistent blackholes could be caused by match-actions table corruptions [12, 38] or the controller misconfiguring the rules.

Transient blackholes: The transient blackhole is similar to the persistent blackhole, but only during a short period. Transient blackholes could be caused by non-atomic rule updates [19] or network-wide inconsistent updates [14].

Random drops: The random drop is a type that the switch persistently drops packets randomly without reporting [12, 38]. Random drops could be caused by component problems such as not-well-seated linecards or faulty links [12].

Of course, there are many other losses we have not mentioned here. Such a diversity of losses demands a tool that can detect all losses regardless of the types.

Capturing location information. The first step to diagnose a packet loss is to identify the location of it—which switch, NIC, or host that incur the packet loss. Knowing the location, we can take quick actions (e.g., reboot the switch, reroute the traffic) to quickly mitigate the impact [36] or analyze the flows at the location to help diagnose the root causes [12].

Capturing packet header information. We need to distinguish different types of losses to identify the best solution to mitigate these losses (Table 1). For example, if there are

Loss types	Example error places	Mitigation	Details
Congestion	Buffer	Re-schedule flows	Flows, timing
Persistent blackholes	Switch match-action tables, Controller	Reboot, fix config	Flows, sequences
Transient blackholes	Controller, switch software	Debug controller or switch software	Flows, sequences, timing
Random drops	switch hardware	direct traffic away	Flows, timing

Table 1: Example types of packet losses

congestions, operators can re-schedule the flows to avoid congested links or move the applications to other servers. For persistent blackholes, we need to reboot the switches to fix table corruptions, or to correct the misconfigurations at the controller. If there is a transient blackhole, it helps operators to identify and fix the bugs in the controllers or switch softwares. If there are silent random drops, operators have to shut down and RMA (return merchandise authorization) the faulty components [12].

To distinguish different types of losses and understand the impacts of losses, we need different types of details. (1) flow information such as 5 tuples (source, destination IP addresses, ports, and protocol). Different flows may have different loss patterns, so it is important to know the 5 tuples. Also, such 5-tuple information can indicate the applications affected by the losses and allow operators to take fine-grained actions to help these applications. (2) Timing information. Many types of losses are very short, such as congestions or transient blackholes, so it is good to know the timing of each individual loss, which can help identify the temporal pattern of the losses. (3) Loss pattern. It is also important to identify the loss pattern, because different types of losses have different loss patterns (e.g., blackhole losses are continuous, while others may not be).

2.2 Existing monitoring tools

In data centers, there have already been a lot of monitoring tools. Some of them can be used for detecting packet losses. However, we now show that these solutions fall short in achieving the above goals. Due to the importance of loss detection in data centers, we need a new monitoring tool that is specifically designed for packet losses.

End host based solutions. End host based solutions [37, 25] can capture losses at the TCP level or by inspecting packets at the hypervisors. Although these solutions can capture all the details of lost packets independent of their types, it is hard to pinpoint the exact location of packet losses.

Pingmesh [12] can infer the location of packet losses through all-to-all probing, but cannot identify the exact switch that causes the problem within the leaf or spine groups. However, it does not track the packet losses of the actual applications. Moreover, its probing frequency is at least 10 seconds to reduce its overhead. Thus, it may miss many transient losses that happen between probes.

Packet mirroring at switches. Everflow [38] selectively mirrors packets to the collector, so the operators can know the traces of packets matching pre-selection filters and can detect losses by identifying those packets that do not have complete traces to reach the destination. However, it is not good at capturing most types of losses, and it is unclear if the packets get lost or if the report to the collectors gets lost.

Counters at switches. Switch vendors provide various counters (e.g., SNMP) to indicate the health conditions and operations at switches, some of which are related to losses. However, due to the diversity of loss types, it is difficult to identify and implement the right set of counters.

FlowRadar [21] captures per-flow counters at each switch at a fine time scale (e.g., 10s of milliseconds). One can use FlowRadar to detect packet losses by comparing the counters at two nearby hops. However, since it is hard to synchronize the counters to capture the same batch of packets, we often have to wait for a flow ends (or a flowlet ends with enough idle time after it [3]). The memory usage for FlowRadar is proportional to the number of flows we need to capture.

2.3 LossRadar Design

Instead of relying on generic monitoring tools which often fall short in loss detection, we propose to build a monitoring tool specific to loss detection given the prevalence and importance of packet losses in data centers. We propose to design a new loss detection solution that can capture all types of losses that happen at anywhere anytime and identify individual lost packets, while keeping the overhead low.

Generic to all types of losses. To detect losses regardless of their types and root causes, our key idea follows the *Flow Conservation Rule*: The set of packets that comes into any network domain (a link, a switch module, a switch, or an area of network devices) should be equal to the set of packets that leaves the domain, as long as the domain does not include any packet sources and sinks (e.g., middleboxes, hosts)[1].

Fast detection of losses and their locations. Based on the Flow Conservation Rule, we design LossRadar as shown in Figure 2. LossRadar installs *meters* to capture unidirectional traffic before it enters and after it leaves a domain in the network (called *upstream* and *downstream meters*). Each meter encodes the unique identifiers for individual packets into a traffic digest, and periodically reports traffic digests

[1]We recognize that a network switch can occasionally behave as a packet source or sink for certain types of packets (e.g., routing protocol messages, switch management protocol messages). Those packets, however, can be easily and unambiguously distinguished and hence can be filtered out from the LossRadar mechanism.

Loss detection tools	Detection time	Generic	Location	packet header info	Overhead
Host monitoring [37, 25]	10s of ms	Yes	No	Yes	Host CPU
Pingmesh [12]	10s of seconds	Only persistent losses	Infer	No	Host CPU
Mirroring [38]	10s of ms	Miss congestion	Yes	Yes	B/w prop. to # packets
Flow-level counters [21]	After a flow(let) ends	Yes	Yes	Only per-flow info	Memory prop. to # flows
LossRadar	10s of ms	Yes	Yes	Yes	Memory prop. to # losses

Table 2: State-of-the-art datacenter monitoring solutions

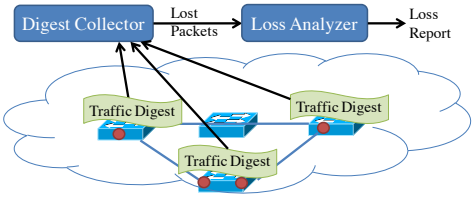


Figure 2: LossRadar architecture (Red circles are meters)

to a central LossRadar collector. LossRadar collector compares traffic digests from upstream and downstream meters, detects violations of the Flow Conservation Rule, and reports detailed information of lost packets.

Capture packet header information with low overhead. We want to store detailed information (e.g., 5 tuples, timing, sequences) to help the diagnosis of lost packets. However, switches have limited memory (only tens of MB for all the counters and match-action tables). Thus, it is important to capture important information about packet losses in limited memory. Rather than keeping information about all the traffic (e.g., FlowRadar [21]), we propose to build a traffic digest whose size is proportional to the number of packet losses independent of the total traffic or flows in the system.

With LossRadar, operators only need to provision the switch memory based on the number of losses they expect and the amount of details they need. When the number of losses goes beyond the expected number, LossRadar can still report the total number of losses at each location but may miss the detailed information such as 5 tuples for some losses. This is reasonable because when there is a large number of losses at a place, it is more important to reboot the entire switch or direct traffic away from it.

To collect more useful details of unknown lost packets, operators may also choose to pre-filter expected losses (e.g., those packets dropped by access control list) by specifying the right set of packets that LossRadar needs to keep in its traffic digests.

Challenges: There are three key challenges in building LossRadar:

Small size digests: To capture individual lost packets and their detailed information with low overhead, we proposed data structures that can capture all the packets in the digest while keeping the digest small enough to reduce the storage, bandwidth, and processing overhead. The key insight is we leverage coding techniques[10] to only keep information about lost packets. (Section 3)

Meter placement: To locate where losses happen in the network, we identify the right places to install meters that can cover the entire network. In some networks where we cannot deploy meters at all places, we can still install the meters at some locations that can narrow down the potential location of losses. (Section 4)

Loss analyzer: Given the details of losses, we design temporal and spatial loss analysis algorithms that can quickly identify some root causes such as blackholes, congestions and random drops. Moreover, we develop algorithms that can identify the set of rules that may cause the blackhole. (Section 5)

3. LossRadar TRAFFIC DIGESTS

In this section, we describe the design of traffic digests with low memory overhead and the benefits of our design, and then discuss the challenges in collecting the digests.

3.1 Capturing losses with small digests

We observe that although the number of packets in the network is too large for the meters to keep track of, the number of lost packets are much smaller. Thus if we can only keep track of the lost packets rather than all packets, we can reduce the overhead a lot. The question is how to track lost packets? Our idea is to keep traffic digests at an upstream meter and a downstream meter. When we compare the two meters, the same packets at both digests will cancel out, so only the packets that do not appear at the downstream (i.e., lost packets) remain in the result. In this way, the digest size only needs to be large enough for the number of losses, not the total number of packets.

We build traffic digests using Invertible Bloom Filter (IBF) [10], as shown in Figure 3. The digest has an array of cells. Each cell contains two values: $xorSum$ and $count$. When a packet x arrives, we insert it into k cells. Each cell is updated as $xorSum = xorSum \oplus x.sig$, $count = count + 1$. The $x.sig$ denotes x's *signature*, which includes the immutable header fields that can uniquely identify the packet (e.g. 5-tuple and IP_ID) and other information about the packet, which we will discuss in Section 3.2. The k cells are indexed by k hash functions ($h_1 \ldots h_k$) calculated on $x.sig$.

Figure 3 (a) and (b) shows two traffic digests, each of which is taken at the upstream meter UM and the downstream meter DM respectively. The upstream digest contains four packets from two flows, and the downstream one contains only one packet – the other three are lost.

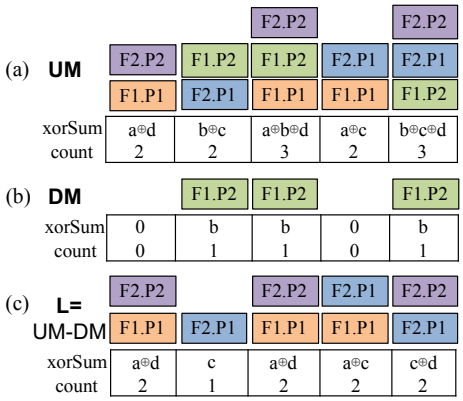


Figure 3: Traffic digests to get individual lost packets. (a) two flows, each with two packets at the upstream meter, indicated as a (F1.P1), b (F1.P2), c (F2.P1), d(F2.P2); (b) only F1.P2 is received at the downstream meter; (c) the result of the subtraction contains only lost packets

To identify lost packets, the digest collector subtracts each cell i in DM's digest from the corresponding one in UM's digest. That is: $L_i.xorSum = UM_i.xorSum \oplus DM_i.xorSum$; $L_i.count = UM_i.count - DM_i.count$. We can decode lost packets from the digest difference L as follows: We first find the cells of L that contain exactly one packet (i.e., $count = 1$) which are called *pure cells*. The $xorSum$ of a pure cell is exactly the signature of the packet in it. We then remove the packet from all the k cells in which the packet was hashed into. Once this is done, some other pure cells may appear, and we repeat the same process for them. By doing this iteratively, we can retrieve all packets in L with a very high probability (we will explain this in Section 3.2). For example, in Figure 3 (c), we first identify a pure cell whose count is 1 (F2.P1). After removing F2.P1, we get the packet F1.P1 (the 4^{th} cell's updated count is 1); and finally after removing F1.P1, we decode F2.P2.

3.2 Benefits of digests

There are two benefits of the digests.

The size of the digest in the meter is proportional to the number of lost packets, but not the total number of packets that go through the meter. Suppose both UM and DM have n_{cell} cells. Then L, as a result of the subtraction, also has n_{cell} cells. Because L only needs to be large enough to decode the n_{loss} lost packets, n_{cell} is determined by the n_{loss}. Existing works have shown that n_{cell} only needs to be proportional to n_{loss} (i.e., $c = \frac{n_{cell}}{n_{loss}}$ is a constant) to achieve a high success rate for decoding all losses[2]. Eppstein et al. [10] shows empirically that $c = 1.3$ to achieve a 99% success rate when $n_{loss} > 1000$ for $k = 3$. Thus, we only need to set the size of the digest in data plane with a small c times the expected number of losses to achieve a high success rate.

[2]A success for decoding all losses means we can always find a pure cell until all lost packets are decoded.

Because packet loss rate is very low (less than 0.01%) in data centers [12], we only need small sized digests. Thus we set our digest size small enough to capture lost packets based on an upper bound of expected loss rate R (e.g., 0.1%). When the loss rate is below R, we can decode individual lost packets with high probability (e.g., >99%). When the loss rate is above R, however, we can still detect the total number of lost packets per port, but may fail to identify individual lost packets. This is because the digest may have so many lost packets in each cell, and thus the occurrence of pure cells become progressively less probable. Note, however, this may be still be quite acceptable for most network admins because, when the loss rate is this high, it is much more crucial to take urgent action to fix the problem (e.g., shut down the link or the switch) than to diagnose the problem or to account for individual losses to particular applications that are affected.

We can include a lot of details we are interested in of each lost packet. The basic setting is to include two pieces of information in the signature of a packet. (1) Flow-level information. We store the 5-tuple information to provide enough details of individual packets. (2) Packet identifier. We use the IP_ID field because the IP_ID field is consecutive within each 5-tuple, so it can be used to uniquely identify a packet. In case the IP_ID field is not sufficient (e.g., if more than 64K packets from a single flow are monitored in a single monitoring window), we can include other fields such as fragment offset, TCP sequence number or a small fraction of the payload in the signature. In the basic setting, each signature is 120 bits[3].
Additionally, operators can configure to include any other fields of a packet that they are interested in (e.g., TCP flags). Note that if the field changes across hops, we need to consider that. For example, if we want to include TTL, and the TTL will decrement h in between, operators need to configure the DM to insert TTL + h instead of TTL into the digest. Operators can also include other metadata that are related to but not part of the packet, such as the timestamp when the packet arrives at the UM. In this case, the UM needs to tag the metadata in the packet header, so that the DM can use the same value. Including other fields require more bits in the signature. For example, if we include the TCP-SYN bit, TTL (only need 1 bit to distinguish zero and non-zero) and timestamp, we need to extend the signature by 16 bits[4], which are used in our loss root cause analysis (Section 5).

3.3 Digest configuration

With LOSSRADAR, operators only need to provision the switch memory based on the number of losses they expect and the amount of details they need. When the number of losses goes beyond the expected number, LOSSRADAR can still report the total number of losses at each location but may miss the detailed information such as 5 tuples for some losses. This is reasonable because when there is a large number of

[3]104 bits for 5-tuple, and 16 bits for IP_ID.

[4]1 bit for TCP-SYN, 1 bits for TTL, and 14 bits for microsecond-level timestamp in 10 ms (2^{14}us>10ms).

losses at a place, it is more important to reboot the entire switch or direct traffic away from it.

To collect more useful details of unknown lost packets, operators may also choose to pre-filter expected losses (e.g., those packets dropped by access control list) by specifying the right set of packets that LOSSRADAR needs to keep in its traffic digests. Recent programmable switches can export information of which packets they decide to drop because of access control lists or buffer overflow (e.g., using negative mirroring, which is presented at P4 workshop). With these techniques, we can also exclude the lost packets known by switches, and thus focus LOSSRADAR digests on those losses the switches cannot diagnose.

3.4 Measurement batch

To capture packet losses in near real time, we capture packets in batches at both upstream and downstream. So the operator needs to ensure the same packet goes to the same batch in both the upstream meter and the downstream meter.

To report packet losses in near real time, meters should keep collecting traffic digests for a small batch of packets every T (e.g., 10 ms) and immediately report the digests to the central collector as soon as the batch is over. To compare traffic digests correctly across meters, however, we need to ensure that a pair of upstream and downstream meters must agree on the boundary of batches so that they can put the same packet in the same batch. Otherwise, digests will always end up with mismatches. Unfortunately, relying on time-synchronization protocols or devices (e.g., NTP [27], PTP [32], GPS) to let meters agree on batch boundaries has fundamental limitations: no matter how accurate the time-sync protocol can ensure, there could always be packets on the fly, i.e. packets that leave the upstream meter, but have not arrived at the downstream meter. Hence, if we rely on the time-sync protocols, batch disagreement is inevitable.

Another possible solution is to use signal packets to synchronize the starting and ending times of a pair of upstream and downstream batches. However, if the signal packet is dropped or gets reordered, some packets may be classified into the wrong batch.

To work with packet losses and reordering, we propose to let each packet carry a batch identifier to tell the downstream meter which upstream batch the packet belongs to. Thus which batch at the downstream a packet belongs to is not based on the packet's arrival time, but the batch_ID it carries. If there is no packet reordering, a downstream batch can be terminated when a packet with a new batch_ID arrives. However, if there is reordering (e.g., due to different priority queues), packets from different batches may interleave at the downstream meter, so multiple downstream batches have to coexist. As a result, we set the lifetime of a downstream batch to $T + timeout$, starting at the first packet's arrival of this batch. The downstream batch is $timeout$ longer than the upstream batch, which is for the delay in between. The operator can set the value of $timeout$ based on their expectation of the maximum delay.

There is one question raised that how many downstream batches can coexist at the same time, which is related to the memory consumption. In data centers, the RTT is a few hundreds of microseconds [4], and per-hop delay should be shorter, so for $T > 1ms$, there will be at most two batches coexist at a downstream meter.

4. NETWORK WIDE DEPLOYMENT

In this section, we discuss where to install the meters to detect packet losses that happen at any place in the network, and pinpoint the exact loss locations. We consider two cases: When we have access to all the devices in the network, or we only have access to some switches.

4.1 Cover the entire network

We need to install meters to meet two goals: (1) Each pair of meters covers a segment of paths where packets have a unique path. In this way, we can easily compare the traffic digests taken at a pair of meters (upstream and downstream) to identify lost packets; (2) All pairs of meters together cover the entire network. In this way, we would not miss any packet loss event.

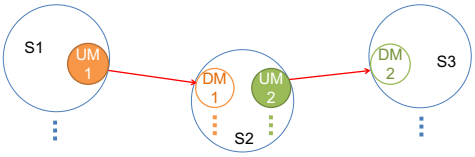
The high-level idea is to install one upstream meter at each output port of every switch, and one downstream meter at each input port of every switch as shown in Figure 4(a). In this way, both directions of every link are covered. Note that although we do not draw in Figure 4(a), there is also traffic from S2 to S1, so actually we also need an upstream meter at S2's output port connected to S1, and a downstream meter at S1's input port connected to S2. In fact, we need two pairs of upstream meter and downstream meter for each bidirectional link. We only draw one pair in Figure 4(a) for conciseness.

Then, given the packet processing pipeline of a switch, we need to figure out at which stages to install the meters. Considering that losses could happen at any stage of the pipeline, we need to make sure the meters cover every part of the data plane. We choose to put the meters at the end of the ingress pipeline of tables, and before the shared buffer, as shown in Figure 4(b). And, we put the upstream meters before the downstream meters within a switch, rather than the other way around, so that the segments between the upstream meters and downstream meters are also covered. In addition, we install meters for each network interface at hosts.

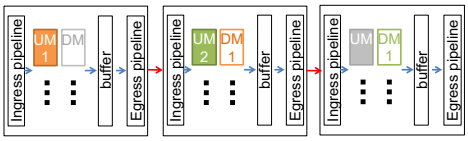
We choose this place to install meters for 2 reasons. First, at the place where upstream meters are installed, the decision of which output port(s) a packet goes to need to have been made, so that the switch knows which upstream meter (output port) to insert the packet into. Thus, the meters must be after all the ingress pipeline of tables, because the decision could be made at any of the ingress tables. Note, by doing so, we can handle multicast by inserting the packet into the upstream meters of all the output ports it goes to. Second, the timestamp collected at this stage is the closest to the time a packet is enqueued, so the timestamp can be used to describe the temporal pattern of congestion losses. By doing so, we make the most use of the timestamp of the packet.

4.2 Incremental deployment

We do not expect LOSSRADAR to be deployed at all the switches in data centers at once. Now we consider the incre-

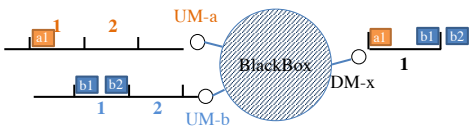


(a) Full deployment at all the switches

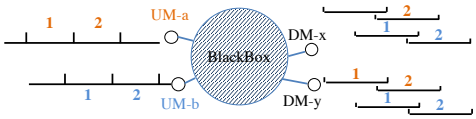


(b) Meter deployment details inside switches

Figure 4: Placing meters to cover the entire network (solid and empty circles are respective upstream and downstream meters)



(a) Sync. Issue for blackbox



(b) Solution for blackbox case

Figure 5: Challenge and solution to blackbox case

mental deployment case where only a subset of the switches support LossRadar. We define blackboxes as those areas where we cannot install the meters. The strawman approach is to install one meter at each port of the blackbox and compare across meters. When meters do not have synchronized clocks, we propose to let each meter maintain multiple small digests instead of one. Moreover, our solution can reduce the memory usage of the strawman approach by almost 2 times.

Strawman approach. Intuitively, we can insert an upstream meter at each entrance to the blackbox, and a downstream meter at each exit of the blackbox. Each upstream meter and downstream meter maintains one digest. To retrieve the losses, we can add up the digests at all upstream meters and the digests at all downstream meters respectively, and compare the two sums. Here adding up two digests a and b means for each corresponding cell i, $a_i.xorSum \oplus b_i.xorSum$, $a_i.count + b_i.count$. So the sum is also a digest of the same size with a and b.

The memory usage of each digest in this case is larger than in a non-blackbox case. The reason is as follows. Let us call the size of a digest in a non-blackbox case as *normal size*. Suppose there are n upstream meters and n downstream meters. During the comparison of the two sums, the total number of losses could be n times of the number of lost packets from a single upstream meter. Thus, the size of the sum digest has to be n times of the normal size. As a result, the size of the digest at each meter has to be the same with the sum, which is n times of the normal size.

Handle unsynchronized clock across meters. The problem with the strawman solution is that if the clocks of the upstream meters are not well synchronized, packets from the same batch of the same upstream meter may arrive at different batches at one downstream meter. For example (Figure 5(a)), UM-a's batch 1 is earlier than UM-b's batch 1. The packet a1 from UM-a's batch 1 goes to DM-x, and starts the downstream batch 1. The packets from UM-b's batch 1 also goes to DM-x. The packet b1 from the *early* part of UM-b's batch 1 arrives at the *late* part of DM-x's batch 1, since DM-x's batch 1 has already started for a while. The packet b2, however, arrives after DM-x's batch 1 ended. So b1 and b2 are in the same upstream batch, but end up in different downstream batches.

As long as the clocks of the upstream meters are not accurately synchronized, there is always a chance of batch disagreement.

Solution. Our solution is that each downstream meter maintains separate digests for each upstream meter, as shown in Figure 5(b). To detect the packet losses, we can add up all the digests for the same upstream meter across all the downstream meters (e.g., the digests for UM-a's batch 1 at DM-x and DM-y respectively), and compare the sum with the digest at the upstream meter (e.g., UM-a's batch 1). Thus, we can identify all the lost packets between one upstream meter and all its downstream meters.

Note that the downstream meters do not always know which upstream meter a packet comes from. Our solution is to let each upstream meter mark its meter ID in the packet header.

This solution has another benefit that it reduces the memory usage. In the strawman approach, the size of each digest (no matter at an upstream or a downstream) is n times of normal size. In our solution, however, we compare one upstream digest with the sum of its corresponding downstream digest, so the total number of losses in each comparison is just the number of lost packet from one upstream meter, not all upstream meters. So the size of each digest only needs to be normal size. Thus, although each downstream meter maintains n separate digests, the total size is still n times of normal size. But for each upstream meter, the total size is only normal size, which is reduced by n times.

In practice, each port of a blackbox can be both entrance and exit, so each switch connects to the blackbox need to install both upstream meter and downstream meter. In this case, the memory usage reduction is $2n/(n+1)$ per switch, which is almost 2 times.

5. LOSS ANALYZER

Using the detailed information of lost packets, we build an analyzer to classify the causes of packet losses by analyzing

traffic digests across the network over time. The loss analyzer consists of (1) a root cause analyzer, which uses only the loss information from LOSSRADAR to analyse the root causes of the losses, and (2) an ACL rule corruption analyzer as an application of the root cause analyzer, which can infer the corrupted rules of an ACL table based on the correct version of the ACL (from controller) and the blackhole losses from the root cause analyzer.

5.1 Root cause analyzer

The root cause analyzer classifies the reasons for the losses on each individual switch. In a short time (e.g. 10 ms) at one switch, we assume losses are dominated by one reason, so we can identify the root causes by recognizing the unique patterns of different types of losses, given the details collected by LOSSRADAR. We then relax the assumption to handle multiple problems all causing losses.

5.1.1 Classification based on loss patterns

We can classify the losses based on the unique patterns of different types of losses, given the details such as 5 tuples, IP_ID, TTL, timestamps and TCP flags. We list the common types of losses here. For other uncommon types, we classify them as unknown. However, after diagnosis, we can extract patterns of them based on the details we have, and use the patterns for future classification.

Congestion. We observed that *the congestion losses are bursty, and the gap between back-to-back losses is only a few microseconds*. The TCP sends a batch of packets every RTT, and the flows that experience losses will shrink their congestion windows, so in the next RTT congestion is much less likely to happen. Thus the congestion losses are bursty. The gap between back-to-back losses is only a few microseconds because the transmission time of a packet is microsecond level. For 10G link, the transmission time of a 1500B packet is 1.2us. Thus, even if the total receiving rate is only 10% higher than the sending rate of an output port, there will be one loss every 11 packets, which is one loss every 13.2us.

Blackholes. We observed three different loss patterns.

(1) Bursty and consecutive losses within a flow. If a blackhole happens and affects ongoing large flows (e.g., file transferring, video delivering), the affected ongoing large flows have consecutive lost packets, and the losses are bursty.

(2) Non-bursty and consecutive losses within a flow. If a blackhole happens and affects some ongoing small flows, which only carries small pieces of data (e.g., sending messages), the affected ongoing small flows have consecutive, but non-bursty losses.

(3) Only a SYN packet loss from each flow. In the above two cases, each flow has multiple losses, but there are cases some flows only have one loss for more than a few seconds. After further investigation, we found that these lost packets are mostly SYN or SYN-ACK. This means that for the new flows, their handshake packets are dropped, so there are no follow-up packets until the retransmission of the SYN packet, which waits for a few seconds (normally 3 seconds for the first retransmission, then the waiting time doubles).

Random drops. *The random drops are evenly distributed over time*. This is because random drops are caused by problematic hardware components of a switch, which randomly drops packets passing it with certain drop rates.

Loop. *Packets dropped by loops have TTL = 0*. So we can use the TTL field to detect the losses caused by loops.

5.1.2 Root cause inference algorithms

There is still a small chance that during a given time period, there may be multiple types of losses together, which blurs the patterns. Thus, we design the root cause inference algorithms based on two principles. **(1) Identify bursty losses first**. We identify bursty losses first because other types of losses have long term patterns (e.g. random drops are evenly distributed over time, and the SYN packets dropped by blackholes should represent a large fraction of losses), which can be blurred by the burst of losses. **(2) It is fine to correctly classify only one type of losses when multiple types are mixed**. This is because we can fix the problems for one type of losses first. Moreover, if the missed types are caused by persistent problems, we can detect them in the future when they are not mixed. **(3) Keep the false alarm rate low for blackholes and random drops**, because both of them require human involvement: operator needs to check the switch configuration for blackholes, and hardware components for random drops. Thus, when blackholes/random drops are mixed with congestion, we believe classifying the losses as congestion losses is better[5].

Before performing the inference algorithm, we first exclude the expected losses such as packets dropped by ACL tables. All the expected losses can be pre-filtered by programmable switches. If the switch does not support pre-filtering, the collector can easily identify this kind of losses by matching the lost packets against the table rules. We also exclude losses with TTL = 0, which could be dropped by loops, or are expected by the applications (e.g., traceroute). We can map the 5-tuple of the lost packets to the application to decide if the loss is expected or not.

The algorithm has two steps on different time scales: (1) For each new batch, we classify the bursty losses; (2) Every $long\_term\_interval$ (e.g. 100 ms), we classify the remaining losses (from multiple batches) into non-bursty types. We now describe the algorithm.

Analysis of bursty losses. We first perform an analysis of bursty losses for each new batch at each port (Algorithm 1). A burst is identified by at least n_{burst} losses with gaps less than t_{gap} between back-to-back losses. A burst of losses may be caused by either congestion or blackholes. If a flow's losses are non-consecutive in this burst, it cannot be caused by a blackhole, so we know there is congestion. In this case, according to the design principle (3), we classify the whole burst as congestion losses (Line 15 to 20). If all losses are consecutive, we only classify the flows that lost almost one

[5]We understand that different networks have different requirements, so operators may use other principles if necessary.

congestion window ($\#losses > n_{bb}$[6]) as blackhole losses, and the rest as congestion losses (Line 21 to 24).

Our idea to measure the consecutiveness is to examine the IP_ID of the lost packets. In current Linux, the IP_ID of each 5-tuple is consecutive [5], so we can use the IP_ID to determine if the losses are consecutive or not.

We should set t_{gap} large enough and n_{burst} small enough to detect slight and short congestion, but still keep the probability of randomly dropping n_{burst} packets with less than t_{gap} back-to-back gap low. n_{bb} should be set slightly lower than the bandwidth-delay product for flows with still-growing windows, while large enough to keep the probability of congestion dropping n_{bb} consecutive packets of a flow low.

Long term analysis. Every *long_term_interval* (e.g. 100 ms), we classify the rest of the losses (Algorithm 2).

We first classify the consecutive losses within a flow as blackhole losses (Line 2 to 6), because its per-flow pattern is not affected by other types of losses, and after removing these losses, the pattern of other losses are clearer. We set a small threshold n_{nb}[7] (line 3) for the length of the consecutive losses. n_{nb} needs to be large enough to keep the probability of randomly dropping n_{nb} consecutive packets of a flow low.

After that, for the rest of the packets, we first classify SYN losses as blackhole losses (line 8 to 11) if the fraction of SYN packets in an interval is large ($> p_{syn}$). p_{syn} should be set large enough to keep false alarm rate low. Finally, we measure the distribution of the rest of the losses over time (line 12 to 14). We divide the time interval into n (number of losses) equal bins, and count the fraction of non-empty bins (bins with at least one loss) in all bins. We empirically estimate this fraction using a simple experiment: we randomly throw m balls into m bins following even distribution, and count the fraction of non-empty bins. The fraction is around 0.63 for different m. This fraction gives an upper bound of p_{rand}, because in practice packets' arrivals are non-uniform.

In practice, we should set $n_{burst} \leq n_{nb}$, because otherwise if a congestion loss burst of n packets where $n < n_{burst}$ contains $\geq n_{nb}$ consecutive losses of a flow, the losses of that flow would be classified as blackhole losses.

5.2 Rule corruption detection

For blackhole losses, after confirming with the configuration, we can determine if there are misconfigurations or not. If not, there must be rule corruptions. We develop algorithms that can locate the set of rules whose corruption may cause the blackhole. We take the access control list (ACL) table as an example. Given the list of blackhole losses and ACLs, we need to identify the rule that may get corrupted. For simplicity, we only consider one bit flip in the table, which is the most common case. There are three cases for unexpected losses: (a) a deny rule's match field corrupted, so it denies flows originally not covered by it; (b) an accept rule's match field corrupted, and the flows originally covered by it now match other deny rules; (c) an accept rule's action changes to deny. We enumerate all three cases. For case (a), we enumerate all deny rules in the correct ACL table, and find the candidates whose match field corrupted by one bit would match all unexpected loss. For case (b) and (c), we check if all unexpected loss all match the same rule in the correct ACL table. If so, then this rule is a candidate. Our evaluation

[6]bb stands for Bursty Blackhole

[7]nb stands for Non-bursty Blackhole

Algorithm 1: LOSSRADAR bursty loss analysis

```
1  Function bursty_loss_analysis(B: a new batch of losses)
2      Mark the types of all losses in B as unknown;
3      burst = find_next_burst(B);
4      while burst != Null do
5          classify_burst(burst);
6          burst = find_next_burst(B);
7      end
8  Function find_next_burst(B)
9      Find the next B_{i...j} that B_{k+1}.time − B_k.time < t_gap, and
       the gap before B_i and after B_j is larger than t_gap;
10     while j − i + 1 < n_burst do
11         Find the next B_{i...j};
12     end
13     return B_{i...j};
14 Function classify_burst(burst)
15     foreach flow f in burst do
16         if f's lost packets have non-consecutive IP_ID then
17             Mark all losses in burst as congestion;
18             Return;
19         end
20     end
21     foreach flow f that has more than n_bb losses in burst do
22         Mark all losses of f as blackhole;
23     end
24     Mark losses in burst whose types are unknown as congestion;
```

Algorithm 2: LOSSRADAR root cause inference

```
1  Function long_term_analysis(L: all losses at a port in
   long_term_interval)
2      foreach flow f in L do
3          if f's lost packets have consecutive IP_ID and
           #(unclassified losses) >= n_nb then
4              Mark losses in f whose types are still unknown as
               blackhole;
5          end
6      end
7      rest = all the losses in L whose types are still unknown;
8      if fraction of SYN packets in rest > p_syn then
9          Mark all the SYN packets' types in rest as blackhole;
10         Remove the SYN packets from rest;
11     end
12     if random_distribution(rest) then
13         Mark all packets' types in rest as random;
14     end
15 Function random_distribution(L)
16     n = number of losses in L;
17     Divide the time period of L into n equal time interval T_{1...n};
18     Count the number of losses in each T_i;
19     if fraction of T_i that has losses > p_rand then
20         return True;
21     end
22     return False;
```

in Section 7.5 shows that this algorithm often narrows down the candidate rule set of 2-2.5 rules and always successfully cover the corrupted rules.

6. IMPLEMENTATION

We implemented LOSSRADAR in both Open vSwitch [28] and P4 behavioral model [29].

6.1 Open vSwitch implementation

We modify Open vSwitch to support traffic digest collection. We direct all the packets to the user space and collect traffic digests there. We create a sending thread which sends digests at the end of each batch to a digest collector via a persistent TCP connection. Each time the collector receives both the traffic digests from a pair of upstream meter (UM) and downstream meters (DM), it runs the decoding process to report lost packets.

6.2 P4 implementation

At a high level, we put LOSSRADAR's functions in two tables, at the end of the ingress pipeline. The first table is for updating UMs, and the second is for DMs. We now inspect the different components needed by LOSSRADAR.

Storing traffic digests. LOSSRADAR's traffic digest is implemented on the $register$ in P4. We defined two arrays of registers for $xorSum$ and $count$ respectively. The width of each register in the $xorSum$ array is set to the total number of bits in a packet's $signature$ (5-tuple, IP_ID, TTL, TCP flag, timestamp, etc.). The lengths of the arrays are the number of cells n_{cell}, which is defined by users according to their expected number of losses.

Each port of a switch has a UM and a DM. Each meter keeps two digests for different batches of packets. We put all $2n_{port}$ upstream digests in an array[8], and all $2n_{port}$ downstream digests in another array. The i-th cell in the b-th batch of the p-th port is indexed by $i + (b \cdot n_{port} + p) \cdot n_{cell}$.

Updating digests. This includes calculating hash values based on the packets' $signatures$, and updating the cells indexed by these hash values. We define 3 hash functions, calculated on a $field\_list$ that is defined to include the $signature$ of each packet. The $signature$ is a bit different for UM and DM. The mutable part of the $signature$ (e.g. timestamp and TTL) for the DM should be the same as the part for the UM at the *previous hop*. So each UM uses the timestamp at this hop, and each DM uses the timestamp carried by the packet. The DM uses TTL+1 instead of TTL to cancel out the decrement between hops.

Getting timestamps. LOSSRADAR uses timestamps when packets are inserted into the UM. In current P4 behavioral model [29], among the timestamps exposed, the closest to our need is the timestamp when the switch starts processing the packet ($ingress\_global\_timestamp$). This choice sacrifices the time spent in the ingress pipeline, which, however, is stabler and smaller (sub-microsecond) than the queuing delay, so it is also useful for congestion detection. We believe in future programmable switch design, exposing timestamp at arbitrary stages in the pipeline is possible.

Pre-filtering. In P4, every packet go through the whole pipeline, including the ones dropped by tables. So we can pre-filter the table-dropped packets by inserting them into only the DM (not the UM). In this way, these packets will cancel out with the UM in the previous hop, without affecting decoding other lost packets. We add a metadata $drop$ with a default value 0 for each packet, and update it to 1 in the dropping actions of the tables. At the UM, we use a matching field to filter out packet with $drop = 1$.

Header modification. LOSSRADAR needs the header to carry batchID and optionally the timestamp and meterID (if in a blackbox case). We either store these data in unused header fields (e.g. VLAN, DSCP) or add a header, and modify parsers or add a parser for these data accordingly.

Each hop needs to modify the batchID and timestamp in the header to the ones the UM uses. However, this modification cannot be performed before or at the UM, because after the UM, the DM needs the previous hop's batchID and timestamp, which is carried by the header. Thus, we store the new batchID and timestamp in a metadata at the UM, and copy the metadata to the header after the DM finishes the update.

7. EVALUATION

Our evaluation focuses on answering five key questions:

1. What is the memory and bandwidth usage of LOSSRADAR, and how they compare to other solutions such as FlowRadar [21] and full packet mirroring [38, 13]? We test them with a single pair of switches with traffics of different loads and different loss rates. We use simulation to test up to 100G traffic load. The result shows that LOSSRADAR always uses less memory than FlowRadar in large data centers where the number of flows is large and less bandwidth than full mirroring.

2. Under realistic network traffic, how effective are the detection and the root cause inference of LOSSRADAR? We run a network with 80 switches and 128 hosts in ns-3 simulations and evaluate LOSSRADAR with different traffic, different loss patterns and loss rates. The result shows that we only need a few KB per digest to capture more than 99.99% lost packets. Our root cause inference algorithm can capture all the root causes within a few 100s ms.

3. What is the memory usage per switch for different deployments? We evaluate a full deployment and different types of blackboxes cases, based on the same topology, traffic and loss problems in the previous question. The result shows that the memory usages per switch in blackbox cases are higher than in a full deployment, but are still acceptable.

4. How fast is the loss detection? Our result in a testbed with Open vSwitch [28] on a FatTree topology shows that the detection delay is only 12 ms.[9]

5. Is the information LOSSRADAR provides useful? We show two applications that use the loss details to improve performance and identify corrupted rules.

[8]For brevity, we conceptually treat the two arrays for respective $xorSum$ and $count$ as one array.

[9]We evaluate the delay with Open vSwitch rather than P4 behavior model due to performance concerns.

Figure 6: Bandwidth usage: LossRadar vs. NetSight

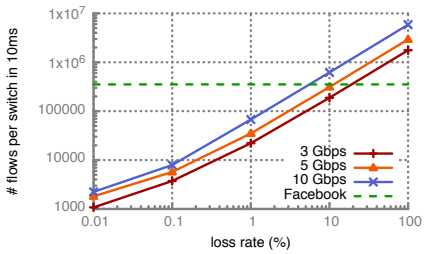


Figure 7: Memory usage: LossRadar vs. FlowRadar.

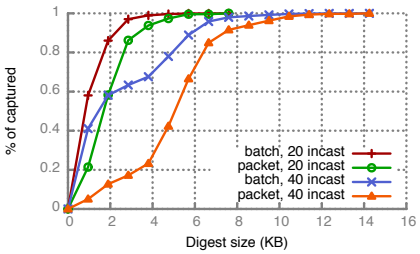


Figure 8: Memory usage under realistic traffic

7.1 Bandwidth and memory overhead

We set up a simulation with a simple topology where there are two switches. By default, we use 10G traffic, and the average packet size is 175B according to Facebook's data center traffic [33]. We change the traffic volume and the number of concurrent flows during the experiment for comparison with different state-of-the-art solutions. For LossRadar, each meter in LossRadar reports the digests every 10ms. Because it is possible that we fail to decode a digest, the size of the digest we tested is large enough so that it can decode all losses with 99% probability.

Bandwidth usage of LossRadar is much less than full packet mirroring. Both Everflow [38] and NetSight [13] capture packets at every hop in the entire network and sends them to a centralized analyzer. While Everflow mirrors raw packets, NetSight introduces compression to reduce the overhead. Thus we compare with NetSight's best compression (Van Jacobson style compression plus gzip compression), which is reported 10Bytes per packet. In order to detect every loss, we configure it to capture every packet.

We evaluated the bandwidth usage with different traffic volume from 4Gbps to 100Gbps, and two loss rates 0.1% and 1%, and show the percentage of LossRadar bandwidth usage to full packet mirroring bandwidth usage in Figure 6. The result shows that LossRadar saves 95% and 99.5% of the bandwidth usage, for 1% and 0.1% loss rates respectively. For example, for 10Gbps traffic, full packet mirroring uses 513Mbps, while LossRadar only uses 2.9Mbps for 0.1% loss rate, and 25.6Mbps for 1% loss rate.

LossRadar saves memory when the number of concurrent flow is large or loss rate is low, compared with FlowRadar. FlowRadar [21] maintains per-flow counters at switches. It can get the loss counter by comparing the packet counters across hops. Its memory usage is proportional to the number of concurrent flows.

We perform experiments to find the number of concurrent flows, loss rate and throughput at which LossRadar and FlowRadar need the same amount of memory, and plot the curves in Figure 7. The red, orange and blue curves are for 3 Gbps, 5 Gbps and 10 Gbps (which is also a 40G link at 25% utilization) throughput respectively. In the area above the curves LossRadar uses less memory; below the curves FlowRadar uses less memory.

We also calculate and plot the average number of flows in 10ms at a cluster switch in Facebook. In Facebook, each server talks to 250 different racks in 5ms on average, and a cluster has 4 cluster switches and at least 64 racks. Thus, assuming each rack has 44 hosts (assuming the top of rack switch has 48 ports, 4 of which is connected to cluster switches), the total number of concurrent flows on average could be 352K in 10ms at a cluster switch. The green dotted line represents this number. This means that as long as the loss rate <5% at 10 Gbps, LossRadar uses less memory than FlowRadar. At a lower throughput, LossRadar can even support a higher loss rate (up to 10% and 20% for 5 Gbps and 3Gbps) while using less memory than FlowRadar.

7.2 Detection and inference effectiveness

We evaluate the fraction of losses being detected given different amounts of memory and the accuracy of the root cause inference, under realistic traffic with injected random drops and blackholes.

We run larger networks and higher link speed in the ns-3 simulator. We simulate a k=8 FatTree topology, which includes 16 core switches, 8 pods (8 switches per pod), and 128 hosts, all connected with 10G links. The simulated switch per-packet processing time is 5 us (not including queueing delay). We deploy 1536 meters in total at switches and hosts. To get realistic packet inter-arrival times, traffic distribution across switches, and realistic loss patterns, we implement DCTCP [4] and ECMP in the simulator. We take the same workload distribution from production data center [4], but add 1000 partition-aggregate queries per second to generate incast-related congestion losses. We set the default incast degree (i.e., the number of senders to one receiver in a query) as 20. The query traffic characteristics also follow the study in [4]. By default, we set the total traffic volume as 40% of the input bandwidth (i.e., 40% $\times$ 1280Gbps in our topology), and a network-wide loss rate of 0.1%. (Note that the per switch port loss rate can go much higher.)

We also inject blackhole and random drops. We select 3 bidirectional links to have 1% random drops, one for each layer (host to ToR, ToR to Aggr, Aggr To Core). We also select 3 switches (a ToR, an Aggr and a Core) to have blackholes. The blackhole is blocking a randomly selected specific destination IP address. Both the random drops and the blackholes are

	t_{gap} (us)	n_{burst}	n_{bb}	n_{nb}	p_{syn}	p_{rand}
Default	20	5	30	5	0.9	0.5
Range	[10,30]	[3,5]	[20,50]	[5,10]	[0.85,0.95]	[0.45,0.55]

Table 3: Root cause inference parameter settings

inserted in the middle of the simulation (not at the beginning). Moreover, we choose the time to inject the blackholes so that they will drop some ongoing large flows, in order to have all types of loss patterns (discussed in section 5.1.1).

7.2.1 Loss detection accuracy

LOSSRADAR needs a small digest size to capture most losses. We evaluate different digest sizes and show the percentage of lost packets being captured in Figure 8 for different incast degrees. A captured lost packet means the packet is retrieved from the digest. With a 20 incast degree, we just need 2KB digest size to capture around 90% of lost packets. Note that LOSSRADAR can still get the right number of losses at each switch, but only provide details for 90% of these packets. We need 7.6 KB to capture the details for more than 99.99% of the lost packets. With a 40 incast degree, the network-wide loss rate grows from 0.1% to 0.3%. Thus we need 14.25 KB to capture more than 99.99% of the lost packets.

LOSSRADAR can capture most batches successfully with a small digest. We also evaluate the percentage of batches where *all* lost packets in the digest of that batch are successfully retrieved (Figure 8) as compared with all the batches with at least one loss here. With a 40 incast degree, we need 6 KB per digest to capture around 90% of the batches. The percentage of captured batches is more than the percentage of captured lost packets because LOSSRADAR sometimes fails to decode some packets when there are too many lost packets in a batch. For these batches, it is ok to just get the total numbers of losses because they have large numbers of losses and operators should inspect the entire switch port.

7.2.2 Root cause inference accuracy

We run our root cause inference on all the losses we get from the trace, and classify each loss into congestion loss, random drop, blackhole loss, or unknown reason. We set the digest size to 7.6 KB according to section 7.2.1. We set the parameters according to the guideline described in Section 5.1.2, as shown in Table 3. We compare the inference result with the ground truth, and show the precision and recall of the three types of losses in Figure 9. We also vary each parameter within a range while keeping other parameters with default values; the error bars in the figure shows the range of the precisions and recalls over all these different settings.

The precision is close to 100%. We design the algorithm to keep the false alarm rates of blackholes and random drops low. The high precision shows our algorithm achieves this goal. The precision of blackholes is 100%, which means there is no false alarm.
With different parameter settings, the precisions of random drops are all above 97.6%. The precision of blackhole losses are almost all 100%, except when $n_{bb} = 20$ it becomes 94.7%. We inspect this case, and find that all false positives are 27 consecutive losses of a flow in a congestion. The reason for such long consecutive losses is similar to the outcast problem [31]. However, this phenomenon is exaggerated in simulation because all delays are deterministic.

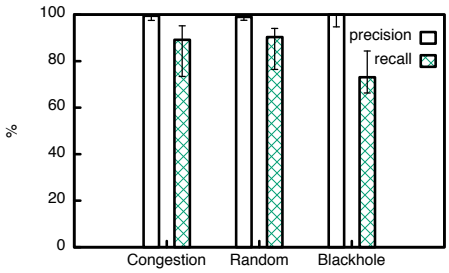


Figure 9: Root cause inference accuracy

Every injected problem is captured. We correctly classify 90.3% of the random drops and 73.1% of the blackhole losses. The missed blackhole losses are all short bursts (e.g. 5 losses) of consecutive losses, so they are classified as congestion losses. The missed random drops have two types: (1) a small number of random drops happen closely (e.g. 3 random drops with gaps less than 10 ms), so they are classified as congestion losses; (2) during some intervals, there are other losses being classified as unknown together with random drops, so the distribution is uneven, and we classify all the losses in these intervals as unknown. However, because the blackholes and the random drops are all persistent, we always correctly classify the losses caused by them in the next few intervals. Thus, the classified losses are sufficient for us to detect all problems.
With different parameter settings, most losses of each type are correctly classified, and we detect all problems as well.

7.3 Memory usage in partial deployments

We evaluate the memory usage per switch in different deployment scenarios based on the same topology, traffic pattern and injected problem in Section 7.2.
With a full deployment, we need 243.2 KB for all the digests in each switch[10]. We also evaluate the memory usage when we do not have access to some parts of the network, which is treated as blackboxes. In data centers, upgrading a ToR switch requires notifying applications or tenants running under the ToR on the planned downtime, or migrating the applications or tenants to other racks. On the other hand, upgrading an aggregation or a core switch is easier, because the traffic can be automatically rerouted to other paths when the switch is shut down without affecting applications.
Thus we evaluate the case that we have access to aggregation and core switches but treat ToR switches as blackboxes. We

[10]7.6 KB for each digest, two digests (batches) per meter, two meters (UM and DM) per port and eight ports.

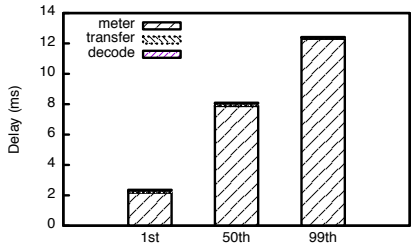


Figure 10: Breakdown of loss detection latency

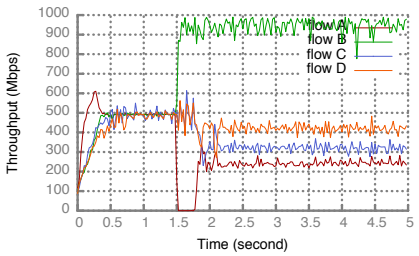


Figure 11: Recovery from blackhole

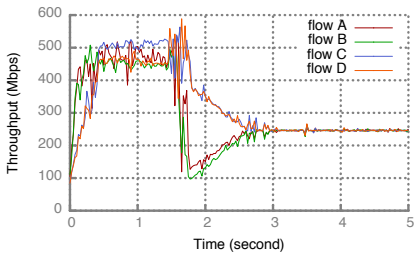


Figure 12: Recovery from random drops

deploy more meters at the devices around the ToR switches. On an aggregation switch, each of the 4 ports that connect to ToR switches receives packets from the 4 hosts under the ToR switch, which requires 4 downstream meters; each port that connects to a core switch only needs 1 downstream meter. So each aggregation switch needs 20 downstream meters, plus 8 upstream meters (one per port). We need 425.6 KB for all the digests in each aggregation switch.

We also evaluate other scenarios. If each aggregation switch is a blackbox, we need 608 KB in each core switch and each ToR switch. If each core switch is a blackbox, we need 608 KB in each aggregation switch.

7.4 Detection latency

We evaluate the detection latency in a testbed. We build a topology of k=4 FatTree in Deterlab [9] with our modified version of Open vSwitch [28] that supports LossRadar. Every 10 ms the upstream switch sends a batch to the collector. The downstream switch keeps each batch for 12 ms ($Timeout = 2$ ms). The size of each digest is set to 5 KB.

Detection latency is bound by the batch length plus the timeout. We break down the detection delay into meter waiting time, transfer time and decoding time (Figure 10). The meter waiting time dominates, during which the packets dropped, but its digest has not been sent out. The transfer time and the decoding time is negligible. For the waiting time, the 99th percentile of packets are close to 12 ms, bound by the waiting time at the downstream.

7.5 LossRadar applications

Improving flow throughput. We run four iperf flows (A, B, C, D) through two ECMP paths in our FatTree testbed, with two flows on each path. The two paths share the same edge and aggregation switches but differ at the core. We inject two types of losses. One is blackholing flow A (Figure 11). LossRadar quickly identifies that flow A is experiencing blackhole, and installs an entry matching flow A at the aggregation switch to moves A's path to the other. The throughput recovers after 250ms. The other case is randomly dropping 1% packets on A and B's link (Figure 12). LossRadar detects random drops on this link, so it moves both flows to the other path.

Correlating lost packets with ACLs. We use ClassBench ACL rules [34] and staggered data center traffic matching these rules as used in [24]. ClassBench has been shown to generates rules representative of real-world access control with accept and deny actions. The traffic is created by first generating staggered traffic for VMs and then assigning VMs with IP address ranges that match the ACL rules. The details are described in [24]. We take four rule sets of 50K, 100K, 150K, and 200K rules. We run the experiment 100 times by flipping a different random bit each time. We install the rules at a single switch, and send the traffic through the switch.

We compare the upstream and downstream traffic digests collected to get the set of lost packets. Given the set of lost packets, our loss analyzer first excludes the intended drops based on the correct set of rules, and then runs flow space correlation algorithm to narrow down the set of candidate rules that may cause the loss. The whole process takes 0.7-2.5ms. Our result shows that we have zero false negatives (i.e., the rule with flipped bit is always in our reported set). From 50K to 200K rules, we narrow down to 2-2.5 candidate rules that may get bit flip. Sometimes a high priority rule which covers all the losses is treated as a candidate rule but sometimes it is not the corrupted rule.

8. DISCUSSION

Avoid modifying packet header fields. It is possible to avoid tagging headers with the batch ID (section 3.4) and the meter ID (section 4.2), by tolerating on-the-fly packets.

Without the batch ID and the meter ID to synchronize batches, an on-the-fly packet would appear in two different batches at the upstream meter and the downstream meter, so it would appear as a difference in both the two batches' subtractions. Thus, we need to compare the results across batches and identify the true losses (only appear once not twice). The number of batches to compare depends on the time a packet takes from the upstream to the downstream, which is less than an RTT—100s of microseconds in data centers. Thus, as long as the batch length $T > 1$ ms, we only need to wait for one more batch to conclude a packet is lost or on-the-fly. Now the downstream digest is no longer a subset of the upstream digest. We leverage the original IBF design [10], which can handle two-way set difference. We also need larger digests, because the difference also include the on-the-fly packets besides the losses. The new trend of more synchro-

nized clocks across switches (e.g. down to sub-microsecond level [20]) can help reduce the extra size. We leave how to further reduce the size to future works.

Improving root cause inferences. Our root cause inference algorithms take the first step in identifying root causes for packet losses. There are a few ways to improve the inference: One way is to use machine learning instead of threshold-based conditions to better classify different types of losses with dynamics. Another way is to incorporate the header patterns of blackholes (e.g. specific 5-tuple or specific dest IP) to help better distinguishing blackhole losses from other types. In addition, when a previously unknown type of losses happen, how the analyzer correctly reports the existence of an unknown type is an open question.

Reducing memory usage for incremental deployment cases. In a blackbox case, each downstream meter needs to maintain one digest for each upstream meter, and the size of each downstream digest has to be equal to the size of an upstream digest, which results in a high memory usage per switch. If we can partition the the upstream digest based on which downstream each packet goes to, the downstream digests can be smaller. This requires path-awareness at the upstream; we leave it to future works.

9. RELATED WORK

We discussed host passive mirroring [37, 25], Pingmesh [12], Mirroring [38], and FlowRadar [21] in Section 2.2. FlowRadar is the closest work to LOSSRADAR, but differs in four aspects: (1) FlowRadar collects per-flow counters at each switch, which can also be used for counting the aggregated number of losses per flow. In contrast, LOSSRADAR provides details of individual losses. (2) The memory usage of FlowRadar is proportional to the number of flows, while LOSSRADAR's is proportional to the number of losses. Thus, the memory usage of LOSSRADAR is lower than FlowRadar when the loss rate is low, as shown in Section 7.1. (3) To detect loss, FlowRadar needs to compare flow counters between two switches, and thus has to wait till an idle reporting interval (at least 10ms) to synchronize the two counters. In contrast, LOSSRADAR immediately detects the packet loss soon after a reporting interval. (4) FlowRadar uses Invertible Bloom Lookup Table [11] which also uses multiple hash functions per item and has a similar decoding process to IBF [10]. The key difference is that LOSSRADAR utilizes the subtraction operation to cancel out the packets between upstream and downstream meters. FlowRadar cannot use the subtraction, because each flow's 5-tuple would cancel out in the subtraction, resulting in useless data.

Some other research solutions are available for loss detection. LDA [18] leverages packet counters to identify losses, but cannot identify individual lost packets and their flow information. NetSight [13] can detect packet loss by sending the packet headers of all the packets at each hop to a centralized controller, but incurs significant bandwidth overhead. In contrast, LOSSRADAR provides detailed loss information with low memory and bandwidth overhead.

Many research works on network verification [15], [16], [17] performs static analysis of forwarding tables, and thus can detect blackholes caused by misconfiguration. Complementary to network verification, LOSSRADAR capture all lost packets in the network in real time.

The paper [6] allows the switch to send to end hosts the packet header information of those packets that are dropped due to congestion. Rather than focusing on one type of losses (misconfiguration or congestion), LOSSRADAR is generic to all types of packet losses.

Our traffic digest is inspired by Invertible Bloom Filters (IBFs). IBFs were introduced in [10] to identify *two-way* set differences for application-layer systems such as peer-to-peer networks and link-state databases. Our loss detection is a *one way* set difference problem, where one set is always a subset of the other. Therefore, we simplify the IBF design with XOR-based codes and counters, but store more flow information in the XOR-based codes. Moreover, because our data structure is used at switches with high-speed links, we face new challenges of synchronizing the data collection across nodes, and identifying the lost packets among multiple upstream and downstream meters around a blackbox. Counter Braids [22] are novel data structures to store *estimated* flow-based counters with low memory usage, but does not work for loss detection because it does not provide accurate counts and does not store flow information with counters.

10. CONCLUSION

With diverse types of packet losses in data center networks, and their significant impact on application performance and network operations, it is important to detect these losses in time and with detailed information of individual lost packets. We presented LOSSRADAR, a generic, light-weight loss detection system that can identify individual lost packets, their locations, and the flows they belong to in near real time. LOSSRADAR is easy to implement with programmable switch hardware, and can capture all the individual lost packets with small memory and bandwidth overhead. We also propose root cause inference algorithms based on the loss information from LOSSRADAR.

11. ACKNOWLEDGEMENT

We thank our shepherd Marco Canini and CoNEXT reviewers for their helpful feedbacks. We thank Sylvia Ratnasamy, George Varghese and Ori Rottenstreich for inspiring discussions. We thank Antonin Bas for the help on the P4 implementation. This paper is partially supported by the NSF grants CNS-1453662, CNS-1423505 and CNS-1413972.

References

[1] https://aphyr.com/posts/288-the-network-is-reliable.

[2] http://www.eweek.com/cloud/amazon-us-east-cloud-goes-down.html.

[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. "CONGA: Distributed Congestion-aware Load Balancing for Datacenters". In: *SIGCOMM*. 2014.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data Center TCP (DCTCP)". In: *SIGCOMM*. 2010.

[5] Weifeng Chen, Yong Huang, Bruno F. Ribeiro, Kyoungwon Suh, Honggang Zhang, Edmundo de Souza e Silva, Jim Kurose, and Don Towsley. "Exploiting the IPID Field to Infer Network Path and End-system Characteristics". In: *PAM*. 2005.

[6] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. "Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Centers". In: *NSDI*. 2014.

[7] Jeff Dean. "Designs, Lessons and Advice from Building Large Distributed Systems". In: *LADIS keynote*. 2009.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's highly available key-value store". In: *SIGOPS*. 2007.

[9] *DeterLab*. https://www.isi.deterlab.net/.

[10] D. Eppstein, M. Goodrich, F. Uyeda, and G. Varghese. "What's the Difference? Efficient Set Difference without Prior Context". In: *SIGCOMM*. 2011.

[11] Michael T. Goodrich and Michael Mitzenmacher. "Invertible Bloom Lookup Tables". In: *arXiv:1101.2245v2*. 2011.

[12] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis". In: *SIGCOMM*. 2015.

[13] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks". In: *NSDI*. 2014.

[14] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. "Dynamic Scheduling of Network Updates". In: *SIGCOMM*. 2014.

[15] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks". In: *NSDI*. 2012.

[16] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. "Real Time Network Policy Checking Using Header Space Analysis". In: *NSDI*. 2013.

[17] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. "VeriFlow: Verifying Network-Wide Invariants in Real Time". In: *NSDI*. 2013.

[18] R. Kompella, K. Levchenko, A. Snoeren, and G. Varghese. "Every Microsecond Counts: Tracking Fine-Grain Latencies with a Loss Difference Aggregator". In: *SIGCOMM*. 2009.

[19] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. "What You Need to Know About SDN Flow Tables". In: *PAM*. 2015.

[20] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. "Globally Synchronized Time via Datacenter Networks". In: *SIGCOMM*. 2016.

[21] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. "FlowRadar: A Better NetFlow for Data Centers". In: *NSDI*. 2016.

[22] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement". In: *SIGMETRICS*. 2010.

[23] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm". In: *SIGCOMM Comput. Commun. Rev.* 1997.

[24] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. "Scalable Rule Management for Data Centers". In: *NSDI*. 2013.

[25] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. "Trumpet: Timely and Precise Triggers in Data Centers". In: *SIGCOMM*. 2016.

[26] *NetFlow*. http://www.cisco.com/go/netflow/.

[27] *NTP FAQ*. http://www.ntp.org/ntpfaq/NTP-s-algo.htm#Q-ACCURATE-CLOCK.

[28] *Open vSwitch*. http://openvswitch.org/.

[29] *P4 behavioral model*. https://github.com/p4lang/behavioral-model/blob/master/targets/simple_switch/simple_switch.cpp.

[30] *Packet loss impact on TCP throughput in ESNet*. http://fasterdata.es.net/network-tuning/tcp-issues-explained/packet-loss/.

[31] Pawan Prakash, Advait Dixit, Y. Charlie Hu, and Ramana Kompella. "The TCP Outcast Problem: Exposing Unfairness in Data Center Networks". In: *NSDI*. 2012.

[32] *Precision time protocol*. http://www.ieee1588.com/.

[33] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. "Inside the Social Network's (Datacenter) Network". In: *SIGCOMM*. 2015.

[34] D. E Taylor and J. S Turner. "ClassBench: A Packet Classification Benchmark". In: *Transactions on Networking* 15.3 (2007).

[35] W. Vogels. *Performance and Scalability*. http://www.allthingsdistributed.com/2006/04/performance_and_scalability.html. 2009.

[36] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. "NetPilot: Automating Datacenter Network Failure Mitigation". In: *SIGCOMM*. 2012.

[37] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. "Profiling Network Performance for Multi-tier Data Center Applications". In: *NSDI*. 2011.

[38] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. "Packet-Level Telemetry in Large Datacenter Networks". In: *SIGCOMM*. 2015.