Masking Corruption Packet Losses in Datacenter Networks with Link-local Retransmission

Raj Joshi[†], Cha Hwan Song[†], Xin Zhe Khooi[†], Nishant Budhdev[‡], Ayush Mishra[†],
Mun Choon Chan[†], Ben Leong[†]
[†]National University of Singapore [‡]Nokia Bell Labs

ABSTRACT

Packet loss due to link corruption is a major problem in large warehouse-scale datacenters. The current state-of-the-art approach of disabling corrupting links is not adequate because, in practice, all the corrupting links cannot be disabled due to capacity constraints. In this paper, we show that, it is feasible to implement link-local retransmission at sub-RTT timescales to completely mask corruption packet losses from the transport endpoints. Our system, LinkGuardian, employs a range of techniques to (i) keep the packet buffer requirement low, (ii) recover from tail packet losses without employing timeouts, and (iii) preserve packet ordering. We implement LinkGuardian on the Intel Tofino switch and show that for a 100G link with a loss rate of 10^{-3}, LinkGuardian can reduce the loss rate by up to 6 orders of magnitude while incurring only 8% reduction in effective link speed. By eliminating tail packet losses, LinkGuardian improves the 99.9^{th} percentile flow completion time (FCT) for TCP and RDMA by 51x and 66x respectively. Finally, we also show that in the context of datacenter networks, simple out-of-order retransmission is often sufficient to significantly mitigate the impact of corruption packet loss for short TCP flows.

CCS CONCEPTS

• **Hardware** → **Failure recovery, maintenance and self-repair**; • **Networks** → **In-network processing**; **Physical links**; *Data center networks*; *Programmable networks*; Link-layer protocols.

KEYWORDS

Packet corruption, Link failures, Optical links, Link-local retransmission, Programmable switches, In-network packet loss recovery

ACM Reference Format:
Raj Joshi, Cha Hwan Song, Xin Zhe Khooi, Nishant Budhdev, Ayush Mishra, Mun Choon Chan and Ben Leong. 2023. Masking Corruption Packet Losses in Datacenter Networks with Link-local Retransmission . In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3603269.3604853

1 INTRODUCTION

Optical links are commonly used as switch-to-switch links in modern datacenter networks [61]. Unfortunately, external factors such as physical damage, bending, or contamination due to airborne dirt particles, can cause *optical attenuation* and make optical links susceptible to data transmission errors [15, 61]. As a result, packet losses due to corruption on optical links in large warehouse-scale datacenters are common. Alibaba's recent study of hundreds of real-world service tickets showed that about 18% of the packet drops that caused network performance anomalies (NPAs) were due to packet corruption [59]. Another large-scale study across 15 Microsoft datacenters consisting of 350K optical links showed that the number of packets lost due to corruption is comparable to those lost due to congestion [61].

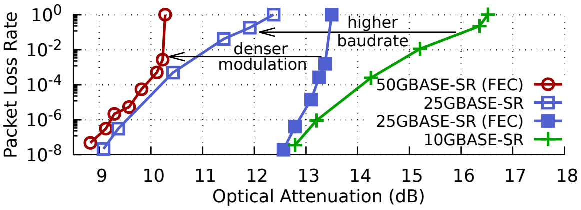


Figure 1: Effect of optical attenuation on various Ethernet link speeds (1518 B frames).

At the same time, Ethernet link speeds continue to increase, having increased from 25G [26] in 2016 to 400G [29] in recent years. This increase has been achieved through a combination of using multiple parallel PHY lanes, higher baudrate, and denser modulation. Figure 1 shows the result of a measurement experiment (details in §2) where we can see that, as the link speeds continue to increase through the use of higher baudrate (from 10G to 25G) and denser modulation (from 25G to 50G), optical links are becoming more susceptible to optical attenuation and thus corruption packet loss.

Optical corruption can only be remedied by physically repairing the damaged links, which can take between several hours to days [61]. During this time, the impact of corruption can only be *mitigated*. The current state-of-the-art approach to mitigate corruption packet loss is to disable the corrupting links while maintaining a certain minimum network capacity [56, 61]. However, this approach is not sufficient, as it is often not feasible for some corrupting links to be disabled without violating capacity constraints. Such links will continue to cause packet drops thereby negatively impacting both throughput and latency-sensitive flows. Data from Microsoft datacenters shows that up to 15% of the corrupting links cannot be disabled under realistic capacity constraints [61].

In this paper, we apply the classical loss recovery strategy of link-local retransmission for mitigating corruption packet loss in datacenter networks. Link-local retransmission has been studied extensively [8, 9, 44] and deployed widely in wireless networks [1, 2, 23, 24]. It has desirable properties such as the recovery overheads are proportional to the corruption loss rate and are localized to only the corrupting link. It can achieve sub-RTT recovery,

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604853

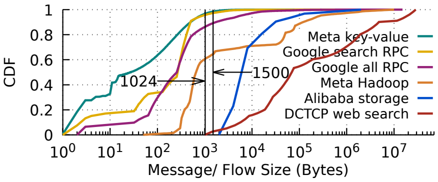


Figure 2: Flows size distribution of several datacenter workloads from 2008 to 2019 [3, 7, 34, 47, 52].

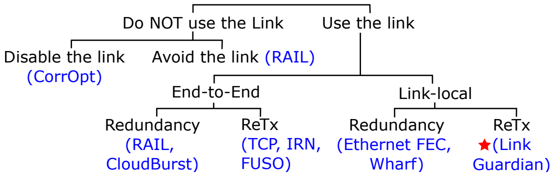


Figure 3: Design space for handling corrupting links in datacenter networks.

and since it is agnostic to the end-hosts, it is amenable to any transport protocol including RDMA. Yet, despite these advantages, link-local retransmissions have never been deployed in the context of datacenter networks to the best of our knowledge.

We suspect that this is because deploying link-local retransmission in datacenter networks is challenging for the following reasons: first, link-local retransmission requires packet buffering while datacenter switch buffers are generally small. The problem is further exacerbated by high link speeds that will generally require more buffering. Second, most flows in datacenter networks are short (see Figure 2), which increases the probability of tail packet loss. Such tail losses need to be detected and recovered at microsecond scales to provide bounded tail FCT guarantees and meet the stringent Service Level Agreements (SLAs) [13, 37, 54, 59]. Third, RDMA is being widely deployed in modern datacenters [19, 21, 37, 60] which is more sensitive to packet reordering than TCP [22]. Therefore, packet ordering needs to be preserved while performing link-local retransmission.

In this paper, we show that, with modern programmable switches, it is now feasible to implement link-local retransmission in datacenter networks. Our system, *LinkGuardian*, can overcome the above challenges by implementing the following mechanisms: (1) a fast and efficient (low overhead) loss detection and recovery protocol to keep the recovery delay and thus the buffering requirement small (§3.1 and §3.4); (2) a novel mechanism to detect tail packet losses quickly and efficiently using a *self-replenishing* queue of "dummy packets" without the need for a timeout (§3.2); and (3) a "reordering buffer" at the receiver switch to maintain packet ordering along with a *backpressure* mechanism to ensure that the buffer does not overflow (§3.3). While individually these techniques are relatively straightforward, our key insight is that their combination is *sufficient* to make link-local retransmission *feasible* in modern datacenter networks.

Conventional wisdom says that link-local retransmissions need to preserve packet ordering to prevent the transport layer from triggering spurious loss recovery and reduction of the sending rate [3, 4, 8, 10, 60]. We will show that in the context of datacenters, it is not *always* necessary to preserve packet ordering (§4.3). The key insight is that most flows in datacenter networks are short [37, 46] and most flows fit within one packet requiring only 1 RTT to complete [37] (see Figure 2). When a flow fits within a single packet, we do not need to worry about ordering for both TCP and RDMA. For multi-packet TCP flows, out-of-order retransmission can still provide significant corruption loss mitigation for TCP flows at 100G speeds even if we cannot retransmit within TCP's reordering window. This is because even when a TCP flow spans multiple packets, it lasts only a few RTTs (flows being short). This means that if there is a corruption loss, it mostly occurs just once and thus reordering happens at most once which has minimal impact on the FCT (§4.4). To this end, we show that a non-blocking variant of LinkGuardian (that implements out-of-order retransmission) not only has lower overheads but can scale better to higher link speeds (§4.1). However, for multi-packet RDMA flows, we currently still need to preserve packet ordering due to its go-back-N transport recovery.

LinkGuardian is currently implemented on an Intel Tofino switch and our testbed evaluation shows that (i) for a 100G link with a loss rate of 10^{-3}, LinkGuardian can reduce the loss rate by up to 6 orders of magnitude while incurring only 8% reduction in the link's effective link speed and requiring less than 90 KB of packet buffer; and (ii) LinkGuardian improves the 99.9th percentile FCT for TCP and RDMA by 51x and 66x respectively by handling tail packet losses at sub-RTT timescales. Furthermore, LinkGuardian is complementary to existing solutions for handling corrupting links. By augmenting CorrOpt [61] (current state-of-the-art) with LinkGuardian, we can reduce the total loss rate in a large datacenter network by at least 4 orders of magnitude, and allow network operators to operate the network at a higher average capacity that was not previously possible.

The main limitation of our current implementation is that recirculation is used for packet buffering because of hardware constraints (Tofino). With more advanced hardware like the Tofino2 [33], it will be possible to implement LinkGuardian more efficiently.

2 BACKGROUND & RELATED WORK

There is a large body of literature on the mitigation of network faults. In particular, we lay out the design space for mitigating the impact of corruption packet loss in Figure 3 and discuss below the tradeoffs involved in previous approaches.

Why can't we simply disable/avoid the faulty links? The current state-of-the-art approach to deal with corrupting links is indeed to disable or avoid them [56, 61]. Doing so, however, reduces network capacity, and therefore links can only be disabled as long as the capacity constraints of the network are not violated.

Network capacity constraints are specified as the minimum number of valley-free paths from a top-of-rack (ToR) switch to the highest level (spine) of the network [61]. In Figure 4, we show the configuration for a typical "pod" from Facebook's state-of-the-art datacenter fabric network [5], where each ToR switch has 192 (4 fabric switches × 48 uplinks) paths to the spine layer. If the capacity constraint is 75% and link A starts corrupting packets, it can be easily disabled and sent for repair as every ToR switch will lose only 1 out of 192 paths to the spine layer. However, if link B also starts corrupting packets while link A is being repaired (which can take 2 to 4 days), link B cannot be disabled since by doing so switch 1 will

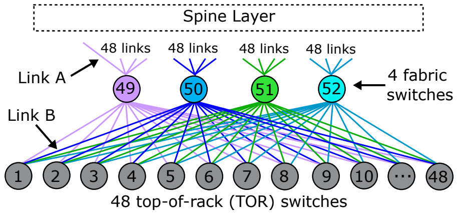


Figure 4: A single pod from Facebook's state-of-the-art datacenter fabric network [5].

lose more than 25% of paths to the spine and violate the capacity constraint.

A recent study of Microsoft datacenters by Zhuo et al. showed that under realistic capacity constraints, about 15% of the corrupting links *cannot be disabled* [61]. Zhuo et al. hence proposed a solution called *CorrOpt* that finds a subset of corrupting links that can be disabled such that the impact of the *remaining* corrupting links is minimized. It is also possible to *avoid* the corrupting links via source routing or by using virtual network topologies (e.g. RAIL [62]). However, again due to capacity constraints, it is not always possible to *avoid* the faulty links.

Why not rely on end-to-end recovery? Since there is currently no way to eliminate corruption losses, recovery is left to the end-to-end transport protocol (TCP/RDMA) by default. However, as shown in Figure 2, most flows in datacenters fit within a single packet and complete within 1 RTT under normal conditions. For such flows, under 10^{-3} corruption packet loss rate, we found that the 99.9th percentile FCTs increase by 66x and 51x when using RDMA and DCTCP, respectively (see Figure 10 in §4.3). In other words, when the flows are very short, high-tail FCTs become more likely since the corrupted packets are more likely to be the "tail" packets that cause retransmission timeout (RTO).

Using an adaptive RTO [39] with NIC-offloaded [6, 39, 50] and/or multipath [11] transport stacks as well as explicit probing (RACK-TLP [12]) can reduce the recovery delay in case of tail packet loss. However, the fundamental limitation of any end-to-end recovery is that it cannot completely eliminate the use of RTO to detect tail packet losses and even the most aggressive RTO cannot be lowered below 1 RTT. Lim et al. proposed a timeout-less design to handle tail packet loss due to congestion, but it does not help with corruption [35]. LinkGuardian, on the other hand, does not employ timeouts and performs corruption loss recovery at sub-RTT timescales.

End-to-end recovery can also be achieved via end-to-end forward error correction (FEC) [57, 62] or packet duplication [53]. However, this adds encoding/decoding latency and also risks worsening congestion by adding redundant bytes for *all* the packets across the *entire* path. Further, the required decoding at the receiving end makes it off-limits for supporting one-sided RDMA operations where no CPU is involved on one end.

Why not use link-local FEC or specialized transceivers? The Ethernet standards for 25G/100G [25, 26] and 50G/200G/400G [27, 28] specify optional and compulsory FEC at the PHY layer respectively. However, the redundancy parameters are fixed in the current standards and cannot be adjusted according to the loss rate. To investigate the effectiveness of Ethernet FEC, we followed the methodology proposed by Zhuo et al. [62] to add a configurable optical attenuation on an OM4 grade fiber using a Variable Optical Attenuator (VOA). We then measured the packet loss rates using pairs of three different transceivers – 10GBASE-SR [17], 25GBASE-SR (with and without FEC) [16], and 50GBASE-SR [18]. As shown in Figure 1, the state-of-the-art 50GBASE-SR suffers significantly from optical attenuation even with FEC. The trends in Figure 1 suggest that as link speeds are increasing using higher baudrate and denser modulation, the effectiveness of Ethernet's built-in FEC is diminishing. It is possible that future Ethernet standards could include a runtime configurable adaptive FEC. However, to the best of our knowledge, currently, there is no hardware support to do so at 100's of Gbps link speeds. Besides, FEC leads to increased per-hop latency for *all* the packets including those that are not affected by corruption [49].

Table 1: Corruption loss rates observed in Microsoft Datacenters [61].

Loss Bucket	% Links
$[10^{-8}, 10^{-5})$	47.23%
$[10^{-5}, 10^{-4})$	18.43%
$[10^{-4}, 10^{-3})$	21.66%
$[10^{-3}+)$	12.67%
Total	100%

Wharf [20] uses link-local FEC at the level of an Ethernet frame (L2). Its main drawback is that the redundancy is added to *all* the packets even when the corruption loss rates are very small (see Table 1). Furthermore, it performs meter-based packet dropping to signal reduced link capacity which may not work well with delay-based transports [32, 38] and most definitely will not work well with loss-sensitive RDMA. Wharf requires FPGA support on switches, and it is unclear if the expensive frame-level FEC encoding/decoding can scale to higher link speeds (>=100G).

RADWAN [51] uses bandwidth variable transceivers (BVTs) to dynamically reduce/adapt the modulation rate (PHY link speed) for WAN links based on the optical attenuation. While Figure 1 suggests that such an approach could work for intra-datacenter optical links, BVTs are currently not used for intra-datacenter links as they are much more bulky and expensive compared to the small form-factor pluggable (SFP) optical transceivers.

Has anyone else tried link-local retransmission in datacenters? For Infiniband networks, LLR [41] is an NVIDIA proprietary feature that breaks an Infiniband L2 datastream into "cells" and performs cell-level retransmission for links that are not longer than 30m. For Ethernet networks, SQR [45] performs link-local retransmission to recover packet loss during fail-stop link failures, but it does not work for corrupting links. LinkGuardian hence represents a new and unexplored point in the solution design space for handling packet corruption in (Ethernet-based) datacenter networks.

Our prior workshop paper [30] investigated the potential of this general idea by implementing out-of-order retransmission within the TCP's reordering window of 3 packets on 10G links. In this paper, we build upon that work to show that out-of-order retransmission outside the TCP's reordering window can still be effective at 100G speeds. Furthermore, our prior work was a work-in-progress and it did not describe a complete solution that: (i) completely masks the corruption packet loss with in-order retransmission (and is hence amenable to RDMA); (ii) handles tail packet loss; (iii) handles consecutive packet loss; (iv) works at high link speeds; and (v) can be deployed effectively on a large-scale network. Therefore, to the best of our knowledge, LinkGuardian is the first complete solution for mitigating corruption packet loss in datacenter networks using link-local retransmission.

3 LINKGUARDIAN

The corruption loss rates in real-world datacenters tend to be small (see Table 1). This makes it possible for LinkGuardian to *mitigate* the impact of corruption packet loss using link-local retransmission. To *detect* link corruption, we use a low-cost control plane scheme (called *corruptd*) that continuously monitors all optical links in the network (see Appendix C) and activates LinkGuardian once a link is found to be corrupting packets. Until it is activated, LinkGuardian lies dormant and imposes no cost on the network.

In this section, we provide an overview of LinkGuardian's design by describing a basic link-local retransmission (LL-ReTx) scheme, the challenges of implementing LL-ReTx at line rates, and, finally, the key ideas that make LL-ReTx practical in the context of datacenter networks.

Basic LL-ReTx. LinkGuardian can be modeled as a protocol[1] running between a "sender" switch and a "receiver" switch (see Figure 5). The sender adds a monotonically increasing sequence number (seqNo) to the transmitted packets and buffers a copy of the recently sent packets (in the Tx buffer). These sequence numbers are used by the receiver to detect corruption packet losses. When there is no packet loss (seqNo 1-2), the receiver piggybacks the cumulative ACK information on top of reverse direction traffic (Ack2). The sender then drops the buffered copies of successfully delivered packets (seqNo 1-2). In case of a corruption packet loss (seqNo 3 in Figure 5), the receiver detects the gap in the sequence numbers when it receives the subsequent packet (seqNo 4). The receiver then sends a high-priority loss notification to the sender (Lost3) and the sender retransmits the packet with seqNo 3 using a high priority queue. We provide further details on the basic LL-ReTx protocol in Appendix A.

Challenges. While this basic LL-ReTx scheme is sufficient to achieve LL-ReTx, it is not practical in a datacenter because of the following reasons:

(1) **Small buffers:** Since the switches in datacenter networks have shallow buffers, the sender needs to receive the ACKs quickly so that it can drop the buffered packets fast enough to keep the Tx buffer usage small. If we piggyback the ACKs naively, they could get delayed by an arbitrary amount depending on the reverse direction traffic.
(2) **Short flows:** Since most datacenter flows are short (see Figure 2), mostly 1 packet, it is not always possible to detect the loss of such packets based on the gap in the sequence numbers. In Figure 5, if the packet with seqNo 5 belonging to a short flow is lost, then the basic LL-ReTx scheme cannot detect the same until a subsequent packet (seqNo 6) is transmitted. This can lead to high-tail FCTs.
(3) **RDMA flows:** The use of RDMA in datacenters networks is now becoming increasingly commonplace [19, 21, 37, 60]. Compared to TCP, RDMA performance is very sensitive to packet ordering due to the lack of a "reordering window" [22]. The basic LL-ReTx above does not preserve the original packet ordering e.g. when seqNo 3 is lost in Figure 5.

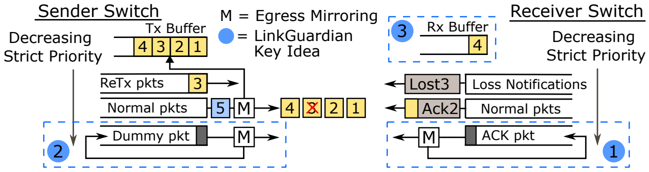


Figure 5: LinkGuardian Design Overview.

LinkGuardian incorporates three key ideas on top of the basic LL-ReTx scheme to address these challenges and make it practical in datacenter networks:

(1) **Self-replenishing queue of ACK packets (§3.1):** LinkGuardian implements a strictly low-priority queue with one ACK packet at the receiver switch (① in Figure 5). This means that there will always be packets in the reverse direction even when there is no reverse direction traffic to piggyback the ACKs.
(2) **Self-replenishing queue of dummy packets (§3.2):** LinkGuardian also implements a similar strictly low-priority queue of dummy packets at the sender switch (② in Figure 5). The dummy packets get sent out as soon as there is no regular traffic to allow the receiver to quickly detect tail packet losses (e.g. seqNo 5 in Figure 5).
(3) Reordering Buffer without Overflow (§3.3): To preserve packet ordering, LinkGuardian implements a reordering buffer on the receiver (③ in Figure 5). A naive design would result in buffer overflow at today's datacenter link speeds. To prevent this, we use a backpressure algorithm to throttle the sender when necessary.

Scope and assumptions. Our goal is not to completely eliminate corruption packet loss because it is too costly to achieve such a guarantee. Instead, we focus on the more modest goal of reducing the corruption packet loss rate to an operator-specified target level. To achieve the target effective loss rate, LinkGuardian also handles the case that the retransmitted copy of the packets could get lost too (§3.4). For the following sections, we assume that a corrupting link corrupts packets only in one direction which is the case with 91.8% of corrupting links in production [61]. However, we should highlight that handling bidirectional corruption would require only minor modifications which we describe in §5.

Operation modes. LinkGuardian in its *default* mode preserves packet ordering. However, we also allow running LinkGuardian in a simple mode called LinkGuardianNB, where we disable the mechanism that maintains packet ordering. Our results in §4.3 show that LinkGuardianNB is effective in mitigating corruption packet loss for short TCP flows because of the small flow sizes as well as TCP's support for reordering window and selective recovery.

3.1 Fast ACKs to prevent buffer overflow

When there are no corruption losses, the sender uses the ACK information from the receiver to clear its buffer by dropping the buffered packets that were successfully received. Therefore, the receiver must send the ACK information as soon as possible to keep the Tx buffer overhead low. While this can be achieved by maintaining a continuous stream of ACK packets, it would add

[1] Note that the protocol runs per link (per port) rather than per flow.

Algorithm 1: De-Duplication & In-Order Recovery

```
Apply to: protected, protected-reTx, recirculating rx-buffered pkts
if pkt.seq_no == ackNo then
    forward();
    ackNo = ackNo + 1;
else if pkt.seq_no > ackNo then
    mark_pkt_as_rx_buffered();
    recirculate(); // will be subjected to the algo again
else if pkt.seq_no < ackNo then
    drop(); // de-duplication
```

significant overhead in the reverse direction. The overhead can be minimized by piggybacking the ACK information on regular packets, but this can cause the ACK signal to be delayed when there is no traffic in the reverse direction.

To address this problem, we introduce a novel *self-replenishing* ACK packet queue that has a strictly lower priority compared to the normal packet queue at the receiver (see Figure 5). The ACK packet queue is initialized with a single minimum-sized explicit ACK packet which will be sent as soon as the normal packet queue is empty. When the normal packet queue is not empty, the ACK information would be piggybacked on a normal packet. In addition, every time an explicit ACK packet is sent, we replenish the queue by adding a new explicit ACK packet back to the same queue using egress mirroring.

3.2 Detecting Tail Losses for Single-Packet Flows

Single-packet flows are common in datacenters [7, 34, 37, 47, 52]. Since losses can only be detected at the receiver from the gap in the sequence numbers, when the last packet before a short break in the transmission is corrupted and lost, the receiver would not detect the loss until the packet transmission resumes. The most common approach to detect such tail losses is to employ retransmission timeouts [48]. However, in order to avoid spurious retransmissions, retransmission timeouts are required to be set conservatively considering worst-case delays [35]. To eliminate the need for a timeout, we add another *self-replenishing* queue at the sender with a single "dummy" packet that has a strictly lower priority compared to the normal packet queue (see Figure 5). Each time when the normal packet queue at the sender is empty, the "dummy" packet will be transmitted and the gap in sequence numbers can be detected immediately at the receiver.

3.3 Reordering Buffer without Overflow

To preserve packet ordering after a corruption loss is detected, the receiver will need to buffer the subsequent out-of-order packets until the retransmission is received from the sender switch. We implement this buffering by using a recirculation port queue as the "reordering buffer" (Rx Buffer in Figure 5). Packets received after the lost packet are buffered using recirculation, and this means that we need a way to ensure that the packets are forwarded in the right order after the lost packet is received from the sender. Furthermore, if extra copies of the retransmitted packet were to be received (§3.4), the extra copies need to be dropped (*de-duplication*). We achieve this by using a single state variable called `ackNo` which determines the correct next packet to be forwarded ahead. The (protected) packets from the sender as well as the receiver-buffered packets are continuously checked against the `ackNo` and sent back into the recirculation queue until it is their turn to be forwarded. The pseudo-code for this is shown in Algorithm 1.

Algorithm 2: Backpressure Mechanism

```
Input: curr_qdepth; // recirculation port's queue size
Initialization: curr_state = resume;
if curr_qdepth >= pauseThreshold && curr_state == resume then
    send_pause();
    curr_state = pause;
else if curr_qdepth <= resumeThreshold && curr_state == pause then
    send_resume();
    curr_state = resume;
```

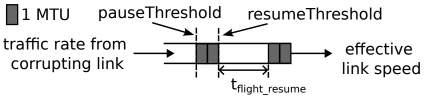


Figure 6: Logical view of receiver-side reordering buffer (recirculation port queue).

Since each retransmission takes a small but non-negligible delay, the reordering buffer will keep filling up with each packet loss if the subsequent packets continue to arrive at line rate, and eventually, the reordering buffer would overflow. To prevent this, we employ a backpressure mechanism where the receiver switch sends pause/resume messages to the sender switch. We only pause/resume the normal packet queue on the sender switch (see Figure 5) so as to not affect the retransmission of the lost packets. The underlying principle is that we want to pause the transmission of the normal packet queue on the sender just enough to keep the reordering buffer usage on the receiver switch to a small non-zero value which we set as 2 MTU (see Figure 6).

We note that after the receiver *decides* to send a resume message, there is a short delay called t_{flight_resume} before the normal packet queue on the sender is resumed. The `resumeThreshold` is therefore set to a value such that during the t_{flight_resume} time, the reordering buffer will not be fully emptied (see Figure 6). Since the thresholds in our backpressure mechanism are similar *in spirit* to the PFC-based backpressure, we follow DCQCN's recommendation [60] to set the `pauseThreshold` by leaving 2 MTU worth of space as hysteresis (see Figure 6). The overall backpressure mechanism is described in Algorithm 2. Essentially, a pause message is sent when the buffer level reaches the `pauseThreshold`; and a resume message is sent when the buffer falls below the `resumeThreshold`. Since Algorithm 2 operates on a per-packet basis, we use a flag `curr_state` to avoid sending redundant pause/resume messages.

We note that LinkGuardian's backpressure mechanism is not always activated because datacenter link utilization is typically low – less than 30% for ~85% of time [58]. It is activated only if a corruption packet loss occurs during a high (>90%) utilization burst which lasts long enough for the reordering buffer to build up to the `pauseThreshold`.

3.4 Mitigating Potential ReTx Losses

If the link corruption rate is high, it is plausible that a retransmitted packet might also be lost. Therefore, to improve the odds of a successful retransmission, the sender retransmits not one, but multiple (N) copies of a buffered packet in response to a loss notification.

Since the original packet was already transmitted (and lost), the total number of copies transmitted for the lost packet would be $N+1$, giving us an effective loss rate of $(actual\_loss\_rate)^{(N+1)}$. Since our goal is not to completely eliminate corruption packet losses but to reduce the effective loss rate to an operator-specified target level, we have the following relation:

$$(actual\_loss\_rate)^{(N+1)} <= (target\_loss\_rate) \qquad (1)$$

For example, if a target loss rate of 10^{-8} is desired by a network operator and the actual loss rate on a corrupting link is 10^{-4}, then retransmitting a single copy of the buffered packet ($N = 1$) would suffice to achieve an effective loss rate of 10^{-8}. Now, solving Equation 1, we get the number of retransmitted copies (N) as:

$$N => \frac{log_{10}(target\_loss\_rate)}{log_{10}(actual\_loss\_rate)} - 1 \qquad (2)$$

Since N is an integer in practice, we assign N the next integer value by doing a *ceil* on the RHS term of Equation 2. Also, note that since the loss rates are typically very low (Table 1), this strategy to retransmit multiple copies adds a very small overhead.

3.5 Implementation Details

LinkGuardian is implemented on an Intel Tofino programmable switch in about 1,800 lines of P4 code and runs entirely in the dataplane. For each packet to be protected, the sender switch adds a 3-byte LinkGuardian data header, consisting of a 16-bit `seqNo` and other metadata: the `seqNo era` and the packet type (original or retransmitted). To piggyback the ACK information on the reverse direction traffic, the receiver switch adds a similar 3 byte LinkGuardian ACK header. During bootstrapping, the *self-replenishing* queues of the dummy and the ACK packets are initialized by injecting a single minimum-sized packet from the switch control plane. All the state variables are maintained on a per-port basis using SRAM-based register memory. By default, LinkGuardian preserves ordering (§3.3) and provides a runtime option to switch to the *non-blocking* mode (LinkGuardianNB) where ordering is not preserved.

Backpressuring the normal packet queue. The normal packet queue on the sender switch can be paused or resumed (§3.3) using Tofino2's advanced flow control primitives [33]. However, since our current implementation is on Tofino, we use PFC pause/resume frames. Specifically, the receiver switch generates PFC pause/resume frames as dictated by Algorithm 2, which are then absorbed and processed by the RX MAC of the corrupting link on the sender switch. We note that such an implementation does not risk a PFC storm or deadlock since the normal packet queue on the sender switch does not further generate any PFC pause/resume frames.

Handling `seqNo` Wrap-around. We handle `seqNo` wrap-around by including an additional "era bit" along with the sequence number which toggles between 0 and 1 each time the sequence number wraps around. We perform an "era correction" when comparing two sequence numbers belonging to different eras, where we subtract a constant N/2 from both the sequence numbers (N is the sequence number range). This works correctly as long as the two different-era sequence numbers are not more than N/2 apart.

Handling consecutive packet losses. To decide which packets to retransmit, the sender switch maintains a lookup table `reTxReqs` which is updated by the receiver (details in Appendix A.1). When consecutive packets are lost, multiple entries in `reTxReqs` need to be updated simultaneously by the loss notification packet. If `reTxReqs` is implemented as a single register, such a simultaneous update is not possible due to hardware limitations. Therefore, we implement `reTxReqs` across multiple 1-bit registers (details omitted for brevity) where the number of registers required is equal to the maximum number of consecutive packets lost. In our current implementation, we provision 5 1-bit registers (across 2 pipeline stages) which based on our measurement results (details in Appendix B.2) can handle 99.9999% of corruption loss events at an unreasonably high packet loss rate of 5%.

Preventing transmission stalls. In spite of our best efforts, there is still a small but non-zero probability that a retransmission will not be successful. Because we buffer packets at the receiver until the retransmission for the missing packet is received, this could stall the transmission indefinitely and cause the reordering buffer to overflow. To handle this rare but potentially fatal event, we implement a timeout called `ackNoTimeout` at the receiver. If a retransmission does not occur within the timeout period, the receiver ignores the lost packet, increments the `ackNo`, and continues with the remaining packet transmissions. The `ackNoTimeout` is set to a value greater than the maximum expected delay in receiving the retransmission after a packet has been found to be lost (details in Appendix B.1). To update the `ackNo` at the receiver when there is an ACK timeout (see §3.3), we use periodic packets from the switch's packet generator for timekeeping [31]. In our implementation, we set the rate of these timer packets to 10 Mpps (~1% of the switch's pipeline processing capacity).

Packet Generation. To create multiple copies of a buffered packet during retransmission (in case of a high loss rate), the sender switch uses the multicast primitive. Upon detecting a loss, the receiver switch uses ingress mirroring to generate the loss notifications. Whenever PFC pause/resume packets need to be sent by the receiver, we modify the timer packets and send them to the sender switch.

3.6 Repairing Corrupting Links in Practice

Recall that LinkGuardian is activated on a link only when the link is found to be corrupting packets (§3). However, if we only enable LinkGuardian and do nothing to repair the corrupting links, then over a long period of time (~1-2 years), we might end up having LinkGuardian activated on the majority of links in a large datacenter network. Therefore, as a long-term strategy for maintaining the network, periodically, we will need to bring down the corrupting links so that they can be repaired.

A simple way to do this is to run an algorithm like CorrOpt [61] to safely schedule LinkGuardian-enabled links for repair without violating capacity constraints. In particular, when a link starts corrupting packets, we immediately enable LinkGuardian on it to reduce the effective loss rate to an acceptable rate. Then we run CorrOpt's fast checker algorithm to check if the link can be safely disabled and scheduled for repair. If so, we disable the link and schedule for repair. Otherwise, the link continues to operate with corruption while LinkGuardian mitigates the impact on application performance. As links get enabled again after their repair is complete, we run CorrOpt's optimizer algorithm to see if any of the

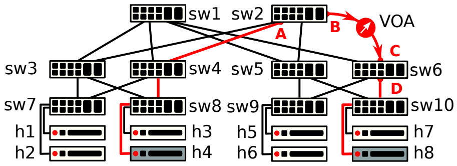


Figure 7: Testbed w/ Variable Optical Attenuator (VOA).

LinkGuardian-enabled corrupting links can be safely disabled and scheduled for repair. What this joint strategy also demonstrates is that instead of being in competition with previously proposed algorithms, LinkGuardian is complementary to them.

4 EVALUATION

In this section, we present our evaluation results for LinkGuardian (LG) and its out-of-order recovery variant LinkGuardianNB (LG_NB). In particular, we seek to answer the following questions:

(1) How effective is LinkGuardian at masking the corruption packet losses? Are we able to reduce the effective loss rate to the operator-specified target as desired? And what is the corresponding reduction in link speed? (§4.1)

(2) How does LinkGuardian interact with the end-point transport protocols? (§4.2)

(3) How well does LinkGuardian handle tail packet loss and improve FCTs for short and single-packet flows? (§4.3, §4.4)

(4) How do the various mechanisms of LinkGuardian contribute to its good performance? (§4.5)

(5) How much buffering do we need and what are the associated overheads and costs of deploying LinkGuardian? (§4.6)

(6) How does LinkGuardian's performance compare with Wharf [20], the state-of-art link-local FEC solution? (§4.7)

(7) When deployed in a large-scale network, how effective is LinkGuardian in reducing the corruption packet loss and improving the overall network capacity? (§4.8)

Testbed Setup. We use the testbed setup shown in Figure 7, where sw2 and sw6 are connected by an OM4 grade fiber optical fiber link. Depending on the experiment, links are either all 25G or all 100G. sw2 and sw6 act as the LinkGuardian sender and receiver respectively and we restrict their recirculation buffers to 200 KB. Following the methodology used in [62], we introduce corruption packet loss on the link between sw2 and sw6 using a VOA. We set LinkGuardian's target loss rate[2] to 10^{-8} and the number of retransmitted packet copies is then determined by Equation 2 depending on the actual loss rate.

Using the switch control plane, we poll the port counters for ports denoted by A, B, C and D in Figure 7. These counters enable us to measure the sending rate/throughput of an endpoint sender, the actual loss rate incurred due to the VOA, and the effective loss rate and link speed achieved by LinkGuardian. Before starting an experiment, we measure the actual loss rate by sending 1B MTU-sized packets across the corrupting link and checking the difference between counters B and C. We also poll the queue occupancies on sw2 and sw6 using the local control plane.

The servers are equipped with Intel Xeon Silver/Gold CPUs, 128 GB memory, NVIDIA CX5 and CX6-DX NICs (25G/100G), and run Linux kernel 5.4.0-91-lowlatency on Ubuntu 20.04.3. For our experiments, we use kernel-based DCTCP and NIC-based RoCEv2 (RDMA) transports. For TCP, TSO, SACK, RACK-TLP, and ECN (100 KB marking threshold [14]) are enabled and RTO_{min} is set to 1 ms. The network RTT for a TCP sender is ~30 μs. For RoCEv2, we use a one-sided RDMA_WRITE operation using NIC-based reliable delivery (RC [42]) which we found to have a RTO of ~1 ms.

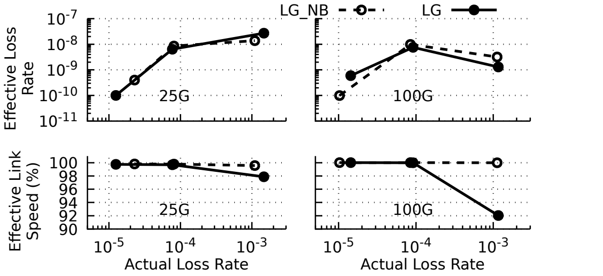


Figure 8: Effective loss rates achieved by LinkGuardian (LG) and LinkGuardianNB (LG_NB) and the effective link speeds.

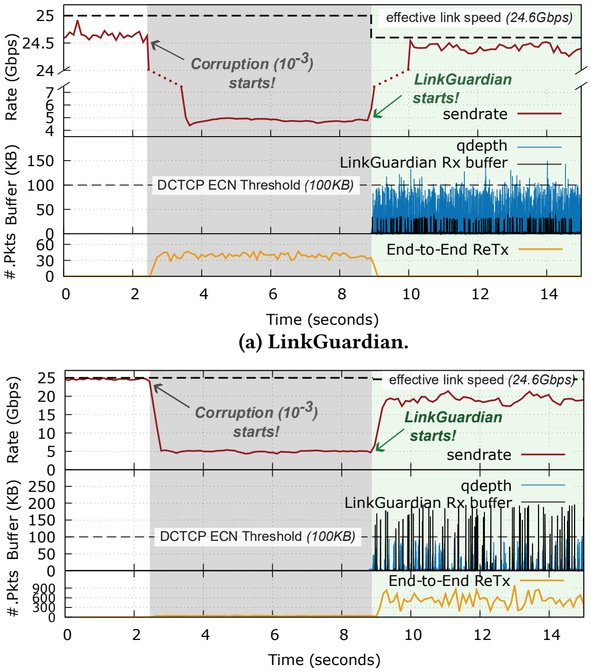
Parameters. LinkGuardian uses 3 parameters: ackNoTimeout, resumeThreshold, and pauseThreshold. As discussed in §3.5, we set the ackNoTimeout to 7.5 μs and 7 μs as we found the maximum retransmission delays to be 6 μs and 5.5 μs for 25G and 100G links respectively. For the resumeThreshold (§3.3 and Figure 6), we measured the maximum t_{flight_resume} values to be 1.9 μs and 1.6 μs for 25G and 100G links respectively. Therefore, we conservatively set the resumeThreshold at 40 KB and 37 KB for 25G and 100G links respectively as the recirculation-based buffer drains at 100G. Since we use a fixed hysteresis of 2 MTU, the pauseThreshold is resumeThreshold + 2 MTU. We provide more details on parameter tuning in Appendix B.1.

4.1 Effective Loss Rate & Link Speed

Using the packet generator on sw2 (see Figure 7), we conduct a "stress test" by sending MTU-sized packets at line rate and evaluate LinkGuardian using three representative loss rates observed in production (see Table 1): 10^{-5}, 10^{-4}, and 10^{-3}. As prescribed by Equation 2, LinkGuardian retransmits 1, 1, and 2 copies for each lost packet for these loss rates, respectively. This should theoretically result in loss rates of 10^{-10}, 10^{-8}, and 10^{-9}, respectively. In Figure 8, we plot the observed (effective) loss rates achieved by LinkGuardian and the corresponding effective link speeds for 25GBASE-SR and 100GBASE-SR4 links. We observe that, except for the 25G link with 10^{-3} loss rate, the effective loss rates for both LinkGuardian and LinkGuardianNB closely match the theoretically expected loss rates. For the 25G link at the 10^{-3} loss rate, our investigations showed that the corruption losses are not independent and identically distributed (i.i.d.) and we suspect that this is the reason why the effective loss rate deviates from the theoretically expected loss rate of 10^{-9}. However, it is still very close to the target loss rate of 10^{-8}.

For effective link speed, we see that LinkGuardianNB scales much better to higher link speeds and higher loss rates compared to LinkGuardian while achieving similar effective loss rates. This is because, unlike LinkGuardian, LinkGuardianNB does not preserve packet ordering and therefore does not incur intermittent pauses in the link transmission. Nevertheless, for a 100G link with a high

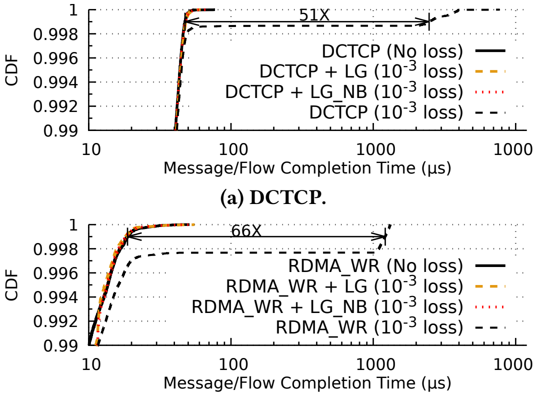
[2] For MTU-sized packets, a loss rate of 10^{-8} corresponds to a bit error rate (BER) of 10^{-12} which is considered a healthy/non-corrupting link [62].



(a) LinkGuardian.

(b) LinkGuardian with backpressure mechanism disabled.

Figure 9: Performance of LinkGuardian for DCTCP on a 25G link with 10^{-3} loss.



(a) DCTCP.

(b) RDMA WRITE

Figure 10: Top 1% FCTs for 143B flows on a 100G link.

actual loss rate of 10^{-3}, LinkGuardian can reduce the loss rate by up to 6 orders of magnitude while incurring only an 8% reduction in the link's effective link speed while preserving packet ordering.

Timeouts in practice. Recall from §3.5 that when LinkGuardian preserves ordering, it implements an `ackNoTimeout` which is triggered when LinkGuardian fails to recover a lost packet i.e. when all the retransmitted copies are lost. We used a simple counter on the receiver switch to measure the total number of timeouts across all the LinkGuardian (LG) experiments in Figure 8. We found that for all the packet loss events (~31M), only 476 (0.0016%) of them had timeouts. This confirms that the `ackNoTimeout` is merely a fallback mechanism that occurs rarely in practice.

4.2 Impact on Transport Protocols

Our high-level goal is to mask the corruption packet losses from the transport layer. While we showed in §4.1 that LinkGuardian can reduce the effective loss rates, what matters is the net impact on transport protocols. To understand the impact of LinkGuardian, we send single flow TCP traffic from `h4` to `h8` using `iperf` with all links set to 25G. We evaluate three different TCP variants: CUBIC, DCTCP, and BBR, as they use congestion loss, ECN, and delay as congestion signals respectively. We consider BBR to be representative of delay-based transport protocols since the implementations for TIMELY [38] and Swift [32] are not readily available.

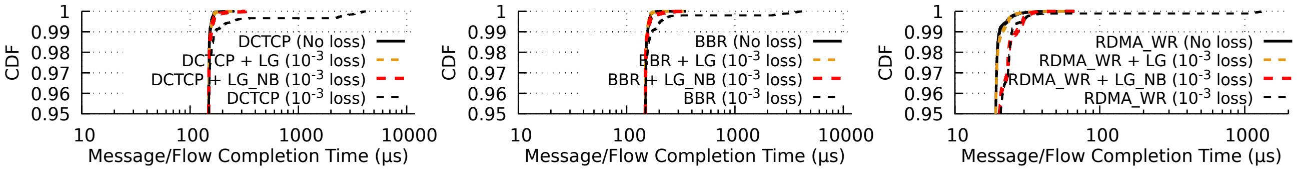
In each experiment, we start the setup with no corruption loss. At the 2-second mark, we introduce a loss rate of 10^{-3} on the link, and approximately 5 seconds later, we enable LinkGuardian. We plot the results for DCTCP in Figure 9a. The effective link speed in the figure is measured separately by sending a line rate UDP flow under the same experiment conditions. We see that the throughput is reduced sharply once corruption losses are introduced. Upon enabling LinkGuardian, the corruption losses are eliminated and the throughput returns to a level comparable to the effective link speed. We also notice that the slightly lower effective link speed leads to a build-up in the flow's buffer at the sender switch (shown as "qdepth") triggering ECN marking. This result also demonstrates that since LinkGuardian only deals with packets *transmitted* on the link, it works well even if the link has congestion. Overall, we see that LinkGuardian's backpressure mechanism is effective at keeping its receiver-side buffer occupancy (labeled as "Rx buffer") low. We observe similar results with CUBIC and BBR (see Appendix B.3).

Backpressure Not Considered Optional. In Figure 9b, we also plot the results when the backpressure mechanism is disabled. We now see a large number of end-to-end retransmissions because the reordering buffer (Rx buffer) periodically builds up and overflows. In fact, the observed packet losses after enabling LinkGuardian are so severe that the random corruption packet losses in the period between 2 and 8 seconds are barely visible in Figure 9b. The throughput is also lower compared to the earlier results shown in Figure 9a. In other words, the backpressure mechanism is critical for ensuring that the buffering at the receiver switch works as intended.

4.3 Tail Packet Loss and Short Flows

One-packet Flows. To evaluate how effectively LinkGuardian handles tail packet losses, we measure the FCT of 143 B DCTCP and RDMA write (RDMA_WR) flows (300K trials) in our testbed with all links set to 100G while introducing a corruption loss rate of ~10^{-3}. 143 B is the most frequent flow size in the Google all RPC workload [52]. It is clear from our results in Figure 10 that both LinkGuardian and LinkGuardianNB are able to mask the corruption losses so effectively that the performance at 10^{-3} loss rate becomes indistinguishable from the case when the link is lossless. LinkGuardian and LinkGuardianNB achieve the same performance since we do not need to worry about ordering in case of single packet flows. We note that the result in Figure 10 is also representative of all other flow sizes for workloads in Figure 2 that fit within a single packet.

Multi-packet Flows. Next, we repeat the experiment with 24,387 B-sized flows which is the most frequent flow size in the



(a) DCTCP **(b) BBR** **(c) RDMA WRITE**

Figure 11: Top 5% FCTs for 24,387B flows (17 packets) on a 100G link.

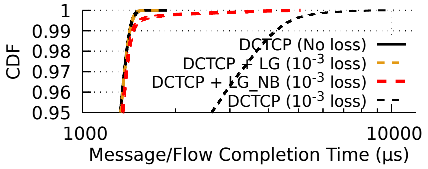


Figure 12: Top 5% FCTs for 2MB DCTCP flows on a 100G Link

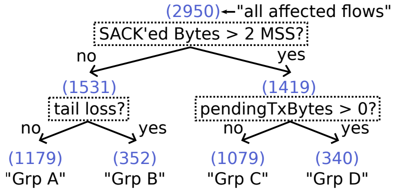


Figure 13: Classification of "affected" 24,387B DCTCP flows with LG_NB. Numbers denote the number of flows (trials).

DCTCP web search workload [3]. We plot the results when using DCTCP, BBR, and RDMA_WR transports in Figure 11. We can see that the lines for LinkGuardian and no loss mostly overlap. While BBR is mostly agnostic to packet loss, this experiment shows that corruption packet loss does affect the FCTs of short BBR flows and therefore mitigating corruption loss is necessary for BBR and similar rate-based/loss-agnostic transport protocols. In Figure 11, we also see that for RDMA, LinkGuardianNB provides no improvement over the loss case other than preventing RTO by handling tail packet losses. This is because RDMA's NIC-based reliable delivery has no reordering tolerance and LinkGuardianNB does not cause any reordering when it recovers the tail packet loss. On the other hand, for DCTCP and BBR, LinkGuardianNB performs nearly as well as LinkGuardian except at very high percentiles (> 99.9th) where it performs marginally worse.

To evaluate the performance with somewhat longer flows, we repeat the FCT experiment with 2MB flows which is the maximum flow size in the Alibaba Storage workload [34]. Figure 12 shows the results for DCTCP transport. We can see that both LinkGuardian and LinkGuardianNB perform similarly for 2MB flows as they did for 24,387B flows (Figure 11a). We expect the results for 2MB flows while using BBR and RDMA transports to be qualitatively similar to Figures 11b and 11c, respectively. While not shown in Figure 12, the solid and dotted black lines start to diverge around 20th percentile indicating ~80% flows were affected by packet corruption.

In summary, both LinkGuardian and LinkGuardianNB improve the 99.9th percentile FCT for single packet DCTCP and RDMA flows by 51x and 66x respectively. For 24,387B flows, the 99.9th percentile gains for LinkGuardian are 19x for DCTCP and BBR, and 39x for RDMA. On the other hand, for 2MB DCTCP flows, LinkGuardian and LinkGuardianNB improve the 99.9th percentile by 4x and 2x respectively. While LinkGuardianNB performs similarly to LinkGuardian for multi-packet TCP flows (up to 99th percentile), it provides little benefit in case of reordering-sensitive RDMA but does eliminate the long tail FCTs due to RTOs.

4.4 Why does out-of-order recovery work for TCP?

To understand why LinkGuardianNB works well for TCP, we studied the packet traces of the 24,387B DCTCP flows that received at least one SACK while LinkGuardianNB performed out-of-order recovery (Figure 11a). We then classified these affected flows into 4 different groups as shown in Figure 13. The condition "SACK'ed bytes > 2 MSS" refers to the case where a DCTCP sender received sufficient SACK'ed bytes to reduce in its cwnd [3, 10]. The "tail loss?" condition refers to the case where a packet was lost (and recovered by LinkGuardianNB) within the last 3 packets of the flow. pendingTxBytes refers to the remaining bytes of the flow that a DCTCP sender is pending to transmit when it received the SACK.

We found that for the flows in group A, LinkGuardianNB was able to exploit the gaps in packet transmissions (TSO segments) to retransmit the lost packet within TCP's reordering window such that the total SACK'ed bytes were less than 2 MSS, and the cwnd was not reduced. The flows in group C received more than 2 MSS of SACK'ed bytes but only *after* the DCTCP sender had finished sending the entire flow. Overall, the flows in groups A and B did not experience a reduction in cwnd due to ≤ 2 MSS SACK'ed bytes while the flows in group C suffered cwnd reduction but the FCT was not affected since there was no data remaining to be sent. Only the flows in group D (a small fraction) experienced somewhat higher FCT because the flows had pending bytes when they received > 2 MSS SACK'ed bytes. However, we found that the pending bytes were no more than 7 MSS, and therefore the impact on FCT was rather limited. A similar explanation was found for LinkGuardianNB's good performance with 2MB DCTCP flows (Figure 12). However, in this latter case, the fraction of flows in group D is relatively larger because of the larger flow size, and there were flows with ~1.3K MSS pending bytes which experienced a much higher increase in FCT leading to the longer tail for LinkGuardianNB in Figure 12.

For BBR, there is no reduction in the sending rate since BBR is loss-agnostic. However, BBR performance still improves with LinkGuardianNB by avoiding a 1 RTT delay as well as TCP end-host stack delays arising from end-to-end recovery.

4.5 Contribution of different mechanisms

To understand the contributions of the different mechanisms implemented by LinkGuardian, we repeat the above experiment (24,387 B) with a variant of LinkGuardian implementing only link-local retransmission (ReTx) and then selectively enable LinkGuardian's

Table 2: Top 1% FCT (μs) for 24,387B DCTCP flows for different LinkGuardian mechanisms: tail loss handling ("Tail") and preserving packet order ("Order")

	No Loss	Loss (10^{-3})	ReTx	ReTx +Order	ReTx +Tail	ReTx+Tail +Order
99.00%	152.293	169.044	161.959	161.168	156.627	155.669
99.90%	166.877	3399.743	212.378	193.252	195.588	168.21
99.99%	197.536	4036.167	3606.115	3773.866	314.128	194.085
99.999%	253.207	4159.96	4107.404	4088.288	356.503	235.793
std dev	21.3	172.294	63.695	80.148	22.629	22.286

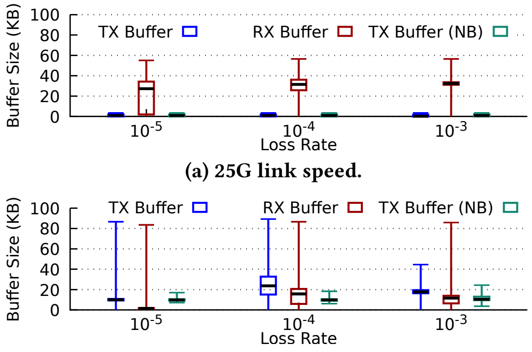
packet order preserving (Order) and tail loss handling (Loss) mechanisms. In Table 2, we show the top 1% FCT results for DCTCP. Simple link-local retransmission improves the 99.9% FCT significantly as it recovers the loss of the 3rd last and the 2nd last packets in the flow which would otherwise cause an RTO due to lack of 3 MSS SACKed bytes. Additionally handling packet ordering only provides marginal gains up to 99.9%. Tail loss handling on the other hand significantly reduces FCT at all top percentiles. Notice that the two right-most columns represent LinkGuardianNB and LinkGuardian respectively, and the additional packet order preserving by LinkGuardian improves the FCT by ~33% at 99.99% and above percentiles thereby nearly matching the performance of the no loss case. Results for BBR and RDMA (omitted for brevity) show similar trends except that for RDMA at 99.9%, ReTx+Order shows 3.75x improvement than ReTx since RDMA is more reordering intolerant compared to TCP. One may erroneously conclude that tail loss handling only helps for FCTs at 99.99% and above. However, our results in Figure 10 show that tail loss handling in crucial for single-packet flows.

We note here that for the loss (10^{-3}) case, performance deficits exist even though RACK-TLP is enabled in our experiments. While the exact reason is under investigation, we believe that this is because for very short flows RACK-TLP does not have a reliable estimate of the network RTT.

4.6 Overhead

In this section, we present the overhead results corresponding to the "stress test" experiments in §4.1 where we run continuous line-rate traffic. These results, therefore, show the "worst case" cost of running LinkGuardian as real-world link utilization exceeds 90% only about 10% of the time [58].

Packet Buffer Usage. LinkGuardian requires a packet buffer at the sender switch (TX buffer) and additionally at the receiver switch (RX buffer) when packet ordering is to be preserved. We used control plane APIs to measure the packet buffer usage which we plot in Figure 14 for 25G and 100G links running at three different loss rates. The key takeaway from these results is that at 25G, the TX and RX buffer usage for LinkGuardian is at most 3.6 KB (~2 MTU) and 60 KB respectively for all evaluated loss rates; at 100G, the TX and RX buffer usage are both at most 90 KB. LinkGuardianNB requires no RX buffer, while its TX buffer requirement is the same as LinkGuardian at 25G and about 3x lower (24.4 KB) at 100G. This is because LinkGuardianNB has no backpressure mechanism that could potentially delay the ACKs. To put these numbers in context, 100G datacenter switches have 16-42 MB of packet buffer [55]. In other words, the required buffering to deploy LinkGuardian is negligible for modern switches.



(a) 25G link speed.

(b) 100G link speed.

Figure 14: LinkGuardian's packet buffer usage for different link speeds and packet loss rates. Whiskers show min, max, 25th, 50th, 75th percentiles.

Table 3: TCP CUBIC goodput (Gb/s) on a 10G Link

Loss Rate →	0	10^{-5}	10^{-4}	10^{-3}	10^{-2}
None	9.49	9.48	8.01	3.48	1.46
Wharf	n/a	9.13	9.13	9.13	7.91
LinkGuardian	9.47	9.47	9.47	9.46	9.2
LinkGuardianNB	9.47	9.47	9.47	9.46	9.2

Protocol Overhead. LinkGuardian adds a 3-byte header to each packet in both forward and reverse (ACK) directions. Since the standard MTU-sized frame is 1,538 octets on wire, this overhead amounts to a ~0.2% degradation of link capacity and occurs only when LinkGuardian is activated. Both the dummy packets and explicit ACK packets do not add any overheads as they are transmitted only when there is no regular traffic.

Recirculation Overhead. Across 3 loss rates and 2 link speeds, we found the worst-case recirculation overhead to be 0.664% of the switch pipeline's processing capacity (more details in Appendix B.4). LinkGuardianNB has the same recirculation overhead on the sender switch but zero on the receiver switch. The key takeaway is that recirculation takes up less than 1% of the switch pipeline's processing capacity, and thus the overhead is negligible for modern switches.

Dataplane Resources. LinkGuardian needs to maintain state in the dataplane on a per-port basis and uses stateful ALUs (SALUs) for stateful operations. With state provisioned for 256 ports, LinkGuardian requires only ~9% of the total SRAM memory and uses ~25% of the available SALUs. We believe that future switches are likely to incorporate more SALUs, while LinkGuardian will be able to support higher link speeds without the need for more SALUs.

We note that, except for the dataplane resources, the above overheads are per LinkGuardian-enabled link. However, the results from our large-scale simulation (§4.8) suggest there could be no more than 2-4 LinkGuardian-enabled links on a switch pipe.

4.7 Comparison with Wharf

Link-local FEC is a natural alternative to link-local retransmissions and therefore we compare LinkGuardian with Wharf [20], the state-of-the-art link-local FEC scheme. We were not able to reproduce Wharf's results experimentally because we did not have access to the required FPGA hardware. In Table 3, we reproduce Wharf's results *numerically* by picking the Wharf FEC parameters

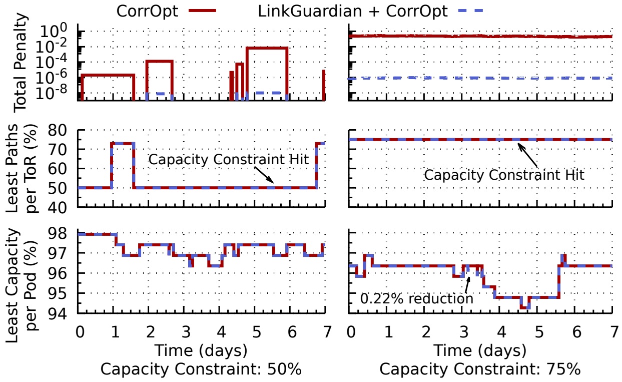


Figure 15: 1-week snapshot of simulation results for FB fabric topology (100K optical links).

that gave their best-reported goodput for each loss rate (c.f. Figure 8 in [20]). In our experiments, we used the same experimental setup as Giesan et al.: 10G link, TCP CUBIC, and Tofino-based random packet dropping. Our results show that both LinkGuardian and LinkGuardianNB compare favorably at all loss rates. For LinkGuardianNB, we observed that it retransmits within TCP's reordering window for the majority of times and thereby prevents the TCP sender from reducing its cwnd below the network BDP.

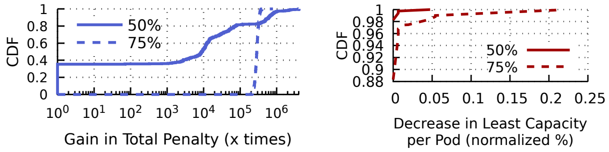
4.8 Effectiveness in large-scale deployment

In this section, we use simulation to evaluate the effectiveness of LinkGuardian when deployed in a large datacenter network. We use the same methodology that was used to evaluate CorrOpt [61] and compare vanilla CorrOpt with LinkGuardian + CorrOpt.

Setup. We contacted the authors of CorrOpt [61] for details on their evaluation setup. However, due to confidentiality reasons, they were unable to provide us with any traces, source code, or topology information. Therefore, we re-implemented their evaluation methodology in about 3K lines of Python code[3]. For the topology, we use the state-of-the-art Facebook fabric [5] (see Figure 4) datacenter network with about 100K switch-to-switch optical links (all 100G) and 1:1 oversubscription ratio[4]. For link corruption trace, we implemented a trace generator that uses the corruption loss rate and link spatial distribution data from Microsoft's datacenters [61], and a per-link Weibull distribution with a mean-time-to-failure (MTTF) of 10K hours [36] (details in Appendix D). When activated, LinkGuardian performs ordered retransmission and the link's effective speed is as per Figure 8. We assume that when sent for repair, 80% of the corrupting links are repaired in ~2 days while the remaining take ~4 days [61].

Evaluation Metrics. We adopted the following metrics used by Zhuo et al. to evaluate CorrOpt [61]:

(1) **Total penalty:** sum of the loss rates for all the active (remaining) corrupting links in the network;
(2) **Least paths per ToR:** the least fraction of paths to the spine (top) layer of the network for the worst-case ToR. This metric captures the impact on per-ToR path diversity as corrupting links are disabled for repair.



(a)

(b)

Figure 16: CDF of (a) Gain in the total penalty; and (b) Decrease in least capacity per pod; for LinkGuardian + CorrOpt compared to vanilla CorrOpt (1 year simulation).

To quantify LinkGuardian's cost which is the reduction in the link's effective capacity, we introduce an additional metric: *Least capacity per pod*, that we define as the total capacity in a network pod from the ToR-layer to the spine (top) layer for the worst-case pod in the network (see Figure 4).

In Figure 15, we plot a 1-week snapshot of a year-long simulation result for capacity constraints (defined in §2) of 50% and 75%. We see that when the capacity constraint (least paths per ToR) is hit, vanilla CorrOpt fails to disable the corrupting links resulting in a higher total penalty. Overall, we see that compared to vanilla CorrOpt, LinkGuardian + CorrOpt reduces the total penalty by about 6 and 4 orders of magnitude for capacity constraints of 50% and 75%, respectively.

To investigate the benefits and costs over the entire simulation period, we plot the CDFs of (a) the gain in total penalty i.e. how many times the combined solution reduces the total penalty compared to vanilla CorrOpt; and (b) the corresponding cost in terms of decrease in the least capacity per pod in Figure 16. We can see that for capacity constraint of 50%, about 35% of the time, there is no difference in the total penalty (gain = 1) as all corrupting links are disabled successfully. However, for the remaining 65% of the time and for nearly all times with 75% capacity constraint, the combined solution offers significant benefits while causing very little reduction in the per pod's capacity to the spine layer (Figure 16b).

Overall, our results demonstrate that when augmented with LinkGuardian, CorrOpt can reduce the total penalty by orders of magnitude while allowing the network to be operated at higher capacity constraints, with a minimal reduction in network capacity.

5 DISCUSSION & FUTURE WORK

In this section, we discuss a few corner cases, address the current implementation constraints with next generation programmable hardware, and discuss future extensions.

Handling bidirectional corruption. Handling bidirectional corruption is relatively straightforward by extending LinkGuardian's current implementation. First, we increase the reliability of the control messages in the reverse direction by sending multiple copies. These messages include loss notifications, explicit ACK packets, and pause/resume messages. Then it is simply a matter of running a parallel instance of LinkGuardian in the reverse direction.

Handling multiple corrupting links on the same switch. In our simulations (§4.8), we observed that in the worst case, there could be 2 and 4 concurrently LinkGuardian-enabled links per switch pipeline for capacity constraints of 50% and 75% respectively. Switch pipelines support ~2 internal recirculation ports per pipe [40] and more recirculation ports can be added by running any

[3] https://github.com/NUS-SNL/linkguardian-sim
[4] supports about 500K 10G-connected or 125K 40G-connected servers

free ports in loopback mode. Nevertheless, how we can use ∼2 recirculation ports to buffer packets for >2 LinkGuardian ports remains to be explored. We believe that this could be plausible as datacenter link utilization is bursty and not all ports run at 100% utilization at the same time [58]. Also, since Tofino2 can likely implement retransmission without recirculation, Tofino2 can naturally support multiple corrupting links.

Handling bursty losses. Bursty or consecutive packet losses can occur on a corrupting link. Therefore, it is possible that the last (tail) normal packet from the sender switch and the subsequent dummy packet are both lost (refer Figure 5). In this case, the tail packet loss would go undetected. LinkGuardian handles this scenario by sending multiple copies of the dummy packet each time the normal packet queue becomes empty. This ensures that, to detect the tail packet loss, the receiver switch receives the dummy packet with a high probability. A similar mechanism is used by the ACK packets when there is corruption in the reverse direction.

Multiple corrupting links on a path. LinkGuardian naturally handles such a scenario since it operates on each link independently. While we could not evaluate this scenario due to the lack of sufficient optical hardware to introduce corruption on multiple links, we expect the end-to-end performance to benefit significantly from LinkGuardian. This is because the baseline loss case would be even worse as multiple corrupting links on a path would lead to a greater fraction of the flows suffering corruption packet loss, with individual flows being more likely to suffer multiple packet losses.

Implementing LinkGuardian with Tofino2. In our measurements (detailed in Appendix B.1), we found that LinkGuardian takes up to 5.25 μs to recover an MTU-sized (1,538 B on wire) packet on a 100G link. Given that it takes only about ∼123 ns to serialize 1,538 bytes on a 100G link, this delay is surprisingly long. It turns out that this large delay is an artifact of our current recirculation-based buffering on the Intel Tofino. Tofino2 [33] offers new advanced flow control primitives that could be used to pause/unpause as well as achieve credit-based scheduling of a queue in the dataplane. These primitives could in theory allow us to implement retransmission without recirculation, but this thesis remains to be validated.

LinkGuardian vs. LinkGuardianNB. Our results in §4 suggest that LinkGuardianNB is generally more scalable to higher loss rates and link speeds and incurs fewer overheads compared to LinkGuardian. While LinkGuardianNB performs comparably to LinkGuardian for TCP (up to 99th percentile), the difference is more significant for RDMA's NIC-based reliable transport. Depending on the application mix and the desired SLA guarantees, a network operator could do a runtime configuration to run either LinkGuardian or LinkGuardianNB on a corrupting link. In fact, while currently not implemented in our prototype, it is reasonably straightforward to allow both LinkGuardian and LinkGuardianNB to run simultaneously on a corrupting link, each protecting a different class of traffic with different ordering guarantees.

Incremental Deployment. LinkGuardian is suitable for incremental deployment as switches are upgraded over time in a network. As discussed in §2, not all links are equal, and therefore network operators can prioritize deploying LinkGuardian for links that cannot be easily disabled without violating capacity constraints. The exact partial deployment strategy that yields maximum benefits remains as future work.

Higher Link Speeds. LinkGuardian is agnostic to the overall scale of the network as it works locally on the link between adjacent switches. The question is whether LinkGuardian would continue to work as link speeds continue to grow. In principle, LinkGuardianNB would work well for higher link speeds of 400G and above due to its lower overheads and better scalability. LinkGuardian, on the other hand, might achieve a proportionally lower effective link speed and higher buffer overhead if the switch pipeline latency hugely dominates the retransmission delay. However, we believe that with a Tofino2-based implementation and further dataplane optimizations, LinkGuardian should still achieve good performance with low overheads. We plan to investigate this once the hardware becomes available to us.

Automatic fallback. LinkGuardian is designed to work for low loss rates observed in today's datacenter networks (Table 1). However, in the rare event of sudden high loss rates, LinkGuardian's performance can degrade, especially when the packet ordering is being preserved. Such scenarios can be handled by extending our control-plane-based monitoring system (Appendix C) to detect such situations and automatically fall back to LinkGuardianNB or completely disable LinkGuardian on the affected corrupting link.

Reordering tolerance in modern transport protocols. Recently, a new feature called the "reordering window adaptation" was proposed in RFC8985 [12]. Also, RoCEv2's NIC-based reliable transport has a new "selective repeat" feature [43] that allows more efficient selective retransmission than Go-back-N recovery. We plan to investigate the implication of these new features for LinkGuardianNB.

6 CONCLUSION

To the best of our knowledge, we are the first to validate that a combination of simple techniques can make link-local retransmission practical in modern datacenter networks. We also show that, for short TCP flows in datacenter networks, simple out-of-order retransmission is often sufficient to significantly mitigate the impact of corruption packet loss. With LinkGuardian, network operators can work with corrupting links with moderate loss rates (between 10^{-3} and 10^{-5}) at a marginally reduced link speed and with little overhead. Since LinkGuardian is amenable to incremental deployment, deploying LinkGuardian with CorrOpt will allow network operators to not only reduce the network-wide corruption loss rate, but also operate networks at higher capacity constraints that were not previously feasible. Overall, we believe that we have made a strong case that link-local retransmission is both practical and effective for modern datacenter networks.

ACKNOWLEDGMENTS

We thank the anonymous SIGCOMM reviewers for their valuable feedback and Danyang Zhuo for answering our relentless questions regarding CorrOpt [61]. This work was supported by the Singapore Ministry of Education Academic Research Fund Tier 1 (T1 251RES1917) and Tier 2 (MOE2019-T2-2-134).

Ethics statement: This work does not raise any ethical issues.

REFERENCES

[1] 3GPP. 2007. TS 36.321: E-UTRA; Medium Access Protocol Specification (Release 8). (2007).

[2] 3GPP. 2020. TS 36.321: LTE; E-UTRA; Medium Access Protocol Specification (Release 16). (2020).

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

[4] Mark Allman, Vern Paxson, and Ethan Blanton. 2009. TCP Congestion Control. *RFC 5681* (2009).

[5] Alexey Andreyev. 2014. Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-SpeedNICs. In *Proceedings of NSDI.*

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of SIGMETRICS.*

[8] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. 1996. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *Proceedings of SIGCOMM.*

[9] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. 1995. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of MOBICOM.*

[10] Ethan Blanton, Mark Allman, Lili Wang, Ilpo Jarvinen, Markku Kojo, and Yoshifumi Nishida. 2012. A conservative loss recovery algorithm based on selective acknowledgment (SACK) for TCP. *RFC 6675* (2012).

[11] Guo Chen, Yuanwei Lu, Yuan Meng, Bojie Li, Kun Tan, Dan Pei, Peng Cheng, Layong Larry Luo, Yongqiang Xiong, Xiaoliang Wang, et al. 2016. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *Proceedings of NSDI.*

[12] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. 2021. The RACK-TLP Loss Detection Algorithm for TCP. *RFC 8985* (2021).

[13] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013).

[14] Linux Networking Documentation. 2022. DCTCP (DataCenter TCP). https://www.kernel.org/doc/html/latest/networking/dctcp.html.

[15] Edward John Forrest Jr. 2014. How to Precision Clean All Fiber Optic Connections: A Step By Step Guide.

[16] fs.com. 2023. EdgeCore ET7302 SR compatible 25GBASE-SR optical transceiver. https://www.fs.com/sg/products/84279.html?attribute=739&id=393015.

[17] fs.com. 2023. Optical transceiver 10GBASE-SR SFP. https://www.fs.com/sg/products/11589.html.

[18] fs.com. 2023. Optical transceiver 50GBASE-SR SFP56. https://www.fs.com/sg/products/146526.html.

[19] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When Cloud Storage Meets RDMA. In *Proceedings of NSDI.*

[20] Hans Giesen, Lei Shi, John Sonchack, Anirudh Chelluri, Nishanth Prabhu, Nik Sultana, Latha Kant, Anthony J McAuley, Alexander Poylisher, André DeHon, et al. 2018. In-network computing to the rescue of faulty links. In *Proceedings of the NetCompute Workshop.*

[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of SIGCOMM.*

[22] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyan Shen, Abdul Kabbani, Moray McLaren, and Steve Scott. 2023. Datacenter Ethernet and RDMA: Issues at Hyperscale. *arXiv preprint arXiv:2302.03337* (2023).

[23] IEEE. 2009. 802.11n-2009 Standard. https://standards.ieee.org/ieee/802.11n/3952/.

[24] IEEE. 2013. 802.11ac-2013 Standard. https://ieeexplore.ieee.org/document/6687187.

[25] IEEE. 2015. IEEE Standard for Ethernet - Amendment 3: Physical Layer Specifications and Management Parameters for 40 Gb/s and 100 Gb/s Operation over Fiber Optic Cables. *IEEE Std 802.3bm-2015 (Amendment to IEEE Std 802.3-2012 as amended by IEEE Std 802.3bk-2013 and IEEE Std 802.3bj-2014)* (2015).

[26] IEEE. 2016. IEEE Standard for Ethernet – Amendment 2: Media Access Control Parameters, Physical Layers, and Management Parameters for 25 Gb/s Operation Amendment 2: Media Access Control Parameters, Physical Layers, and Management Parameters for 25 Gb/s Operation. *IEEE Std 802.3by-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE Std 802.3bw-2015)* (2016).

[27] IEEE. 2017. IEEE Standard for Ethernet - Amendment 10: Media Access Control Parameters, Physical Layers, and Management Parameters for 200 Gb/s and 400 Gb/s Operation. *IEEE Std 802.3bs-2017 (Amendment to IEEE 802.3-2015 as amended by IEEE's 802.3bw-2015, 802.3by-2016, 802.3bq-2016, 802.3bp-2016, 802.3br-2016, 802.3bn-2016, 802.3bz-2016, 802.3bu-2016, 802.3bv-2017, and IEEE 802.3-2015/Cor1-2017)* (2017).

[28] IEEE. 2019. IEEE Standard for Ethernet - Amendment 3: Media Access Control Parameters for 50 Gb/s and Physical Layers and Management Parameters for 50 Gb/s, 100 Gb/s, and 200 Gb/s Operation. *IEEE Std 802.3cd-2018 (Amendment to IEEE Std 802.3-2018 as amended by IEEE Std 802.3cb-2018 and IEEE Std 802.3bt-2018)* (2019).

[29] IEEE. 2020. IEEE Standard for Ethernet – Amendment 7: Physical Layer and Management Parameters for 400 Gb/s over Multimode Fiber. *IEEE Std 802.3cm-2020 (Amendment to IEEE Std 802.3-2018 as amended by IEEE Std 802.3cb-2018, IEEE Std 802.3bt-2018, IEEE Std 802.3cd-2018, IEEE Std 802.3cn-2019, IEEE Std 802.3cg-2019, and IEEE Std 802.3cq-2020)* (2020).

[30] Raj Joshi, Qi Guo, Nishant Budhdev, Ayush Mishra, Mun Choon Chan, and Ben Leong. 2022. LinkGuardian: Mitigating the impact of packet corruption loss with link-local retransmission. In *Proceedings of APNet.*

[31] Raj Joshi, Ben Leong, and Mun Choon Chan. 2019. Timertasks: Towards time-driven execution in programmable dataplanes. In *Proceedings of SIGCOMM (Posters and Demos).*

[32] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay Is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of SIGCOMM.*

[33] Jeongkeun Lee. 2020. Advanced Congestion & Flow Control with Programmable Switches. In *P4 Expert Roundtable Series.* https://opennetworking.org/wp-content/uploads/2020/04/JK-Lee-Slide-Deck.pdf

[34] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of SIGCOMM.*

[35] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. 2021. Towards timeout-less transport in commodity datacenter networks. In *Proceedings EuroSys.*

[36] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A Large Scale Study of Data Center Network Reliability. In *Proceedings of IMC.*

[37] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. 2022. From luna to solar: the evolutions of the compute-to-storage networks in Alibaba cloud. In *Proceedings of SIGCOMM.*

[38] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of SIGCOMM.*

[39] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *Proceedings of SIGCOMM.*

[40] EdgeCore Networks. 2022. DCS802. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=334.

[41] NVIDIA. 2020. Unbreakable Links - MLNX-OS v3.9.0300 - NVIDIA Networking Docs. https://docs.nvidia.com/networking/display/MLNXOSv390300/Unbreakable+Links.

[42] NVIDIA. 2022. RDMA Transport Modes. https://docs.nvidia.com/networking/display/RDMAAwareProgrammingv17/Transport+Modes.

[43] NVIDIA. 2022. RoCE Selective Repeat. https://docs.nvidia.com/networking/m/view-rendered-page.action?abstractPageId=25137694.

[44] Christina Parsa and JJ Garcia-Luna-Aceves. 1999. TULIP: A Link-Level Protocol for Improving TCP over Wireless Links. In *Proceedings of WCNC.*

[45] Ting Qu, Raj Joshi, Mun Choon Chan, Ben Leong, Deke Guo, and Zhong Liu. 2019. SQR: In-network packet loss recovery from link failures for highly reliable datacenter networks. In *Proceedings of ICNP.*

[46] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. 2022. PLB: Congestion Signals Are Simple and Effective for Network Load Balancing. In *Proceedings of SIGCOMM.*

[47] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network's datacenter network. In *Proceedings of SIGCOMM.*

[48] Matt Sargent, Jerry Chu, Vern Paxson, and Mark Allman. 2011. Computing TCP's Retransmission Timer. *RFC 6298* (2011).

[49] Omer S. Sella, Andrew W. Moore, and Noa Zilberman. 2018. FEC Killed The Cut-Through Switch. In *Proceedings of NEAT.*

[50] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *Proceedings of NSDI.*

[51] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. 2018. RADWAN: Rate Adaptive Wide Area Network. In *Proceedings of SIGCOMM.*

[52] R Sivaram. 2008. Some Measured Google Flow Sizes. *Google internal memo, available on request* (2008).

[53] Ashish Vulimiri, Oliver Michel, P Brighten Godfrey, and Scott Shenker. 2012. More is less: Reducing latency via redundancy. In *Proceedings of HotNets*.

[54] Shuai Wang, Kaihui Gao, Kun Qian, Dan Li, Rui Miao, Bo Li, Yu Zhou, Ennan Zhai, Chen Sun, Jiaqi Gao, Dai Zhang, Binzhang Fu, Frank Kelly, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. 2022. Predictable vFabric on Informative Data Plane. In *Proceedings of SIGCOMM*.

[55] Jim Warner. 2022. Packet Buffers. https://people.ucsc.edu/~warner/buffer.html.

[56] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. 2012. NetPilot: Automating datacenter network failure mitigation. In *Proceedings of SIGCOMM*.

[57] Gaoxiong Zeng, Li Chen, Bairen Yi, and Kai Chen. 2022. Cutting Tail Latency in Commodity Datacenters with Cloudburst. In *Proceedings of INFOCOM*.

[58] Qiao Zhang, Vincent Liu, and Hongyi Zeng. 2017. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of IMC*.

[59] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. 2020. Flow event telemetry on programmable data plane. In *Proceedings of SIGCOMM*.

[60] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *Proceedings of SIGCOMM*.

[61] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and mitigating packet corruption in data center networks. In *Proceedings of SIGCOMM*.

[62] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. 2017. RAIL: A Case for Redundant Arrays of Inexpensive Links in Data Center Networks. In *Proceedings of NSDI*.

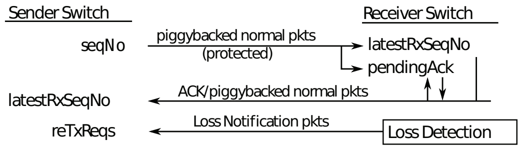


Figure 17: State maintained by LinkGuardian switches and different types of packets that read/update it.

Note: Appendices are supporting material that has NOT been peer-reviewed.

A PROTOCOL DETAILS

In this Appendix, we provide some details that might be helpful for understanding our complete implementation of LinkGuardian, but which are not essential for understanding the key ideas and contributions of our work.

A.1 Loss Detection & Notification

In Figure 17, we list the state variables maintained by the sender and receiver switches and the different packets that are exchanged. The sender maintains a monotonically increasing `seqNo` while the receiver records the latest received `seqNo` as `latestRxSeqNo`. A copy of the `latestRxSeqNo` is also maintained at the sender, which the receiver keeps updating. The sender also maintains a lookup table called `reTxReqs`, which records the sequence numbers of the packets for which retransmission is requested.

For each packet that is transmitted on the corrupting link (protected packet), the sender adds the `seqNo` to the packet (using a custom header) and increments it by 1. The sender uses egress mirroring to also make a copy of the packet along with the added sequence number and buffers it until the receiver notifies that the packet was received successfully. On the receiver, when a protected packet is received, it updates the `latestRxSeqNo` to the `seqNo` in the packet and also sets the `pendingAck` to 1. `pendingAck` set to 1 denotes that the copy of `latestRxSeqNo` on the sender is yet to be updated.

No Loss Scenario. When there are no corruption packet losses, the `latestRxSeqNo` on the receiver would increase by 1, each time a protected packet is received. On every update of the `latestRxSeqNo`, the receiver must update the `latestRxSeqNo` on the sender as soon as possible so that the sender can drop the buffered packets that are successfully delivered. This timely update of the `latestRxSeqNo` on the sender is critical to ensure that LinkGuardian's use of the packet buffer at the sender is kept to a minimum.

Loss Scenario. When a protected packet(s) gets corrupted and dropped by the receiving MAC, the receiver observes that the `latestRxSeqNo` is incremented by more than 1. On noticing this, the receiver activates a `LossDetection()` routine. In this routine, the receiver generates a new packet called "Loss Notification" which contains information about the missing sequence number as well as the `latestRxSeqNo`. This loss notification packet is sent to the sender through a high-priority queue (see Figure 5) to ensure timely recovery. On reaching the sender, the lookup table `reTxReqs` (Figure 17) is updated with the sequence numbers of the packets that need to be retransmitted.

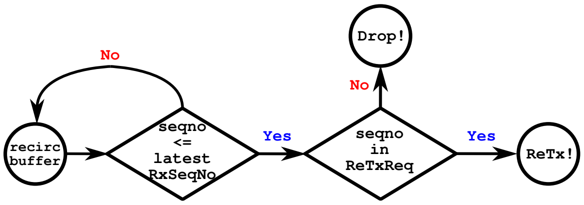


Figure 18: Sender-side buffering and Retransmission.

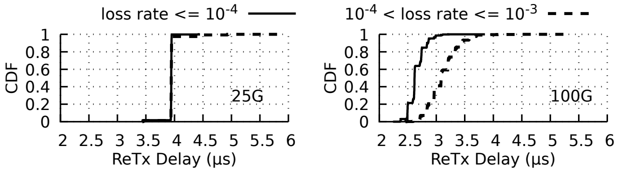


Figure 19: Delay observed by LinkGuardian receiver switch to receive retransmission from the time the loss was detected.

A.2 Sender-side Buffering & Retransmission

For each packet that is sent on the corrupting link, the sender switch adds a monotonically increasing `seqNo` and uses egress mirroring to create a copy of the packet for buffering. The packet buffering on the sender switch is realized through recirculation. Specifically, the buffered copy of the protected packet is sent to the recirculation port of the switch dataplane pipeline. At the same time, as described in §A.1, the receiver switch keeps the `latestRxSeqNo` on the sender switch updated and additionally updates the lookup table `reTxReqs` in case of a corruption packet loss. Each time the buffered packet completes a recirculation loop, the sender switch applies the logic shown in Figure 18 to the packet's sequence number. Essentially, if the buffered packet's sequence number is less than or equal to the `latestRxSeqNo`, the sender switch checks the `reTxReqs` lookup table to see if a retransmission is requested for that sequence number. If so, the packet is retransmitted through a high-priority queue (see Figure 5) or the packet is dropped otherwise. If a packet is retransmitted, its sequence number is cleared in the `reTxReqs` table. If the buffered packet's sequence number is greater than the `latestRxSeqNo`, then we do not know yet if the packet was successfully received or not, and therefore the sender switch continues to buffer the packet through recirculation.

B ADDITIONAL EXPERIMENTS AND RESULTS

This Appendix presents additional experiments as well as more detailed results for the experiments described in the main body of the paper.

B.1 Parameter Tuning

In this section, we describe how we derive the appropriate values for three parameters used by LinkGuardian: `ackNoTimeout`, `resumeThreshold`, and `pauseThreshold`.

Recall that when LinkGuardian preserves packet ordering (default mode), the `ackNoTimeout` prevents LinkGuardian from stalling in the event that a lost packet is never recovered (§3.3). Therefore, `ackNoTimeout` needs to be set to a value larger than the expected maximum retransmission delay. To estimate the retransmission delay, we measured the time from when the receiver switch detects packet loss to when it successfully receives the retransmission from the sender switch. Since high-priority queues are used for loss notification and retransmission, this retransmission delay is a function of the switch pipeline latencies, the link speed, and the number of retransmitted copies. If more than one copy is retransmitted (for higher loss rates), then the worst-case retransmission delay is when only the last copy is received. In Figure 19, we plot the distribution of the retransmission delays for ~1 million loss recoveries for 1,518 B packets. We conservatively set the `ackNoTimeout` to 7.5 μs and 7 μs for 25G and 100G, respectively. A larger `ackNoTimeout` leads to a slightly longer stall in transmission, but only in the unlikely event that the original packet and all retransmitted copies are lost.

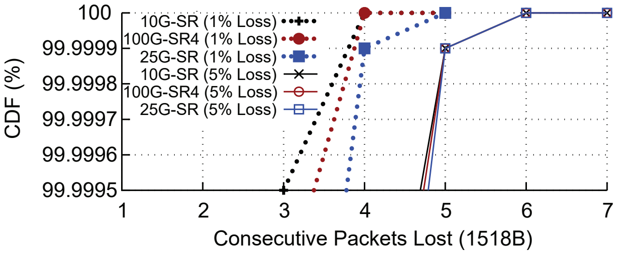


Figure 20: Distribution of consecutive packets lost.

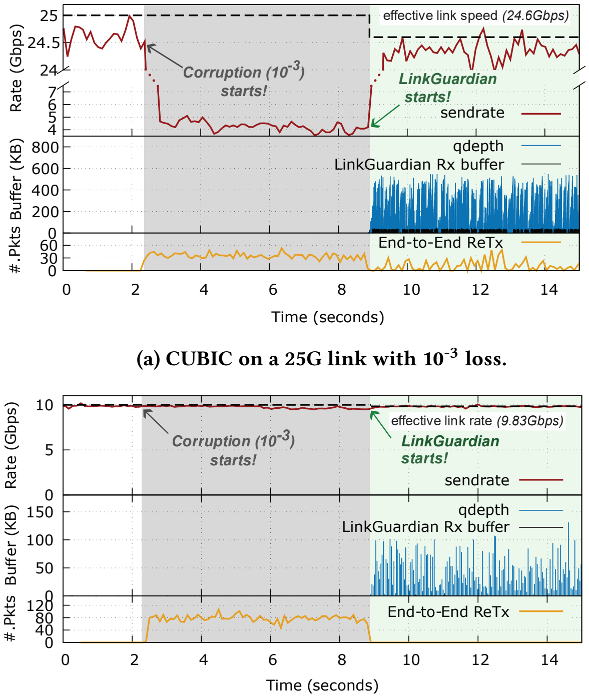
The backpressure mechanism on the LinkGuardian receiver uses the `pauseThreshold` and the `resumeThreshold` (see §3.3 and Figure 6). Recall that a resume message is sent when the reordering buffer at the receiver drops below the `resumeThreshold`. If the `resumeThreshold` is set too small, the reordering buffer will be empty before the sender switch successfully resumes transmission. Hence, we set the `resumeThreshold` to a value that is larger than the amount of data that would drain from the buffer during the time from when the receiver sends a resume message to when the receiver starts receiving traffic again. We refer to this time as t_{flight_resume}. t_{flight_resume} is independent of the corruption loss rate and depends only on the link speed and switch pipeline latencies. We measured the maximum t_{flight_resume} values to be 1.9 μs and 1.6 μs for 25G and 100G links respectively. Since the recirculation-based reordering buffer drains at 100G, we set `resumeThreshold` at 40 KB and 37 KB for 25G and 100G links respectively. Since we use a fixed hysteresis of 2 MTU, the `pauseThreshold` is `resumeThreshold` + 2 MTU.

B.2 Consecutive Corruption Packet Loss

In Figure 20, we plot the distribution of the number of consecutive packets lost that we measured by setting the VOA to induce unreasonably high loss rates of 1% and 5%. Based on Figure 20, our current implementation (§3.5) provisions for handling 5 consecutive packets lost using 5 1-bit registers.

B.3 Impact on CUBIC and BBR transports

In this section, we present the results of the same experiment as in §4.2 but with CUBIC and BRR transports. In Figures 21a and 21b, we plot the results for CUBIC, and BBR respectively. The effective link speed in these figures is measured separately by sending a line rate UDP flow under the same experiment conditions.



(a) CUBIC on a 25G link with 10^{-3} loss.

(b) BBR on a 10G link with 10^{-3} loss.

Figure 21: Performance of LinkGuardian for CUBIC and BBR Transport Protocols.

CUBIC. In Figure 21a, we see that at 10^{-3} corruption loss, the throughput for CUBIC reduces sharply once corruption losses are introduced. Upon enabling LinkGuardian, the corruption losses are nearly eliminated and the throughput returns to a level comparable to that before packet corruption was introduced. We also notice that there is a build-up in the flow's buffer at the sender switch (shown as "qdepth") due to the reduced effective link capacity. CUBIC being loss-based, we can also see congestion loss happening once LinkGuardian is enabled. Note that LinkGuardian only protects and retransmits the packets that are *sent out* on the corrupting link and is not affected by any congestion loss happening due to the overflowing of the normal packet queue.

BBR. Since BBR is mostly agnostic to packet loss, we see in Figure 21b that it suffers minimal degradation when corruption loss is introduced[5]. Nevertheless, it seems that once LinkGuardian is enabled, we still see a small increase in the observed throughput.

These results show that, other than ECN-based DCTCP, even loss-based and delay-based congestion control protocols work correctly with LinkGuardian.

B.4 Overheads

In this section, we present more details for the overhead results presented in §4.6. Recall that these overhead results correspond to the "stress test" experiments in §4.1, where we run continuous line-rate traffic. Therefore, these results represent the "worst case" cost of running LinkGuardian as real-world link utilization exceeds 90% only about 10% of the time [58].

Table 4: Recirculation overhead (% pipe forwarding capacity)

Loss Rate →	10^{-5}	10^{-4}	10^{-3}
25G TX	0.45	0.449	0.444
25G RX	0.661	0.662	0.664
100G TX	0.663	0.657	0.608
100G RX	0.657	0.658	0.662

Recirculation Overhead. In Table 4, we show the recirculation overhead for LinkGuardian at both the sender and the receiver switches in terms of the percentage of the switch pipeline's processing capacity. We see that LinkGuardianNB has the same recirculation overhead on the sender switch but zero on the receiver switch. Overall, we see that the recirculation takes up less than 1% of the switch pipeline's processing capacity and therefore this overhead is negligible. Since this overhead is for 1500B MTU-sized packets sent at line rate (§4.1), we can extrapolate it to obtain the overhead when using smaller packet sizes. Considering the median packet size of 250B observed in datacenter networks [47], we expect the recirculation overhead to be ~6 times compared to the one reported in Table 4. Even then, the maximum expected *worst-case* overhead would be 0.664 x 6 = 3.984%, which is relatively low.

C MONITORING LINKS FOR CORRUPTION

To detect corrupting links, we implemented `corruptd`, a daemon that runs at the local control plane of the programmable switches.

Detecting Corrupting Links. `corruptd` periodically polls the driver (in this paper, we configure the interval as 1 second) to extract the switch port RX statistics, specifically, `framesRxOk` and `framesRxAll`. We maintain a moving window of 100M frames to compute the link loss rates, given by $L = \frac{framesRxOk}{framesRxAll}$. When $L \geq 10^{-8}$ for any particular link, the upstream transmitting switch will be notified to activate LinkGuardian.

Notification and Activation. For scalability, `corruptd` daemons communicate through a publish-subscribe (PubSub) pattern using Redis. Each daemon subscribes to link corruption notifications relating to the local switch's links. Upon receipt of a notification, `corruptd` pushes corresponding dataplane match-action table entries to activate LinkGuardian for the corrupting link depending on the target and the actual loss rates (see Equation 2).

D LINK CORRUPTION TRACE GENERATION

A link corruption trace is essentially a time series of link corruption events where a link corruption event denotes which link started to corrupt packets and at what loss rate. To determine the time at which a link would start corrupting packets, we assume a per-link 1-parameter Weibull distribution with a constant shape parameter (β). This is because the location parameter of the Weibull distribution (γ) is zero since it is not guaranteed that all links in a large warehouse-scale datacenter would not start corrupting packets during a certain initial period. Also, the shape parameter (β) is equal to 1, since the corruption is purely caused by random external events such as connector contamination, fiber bending, etc. Therefore, the per-link Weibull PDF that determines the time until a link's next failure is

[5] We ran BBR on a 10G link instead of a 25G link because in our setup, BBR became CPU-limited when we tried to run the experiment on a 25G link, and it was not able to fully saturate the link.

given by

$$f(t) = \frac{1}{\eta} \times e^{-\left(\frac{t}{\eta}\right)} \tag{3}$$

where the parameter η is the mean-time-to-failure (MTTF) of a link. A study by Meza et al. [36] showed that for fiber links from different vendors considered in their study, the mean time between the link faults was at most 10,000 hours. We conservatively use the value of 10,000 hours as the MTTF (η in Equation 3) since Meza et al. did not specifically consider only intra-datacenter links. What this means is that, on average, it would take 10,000 hours (or 1.15 years) for a fiber link to start corrupting packets from the time it was last repaired.

To generate the trace, we first draw samples from the Weibull distribution independently for each link to determine the times at which each link would start corrupting packets. This gives us the various times of the corruption events and the link involved in each corruption event. Then for each corruption event, we use the corruption loss rate distribution from CorrOpt (c.f. Table 1 in [61]) to determine the loss rate. This list of corruption events sorted by time forms the link corruption trace. We note that the trace generated using the above methodology has a nearly random spatial distribution of simultaneously corrupting links which matches the observation by Zhuo et al. [61] in production datacenters.