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 99.9th 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].

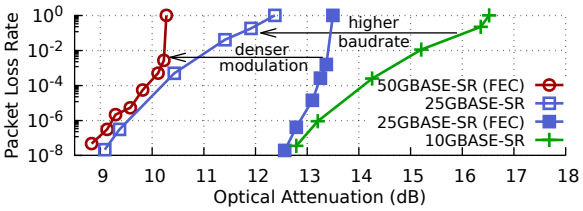

Figure 1: Effect of optical attenuation on various Ethernet link speeds (1518 B frames).

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,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604853

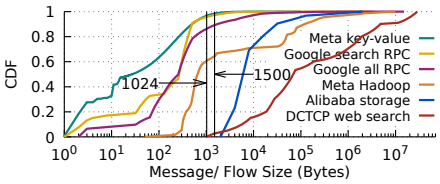

Figure 2: Flows size distribution of several datacenter workloads from 2008 to 2019 [3, 7, 34, 47, 52].

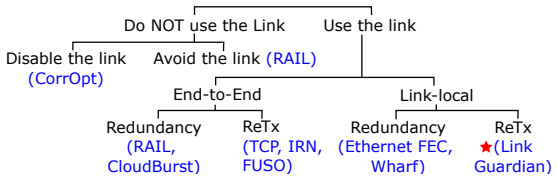

Figure 3: Design space for handling corrupting links in datacenter networks.

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 [42]). 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) [5, 7, 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 e ectively on a large-scale network. Therefore, to the best of our knowledge, LinkGuardian is the rst 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, nally, 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 bu ers a copy of the recently sent packets (in the Tx bu er). 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 tra c (Ack2). The sender then drops the bu ered 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 noti cation 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 su cient to achieve LL-ReTx, it is not practical in a datacenter because of the following reasons:

(1) Small bu ers: Since the switches in datacenter networks have shallow bu ers, the sender needs to receive the ACKs quickly so that it can drop the bu ered packets fast enough to keep the Tx bu er usage small. If we piggyback the ACKs naively, they could get delayed by an arbitrary amount depending on the reverse direction tra c.
(2) Short ows: Since most datacenter ows 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 ow 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 ows: The use of RDMA in datacenters networks is now becoming increasingly commonplace [9, 21, 37, 60]. Compared to TCP, RDMA performance is very sensitive to packet ordering due to the lack of a reordering window [12]. The basic LL-ReTx above does not preserve the original packet ordering e.g. when seqNo 3 is lost in Figure 5.

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

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 (Ÿ3.1): 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 tra c to piggyback the ACKs.
(2) Self-replenishing queue of dummy packets (Ÿ3.2): 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 tra c to allow the receiver to quickly detect tail packet losses (e.g. seqNo 5 in Figure 5).
(3) Reordering Bu er without Over ow (Ÿ3.3): To preserve packet ordering, LinkGuardian implements a reordering bu er on the receiver (③ in Figure 5). A naive design would result in bu er over ow 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-speci ed target level. To achieve the target e ective loss rate, LinkGuardian also handles the case that the retransmitted copy of the packets could get lost too (Ÿ3.4). 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 modi cations which we describe in Ÿ5.

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 Ÿ4.3 show that LinkGuardianNB is e ective in mitigating corruption packet loss for short TCP ows because of the small ow sizes as well as TCP's support for reordering window and selective recovery.

3.1 Fast ACKs to prevent bu er over ow

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

Algorithm 1: De-Duplication & In-Order Recovery

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

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;
```

signi cant 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 tra c 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 ows 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 Bu er without Over ow

To preserve packet ordering after a corruption loss is detected, the receiver will need to bu er the subsequent out-of-order packets until the retransmission is received from the sender switch. We implement this bu ering by using a recirculation port queue as the reordering bu er (Rx Bu er in Figure 5). Packets received after the lost packet are bu ered 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 (Ÿ3.4), 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-bu ered 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.

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

Since each retransmission takes a small but non-negligible delay, the reordering bu er will keep lling up with each packet loss if the subsequent packets continue to arrive at line rate, and eventually, the reordering bu er would over ow. 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 a ect 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 bu er 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 $t_{inflight\_resume}$ before the normal packet queue on the sender is resumed. The resumeThreshold is therefore set to a value such that during the $t_{inflight\_resume}$ time, the reordering bu er 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 bu er level reaches the pauseThreshold; and a resume message is sent when the bu er falls below the resumeThreshold. Since Algorithm 2 operates on a per-packet basis, we use a ag $curr\_state$ 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 bu er 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 (#) copies of a bu ered packet in response to a loss noti cation.

Since the original packet was already transmitted (and lost), the total number of copies transmitted for the lost packet would be # ¸ 1, giving us an e ective loss rate of ¹02CD0;„>BB̲A0C4¹#¸1º. Since our goal is not to completely eliminate corruption packet losses but to reduce the e ective loss rate to an operator-speci ed target level, we have the following relation:

$$^{1}02CD0;„>BB\underline{A}0C4^{1\#\,,\,1^{o}} \ddot{Y}= {}^{1}C0A64;\subseteq BB\underline{A}0C^{o}4 \qquad (1)$$

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 bu ered packet (# = 1) would su ce to achieve an e ective loss rate of 10^{-8}. Now, solving Equation 1, we get the number of retransmitted copies (#) as:

$$\# = i \frac{;>6_{10}{}^{1}C0A64;\subseteq BB\underline{A}0C^{o}4}{;>6_{10}{}^{1}02CD0;„>BB\underline{A}0C^{o}4} \quad 1 \qquad (2)$$

Since # is an integer in practice, we assign # 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 tra c, 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 (Ÿ3.3) 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 (Ÿ3.3) using Tofino2's advanced flow control primitives [33]. However, since our current implementation is on Tofino, we use PFC pause/resume frames. Speci cally, 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 di erent 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 di erent-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 leTxReqs 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 noti cation 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 e orts, there is still a small but non-zero probability that a retransmission will not be successful. Because we bu er packets at the receiver until the retransmission for the missing packet is received, this could stall the transmission inde nitely and cause the reordering bu er to over ow. 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 Ÿ3.3), we use periodic packets from the switch's packet generator for timekeeping (Ÿ3.1). 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 bu ered 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 noti cations. 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 (Ÿ3). 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 e ective 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 e ective is LinkGuardian at masking the corruption packet losses? Are we able to reduce the e ective loss rate to the operator-speci ed target as desired? And what is the corresponding reduction in link speed? (Ÿ4.1)
(2) How does LinkGuardian interact with the end-point transport protocols? (Ÿ4.2)
(3) How well does LinkGuardian handle tail packet loss and improve FCTs for short and single-packet ows? (Ÿ4.3, Ÿ4.4)
(4) How do the various mechanisms of LinkGuardian contribute to its good performance? (Ÿ4.5)
(5) How much bu ering do we need and what are the associated overheads and costs of deploying LinkGuardian? (Ÿ4.6)
(6) How does LinkGuardian's performance compare with Wharton [20], the state-of-art link-local FEC solution? (Ÿ4.7)
(7) When deployed in a large-scale network, how e ective is LinkGuardian in reducing the corruption packet loss and improving the overall network capacity? (Ÿ4.8)

Testbed Setup. We use the testbed setup shown in Figure 7, where sw2 and sw6 are connected by an OM4 grade ber optical ber 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 bu ers 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 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 e ective 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 di erence 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 [4]) are enabled and)$ <8= is set to 1 ms. The network RTT for a TCP sender is 30 `s. For RoCEv2, we use a one-sided RDMA_WRITE operation using NIC-based reliable delivery (RC [42]) which we found to have)$ of 1 ms.

Parameters. LinkGuardian uses 3 parameters: ackNoTimeout , resumeThreshold , and pauseThreshold . As discussed in Ÿ3.5, we set the ackNoTimeout to 7.5 `s and 7 `s as we found the maximum retransmission delays to be 6 `s and 5.5 `s for 25G and 100G links respectively. For the resumeThreshold (Ÿ3.3 and Figure 6), we measured the maximum $n_{inflight\_resume}$ values to be 1.9 `s and 1.6 `s 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 bu er drains at 100G. Since we use a xed hysteresis of 2 MTU, the pauseThreshold is resumeThreshold + 2 MTU. We provide more details on parameter tuning in Appendix B.1.

4.1 E ective 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 (e ective) loss rates achieved by LinkGuardian and the corresponding e ective link speeds for 25GBASE-SR and 100GBASE-SR4 links. We observe that, except for the 25G link with 10^{-3} loss rate, the e ective 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 e ective 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}.

Figure 8: E ective loss rates achieved by LinkGuardian (LG) and LinkGuardianNB (LG_NB) and the e ective link speeds.

For e ective link speed, we see that LinkGuardianNB scales much better to higher link speeds and higher loss rates compared to LinkGuardian while achieving similar e ective 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) DCTCP.

(a) LinkGuardian.

(b) RDMA WRITE

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

(b) LinkGuardian with backpressure mechanism disabled.

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

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 e ective link speed while preserving packet ordering.

Timeouts in practice. Recall from Ÿ3.5 that when LinkGuardian preserves ordering, it implements a 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 con rms 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 Ÿ4.1 that LinkGuardian can reduce the e ective loss rates, what matters is the net impact on transport protocols. To understand the impact of LinkGuardian, we send single ow TCP tra c from h4 to h8 using iperf with all links set to 25G. We evaluate three di erent 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 e ective link speed in the gure is measured separately by sending a line rate UDP ow 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 e ective link speed. We also notice that the slightly lower e ective link speed leads to a build-up in the ow's bu er 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 e ective at keeping its receiver-side bu er occupancy (labeled as Rx bu er) 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 bu er (Rx bu er) periodically builds up and over ows. 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 bu ering at the receiver switch works as intended.

4.3 Tail Packet Loss and Short Flows

One-packet Flows. To evaluate how e ectively LinkGuardian handles tail packet losses, we measure the FCT of 143 B DCTCP and RDMA write (RDMA_WR) ows (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 ow 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 e ectively 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 ows. We note that the result in Figure 10 is also representative of all other ow sizes for workloads in Figure 2 that t within a single packet.

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