MAY is not enough!
QUIC servers SHOULD skip packet numbers

Louis Navarre
louis.navarre@uclouvain.be
UCLouvain, Belgium

Olivier Bonaventure
olivier.bonaventure@uclouvain.be
UCLouvain & WEL-RI, Belgium

Abstract

The QUIC protocol increasingly replaces TCP as the dominant transport protocol on the Internet. Its design closely integrates TLS 1.3 with modern transport features such as faster connection establishment and stream multiplexing. QUIC uses unique and monotonically increasing packet numbers compared to the wrapping TCP sequence number.

In this paper, we analyze the behavior of QUIC stacks against the *optimistic acknowledgment* (OACK) attack, i.e., whenever a peer falsely acknowledges non-received packets to increase the transmission rate to saturate the emitter's network. Although QUIC implementations may skip packet numbers to prevent such an attack, we find that of the 16 existing server implementations from the QUIC Interop Runner, 11 use contiguous packet numbers and are vulnerable to the OACK attack. In a controlled environment, we design a simple OACK client and show that we can increase the server's bit rate up to > 200× depending on the stack. We confirm the results by carefully reproducing the OACK attack on a large Content Delivery Network server and suggest an example of a patch to protect implementations.

CCS Concepts

• **Networks** → **Transport protocols**; **Denial-of-service attacks**.

Keywords

QUIC, Optimistic Acknowledgments Attack

ACM Reference Format:
Louis Navarre and Olivier Bonaventure. 2025. MAY is not enough! QUIC servers SHOULD skip packet numbers. In *Applied Networking Research Workshop (ANRW '25), July 22, 2025, Madrid, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3744200.3744772

1 Introduction

Reliable transport protocols such as TCP [4], SCTP [16], and QUIC [7] use acknowledgments to confirm the correct reception of data. Each acknowledgment indicates that all data, including the returned sequence/packet number, has been correctly received. Modern implementations of these protocols return acknowledgments at different frequencies. A TCP receiver usually acknowledges every second received packet. The QUIC specification recommends sending an ACK frame after receiving at least two ack-eliciting packets [7], but a recent extension proposes to negotiate this frequency [6].

These rules are valid for honest receivers who want to receive data reliably. However, in 1999, Savage *et al.* [11] showed that malicious TCP receivers could manipulate their acknowledgments to launch denial-of-service attacks against servers. In a nutshell, to force the server to send data as quickly as possible, a malicious receiver could send optimistic acknowledgments that ignore packet losses (i.e., never return duplicate acknowledgments of SACK block [5]) or worse, send acknowledgments for data that it did not yet receive. Savage *et al.* proposed extensions to TCP to solve this problem, but they have not been adopted. In 2005, Sherwood *et al.* demonstrated that the attack was possible against different TCP implementations [15]. Several solutions were suggested to prevent such attacks against TCP using cumulative *nonces* [11] and by randomly skipping sequence numbers [15]. In 2016, Schaub and Trey released a user-space implementation of TCP that performs this attack [12]. Laraba *et al.* proposed a solution to protect TCP servers by tracking all flows and identifying misbehaviors [8].

This attack was known during the QUIC protocol standardization, and countermeasures were discussed. Since QUIC encrypts most of the packet headers and data, a firewall cannot analyze the incoming QUIC packets and block optimistic acknowledgments, in contrast to TCP. This paper explores whether attacks using optimistic acknowledgments are possible on current QUIC implementations. Our in-lab results indicate that **this attack is feasible. 11 of the 16 QUIC server implementations from the Interop Runner [13, 14] are vulnerable**. Section 2 gives background to QUIC and the *optimistic acknowledgment* attack. In Section 3, we precisely define the OACK attack in QUIC and identify the QUIC server stacks vulnerable to this attack. In Section 4, we design and implement a simple OACK predictor, demonstrating the exposure of these 11 implementations with in-lab measurements: some stacks send > 200× faster than expected. We then carefully perform the OACK attack on a website hosted on a CDN server. Finally, we propose a patch to *quiche* [3] based on existing resilient implementations to detect OACK attacks (Section 6).

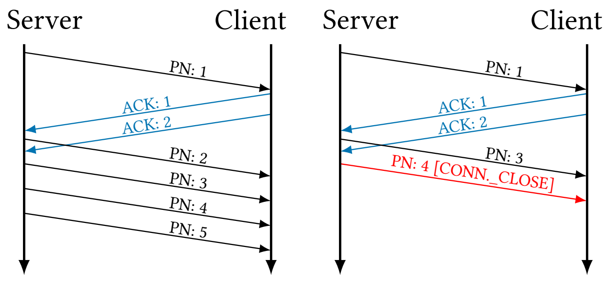
2 Background

This Section gives a brief overview of QUIC [7] and illustrates an *optimistic acknowledgment* (OACK) attack.

QUIC [7] is a standardized transport protocol running above UDP. QUIC closely integrates TLS 1.3 [10] to provide a secure and faster handshake than the TCP/TLS stack. QUIC supports stream multiplexing to avoid the head-of-line blocking problem of TCP and datagrams for unreliable delivery. One key difference between TCP and QUIC is their packet representation. TCP uses a 32-bit sequence number that identifies the position of the first byte of the payload in the bytestream. This sequence number wraps for long connections. When TCP retransmits a packet, it sends it with the same sequence number as the initial transmission. QUIC uses another approach: each packet is a *frame container*. Each packet is uniquely identified by a monotonically increasing, non-wrapping packet number. QUIC endpoints cannot reuse the same packet number twice during a connection. Packet numbers range from 0 to $2^{62} - 1$. A QUIC host does not retransmit lost packets as a TCP host does. QUIC retransmits lost frames in *new* packets with a fresh packet number. QUIC receivers acknowledge packets by sending ACK frames containing the correctly received ranges of packet numbers. In opposition to the clear TCP sequence number, QUIC packet numbers are encrypted in the header.

The QUIC interop runner. More than twenty QUIC implementations coexist in the wild [17]. The QUIC interop runner [13, 14] is an initiative to provide an open-source, free, and fully automated tool to assess the interoperability of QUIC stacks in different scenarios. Seventeen implementations currently participate in the QUIC interop runner; *ngtcp2* and *haproxy* only provide *docker* images for their servers, while *chrome* only provides its client.

The optimistic acknowledgment (OACK) attack was first discussed on TCP [11]. Figure 1a illustrates this attack on QUIC. The malicious client sends acknowledgments for packets that it has not yet received. The server reacts to these



(a) Not skipping packet numbers. (b) Skipping packet numbers.
Figure 1: Optimistic acknowledgment attack example. *PN* stands for *packet number*.

acknowledgments as a sign that it can increase its congestion window and transmission rate since all data in flight was correctly received. By sending acknowledgments beforehand, the malicious receiver *hides the losses* to the sender, thus abnormally increasing the sender's congestion window without considering the underlying network condition. Such an attack can impact both the server and the network. A coordinated attack targeting the same server might saturate its network and potentially impact legitimate clients.

3 Optimistic acknowledgments in QUIC

This Section analyzes the optimistic acknowledgments attack inside QUIC. For simplicity, we consider a client-server connection in which the client requests a large file from the server. The client is malicious and sends optimistic acknowledgments to the server to increase its transmission bit rate. To conduct this attack, the client must guess the sequence of packet numbers that the server will generate. Its objective is to acknowledge these packets *before* they are received; otherwise, it would be harmless since the acknowledgments would follow the network's conditions. We leverage the QUIC Interop Runner to identify vulnerable server implementations to the OACK attack. We assume that the client does not use the transmitted data, i.e., it does not care whether it correctly receives the packets sent by the server. This assumption is reasonable since the client's only interest is to attack the server. A QUIC server is vulnerable to the OACK attack if the client can guess the server's sequence of packet numbers. By knowing this sequence, the client can predict packets it has not received yet and hide the network losses, thus increasing the server's bit rate. Table 1 reports the results of the vulnerable implementations.

Skipping packet numbers. The client must guess the server's packet number sequence to optimistically acknowledge packets sent from the server. This must be done carefully to avoid sending an ACK frame for unsent packets. Indeed, the majority of tested implementations close the connection when they receive an acknowledgment for an unsent packet (Line *(2)* in Table 1). For example, the client could simulate the CUBIC congestion window update mechanism to anticipate the packets sent by the server. QUIC [7] includes

Implem	quic-go v0.50.0	ngtcp2 v.1.11.0	mvfst v2025.03.03	quiche v0.23.4	kwik v0.10.1	picoquic 6304c2e9cc35	aioquic v1.2.0	neqo v0.12.2
(1) Skip PN	✓	✗	✗	✗	✗	✓	✗	✗
(2) Correctness	✓	✓	✓	✗	✗	✓	✗	✓
Implem	nginx 145b228530c3	msquic v2.3.9	xquic v1.8.2	lsquic v4.2.0	haproxy v3.1	quinn v0.5.9	s2n-quic v1.52.0	go-x-net d18fa4cfbd84
(1) Skip PN	✗	✗	✗	✓	✗	✓	✓	✗
(2) Correctness	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: 11 of the 16 server implementations available from the QUIC Interop Runner are vulnerable to optimistic acknowledgment attacks. We report the version used for the measurements (`vX.Y.Z`) or the *docker* image hash. Implementations are vulnerable if they don't skip packet numbers (1). Worse, implementations that do not assess the correctness of the `ACK` frames expose themselves to straightforward OACK attacks (2). Except *msquic*, maintainers of the 11 implementations confirmed their vulnerability to the OACK attack; we did not receive any response from the maintainers of *msquic*; we inspected the source code of *msquic* to confirm its vulnerability. Patched implementations at the time of writing are underlined.

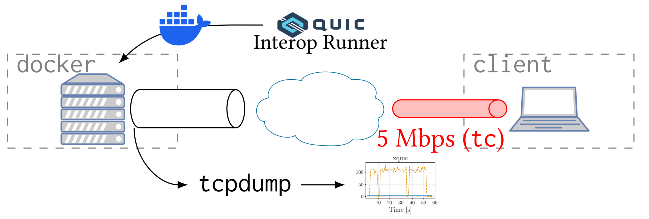


Figure 2: Virtual testbed running on a single machine. A Linux bridge connects the two namespaces.

a mechanism to prevent malicious clients from attacking the server with OACK: the possibility to *skip* packet numbers. Section 21.4 of the QUIC specification [7] mentions that *An endpoint MAY skip packet numbers when sending packets to detect this behavior (i.e., optimistic acknowledgments). An endpoint can immediately close the connection with a connection error of type PROTOCOL_VIOLATION.* There was no consensus on the mandatory skipping of packet numbers with the IETF QUIC working group to prevent these attacks, and Section 21.4 of RFC9000 includes this as a possible feature. If the server skips some packet numbers when emitting packets, the client cannot guess the sequence of packets, as illustrated by Figure 1b. By *randomly* skipping some packet numbers, the server can quickly identify malicious receivers and potentially close the connection.

11 of 16 server implementations are vulnerable to the OACK attack. We run each server implementation in the runner to collect its emitted packets to transfer a 30 MB file. We consider that the server skips packet numbers when we see gaps in the sequence of emitted packets on the server during the data transfer. Out of the 16 tested servers, 11 do *not* skip packet numbers, including the implementations from Meta (*mvfst*), Microsoft (*msquic*), Alibaba (*xquic*), and Mozilla (*neqo*). Except *msquic*, the maintainers of these implementations confirmed their lack of packet number skipping support. We inspected the source code of *msquic* to confirm that it does not skip packet numbers.

Assessing acknowledgments. During this measurement, we noticed that several implementations do not assess the correctness of received acknowledgments, i.e., the server does not verify if the `ACK` frames sent by the client contain ranges of packets that were not (yet) sent by the server. For example, the client acknowledges packets 0..20 while only 12..16 are in flight. Such a server would extract the 12..16 range without considering that the 17..20 range corresponds to packets that were not sent. By skipping this verification step, the server is exposed to straightforward OACK attacks, as the malicious client does not need to accurately predict the packet number sequence.

Quiche, kwik, and aioquic do not verify the correctness of received acknowledgments. To identify which implementations are vulnerable to condition (2), we create a simple client continuously acknowledging the $0..10^6$ range to ensure we do not offload the flow control limits from the QUIC Interop Runner initial values. We observe that *quiche*, *kwik*, and *aioquic* do not assess the correctness of received `ACK` frames. We reported this issue to the maintainers of these three implementations. While this lack of acknowledgment assessment is a strong vulnerability, this paper focuses on skipping packet numbers to protect from OACK attacks.

4 Implementations vulnerabilities

In Section 4.1, we design our OACK predictor and implement it in a client application based on Cloudflare *quiche*[1]. Then, we produce the OACK attack on the 11 vulnerable server implementations in a controlled environment described in Section 4.2. To be as generic as possible, we consider that the server verifies the correctness of the received acknowledgments and closes the connection otherwise.

4.1 Optimistic acknowledgment predictor

An OACK predictor must fulfill two properties: (1) Generate acknowledgment ranges faster than if the corresponding packets were actually received; (2) The server must not

[1]We choose *quiche* only due to our affinity with the implementation.

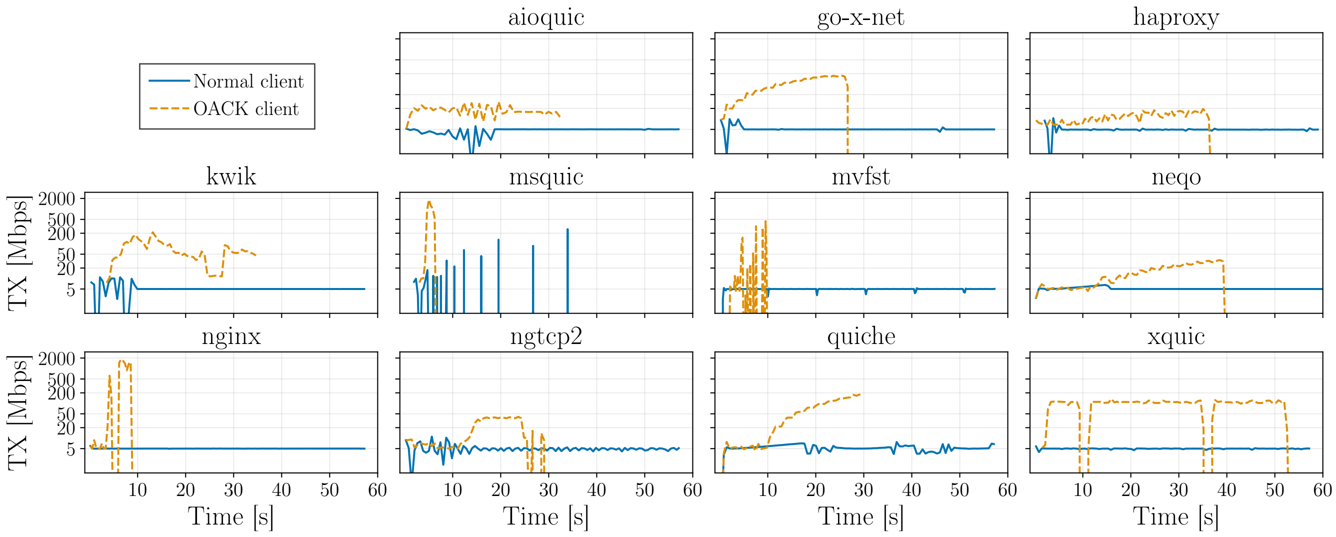


Figure 3: Transmission bit rate measured on the server egress with a client limited at 5 Mbps **reception bandwidth, using the vulnerable servers' implementations taking part in the QUIC Interop Runner.**

receive ranges for unsent packets. Designing this predictor in QUIC is trickier than designing it with TCP. Indeed, RFC9293 [4] (Section 3.5.2) states that upon receiving an invalid acknowledgment (such as an OACK), a TCP server should ignore the ACK and send an empty packet with the correct sequence number without sending a RST to the client. This design follows Postel's principle: "Be conservative in what you send, and liberal in what you accept". OACK predictors in TCP could rely on this mechanism to synchronize the OACK with the server if they send acknowledgments too quickly [15]. On the contrary, a QUIC server immediately closes the connection if it detects invalid acknowledgments, requiring smarter predictors. Algorithm 1 presents our OACK predictor for QUIC, introducing two parameters. To prevent the client from injecting OACKs too early in the connection, we delay the start of the attack by l received packets (line 4). For simplicity, we call n_0 the first packet number used by the server for delivering data. The core idea of the predictor is to acknowledge each received packet with an increasing delta. Whenever the client receives a new packet with an increased packet number n, it sends an ACK frame acknowledging packets $n_0..n+d$. As such, the client acknowledges d more packets that have not yet been received. The trick continuously increases by i_d the value of d because the attack induces congestion in the network while the client keeps increasing the server's bit rate.

Algorithm 1: Simple QUIC OACK predictor.

```
$i_d$: aggressiveness of increase of $d$
$l$: number of packets before starting OACK
$d$: OACK increase. Init: 0
$n_m$: maximum received packet number
$n_0$: first packet number with data. Init: None
$n$: received packet number
Result: Potential OACK ranges to send to the peer
1 if $n_0$ is None then
2     $n_0 \leftarrow n$
3 end
4 if $n_0 + l < n$ or $n < n_m$ then
5     return No range;
6 end
7 $n_m \leftarrow n$;
8 $d \leftarrow d + i_d$;
9 return $n_0..n + d$;
```

4.2 Exploring QUIC stacks vulnerabilities

In this Section, we carry out the OACK attack on the 11 vulnerable QUIC server stacks from Table 1. During the following experiments, we set $i_d = 4$ and $l = 400$. Due to space limitations, we do not explore other sets of values.

Virtual testbed. We use the virtual testbed illustrated in Figure 2 on a single machine to encourage the reproducibility of the experiments. We work with NPF [1] to orchestrate and reproduce our experiments. We leverage the *docker* images from the QUIC Interop Runner [13, 14] to run the servers in their namespace (`docker`). Our malicious client runs in its namespace (`client`), and a Linux bridge connects these two namespaces. Using `tc`, we add a 5 ms delay in both directions on the main namespace. We add a 5 Mbps bandwidth limit towards the client to show the attack's impact in a constrained environment. The client requests a 600 MB file through an HTTP request, i.e., content large enough to expose the OACK behavior. We limit each experiment to 60 s. Our malicious client uses small 10 MB initial flow control

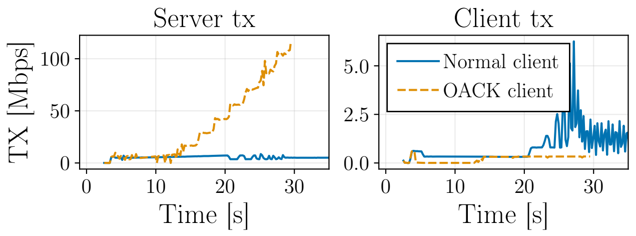


Figure 4: Server (left) and client (right) bandwidth of our OACK client against a *quiche* server.

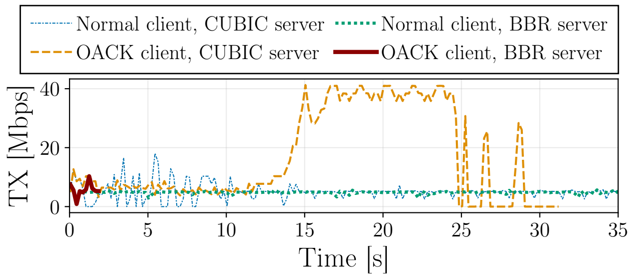


Figure 5: Bandwidth measured on the *ngtcp2* server when using CUBIC against BBRv2.

limits and increases them based on the optimistic acknowledgments it sends. We measure the server's bit rate at its egress.

Our OACK predictor tricks the 11 servers into increasing their transmission bit rate up to 200× compared to normal behavior. Figure 3 shows the server bit rate with and without OACK enabled on the client. The duration and success of the attack vary across QUIC implementations. For example, *msquic* closes the connection because the OACK predictor sent an acknowledgment for an unsent packet after ~2 s of the attack. However, during this duration, the server sends up to 1.5 Gbps of traffic, a 300× increase to the actual network bandwidth. Other implementations, such as *xquic* and *mvfst*, send between 100 and 300 Mbps traffic to the network during 50 s and 5 s respectively. In conclusion, Figure 3 assesses that our OACK predictor can attack these servers by making them send data too quickly into the network despite its simplicity.

The client transmission pattern does not indicate any malicious behavior. Figure 4 analyzes the client transmission bit-rate (right sub-Figure) when sending OACKs. Our predictor only sends OACKs (with extended ranges) when it receives a packet from the server. As such, the bit rate towards the server follows normal behavior. Considering that QUIC packets are end-to-end encrypted, a middlebox cannot identify the attack by analyzing the client's traffic. We even note that without OACK, the client's transmission bit rate is higher. The client sends larger ACK frames with more ranges due to the gaps induced by the congestion losses.

CUBIC vs BBR on the server. Our OACK predictor relies on the hypothesis that the servers use loss-based congestion control algorithms (CCA), such as CUBIC [18]. Such algorithms never decrease their congestion window unless losses have been reported by their peer. However, non-loss-based CCAs such as BBR [2] use a different behavior to adapt the congestion window. In 2019, BBR accounted for ~40 % of Internet traffic [9]. A BBR server may proactively decrease its congestion window during a *probing phase*. If a client cannot guess when the server enters such *probing phase*, it might send an acknowledgment for an unsent packet because of the decreased throughput. Figure 5 shows the impact of using BBR instead of CUBIC with an OACK client and the *ngtcp2* server. Because the server's congestion window decreased without seeing losses, our simpler predictor sent acknowledgments for unsent packets, triggering the connection's closing. While using BBR first seems like a good remedial strategy, a smarter predictor could predict such *probing phases* to attack a BBR server; we leave it as future work. Figure 6 explores the same scenario with *quiche*, which does not assess the correctness of the received ACK frames (condition (2)). Interestingly, the server's bit rate with BBRv2 is more than doubled compared to CUBIC. Sending these acknowledgments optimistically decreases the server's perceived RTT, increasing its transmission bit rate.

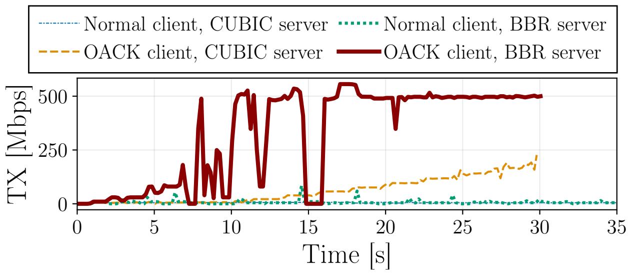


Figure 6: Bandwidth measured on the *quiche* server when using CUBIC against BBRv2.

5 OACK on a deployed CDN server

A large Content Delivery Network (CDN) agreed to let us measure the impact of a *controlled* OACK attack against its network. We freely host a website consisting of static images on the CDN, which handles the deployment and network stack for interacting with the website. The request consists of downloading eight images for a total of 17 MB. The client is connected to the Internet through a Turris Omnia access point to measure the impact of the OACK attack while remaining careful. We limit the bandwidth from the Turris towards the client to 1 Mbps with `tc` and measure the bit rate sent by the server at the ingress of the Turris. We use a hard 1 Mbps limit to better reflect the attack's effect without aggressively attacking the CDN network.

We can perform the OACK attack on the deployed CDN server. We use our predictor from Algorithm 1 with $l = 400$ and $i_d = 10$. Figure 7 reports the results. The *Baseline*

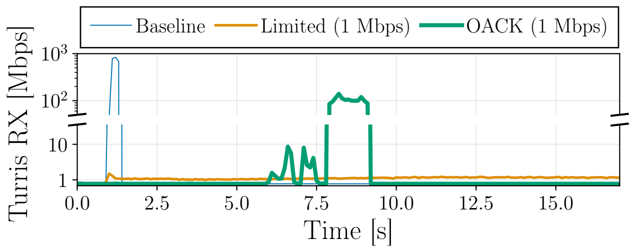


Figure 7: Bandwidth measured on the Turris access point. The client makes an HTTP/3 request to our website hosted on a large Content Delivery Network.

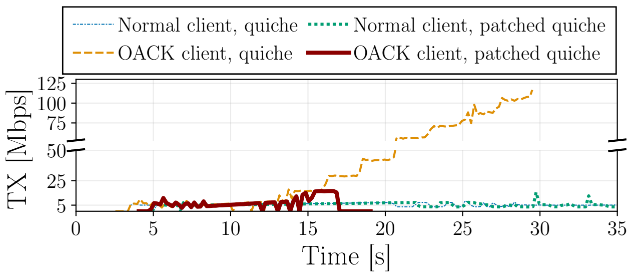


Figure 8: Throughput on the fixed *quiche* server.

curve shows the expected throughput *without* the 1 Mbps bandwidth limit, indicating that we can expect to receive data up to 700 Mbps. With the bandwidth limit and a benign client (*Limited* curve), we see that the reception bit rate does not exceed 1.1 Mbps. However, the *OACK* curve indicates that our malicious client correctly increases the server's transmission bit rate because of the OACKs; we receive up to 100 Mbps traffic despite the hard 1 Mbps theoretical limit. To avoid taking the risk of *actually* attacking the CDN, we do not attempt to reach the 700 Mbps baseline bit rate. However, we argue that the results from Figure 7 show the potential of the attack if executed more aggressively against a deployed server.

6 Preventing the OACK attack

To protect a QUIC host against OACKs, the implementation must (1) assess the validity of the received acknowledgment and (2) randomly skip packet numbers during the transfer.

Assessing acknowledgments' validity. To assess the validity of received ACK frames, QUIC endpoints must match the acknowledged ranges (of packet numbers) to the packet numbers sent. For example, Cloudflare *quiche*[2] loops over all unacknowledged packets, checking whether its packet number lies within the received ranges, not assessing its validity. Supposing that the endpoint uses continuously increasing packet numbers (the case of skipping packet numbers is analyzed later), a correction consists of verifying whether the ranges contain packet numbers higher than the maximum packet number the endpoint sent.

Skipping packet numbers. As stated by RFC9000 [7], QUIC endpoints *may* skip packet numbers to prevent the optimistic acknowledgment attack. It is sufficient to *randomly* skip a few packet numbers during the connection to limit the overhead induced by the increased number of ranges caused by the gaps. For example, *quic-go* randomly skips a packet number at most every 131,072 packets; *picoquic* randomly skips a packet number uniformly in the range $[3, 256^{1+n_a}]$ where n_a is the number of already skipped packet numbers. To detect an OACK attack, both implementations create dummy and unsent packets associated with these skipped numbers. When the endpoint receives an ACK from its peer, it checks whether the ranges acknowledge the dummy packet; if so, they close the connection.

Patching *quiche*. We patched *quiche* with the ACK correctness and packet number skip. It adds 187 and deletes 16 lines of code. We run our OACK client against this modified *quiche* server in the same setup as in Section 4.2. Figure 8 illustrates that the patched server is not vulnerable anymore to the OACK attack. The server skips a packet number with a uniform probability of at most one every 131,072 packets. When the malicious client enters the OACK phase, the skipped packet number triggers the close of the connection on the server (after 17 s). The corrected server sends at most 18 Mbps in its network before noticing the attack. The server could detect the attack faster by skipping more packet numbers, at the cost of increasing the overhead and the number of ranges acknowledged in ACK frames.

7 Conclusion

This paper explores the exposure of the QUIC protocol [7] to the optimistic acknowledgment (OACK) attack, initially discussed in TCP [11]. While QUIC's specification [7] suggests skipping packet numbers to prevent the attack, we showed that 11 of the 16 server stacks participating in the QUIC Interop Runner [13, 14] are exposed to OACKs. With our simple OACK predictor, we illustrated the attack on these implementations and demonstrated its efficiency both in an emulated and a real network. Because QUIC packets are end-to-end encrypted, middleboxes and firewalls cannot identify an OACK attack upstream. Worse, a third party cannot observe its attack because the client's acknowledgment rate follows benign behavior. We propose a 180-line patch to *quiche*, one of the most vulnerable stacks, and show its resilience to the OACK attack when correctly implementing the recommendations. QUIC servers become robust to the optimistic acknowledgment attack by skipping packet numbers. While QUIC's specification [7] exposes this feature as a possibility, we argue that *MAY is not enough* and *QUIC servers SHOULD skip packet numbers.*

[2]Version 0.23.5, released on April 2, 2025

Acknowledgments

The Walloon Region has partially supported this work as part of the funding of the FRFS-WEL-T strategic axis. We thank the ANRW reviewers for their valuable feedback.

Ethical considerations

We contacted the lead developers of the vulnerable implementations before submitting the paper. The discussions are ongoing, and we expect they can update their implementation before the paper is published. Multiple implementations have already been fixed after our discussions. Our patch for *quiche* requires only 180 lines, including comments and updates in existing tests. We conducted almost all our experiments inside our labs. The real network measurements were performed carefully and with the agreement of the target CDN. We constrained our network during this experiment and ensured it did not exceed our *real* bandwidth share during the attack. Finally, the optimistic acknowledgment attack is well-known and even mentioned in QUIC's specification [7].

Artefacts

The conducted experiments are fully reproducible with the current *docker* images of the QUIC Interop Runner. We release these experiment scripts, our OACK predictor client, and our patch to *quiche*: https://github.com/louisna/quic-optimistic-ack-anrw.

References

[1] Tom Barbette. 2024. Poster: NPF: orchestrate and reproduce network experiments. In *2024 ACM Conference on Reproducibility and Replicability.*

[2] Neal Cardwell, Ian Swett, and Joseph Beshay. 2025. *BBR Congestion Control.* Internet-Draft draft-ietf-ccwg-bbr-02. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-ccwg-bbr/02/ Work in Progress.

[3] Cloudflare. 2025. Savoury implementation of the QUIC transport protocol and HTTP/3. https://github.com/cloudflare/quiche Version 0.23.5.

[4] Wesley Eddy. 2022. Transmission Control Protocol (TCP). RFC 9293. doi:10.17487/RFC9293

[5] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Dr. Allyn Romanow. 1996. TCP Selective Acknowledgment Options. RFC 2018. doi:10.17487/RFC2018

[6] Jana Iyengar, Ian Swett, and Mirja Kühlewind. 2025. *QUIC Acknowledgment Frequency.* Internet-Draft draft-ietf-quic-ack-frequency-11. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-ack-frequency/11/ Work in Progress.

[7] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. doi:10.17487/RFC9000

[8] Abir Laraba, Jérôme François, Shihabur Rahman Chowdhury, Isabelle Chrisment, and Raouf Boutaba. 2021. Mitigating TCP protocol misuse with programmable data planes. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 760–774.

[9] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. 2019. The great internet TCP congestion control census. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 3 (2019), 1–24.

[10] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. doi:10.17487/RFC8446

[11] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. 1999. TCP congestion control with a misbehaving receiver. *ACM SIGCOMM Computer Communication Review* 29, 5 (1999), 71–78.

[12] Alexander Schaub and Deitch Trey. 2016. https://reproducingnetworkresearch.wordpress.com/2016/05/30/cs-244-16-misbehaving-tcp-receivers-can-cause-internet-wide-congestion-collapse/comment-page-1/. CS244 '16: Misbehaving TCP receivers can cause Internet-wide congestion collapse.

[13] Marten Seemann. 2025. QUIC Interop Runner. https://interop.seemann.io/ Accessed on March 27, 2025.

[14] Marten Seemann and Jana Iyengar. 2020. Automating QUIC interoperability testing. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC.* 8–13.

[15] Rob Sherwood, Bobby Bhattacharjee, and Ryan Braud. 2005. Misbehaving TCP receivers can cause Internet-wide congestion collapse. In *Proceedings of the 12th ACM conference on Computer and communications security.* 383–392.

[16] Randall R. Stewart. 2007. Stream Control Transmission Protocol. RFC 4960. doi:10.17487/RFC4960

[17] QUIC working group (IETF). 2025. QUIC Implementations. https://github.com/quicwg/base-drafts/wiki/Implementations Accessed on April 4, 2025.

[18] Lisong Xu, Sangtae Ha, Injong Rhee, Vidhi Goel, and Lars Eggert. 2023. CUBIC for Fast and Long-Distance Networks. RFC 9438. doi:10.17487/RFC9438