USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Breaking the Boundaries: Analyzing QUIC Frame-Packet Interactions With QUIC-Attacker

Nurullah Erinola, Marcel Maehren, Marcus Brinkmann, and Jörg Schwenk, *Ruhr University Bochum*

https://www.usenix.org/conference/usenixsecurity26/presentation/erinola

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Breaking the Boundaries: Analyzing QUIC Frame-Packet Interactions With QUIC-Attacker

Nurullah Erinola, Marcel Maehren, Marcus Brinkmann, and Jörg Schwenk

Ruhr University Bochum

Abstract

QUIC is a new network protocol based on UDP that replaces TCP and TLS with an integrated protocol. It provides multiplexing of streams over a single encrypted and authenticated connection. The QUIC standard allows many different combinations of UDP datagrams, and QUIC packets, frames, and streams to transport the same information. This implies that testing the *receiving* side of QUIC is difficult.

We develop probes to explore how different QUIC server implementations handle the coalescence and fragmentation of payloads, covering both valid and invalid combinations of datagrams, packets, and frames. Already at this basic level, we observe significant differences between implementations, some of which pointing towards exploitable vulnerabilities. Previous QUIC research tools were not designed to implement such probes. To address this limitation, we present *QUIC-Attacker*, a testing framework that allows maximum freedom on the *sending* side of QUIC.

We present our results on these probes when applied to 15 QUIC server libraries, uncovering eight DoS vulnerabilities caused by unhandled exceptions and exploitable injection vulnerabilities in Kwik and Alibaba's XQUIC.

1 Introduction

The Internet of today is built on a network stack of independent protocols, including the Internet Protocol (IP), Transmission Control Protocol (TCP), Transport Layer Security (TLS), and application-layer data protocols such as the Hypertext Transfer Protocol (HTTP). The division into several layers of abstraction, assigning a clear role to each layer across the stack, has enabled the rapid growth of the Internet. However, the combination of TCP and TLS has the drawback that both require a handshake for connection establishment. Since TLS and TCP are located at different layers of the network stack, the TLS handshake can only begin after the TCP handshake, thus delaying initial communication by the sum of their Round-Trip-Times.

QUIC QUIC is a new protocol designed to address the TCP/TLS overhead, while also improving security through encryption by default. Launched by Google in 2012, QUIC was submitted to the Internet Engineering Task Force (IETF) in 2015. This initial version also contained a novel key exchange protocol [11], which was later abandoned in favor of the TLS 1.3 handshake [17]. QUIC version 1 was published in the RFCs 8999 to 9002 [9, 10, 25, 26] in 2021. Already one year after publication, Cloudflare reported that approximately 30% of web requests to their servers used QUIC.[1]

Cross-Layer Protocol QUIC is a highly complex protocol that integrates several traditionally separate layers into a single protocol. Internally, QUIC consists of multiple interacting components: *packets*, *frames*, and *streams*. Packets form the outer transport unit; their payload consists of one or more frames. Some frame types contain control information, others carry stream data used to transmit TLS handshake messages and application data. This interaction introduces strong dependencies between packets, frames, and streams: there are strict definitions of allowed packet/frame and frame/stream combinations; the state machine of the TLS 1.3 handshake puts additional restrictions on which packets may be sent at a certain point of time; retransmission handling is now bound to packets, while establishing and resetting data streams is bound to control frames. When implementing this complex communication protocol, Postel's law - *be conservative in what you send, and liberal in what you receive* - imposes a greater burden on the correct implementation of the *receiving* side of QUIC.

In this paper, we perform three systematic probes into one aspect of the complexity of correctly handling received QUIC packets: *coalescence*. QUIC implementations are free to coalesce frames into packets and packets into datagrams, and fragment stream data across multiple frames in arbitrary ways. In the spirit of Postel's law implementations will probably send data in *one* particular form, but must be able to receive

[1] https://blog.cloudflare.com/http3-usage-one-year-on/

data sent in *all* possible forms. This motivates our first research question:

> **RQ1:** Does the coalescence of packets, frames, and stream data affect the processing by QUIC recipients in unexpected ways? Do boundaries of datagrams, packets, and frames influence the processing of the contained packets, frames, and stream data?

Frame Types QUIC introduces a large set of frame types. RFC 9000 defines 20 different core frame types, including frames for stream management, flow control, and connection level control. Certain frame types may only be used after the secure channel has been established. Therefore, RFC 9000 clearly specifies a whitelist about which packet types are allowed to carry which frame types, and under which conditions specific packets and frames may be sent. The complexity of these relationships can lead to subtle implementation pitfalls. This motivates our second research question:

> **RQ2:** Do implementations respect the binding between frame types and packet types as specified, even in different coalescence scenarios? Can violations lead to exploitable security vulnerabilities?

Interaction of Frame Types Some frame types (e.g., `RESET_STREAM` and `STOP_SENDING`) can affect the processing of another, subsequently sent frame type (e.g., `STREAM`). This raises the question if the interpretation of these cases is consistent across different implementations.This motivates our third research question:

> **RQ3:** Are QUIC implementations consistent in how they interpret the interaction of frames affecting stream states, even in coalescence scenarios?

Challenges in Evaluating QUIC Implementations To answer our research questions, we require a *flexible* and *scalable* analysis tool that (a) provides us with full control over the coalescence of packets and frames, and their content (RQ1-3), (b) allows us to bypass the binding of packets and frames (RQ2), and (c) provides control over the cryptographic state machine, since splitting frames of one packet in multiple packets (or vice versa) requires decryption and re-encryption. Ideally, this tool should also be *extensible* to add future test cases with minimal overhead.

Unfortunately, current QUIC analysis tools do not meet these requirements. Prior work primarily relies on modified QUIC/TLS libraries [1, 16, 27], network simulators [20], or general-purpose fuzzing frameworks [2] (cf. Table 1). Tools based on libraries are designed to remain RFC compliant and are influenced by Postel's law. They typically implement a single encapsulation and coalescence strategy, which limits its systematic evaluation of alternative strategies. Network simulation-based approaches are suitable for interoperability testing, but do not provide control over the cryptographic state. General-purpose frameworks offer scalability and automation but lack native support for protocol-specific semantics. This prevents immediate and fine-grained testing. We provide a detailed discussion of the existing analysis tools and their limitations in Section 2. This gap in the current tool landscape motivated us to develop a new reusable and extensible analysis tool.

QUIC-Attacker Our work is based on a novel, complete, and flexible QUIC implementation, *QUIC-Attacker*, that fulfills requirements (a), (b), and (c) stated above. Our implementation is based on the well-established framework *TLS-Attacker* [3, 22], re-using the TLS 1.3 support already implemented, but with added features for QUIC network components (packets, frames, and streams) and the QUIC packet encryption layer (which replaces the TLS 1.3 record layer) (Section 5). Specifically, we implement support for all packets and frames as specified in RFC 9000, and introduce a new *frame layer* and a new *packet layer* that allows sending and receiving arbitrary QUIC messages within a protocol flow. Leveraging the existing architecture of TLS-Attacker, as well as its TLS implementation, extended by our QUIC implementation, the tool provides complete control over the construction and modification of packets and frames, and enables the execution of arbitrary QUIC message flows. Using this extended framework, which we refer to as *QUIC-Attacker*, we implement a comprehensive set of test cases for testing *coalescence* (Section 6 and Table 2). These test cases also demonstrate the flexibility and effectiveness of *QUIC-Attacker* and highlight its suitability for deep analyses of the new protocol.

Methodology To address **RQ1**, we construct a catalog that tests if the coalescence of TLS handshake data or application data in packets and datagrams changes the processing of the data at the receiver. The catalog uses three main packet types (`Initial`, `Handshake`, and `1-RTT`) and puts exemplary, whitelisted data streams (`TLS ClientHello`, `TLS Finished`, and `HTTP/3 GET`) in one or two packets, and one or two datagrams. Table 3 summarizes the results. To address **RQ2**, we construct a test catalog that evaluates if implementations correctly reflect QUIC's restrictions of allowed frames per packet type, under varying coalescence. The same packet types as above are used, as they correspond to different states of the QUIC crypto engine. In addition to sending standalone blacklisted frames in these packets (cf. [1, 6]), we test if coalescence of a whitelisted frame with a blacklisted frame within one packet, one datagram or two datagrams influences processing at the receiver. Table 4 summarizes the results. Finally, to answer **RQ3**, we create a test catalog that focuses on frames responsible for controlling streams. These include `RESET_STREAM`, `STOP_SENDING`, and the exchange of application data itself (`STREAM`). In this catalog, we combine such

frames and analyze whether implementations handle these scenarios consistently. Table 5 summarizes our oberservations.

Findings First, encapsulation and coalescence choices matter in practice, as seven implementations fail to correctly process coalesced `Initial` packets, showing that packet boundaries and coalescing behavior can affect processing in unexpected ways (Section 7.1). Second, we observed that the frame-packet constraints mandated by specification are not consistently enforced. Two implementations accept frames in packet types where they are forbidden, and for one of them we show that these violations enable exploitable behaviors through message injection, including a resource confusion scenario and a truncation attack (Section 7.2). We additionally uncovered that sending a forbidden frame as a client can trigger crashes in one implementation. Third, we found that QUIC implementations are not consistent in how they interpret and process interacting stream state frames, leading to divergent stream states across implementations (Section 7.3). In this context, our tests also uncovered seven additional crashes. Beyond our main evaluation, we further found that QUIC implementations exhibit incorrect interactions between the TLS and QUIC layer (Section 7.4).

Our findings highlight that introducing a new protocol and merging several network layers provides new optimization opportunities, but also increases implementation risks due to the resulting complexity.

Contributions Our main contributions are as follows:

- We provide a novel tool, *QUIC-Attacker*[2], for evaluating QUIC server and client implementations. Its design emphasizes reusability and extensibility while providing complete control over the construction and modification of datagrams, packets, and frames, thereby allowing the execution of arbitrary message flows.
- We introduce the idea of testing *coalescence* in QUIC implementations under systematically varied datagram, packet, and frame compositions.
- We create 27 tests templates across three test catalogs to reveal processing inconsistencies, compliance failures, and security vulnerabilities in QUIC implementations.
- We provide the *QUIC-Docker-Library*[3], a Docker-based collection of 15 open-source QUIC implementations, and use it to evaluate our test catalogs in a reproducible testing environment.
- Our evaluation reveals various processing inconsistencies regarding semantically identical, but differently encapsulated payloads. Among these, we identified eight DoS vulnerabilities and exploitable injection vulnerabilities in Kwik and Alibaba's XQUIC.

[2]We contribute *QUIC-Attacker* as part of the *TLS-Attacker* suite: https://github.com/tls-attacker/TLS-Attacker

[3]https://github.com/tls-attacker/QUIC-Docker-Library

2 Related Work

Analysis of QUIC Implementations QUIC implementations have been evaluated extensively in controlled lab environments. Such analyses were performed both during the standardization process [6, 12, 16, 20, 24] and after the publication of the final specification [1, 2, 5, 7, 23, 27]. Two of these studies have also shown that some QUIC implementations incorrectly process forbidden frame-packet combinations: Ang et al. [1] examined whether implementations incorrectly accept a `CRYPTO` frame inside a `0-RTT` packet, while Gagliardi and Levillain [6] examined whether a `PING` frame is incorrectly processed when received in an `Initial` packet. However, neither of these works systematically examined all frame-packet combinations defined by the specification. In addition, existing studies consider frames and packets in isolation and do not investigate how different encapsulation strategies influence implementation behavior, such as coalescing multiple frames into a single packet or coalescing multiple packets into a single datagram.

Large-Scale Scans on the QUIC Ecosystem The QUIC ecosystem has also been the subject of several large-scale studies, which tracked its deployment and traffic over time [19], analyzed configuration choices of deployed servers [27], collected provided certificates to assess their impact on the performance [14], and fingerprinted QUIC libraries in the wild [28].

Development of QUIC Analysis Tools Multiple studies build tools to conduct their analysis and facilitate future research. These tools differ substantially in their design objectives and are each tailored to *one* distinct methodological focus (cf. Table 1). Piraux et al. developed *QUIC-Tracker* [16], a test suite that evaluates servers against selected statements from the QUIC specification. It builds on a TLS library to implement a minimal QUIC stack based on draft 29, supporting only the protocol functionality required for its specific test suite. Building on this work, Ferreira et al. developed *Prognosis* [5], a framework for state machine learning. They

Tool	Year	Ref.	Goal	Built On	Target
QUIC-Tracker	'18	[16]	Test Suite	picotls	server
Prognosis	'21	[5]	SML	QUIC-Tracker	server
QScanner	'21	[27]	Scanning	quic-go	server
QuicInteropRunner	'20	[20]	IT	ns-3	client, server
QUIC-Fuzz	'25	[2]	Fuzzing	AFLNet	server
QUICTester	'25	[1]	SML	aioquic	server
QUIC-Attacker	This work		Flexible QUIC Stack	TLS-Attacker	client, server

SML State Machine Learning **IT** Interoperability Testing
client Client Implementations server Server Implementations

Table 1: Comparison of research tools for the evaluation of QUIC implementations sorted by publication year.

extended QUIC-Tracker to support their limited input alphabet for state learning, but neither updated the QUIC implementation to the final QUIC specification nor implemented the full set of protocol features. Seemann and Iyengar introduced the *QuicInteropRunner* [20], an automated testing framework designed to assess interoperability between different libraries under configurable network conditions using the ns-3[4] simulator. However, this approach provides no control over the cryptographic state and therefore cannot be used to manipulate the encrypted packets or vary encapsulation boundaries. Zirngibl et al. developed *QScanner* [27], a stateful scanner based on a QUIC library that performs QUIC handshakes to extract supported QUIC versions, QUIC transport parameters, and TLS parameters from servers. While well suited for Internet-wide measurements, it builds on a QUIC library that is subject to Postel's law and therefore only needs to implement a single encapsulation strategy to remain RFC compliant, making it unsuitable for systematic testing of varying strategies. Kian Kai et al. presented *QUIC-Fuzz* [2] and *QUIC-Tester* [1], tools for fuzzing and state learning. These tools are built on top of QUIC libraries (aioquic) or a general-purpose testing framework (AFLNet). However, AFLNet does not provide native QUIC support and cannot be used for immediate testing. In Table 1, we summarize these tools for QUIC analyses along with their design goals. We exclude studies that rely on static analysis [23], reuse existing tools [6, 12, 14, 19, 28], or modify a QUIC library to support a specific, immediate use case [7, 24] without the goal of building a dedicated tool.

Overall, the current tool landscape provides many specialized solutions but offers limited support for systematically applying diverse testing strategies across QUIC implementations.

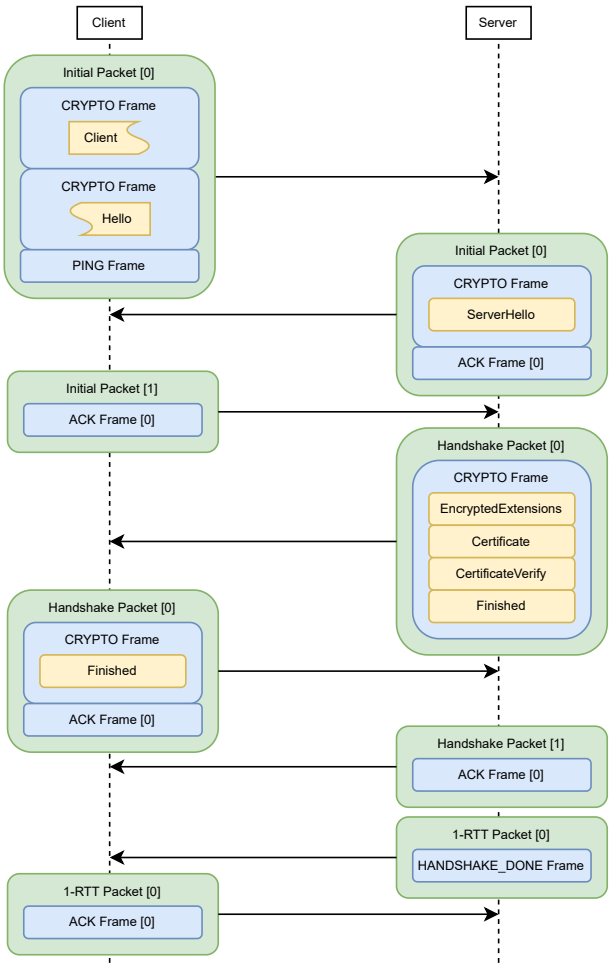


Figure 1: An example QUIC flow illustrating the complex interplay between QUIC packets, QUIC frames, and TLS handshake messages. For simplicity, `PADDING` frames are omitted.

3 Background

QUIC [9, 10, 25, 26] is a secure, general-purpose transport protocol between a server and a client, based on UDP. To achieve security goals like confidentiality, integrity, and authentication. It integrates the handshake from the TLS 1.3 specification (cf. Figure 1), but not the TLS record layer. QUIC is mandatory to use with HTTP/3 (RFC 9114 [4]), and may be used for other protocols like DNS (RFC 9250 [8]).

3.1 QUIC Components

Datagrams QUIC runs over UDP, so UDP datagrams are used to transport QUIC packets. One datagram can be used to transport one or more packets.

Packets QUIC packets are the outer building blocks and are always encrypted and integrity protected using AEAD, with keys depend on the state of the TLS 1.3 state machine. `Initial` packets are used before any key state has been established; their keys are derived from a static value in the specification and a public value contained in the header of the packet. Consequently, these packets are not protected against man-in-the-middle attacks. `Handshake` packets use the TLS 1.3 handshake keys, while `1-RTT` packets use the TLS 1.3 application data keys.

Beyond encryption and key usage, QUIC detects data loss at the packet level. For each type of packet, each QUIC peer maintains a separate packet number starting at 0. Acknowledgements for the reception of a certain packet number must be sent in an `ACK` frame within the same packet type. For example, in Figure 1, the `Initial Packet [0]` containing the `ClientHello` message in two `CRYPTO` frames is acknowledged by an `ACK` frame contained in the server's `Initial Packet [0]`. In this work, we do not consider packet loss.

There are two formats of packets: *long header* packets and *short header* packets. Long header packets are used during

[4] https://www.nsnam.org/

connection establishment and include additional header fields, such as the QUIC version. Short header packets, in contrast, are used once the connection has been established and carry less metadata.

Frames QUIC frames are transported in the payload of QUIC packets. One packet can hold one or more frames of any type (even mixed). Frames convey structured protocol information; each frame has a type of information it carries. Some frame types are essential for connection management, such as `ACK` frames, which acknowledge received packets, and `CRYPTO` frames, which transport TLS handshake messages. Others provide transport functionality, for example `STREAM` frames that deliver application data, or `CONNECTION_CLOSE` frames that signal connection termination.

Streams QUIC uses streams as an abstraction for transporting application data. Streams are ordered byte streams that can be unidirectional or bidirectional. Multiple independent streams can be used concurrently, enabling multiplexing of application data within a single connection. Each stream provides in-order delivery for its own byte sequence while remaining independent of other streams. QUIC also defines simple mechanisms for stream management, including flow control and the opening and closing of streams. Each QUIC stream is identified by a numeric stream ID that also encodes both the initiating endpoint and the directionality of the stream.

Dependencies In HTTP/2 using TLS over TCP, the layers have a simple dependency: Once the TCP channel is established, a single stream of plaintext bytes may be sent. After the TLS channel is established, again a stream of bytes may be sent. In QUIC, dependencies are much more subtle: The acknowledgement of transmissions is tied to QUIC packets, not UDP datagrams. Packet types are tied to the state of the TLS handshake, and this handshake itself is done using special packets and special frames.

3.2 Establishing a QUIC Connection

A QUIC connection always begins with a *handshake* which is responsible for negotiating cryptographic parameters and keys between a client and server. For this purpose, QUIC integrates TLS 1.3 [17] messages embedded within its own packets and frames, as specified in RFC 9001 [26]. Figure 1 illustrates how a QUIC connection is established. The client initiates the connection by sending an `Initial` packet that contains `CRYPTO` frames with the `ClientHello` message. The client's packet count starts with 0. The server responds with an `Initial` packet that includes a `CRYPTO` frame carrying the `ServerHello` message and an `ACK` frame acknowledging the client's `Initial` packet with packet number 0. After exchanging the `Hello` messages, both sides derives TLS handshake traffic keys, so QUIC switches to `Handshake` packets, using these keys for encryption. The server initializes its `Handshake` packet counter to 0 and sends the first packet containing a `CRYPTO` frame with the `EncryptedExtensions`, `Certificate`, `CertificateVerify`, and `Finished` messages. The client first acknowledges the server's `Initial` packet by sending an `Initial` packet with an `ACK` frame. Then it completes the TLS handshake with the `Finished` message in a `CRYPTO` frame in its only `Handshake` packet. The server must acknowledge the `Finished` with its second `Handshake` packet and an `ACK` frame. The TLS state machine has now switched to application data keys, and therefore QUIC is switching to `1-RTT` packets. The server then confirms handshake completion by sending a `HANDSHAKE_DONE` frame. Application data may now be exchanged in one or more streams within `1-RTT` packets. In this work, we do not consider session resumption and pre-shared keys.

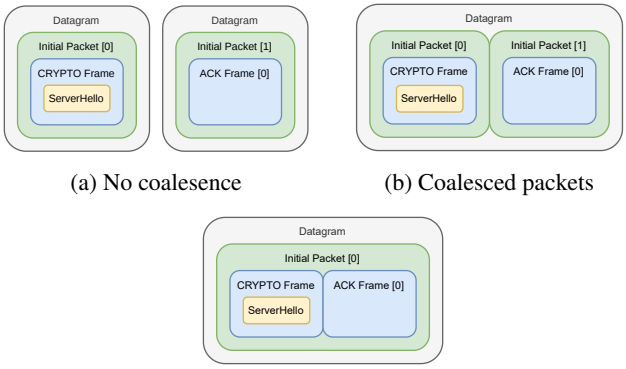
4 Challenges in Packet and Frame Processing

In this section, we carefully analyze RFC 9000 [10] to derive test cases related to *coalescence*.

4.1 Packets and Frames

Packets Within Datagrams A single datagram can contain multiple packets. RFC 9000 mandates that only packets with a *length* field, such as `Initial` or `Handshake` packets, can be followed by additional packets inside the same datagram. `1-RTT` packets, do not contain a *length* field and can therefore only appear as the last packet in a datagram. Table 6 in the appendix summarizes these constraints for all packet types defined in RFC 9000. When receiving a datagram, an endpoint must iterate over all packets it contains and process each one independently. Each packet carries its own header protection, encryption level, and payload. The receiver must attempt to process each packet even if a preceding packet fails to decrypt. A packet that cannot be decrypted may be discarded or buffered. This raises the question of whether all implementations handle these boundaries consistently or whether the combination of packets within a datagram can influence the subsequent processing of frames inside the packets.

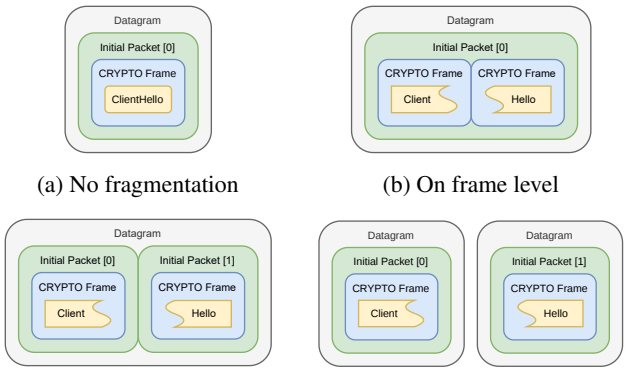
Frames Within Packets In contrast to packet coalescing, the specification does not provide explicit rules regarding which frames may or may not be combined within a single packet. Likewise, it does not define whether the ordering of frames inside a packet is semantically relevant or whether different compositions of frames are expected to yield identical behavior. As a result, it remains unclear whether different frame combinations or frame ordering can influence the behavior of an implementation. The payload of a packet is structured as a sequence of frames. RFC 9000 defines up



(a) No coalesence (b) Coalesced packets

(c) Coalesced frames

Figure 2: Using QUIC, three valid compositions exist for sending two independent frames, illustrated for the first message of the QUIC handshake sent from the server to the client.



(a) No fragmentation (b) On frame level

(c) On packet level (d) On datagram level

Figure 3: Using QUIC, four valid compositions exist for sending a fragmented stream of data, illustrated for the first message sent of the QUIC handshake from the client to the server.

to 20 different frame types, each serving specific protocol purposes. Additional frame types have been standardized in later RFCs (e.g., [15]), and more may be introduced in future specifications. The specification also defines strict rules regarding which frame types are allowed in which packet types. For example, `STREAM` frames, which carry application data, are permitted only in `1-RTT` and `0-RTT` packets. `Initial` and `Handshake` packets, by contrast, are restricted to frames required for connection establishment and key negotiation (e.g, `CRYPTO`). Table 7 in the appendix summarizes these constraints for all frame types defined in RFC 9000. Implementations must therefore validate the consistency between packet and frame types. Upon receiving a packet, an endpoint is required to verify that each contained frame is permitted for the given packet type before processing it. If a prohibited frame type is identified, the endpoint must treat this as a protocol violation and terminate the connection accordingly.

4.2 Coalescence and Fragmentation

To avoid potential denial-of-service or message injection vectors, the specification explicitly forbids packet fragmentation. Instead, larger protocol messages (e.g., certificate chains) as well as application data, may be fragmented across multiple frames, distributed over multiple packets, or even sent in separate UDP datagrams.

Case 1: Sending Multiple Frames (Figure 2) The simplest way to transmit two *independent* frames is to place each frame into its own packet and send each packet in its own datagram (2a). However, the specification explicitly allows additional compositions due to coalescing. For example, each frame may be contained in a separate packet, while both packets are coalesced into a single datagram (2b). Alternatively, both frames may be placed into a single packet and sent within one datagram (2c).

Case 2: Transmission of a Data Stream (Figure 3) A stream of data can be transmitted within a single frame, such as a `CRYPTO` or `STREAM` frame (3a). QUIC also allows fragmenting the data across multiple frames. For example, a data stream may be split into two frames and transmitted together within a single packet and datagram (3b). For `CRYPTO` frames, each frame may also be placed into a separate packet, while both packets are coalesced into a single datagram (3c). A fourth valid option is to send the packets separately in two datagrams (3d).

5 QUIC-Attacker

To study how QUIC implementations process coalescence and fragmentation, a tool is required that provides maximum flexibility in generating combinations of datagrams, packets, and frames. We therefore propose *QUIC-Attacker*, a new tool for the systematic analysis of QUIC server and client implementations. QUIC-Attacker is designed to deliberately relax the robustness principle, i.e. to not be conservative in its sending behavior, but to allow many different sending behaviors to be configured, even combinations forbidden by the standard.

5.1 Design Goals

Control Over Boundaries Our first goal is to provide fine-grained control over boundaries in QUIC. We aim to precisely control how payloads are fragmented into frames, how frames are coalesced into packets, and how packets are combined into datagrams.

Challenging Specification Constraints Our second goal is to challenge dependencies imposed by the specification at all protocol levels. We aim to inject frames into packet types where they are forbidden, to coalesce packet types that must

not appear together, and to transmit packets with arbitrary content.

Manipulation of Protocol Fields Our third goal is to enable unrestricted manipulation of any protocol fields. We aim to directly modify packet headers, manipulate frame contents, and generate cryptographically invalid packets.

5.2 Architectural Foundation

We build QUIC-Attacker on top of TLS-Attacker [3, 22], a well-established framework for the systematic analysis of (D)TLS client and server implementations. Further background on TLS-Attacker is provided in Appendix B.

Test Design in TLS-Attacker TLS-Attacker's behavior can be specified using *workflow traces*. A workflow trace describes the protocol flow of a session at a high level by defining a sequence of *actions*, which represent messages exchanged with a peer. TLS-Attacker distinguishes between *send* actions for transmitting messages and *receive* actions for processing incoming messages. Coarse manipulations, such as injecting an unexpected message into an otherwise benign flow, can be achieved by adding the additional message to a send action. To enable fine-grained manipulations, TLS-Attacker provides the concept of *modifiable variables*. These variables can be integrated into workflow traces to apply modifications to individual message fields at runtime. All remaining fields are generated dynamically by TLS-Attacker.

This architecture makes TLS-Attacker the ideal foundation for QUIC-Attacker, as we aim to give the user full control over the protocol flow while keeping the overhead for test cases minimal.

Expanding TLS-Attacker Internally, TLS-Attacker's processing logic is divided into *layers* that handle the (sub-) protocols involved in a (D)TLS session. For a DTLS flow, the *layer stack* consists of a *UDP* layer handling the data exchange, followed by the *record* layer providing cryptographic operations and (de-)fragmentation logic, and finally the *message* layer handling the TLS handshake messages. To implement QUIC-Attacker, we contribute two new layers: the *packet layer* and *frame layer*. In a QUIC session, these layers operate above TLS-Attacker's UDP layer handling the datagrams and beneath the message layer handling the TLS messages. The frame and packet layers handle the fragmentation logic and cryptographic operations of QUIC. As part of integrating these layers, we also implemented support for all QUIC packets and frames types within TLS-Attacker. A detailed overview of our extensions is shown in Appendix C.

5.3 Core Features

Support for All Frames and Packets QUIC-Attacker supports the full set of six packet types as well as the 20 frame types defined in RFC 9000, as listed in the tables of Appendix A. For each frame and packet, it provides a *preperator* for preparing the protocol fields, a *serializer* for transmission, a *parser* for reception, and a dedicated *handler* to update the internal state upon sending and receiving QUIC messages.

Automatic Wrapping and Unwrapping of TLS Messages By default, QUIC-Attacker automatically encapsulates TLS messages into `CRYPTO` frames and places these frames into suitable QUIC packets using our new layers. This behavior is applied whenever the user does not explicitly define the use of other frame or packet types for transmitting TLS messages. For receiving, QUIC-Attacker automatically collects data from the `CRYPTO` frames (possibly over multiple datagrams, packets and frames) until a complete TLS message has been received, before it is further processed.

Flexible QUIC Frame and QUIC Packet Layers By providing full control over the frame and packet composition process, these layers enable the explicit selection, ordering, and combination of frames and packets used during transmission. This also enables the construction of invalid or forbidden frame-packet combinations.

Flow Control QUIC-Attacker can automatically generate acknowledgments for received packets. This ensures that connections can progress normally without requiring users to explicitly model acknowledgments. QUIC-Attacker also supports automatic retransmissions when expected messages are not received. In such cases, it retransmits the last complete flight of messages. Both features are enabled by default but can be disabled.

Comprehensive Cryptographic State QUIC-Attacker stores all cryptographic parameters derived during a connection. By retaining all parameters, it enables the construction and processing of any protected packet at any time during a connection. This is possible even in protocol states where the specification would normally forbid the use of certain keys.

5.4 Using QUIC-Attacker Interfaces

We now demonstrate how the core features of QUIC-Attacker translate into practical usage. Starting from a *basic* QUIC handshake, we show how users can progressively refine message composition by operating on different abstraction levels, while reusing the same underlying workflow.

Defining a Workflow Trace We begin with a *basic* workflow that establishes a QUIC connection up to the receipt of the server's `HANDSHAKE_DONE` frame (cf. Figure 1):

```
WorkflowTrace wf = new WorkflowTrace();
wf.addTlsAction(new SendAction(new ClientHelloMessage()));
wf.addTlsAction(new ReceiveTillAction(new FinishedMessage()));
wf.addTlsAction(new SendAction(new FinishedMessage()));
wf.addTlsAction(new ReceiveQuicTillAction(new
    ↪ HandshakeDoneFrame()));
```

This workflow explicitly specifies the TLS messages to be sent (`2`, `4`) and which TLS messages (`3`) or QUIC frames (`5`) to wait for. All remaining protocol behavior, such as fragmentation into `CRYPTO` frames, packet construction, encryption, sending acknowledgments, and retransmissions, is done automatically by QUIC-Attacker in accordance with the specification. As a result, the `ClientHello` is serialized into a single `CRYPTO` frame and transmitted in an `Initial` packet, as illustrated in Figure 3a. This baseline reflects QUIC-Attacker's default behavior of generating protocol-compliant messages.

Refining Encapsulation Building on this baseline, users can selectively override the encapsulation to gain finer control over boundaries. To control fragmentation, e.g, the send action carrying the `ClientHello` can be extended with an explicit frame configuration. The following modification splits the `ClientHello` across two `CRYPTO` frames of 50 and 250 bytes:

```
SendAction sndCh = new SendAction(new ClientHelloMessage());
sndCh.setConfiguredQuicFrames(new CryptoFrame(50), new
    ↪ CryptoFrame(250));
```

QUIC-Attacker fragments the `ClientHello` accordingly and serializes the resulting frames into a `Initial` packet, as shown in Figure 3b. Importantly, the workflow trace itself remains unchanged. In the same manner, users can instruct QUIC-Attacker via dedicated configuration flags to serialize each frame into its own packet (cf. Figure 3c) or to place each packet into a separate datagram (cf. Figure 3d).

Adding Additional Payloads QUIC-Attacker allows users to add additional frames into packets. E.g., the following configuration adds a `PING` frame alongside the `CRYPTO` frames, resulting in the client's `Initial` packet shown in Figure 1:

```
sndCh.setConfiguredQuicFrames(new CryptoFrame(50), new
    ↪ CryptoFrame(250), new PingFrame());
```

This enables the construction of (non-)standard or conflicting frame combinations while reusing the same high-level workflow definition.

Modifying Protocol Fields QUIC-Attacker also allows direct manipulation of protocol fields. Users can explicitly construct packets and override individual protocol fields before transmission. The following snippet forces the use of a specific packet number in an `Initial` packet:

```
InitialPacket ip = new InitialPacket();
ip.setUnprotectedPacketNumber(Modifiable.explicit((byte) 3));
sndCh.setConfiguredQuicPackets(ip);
```

6 Methodology

6.1 Testing Fragmented Streams

To evaluate whether QUIC implementations correctly support packet coalescing and fragmentation across packets and datagrams, we construct a test catalog that varies how the *same* logical payload is encapsulated. Across all tests, we fragment the payload into two frames and transmit them in three encapsulation variants: (i) both frames are sent within a single packet (P1, P4, and P7), (ii) the frames are split across two packets that are coalesced into one datagram (P2, P5, and P8), and (iii) the frames are split across two packets transmitted in separate datagrams (P3, P6, and P9). We instantiate this strategy for different packet types and payloads. For `Initial` and `Handshake` packets, we split TLS handshake messages into two `CRYPTO` frames (`ClientHello` and `Finished`). For `1-RTT`, we split application data into two `STREAM` frames (`HTTP/3 GET`). Table 2 summarizes all resulting test cases.

6.2 Testing Frame-Packet Combinations

To evaluate whether QUIC implementations correctly validate frame types within different packet types, we construct a test catalog that injects frames that are disallowed for the respective packet type and vary their encapsulation. For `Initial` and `Handshake` packets, we perform the injection in the context of an TLS handshake message carried in a single `CRYPTO` frame. We combine the injected frame with the client's `ClientHello` in an `Initial` packet and with the `Finished` in a `Handshake` packet. For `1-RTT` packets, we perform the injection in combination with a `HTTP/3 GET` carried in a single `STREAM` frame. For each injected frame type, we again test three encapsulation variants: (i) the injected frame is appended to the same packet as the `CRYPTO`/`STREAM` frame (F1, F4, and F7), (ii) the injected frame is placed into a separate packet that is coalesced into the same datagram (F2, F5, and F8), and (iii) the injected frame is placed into a separate packet transmitted in a different datagram (F3, F6, and F9). Table 2 summarizes all test cases and highlights the packets with the injected frame in red.

6.3 Testing Frame Interactions

We implemented a set of tests to observe how QUIC server implementations process and respond to *contradicting* combinations of frames that are responsible for controlling streams.

1. Test Catalog: Fragmented Streams

Packet	Payload			
Initial	ClientHello	P1	P2	P3
Handshake	Finished	P4	P5	P6
1-RTT	HTTP/3 GET	P7	P8	P9

2. Test Catalog: Frame-Packet Combinations

Packet	Payload			
Initial	ClientHello	F1	F2	F3
Handshake	Finished	F4	F5	F6
1-RTT	HTTP/3 GET	F7	F8	F9

3. Test Catalog: Frame Interactions

Packet	Payload			
1-RTT	SS + ST	S1	-[a]	S2
1-RTT	ST + SS	S3	-[a]	S4
1-RTT	RST + ST	S5	-[a]	S6
1-RTT	ST + RST	S7	-[a]	S8
1-RTT	ST	S9	-[a]	N/A

RST RESET_STREAM **SS** STOP_SENDING **ST** STREAM

1x UDP Datagram with 1x QUIC Packet
1x UDP Datagram with 2x QUIC Packets
2x UDP Datagrams with 1x QUIC Packet each

a: Omitted as correct rejection directly follows from P8 and F8.

Table 2: Overview of our test catalogs. For all packet types, we construct test cases that vary how the payload is encapsulated across packets and datagrams. In the first catalog, we fragment a benign payload into two frames. In the second catalog, we do not fragment the payload but instead inject an additional frame that is disallowed for the respective packet type; red indicate packets containing the injected frame. In the third catalog, we combine two different frames.

These include `RESET_STREAM`, `STOP_SENDING`, and the exchange of application data itself (`STREAM`). To assess the impact of encapsulation choices, each test case is evaluated across different packet–datagram composition variants (cf. Table 2). First, we test combinations of `STOP_SENDING` and the transmission of application data within a `STREAM` frame. This combination is contradicting, as application data is sent on the same stream for which the `STOP_SENDING` requests that the server cease the transmission on this stream. We evaluate this scenario both when the two frames are placed into the same `1-RTT` packet (S1) and when they are split into separate `1-RTT` packets that are transmitted in distinct datagrams (S2). We then repeat the same tests with the order of the two frames reversed to assess whether the relative order of the frames changes the processing (S3 and S4). Second, we repeat the same four tests by replacing the `STOP_SENDING` with a `RESET_STREAM` (S5-S8). In this case, the contradiction arises from the fact that the `RESET_STREAM` explicitly closes the stream on which the application data is transmitted. Finally, we include a single-frame test that focuses on the stream IDs. In this test, we attempt to send application data on a stream with a skipped stream ID (S9).

7 Evaluation of Software Libraries

To systematically evaluate different QUIC implementations, we developed the *QUIC-Docker-Library*. This library provides Docker images for QUIC servers and clients, covering a wide range of versions and implementations. It enables reproducible experiments and large-scale comparative studies by simplifying deployment, reducing manual setup effort, and lowering the barrier for future research on QUIC behavior and interoperability. The library contains Docker images for 15 QUIC implementations. We focused on open-source implementations, as this allowed us to investigate the root causes of the discovered issues. This results in a broad and diverse set of implementations, including aioquic (v1.3.0), Kwik (v0.8.11), LSQUIC (v4.2.0), MsQuic (v2.5.5), mvfst (v2025.02.10.00), Neqo (v0.12.0), ngtcp2 (v1.12.0), picoquic (c2155d4), quic-go (v0.51.0), quicly (cd31aac), Quinn (v0.10.4), s2n-quic (v1.64.0), XQUIC (v1.9.0), and the two independently developed quiche libraries, one from Google (056196) and one from Cloudflare (v0.20.1). We refer to Google's implementation of quiche as quiche^G and to Cloudflare's as quiche^C. We used the server utilities provided by the respective projects to evaluate their behavior. All implementations were executed in a controlled lab environment to ensure consistent testing conditions. Since our tests are derived from mandatory RFC requirements, false positives or false negatives can only arise from an incorrect interpretation of these requirements and can be addressed by refining the corresponding test. We manually confirmed all findings and did not observe any false positives in the evaluation. We reported our observations to the respective library developers, and the details are summarized in Section 10. Table 3, Table 4, and Table 5 summarize the results of our test catalogs. The following sections discuss the findings in detail.

7.1 Fragmented Streams (Table 3)

Overall, QUIC packet coalescing with fragmented streams as defined in the specification is broadly supported by the evaluated implementations. Table 3 summarizes the results of our tests. Across the full test catalog, we observe only one deviation from the expected behavior, namely in test case P2. In this case, multiple implementations fail to correctly process a `ClientHello` when its `CRYPTO` frames are distributed across two `Initial` packets that are coalesced into a single datagram (as shown in Figure 3c). Four libraries (Kwik, picoquic, quic-go, and quicly) show the same characteristic behavior: upon receiving a datagram containing two `Initial` packets, they acknowledge the first packet by sending an `ACK` frame but silently drop the second packet without further processing. mvfst and quiche^G show different behavior. Upon receiving such a datagram, neither server sends any response. XQUIC deviates more severely from the specification. When receiving a datagram containing two `Initial`

	Test	Case	aioquic	Kwik	LSQUIC	MsQuic	Neqo	ngtcp2	picoquic	mvfst	quic-go	quiche^{C}	quiche^{G}	quicly	Quinn	s2n-quic	XQUIC
P1	Sending of a TLS ClientHello in Initial packets	1x Datagram with 1x Packet	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P2		1x Datagram with 2x Packets	✓	✗[a]	✓	✓	✓	✓	✗[a]	✗[b]	✗[a]	✓	✗[b]	✗[a]	✓	✓	✗[c]
P3		2x Datagrams with 1x Packet each	✓	✓	✓[d]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P4	Sending of a TLS Finished in Handshake packets	1x Datagram with 1x Packet	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P5		1x Datagram with 2x Packets	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P6		2x Datagrams with 1x Packet each	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P7	Sending of a HTTP/3 GET in 1-RTT packets	1x Datagram with 1x Packet	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P8		1x Datagram with 2x Packets	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P9		2x Datagrams with 1x Packet each	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

✓ Correct processing of encapsulation ✗ Incorrect processing of encapsulation

a: Acknowledges the first packet and drops the second packet.
b: No response.
c: Closes the connection with a `CONNECTION_CLOSE` frame.
d: Performs a token exchange first.

Table 3: Overview of the introduced *fragmented streams* tests. All tests except P8 examine that the constructed message is accepted and processed correctly by the server. In contrast, P8 sends an intentionally non-compliant message to assess whether the server discards it. Our results show that only one test case (P2) is not processed correctly by multiple servers.

packets, XQUIC immediately terminates the connection by sending a `CONNECTION_CLOSE` frame indicating a protocol error. Interestingly, all seven libraries correctly process test case P3, where the two `Initial` packets are transmitted in separate datagrams. This suggests that the issue is not caused by a general inability to handle multiple `Initial` packets, but rather by limitations in processing multiple `Initial` packets within a single datagram.

7.2 Frame-Packet Combinations (Table 4)

Almost all implementations validate the frame types contained in received packets. Only XQUIC, Kwik and Neqo show deviations from the expected behavior. Table 4 summarizes the results of our tests.

7.2.1 XQUIC

Multiple tests indicate that Alibaba's QUIC implementation (XQUIC) processes frames in packet types in which they are not permitted by the specification (F1, F3, F4, F5, and F6). An exception is test case F2, in which the injected frame is not processed. This behavior is explained by XQUIC's handling of coalesced `Initial` packets observed in P2 (cf. Section 7.1): when receiving a datagram containing multiple `Initial` packets, XQUIC processes only the first packet and ignores subsequent ones. As the injected frame in F2 is placed into a second `Initial` packet within the same datagram, the missing frame-type validation is not detected in this case.

We analyzed the impact of this behavior under the *Internet Threat Model* also considered by QUIC itself [10, Section 21.1]. This model assumes that *"the attacker has nearly complete control of the communications channel over which the end-systems communicate"* [18, Section 3], enabling direct packet manipulation. Under this threat model, the missing validation of frame placement enables multiple attacks that target different aspects of the server's behavior.

Injection of Application Data (CVE-2026-6328) XQUIC buffers `STREAM` frames received in `Initial` and `Handshake` packets and processes them after the handshake completes. However, these packets are intended to carry TLS handshake messages and not application data in `STREAM` frames. By accepting and deferring the processing of `STREAM` frames in these packet types, XQUIC allows an attacker to inject *arbitrary* application data at the beginning of a QUIC connection.

Using HTTP/3 as the application protocol and the official Alibaba web server with XQUIC integration,[5] we escalated the message injection vulnerability to showcase a *confusion attack* similar to the one described for HTTP/1.0 and TLS by Merget et al. in [13] (*Opossum attack*). In our proof of concept, an attacker injects a `STREAM` frame during the QUIC handshake between a Chrome browser and the web server. Once the handshake completes and the QUIC implementation processes the buffered `STREAM` frame, the response is forwarded as the response to the victim's first request. Concretely, a victim that requests `cat.html` is instead served the contents of `dog.html`. Figure 4 illustrates the attack.

[5] https://github.com/alibaba/tengine

	Test	Case	aioquic	Kwik	LSQUIC	MsQuic	Neqo	ngtcp2	picoquic	mvfst	quic-go	quiche^C	quiche^G	quicly	Quinn	s2n-quic	XQUIC
F1	Sending of a TLS ClientHello with a disallowed frame in Initial packets		✓	↯	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	↯
F2			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
F3			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	↯
F4	Sending of a TLS Finished with a disallowed frame in Handshake packets		✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
F5			✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
F6			✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
F7	Sending of a HTTP/3 GET with a disallowed frame in 1-RTT packets		✓	✓	✓	✓	↯	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
F8			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
F9			✓	✓	✓	✓	↯	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

 1x Datagram with 1x Packet  1x Datagram with 2x Packets  2x Datagrams with 1x Packet each

✓ Disallowed frames not processed ✗ Disallowed frames processed ↯ Disallowed frames processed, resulting in a vulnerability

Table 4: Overview of the introduced *frame-packet combination* tests. Each test is executed for all frame types disallowed for the respective packet type, including frame types that must not be sent by the client; consequently, each test corresponds to multiple executions. The permitted combinations are summarized in Table 7. The red packets indicate those containing the injected frame. Our results indicate that the Kwik and XQUIC servers do not implement any frame type validation during packet processing.

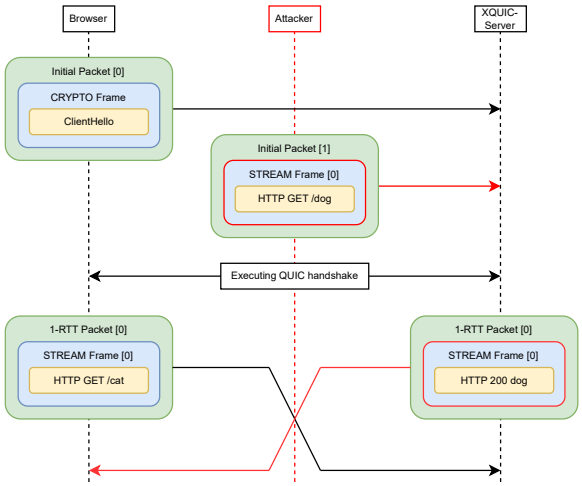
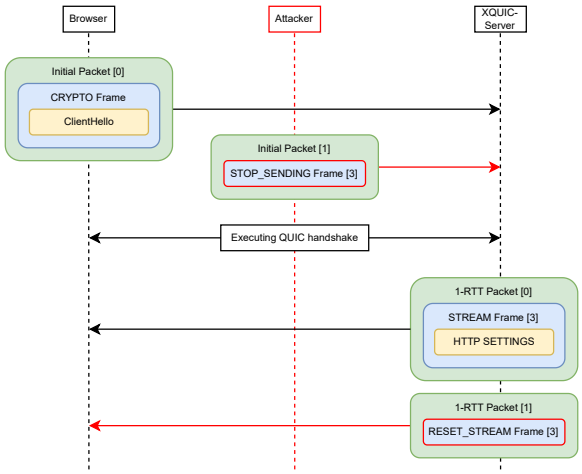


Figure 4: A sketch of the confusion attack on XQUIC (CVE-2026-6328). For simplicity, `ACK` frames are omitted. After the browser's TLS `ClientHello`, the attacker injects a `GET` request for `dog`. After the handshake, XQUIC replies to the attacker's request, which the browser misinterprets as the response to its own `GET` request for `cat`.

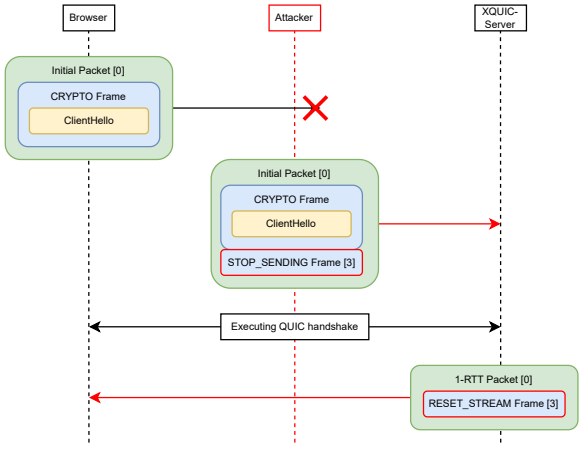
Deletion of Application Data Instead of injecting a `STREAM` frame as in the previous attack, an adversary can also inject stream control frames such as `STOP_SENDING` or `RESET_STREAM` to drop contents of post-handshake streams. XQUIC directly processes these frames although they are intended to manage stream states only after the handshake has been completed. This enables an attacker to manipulate the server's stream states *before* any legitimate application data is exchanged. The attack relies on predicting or guessing stream IDs that will be used by the server, which are commonly allocated sequentially. While in our case the attacker has only limited ability to affect the point of truncation, security vulnerabilities in web servers through suffix truncation have been demonstrated before in the context of HTTP over TLS by Smyth and Pironti [21]. Figure 5 illustrates the two injection variants using the `STOP_SENDING` frame as example. In case (a), the attacker injects a `STOP_SENDING` frame in an additional `Initial` packet. While the handshake proceeds normally, XQUIC still transmits the subsequent application data from the server, but then terminates the stream by sending a `RESET_STREAM`. In case (b), the attacker injects the `STOP_SENDING` frame into the `Initial` packet carrying the TLS `ClientHello`. In this variant, XQUIC suppresses the application data entirely and directly closes the stream, leading to a full truncation effect.

7.2.2 Kwik

Similar to XQUIC, the Kwik server also deviates from the specification when handling forbidden frame-packet combinations (F1, F4, F5, and F6). However, F2 and F3 do not expose the missing validation. In F2, the injected frame is placed into a second `Initial` packet within the same datagram, which is not processed by Kwik (cf. Section 7.1). In F3, Kwik derives the `Handshake` keys immediately after successfully processing the TLS `ClientHello` and discards the `Initial` keys, causing subsequent `Initial` packets (including the injected frame) to be rejected before processing occurs.



(a) The attacker injects a `STOP_SENDING` frame in an additional `Initial` packet. XQUIC still sends the application data, but immediately terminates the stream afterwards.



(b) The attacker injects a `STOP_SENDING` frame into the `Initial` packet carrying the TLS `ClientHello`. XQUIC suppresses the application data and directly closes the stream.

Figure 5: A sketch of the truncation attack on XQUIC. For simplicity, `ACK` frames are omitted. After the browser's TLS `ClientHello`, the attacker injects a `STOP_SENDING` frame for the stream with ID 3. Depending on where the frame is injected, XQUIC reacts differently.

Our tests indicate that at least two frames (`MAX_STREAM_DATA` and `RETIRE_CONNECTION_ID`) are still processed even when they appear in forbidden packet types such as `Initial` or `Handshake` packets. A subsequent manual source code analysis reveals that this behavior is not limited to these two frame types. We found that frame placement validation is entirely missing and frames are processed regardless of their packet type. Our automatic tests could not confirm this behavior for all frame types, as many frames do not trigger *observable* server responses that would allow us to reliably infer their processing from the outside. Notably, at the time of writing, the developers of Kwik implemented frame placement validation independently of our findings.

7.2.3 Neqo

The `NEW_TOKEN` frame is permitted to appear in `1-RTT` packets according to the specification. However, it is restricted to being sent by the server. When the Neqo server receives a `NEW_TOKEN` frame from a client after the handshake has completed, it fails to handle this protocol violation gracefully and instead crashes immediately (F7 and F9).

7.3 Frame Interactions (Table 5)

Crashes by Exploiting STOP_SENDING Frames We discovered that Neqo and quiche^C servers fail to correctly handle a `STREAM` frame containing application data in combination with a `STOP_SENDING` frame for the same stream. Interestingly, when both frames are transmitted within the same packet and datagram, the relative ordering of the `STOP_SENDING` and `STREAM` frames does not affect the outcome (S1 and S3). In this case, both frame orderings consistently trigger a crash. However, when the two frames are split across separate packets and transmitted in distinct datagrams, the processing behavior becomes order-dependent (S2 and S4). A crash occurs only when the `STOP_SENDING` frame is received before the `STREAM` frame. In contrast, when the `STREAM` frame is received first, the application data is processed and responded to as expected, while the subsequently received `STOP_SENDING` frame is silently ignored (S4). The underlying cause for the crashes is, upon processing a `STOP_SENDING` frame, the servers marks the corresponding stream ID in its internal state as no longer available for sending data. When the HTTP/3 layer subsequently attempts to process application data in the following `STREAM` frame, it violates this internal constraint, leading to an immediate crash.

A similar issue was observed in the LSQUIC server. When receiving a `1-RTT` packet that contains both a `STREAM` frame and a `STOP_SENDING` frame sent in this order, the server crashes instead of handling the protocol violation gracefully (S3). In contrast to Neqo and quiche^C, the relative ordering of the two frames is relevant in this case. When the `STOP_SENDING` is placed first within the packet, the server immediately closes the corresponding stream and subsequently ignores the following `STREAM` frame (S1).

Closing of Unused Streams When receiving `STREAM` frames, s2n-quic closes all streams with IDs lower than the received stream in the same category (S9). For example, receiving a `STREAM` frame with ID 8 triggers `STOP_SENDING` frames for streams with IDs 0 and 4. While this behavior may be intended as a flow-control optimization, it can lead to

	Test	Case	aioquic	Kwik	LSQUIC	MsQuic	Neqo	ngtcp2	picoquic	mvfst	quic-go	quiche^C	quiche^G	quicly	Quinn	s2n-quic	XQUIC
S1	Sending of `STOP_SENDING` then `STREAM`	1	RST	ST	RST	RST	↯	RST	RST	RST	RST	↯	RST	RST	RST	RST	RST
S2		2	RST	ST	RST	RST	↯	RST	RST + CC	RST + SS	RST	↯	RST	RST	RST	RST	RST
S3	Sending of `STREAM` then `STOP_SENDING`	1	RST	RST	↯	RST	↯	RST	CC	RST	RST	↯	ST + RST	ST	RST	RST	RST
S4		2	ST + RST	ST	ST	ST	ST	ST	ST	ST	ST + RST	ST	ST	ST	ST + RST	ST	ST + RST
S5	Sending of `RESET_STREAM` then `STREAM`	1	ST	ST	CC	RST	-	-	RST	RST	RST	-	SC	-	ST	ST	RST
S6		2	ST	ST	RST	RST	-	-	RST	RST	RST	-	SC	-	ST	ST	RST
S7	Sending of `STREAM` then `RESET_STREAM`	1	ST	-	RST	RST	ST	ST	CC	RST	RST	-	ST	ST	ST	ST	RST
S8		2	ST	ST	ST	ST	ST	ST	ST	ST	ST	ST	ST	ST	ST	ST	ST + RST
S9	`STREAM` with skipped ID	1	ST	ST	ST	N/A[a]	ST	ST	ST	ST	ST	ST	ST	ST	ST	ST + SS	ST

1 1x Datagram with 1x Packet 2 2x Datagrams with 1x Packet each

\- No response **CC** CONNECTION_CLOSE **RST** RESET_STREAM **SS** STOP_SENDING **ST** STREAM ↯ Results in a crash (uncaught exception)

a: Test not applicable, as the default server allows only a single stream and requires the first available ID, preventing skipped stream IDs.

Table 5: Overview of the introduced *frame interactions* tests. The observed server answers indicate a heterogeneous ecosystem, with different reactions to identical test cases. In addition, some tests triggered crashes in LSQUIC, Neqo, and quiche^C.

closure of streams that the application might still use, causing unexpected data loss.

Different Processing Strategies Across the complete set of tests, we observed that QUIC servers react differently to identical test cases indicating a heterogeneous ecosystem. However, a central observation is that the order of frames within a packet can significantly influence processing behavior. For example, comparing test cases S1 and S3 shows that sending a `STOP_SENDING` frame before a `STREAM` frame can trigger a different reaction than sending the same frames in the opposite order. A similar effect can be observed when comparing S5 and S7.

7.4 Additional Findings

7.4.1 Handshake With Malformed CRYPTO Frames

Beyond our presented main evaluation, we used QUIC-Attacker's capabilities to transmit deliberately malformed messages to explore additional classes of implementation flaws. In particular, we implemented tests that invalidate specific fields of `CRYPTO` frames (e.g., the *offset*). By manipulating the offsets, we evaluate whether server implementations correctly detect and reject malformed `CRYPTO` frames with TLS handshake data. The evaluation across all tested servers reveals a consistent result: all libraries except Alibaba's XQUIC detect the invalid offsets and do not complete the handshake. We therefore investigated XQUIC in more detail using QUIC-Attacker. In the following, we present the results of this analysis.

Missing Offset Validation Our analysis revealed that XQUIC does not enforce strict validation of `CRYPTO` frame offsets during the final stages of the handshake. When transmitting the complete TLS `Finished` message in a single `CRYPTO` frame, the frame can have an *arbitrary* offset value without preventing completion of the handshake. In a valid flow, the `CRYPTO` offset for this final handshake message is expected to 0. Building on this observation, we also found that XQUIC completes the handshake even if the *verify_data* field of the `Finished` message is modified in combination with an invalid offset in the `CRYPTO` frame.

Handshake Without TLS Finished We also found that XQUIC can transition into a handshake complete state even when the client does not send a `Finished` at all. Instead, the handshake can be concluded by sending a `Certificate` followed by a `CertificateVerify`. While these messages are typically used for client authentication when requested by the server, this sequence is insufficient to complete the handshake, as it omits the mandatory `Finished`.

Root Cause A manual source code analysis revealed that these observations are caused by a state machine bug. The server marks the connection as established without verifying that the TLS handshake has actually completed. Any `CRYPTO` frame read event at the `Handshake` packet level can trigger this transition. For example, a `CRYPTO` frame with a non-zero offset and arbitrary payload bytes is already sufficient to trigger the bug. This indicates that the connection state changes without proper validation of the TLS handshake state.

7.4.2 Handshake With Initial Packets

During initial testing of QUIC-Attacker's functionalities, we observed an unexpected specification deviation in aioquic. Both the client and server complete the handshake successfully even when their peer transmits *all* TLS handshake messages in `Initial` packets. According to the specification, TLS handshake messages beyond the initial `ClientHello` and `ServerHello` are expected to be carried in `Handshake` packets.

8 Discussion

Combinatorial Complexity of Frame Compositions Our evaluation covers *three* cases in which *two* frames are combined within a *single* test scenario. We also restrict the fragmentation to at most two frames per data stream, ensuring that the resulting executions remain within the scope of our defined test cases. However, QUIC permits more complex compositions: streams may be fragmented into more than two frames and RFC 9000 defines up to 20 different frame types that can appear in a wide range of combinations. But as the number of frames included in a composition increases, the number of possible tests grows rapidly, making complete coverage within a single study infeasible. Our study represents an initial step toward systematically exploring frame interactions in QUIC and shows that even limited frame compositions can expose striking differences between implementations and reveal exploitable vulnerabilities.

Lessons From Combining Complex Protocols Our results in Section 7.4.1 show that the integration of TLS 1.3 into QUIC is not *straightforward*. Findings such as XQUIC's premature transition to an established connection state indicate that even when the underlying TLS stack operates correctly, *repurposing* parts of this stack outside of the originally intended flow, may lead to critical issues. This indicates that analyzing TLS 1.3 in QUIC is fundamentally different and cannot be replaced by analyzing TLS 1.3 in isolation.

9 Conclusion

In this work, we introduced *QUIC-Attacker*, a framework for the systematic analysis of QUIC server and client implementations, providing fine-grained control over fragmentation and coalescence, as well as protocol field manipulation to construct both RFC-compliant and intentionally malformed QUIC connections. We further presented the *QUIC-Docker-Library*, a deployment artifact that streamlines the setup of heterogeneous QUIC stacks.

Based on *QUIC-Attacker* and a dedicated test catalog, we found that packet coalescing can affect the processing of semantically equivalent messages and introduce interoperability problems (RQ1). Moreover, we identified that the specification's frame-packet constraints are not consistently validated in practice which leads to exploitable vulnerabilities (RQ2). Finally, we observed that implementations differ in their interpretation of interactions between stream related frames, leading to inconsistent stream handling (RQ3). Taken together, this study revealed that QUIC's encapsulation flexibility directly affects how messages are processed and how the internal connection state changes. This observation highlights that testing individual frames or packets in isolation is often insufficient for QUIC, and that testing should additionally cover composition effects.

Looking ahead, *QUIC-Attacker* provides a reusable foundation for extending QUIC evaluation beyond today's specialized tools. Future work can create new test catalogs, systematically evaluate client-side behavior, and apply additional methodologies such as combinatorial testing.

10 Ethical Considerations

Stakeholders For each step of our methodology, we considered potentially affected parties. Most importantly, the release of our research tool affects developers of QUIC libraries and entities using these libraries to secure their communication. While the tool is designed to aid in network security research, lowering the barrier for systematic testing, it could also be used by malicious parties to probe for vulnerabilities. Our research may further affect standardization bodies if our results indicate that aspects of the specification leave ambiguity contributing to the identified issues. Finally, we also considered the broader impact on the general public and particularly on users of affected implementations. These may be affected as with any published vulnerability report, there is always a risk that services fail to implement fixes, rendering their user data vulnerable.

Ethical Principles *Beneficience and Public Interest:* The aim of our research is to improve the state of the QUIC ecosystem and to help achieve secure implementations for users relying on QUIC to protect their data. An important step into this direction is to identify implementation pitfalls that may threaten the security of implementations and potentially already manifest as bugs in QUIC endpoints deployed in production.

Respect for Persons: The only persons directly affected by our study are the developers of the considered QUIC libraries. In our reports (see below), we used the indicated channels, enabling developers to decide which details of an issue shall be published alongside the corresponding fixes.

Justice: Our evaluation is not intended to target any developers in particular. We evaluated QUIC libraries that represent the major actively maintained QUIC stacks from both industry and the open-source community. These libraries have also

been the subject of several prior studies [1, 2, 5, 20].

Respect for Law: Our study was conducted in a controlled lab environment using open-source QUIC implementations. As such, our analysis did not threaten live services and did not enable access to any user data.

Mitigations Our research identified both potential interoperability issues and exploitable vulnerabilities. We responsibly disclosed all of our findings to the respective QUIC library developers. Regarding the specification violations listed in Table 3, we notified all affected projects. At the time of submission, all affected projects except mvfst acknowledged the reported deviation. Of these, only quicly and Kwik implemented a fix. For the security vulnerabilities listed in Table 4, we reported the issues to the maintainers of XQUIC and Neqo. Both projects acknowledged our findings and released patches to mitigate the vulnerabilities. The issue affecting Kwik was identified and fixed by the maintainers independently and in parallel to our work. For the security vulnerabilities listed in Table 5, the maintainers of LSQUIC and Neqo acknowledged our reports. While Neqo subsequently deployed a fix, no mitigation has been released by LSQUIC at the time of submission. We additionally informed the maintainers of quiche^C but did not receive a response prior to submission. Finally, for the additional observations discussed in Section 7.4, we contacted all affected projects. At the time of submission, only XQUIC acknowledged the reported deviations.

Proof of Concept for XQUIC To confirm the vulnerability in Alibaba's XQUIC, we built a proof of concept (PoC) that escalates the identified message injection vulnerability to achieve a confusion attack. To ensure that the issue is not limited to the XQUIC example server, we additionally tested Alibaba's HTTP/3 server (`alibaba.com`). The observed confusion was restricted to requests generated by our PoC simulating a victim client. No real users or user data were affected by our tests.

Decision to Conduct and Publish our Research We chose to conduct this research to identify and help mitigate flaws in QUIC implementations, including issues that could (potentially) already be exploited by adversaries. Reporting details about discovered vulnerabilities always bears the risk that malicious actors try to exploit endpoints that have not been updated to incorporate provided fixes. We mitigated this risk by starting the disclosure process for exploitable vulnerabilities early, ensuring fixes are provided long before publication of this study. We further consider identifying and publishing results highlighting vulnerabilities crucial to help identify systemic issues in the implementation landscape. As such, findings can also benefit future revisions of the protocol specifications as they provide a baseline for discussions on possible improvements.

Similarly, QUIC-Attacker as a research tool may become subject to misuse by malicious actors. However, we believe that sustainable research tools enabling systematic studies are essential to improve the overall security of a protocol's ecosystem. In this context, we point to the history of TLS research, and in particular to research based on TLS-Attacker, which serves as the foundation of QUIC-Attacker. Prior studies by various authors based on TLS-Attacker [3] have helped identify widespread implementation issues and even specification flaws. We publish QUIC-Attacker in the hope of providing similar benefits for the growing QUIC ecosystem.

11 Open Science

As part of our artifacts, we release the full implementation of *QUIC-Attacker* as well as the framework containing the complete set of test cases used in our evaluation. We additionally release the *QUIC-Docker-Library*, which provides Docker images for all evaluated QUIC libraries across multiple versions. All released projects are made available under an open-source license. In addition to these artifacts, we also include the proof of concept exploit for Alibaba's XQUIC implementation.

The released artifacts provide a comprehensive foundation for reproducing our results, extending our analyses, and supporting further research on QUIC implementations. All artifacts can be fount at `https://doi.org/10.5281/zenodo.20280317`.

12 Acknowledgments

We thank the anonymous reviewers for their valuable feedback. We further thank Patrick Weixler and Marten Schmidt, as well as the members of the System Security group at Paderborn University, for their contributions to the *QUIC-Attacker* project. This research was partially supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and by the German Federal Ministry of Research, Technology and Space (BMFTR) through the project KoTeBi.

References

[1] Kian Kai Ang, Guy Farrelly, Cheryl Pope, and Damith C. Ranasinghe. An automated blackbox noncompliance checker for quic server implementations. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '25, page 1459–1475, New York, NY, USA, 2025. Association for Computing Machinery.

[2] Kian Kai Ang and Damith C. Ranasinghe. Quic-fuzz: An effective greybox fuzzer for the quic protocol. In

Computer Security – ESORICS 2025: 30th European Symposium on Research in Computer Security, Toulouse, France, September 22–24, 2025, Proceedings, Part III, page 1–22, Berlin, Heidelberg, 2025. Springer-Verlag.

[3] Fabian Bäumer, Marcus Brinkmann, Nurullah Erinola, Sven Hebrok, Nico Heitmann, Felix Lange, Marcel Maehren, Robert Merget, Niklas Niere, Maximilian Radoy, Conrad Schmidt, Jörg Schwenk, and Juraj Somorovsky. Tls-attacker: A dynamic framework for analyzing tls implementations. *Proceedings of Cybersecurity Artifacts Competition and Impact Award (ACSAC'24)*, 2024.

[4] M. Bishop. HTTP/3. RFC 9114, IETF, June 2022.

[5] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 762–774, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Eva Gagliardi and Olivier Levillain. Analysis of quic session establishment and its implementations. In Maryline Laurent and Thanassis Giannetsos, editors, *Information Security Theory and Practice*, pages 169–184, Cham, 2020. Springer International Publishing.

[7] Konrad Yuri Gbur and Florian Tschorsch. QUICforge: Client-side request forgery in QUIC. In *NDSS 2023*, San Diego, CA, USA, February 2023. The Internet Society.

[8] C. Huitema, S. Dickinson, and A. Mankin. DNS over Dedicated QUIC Connections. RFC 9250, IETF, May 2022.

[9] J. Iyengar and I. Swett. QUIC Loss Detection and Congestion Control. RFC 9002, IETF, May 2021.

[10] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, IETF, May 2021.

[11] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1185–1196, Denver, CO, USA, October 12–16, 2015. ACM Press.

[12] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. Same standards, different decisions: A study of quic and http/3 implementation diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ '20, page 14–20, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Robert Merget, Nurullah Erinola, Marcel Maehren, Lukas Knittel, Sven Hebrok, Marcus Brinkmann, Juraj Somorovsky, and Jörg Schwenk. Opossum attack: Application layer desynchronization using opportunistic TLS. Cryptology ePrint Archive, Paper 2025/1260, 2025.

[14] Marcin Nawrocki, Pouyan Fotouhi Tehrani, Raphael Hiesgen, Jonas Mücke, Thomas C. Schmidt, and Matthias Wählisch. On the interplay between tls certificates and quic performance. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '22, page 204–213, New York, NY, USA, 2022. Association for Computing Machinery.

[15] T. Pauly, E. Kinnear, and D. Schinazi. An Unreliable Datagram Extension to QUIC. RFC 9221, IETF, March 2022.

[16] Maxime Piraux, Quentin De Coninck, and Olivier Bonaventure. Observing the evolution of quic implementations. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ'18, page 8–14, New York, NY, USA, 2018. Association for Computing Machinery.

[17] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, IETF, August 2018.

[18] E. Rescorla and B. Korver. Guidelines for Writing RFC Text on Security Considerations. RFC 3552, IETF, July 2003.

[19] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. A first look at quic in the wild. In Robert Beverly, Georgios Smaragdakis, and Anja Feldmann, editors, *Passive and Active Measurement*, pages 255–268, Cham, 2018. Springer International Publishing.

[20] Marten Seemann and Jana Iyengar. Automating quic interoperability testing. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ '20, page 8–13, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Ben Smyth and Alfredo Pironti. Truncating TLS connections to violate beliefs in web applications. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, Washington, D.C., August 2013. USENIX Association.

[22] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1492–1504, Vienna, Austria, October 24–28, 2016. ACM Press.

[23] Stefan Tatschner, Sebastian N. Peters, David Emeis, John Morris, and Thomas Newe. A quic(k) security overview: A literature research on implemented security recommendations. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Kashyap Thimmaraju and Björn Scheuermann. Count me if you can: Enumerating quic servers behind load balancers. *Electronic Communications of the EASST*, 80, Sep. 2021.

[25] M. Thomson. Version-Independent Properties of QUIC. RFC 8999, IETF, May 2021.

[26] M. Thomson and S. Turner. Using TLS to Secure QUIC. RFC 9001, IETF, May 2021.

[27] Johannes Zirngibl, Philippe Buschmann, Patrick Sattler, Benedikt Jaeger, Juliane Aulbach, and Georg Carle. It's over 9000: analyzing early quic deployments with the standardization on the horizon. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 261–275, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Johannes Zirngibl, Florian Gebauer, Patrick Sattler, Markus Sosnowski, and Georg Carle. Quic hunter: Finding quic deployments and identifying server libraries across the internet. In Philipp Richter, Vaibhav Bajpai, and Esteban Carisimo, editors, *Passive and Active Measurement*, pages 273–290, Cham, 2024. Springer Nature Switzerland.

A QUIC Packets and QUIC Frames

Packet Type	Header Type	Sender	Coalescing
Initial	Long	C S	●
Handshake	Long	C S	●
0-RTT	Long	C	●
1-RTT	Short	C S	◐
Retry	Long	S	◐
Version Negotiation	-	S	◐

● Packet type may appear multiple times within single datagram
◐ Packet type must be last in datagram and only included once

Table 6: Overview of the QUIC packet types as defined by RFC 9000 and their coalescing constraints within a single UDP datagram. QUIC-Attacker supports sending all packet types with arbitrary frames, except for Retry and Version Negotiation packets, which do not follow the regular packet format and never carry frames.

Frame Type	Sender	I	H	0-R	1-R
PADDING	C S	✓	✓	✓	✓
PING	C S	✓	✓	✓	✓
ACK	C S	✓	✓	-	✓
RESET_STREAM	C S	-	-	✓	✓
STOP_SENDING	C S	-	-	✓	✓
CRYPTO	C S	✓	✓	-	✓
NEW_TOKEN	S	-	-	-	✓
STREAM	C S	-	-	✓	✓
MAX_DATA	C S	-	-	✓	✓
MAX_STREAM_DATA	C S	-	-	✓	✓
MAX_STREAMS	C S	-	-	✓	✓
DATA_BLOCKED	C S	-	-	✓	✓
STREAM_DATA_BLOCKED	C S	-	-	✓	✓
STREAMS_BLOCKED	C S	-	-	✓	✓
NEW_CONNECTION_ID	C S	-	-	✓	✓
RETIRE_CONNECTION_ID	C S	-	-	✓	✓
PATH_CHALLENGE	C S	-	-	✓	✓
PATH_RESPONSE	C S	-	-	-	✓
CONNECTION_CLOSE	C S	✓	✓	✓	✓
HANDSHAKE_DONE	S	-	-	-	✓

I Initial **H** Handshake **0-R** 0-RTT **1-R** 1-RTT

Table 7: Overview of the QUIC frame types specified in RFC 9000, their intended senders, and the packet types in which they are allowed to appear. QUIC-Attacker supports all frame types. Our tests explore both valid and invalid frame-packet combinations.

B TLS-Attacker

We build QUIC-Attacker on top of TLS-Attacker [3, 22], a well-established framework for the systematic analysis of (D)TLS client and server implementations. To enable rapid prototyping of compliance and security tests, TLS-Attacker's architecture aims to enable users to focus on the specific parts of the protocol relevant to their evaluation. To this end, message flows are generated with respect to the protocol specification except for modifications defined by the user. TLS-Attacker is further specifically designed to handle non-compliant message flows. As an additional advantage, we can benefit from its flexibility in generating and modifying TLS messages. This was useful in some manual analysis of QUIC implementations, see Section 7.4.1.

C Additional Details on QUIC-Attacker

To provide a better understanding of our newly introduced layers, Figure 6 illustrates the functional behavior of our QUIC stack.

Sending QUIC Messages Sending messages follows a top-down processing model across the new *QUIC frame* and *QUIC packet* layers. Across all layers, outgoing data is handled in the same three-step pattern: the layer *prepares* the next message, *updates* the corresponding protocol state, and

then *serializes* the message into a byte stream and passes it to the next layer. The process starts at the TLS message layer. This layer prepares the user-defined TLS messages and forwards the resulting TLS byte stream to the QUIC frame layer. At the QUIC frame layer, the incoming TLS stream is encapsulated into `CRYPTO` frames. Users may either define the exact `CRYPTO` frames to use or rely on protocol-compliant default frame generation. If the user defines additional frames, the layer prepares and serializes them as well and combines them with the `CRYPTO` frames. The layer then outputs the serialized QUIC frame stream. At the QUIC packet layer, the incoming QUIC frame stream is encapsulated into packets and protected using the newly implemented QUIC encryption. Again, users can either specify packets explicitly or rely on protocol-compliant defaults. If the user defines additional QUIC packets, the layer prepares and protects them as well and combines them with the other packets. The layer then outputs the serialized QUIC packet stream for the UDP layer.

Receiving QUIC Messages Receiving messages follows the reverse direction of the stack. Each layer first *parses* the incoming byte stream and then *updates* the corresponding protocol state before passing the result to the next layer. Incoming UDP datagrams are parsed into QUIC packets at the QUIC packet layer, where packet-level state is updated and protected payloads are decrypted. The packet contents are then forwarded to the QUIC frame layer, which parses the frames and updates the frame-level state. If `CRYPTO` frames are present, the contained TLS handshake bytes are extracted and passed to the TLS message layer for processing.

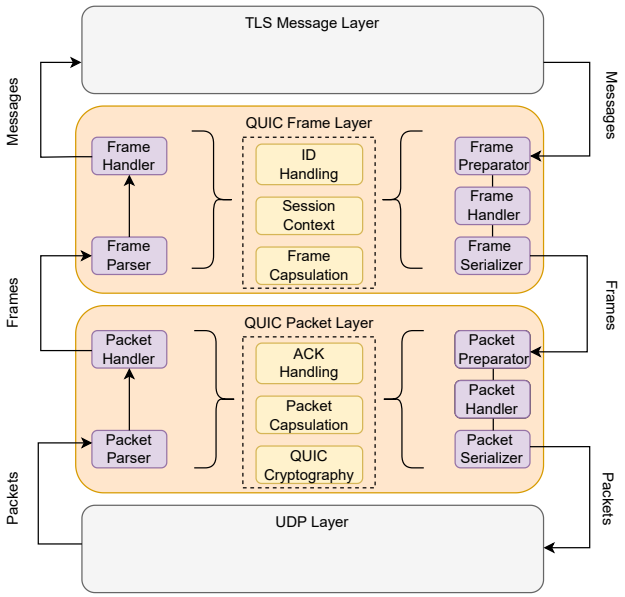


Figure 6: Overview of our layer extensions to TLS-Attacker. The newly introduced layers are depicted in orange. Frame and packet processing elements involved in sending and receiving QUIC message are depicted in purple. We implement all frames and packet types specified by RFC 9000 (cf. Table 6 and Table 7). Core functionalities of QUIC required to provide the QUIC session state, including the cryptographic operations, are depicted in yellow.