Q3Fuzz: Multi-Layered Stateful Fuzzing for the QUIC-HTTP/3 Protocol Stack

Isa Jafarov
City University of New York
New York, USA
ijafarov@gradcenter.cuny.edu

Choongin Lee
Korea University
Seoul, Republic of Korea
choonginlee@korea.ac.kr

Heejo Lee
Korea University
Seoul, Republic of Korea
heejo@korea.ac.kr

Sven Dietrich
City University of New York
New York, USA
spock@ieee.org

***Abstract*—Existing work on protocol state machine inference focuses on modeling a single network-layer protocol at a time, while considering its payload as raw data. Such a model lacks information about how two different protocols interact with each other. In this work, we introduce a novel black-box framework, named Q3Fuzz, to model two network protocols together and conduct multi-layered stateful fuzzing. Q3Fuzz constructs the model based on observed network traces by treating the underlying layer and its encapsulated payload as co-occurring events. It then uses the model to direct both mutation-based and generation-based fuzzing.**

Q3Fuzz is architected for the QUIC-HTTP/3 protocol stack. HTTP/3 is built on top of QUIC, a transport-layer protocol, which was designed to improve performance. While QUIC operates based on a loosely defined state model for individual streams, its integration with HTTP/3 introduces complex behavioral logic that single-protocol models fail to capture.

We evaluated Q3Fuzz against all 14 open-source QUIC-HTTP/3 servers that have 500+ GitHub stars and discovered 17 vulnerabilities (15 previously unknown) across 7 different implementations. The findings include critical issues, such as server-crash bugs and low-rate Denial-of-Service vulnerabilities, leading to two CVE ID assignments to date. In terms of code coverage, Q3Fuzz outperforms the closest existing work, a grey-box QUIC protocol fuzzer, by achieving up to 76.1% higher branch coverage. We make Q3Fuzz publicly available to foster future research [1].

***Index Terms*—HTTP/3, QUIC, Stateful fuzzing, Network security, Denial-of-service**

I. Introduction

The internet is undergoing a foundational shift with the widespread adoption of HTTP/3, the latest web protocol. Standardized in June 2022, HTTP/3 leverages the QUIC protocol [1] to deliver significant performance gains, such as reduced connection latency [2] and the elimination of head-of-line (HOL) blocking [3]. In particular, HTTP/3 delegates features previously handled by HTTP/2, such as stream multiplexing [4], directly to the QUIC transport layer. This delegation resolves the TCP-level HOL blocking that troubled the stack of the previous version [5] by integrating application-level concepts directly into the transport mechanism itself. Consequently, this has led to rapid adoption; HTTP/3 is now supported by over a third of all websites [6] and handles a growing share of global user traffic [7].

[1]https://github.com/IsaJafarov/Q3Fuzz

The wide use of HTTP/3 makes it a high-value target for attackers, especially given its architectural complexity. This is not merely theoretical, as various real-world vulnerabilities have emerged in the intertwined stack. For example, Sudhan et al. [8] demonstrated that essential QUIC features such as connection migration can be exploited to launch Denial-of-Service (DoS) attacks [9] in HTTP/3 servers. Similarly, Chatzoglou et al. [10] demonstrated that attack strategies from legacy protocols can affect HTTP/3 servers, leading to resource exhaustion via malformed frames or manipulated stream parameters in the stack. These cases highlight that subtle implementation flaws in either the QUIC or HTTP/3 layer can lead to severe vulnerabilities.

The fundamental difficulty in finding such vulnerabilities stems from requiring testers to contend with two sophisticated protocols simultaneously. In other words, this intertwining of QUIC's foundational stream state machine (SM) [1] with HTTP/3's application logic [11]–[13] results in a multi-layered behavioral model, necessitating a *state-aware fuzzing* that goes far beyond simple stateless testing. To this end, a couple of fundamental challenges need to be solved: handling the multi-layer state interactions (C1), which requires a model that captures transitions triggered by both transport and application-layer data, and achieving high test coverage in a black-box environment (C2), which is crucial for testing diverse implementations effectively.

Existing fuzzing methodologies for this stack are restricted to the QUIC transport layer [14]–[16], failing to capture the intertwined state transitions between QUIC and the HTTP/3 application logic. For a thorough analysis, we also discuss the challenges faced by the recent work [16] and how Q3Fuzz overcomes them. These challenges are ensuring input integrity (C3) to pass encryption checks, managing protocol non-determinism (C4) to maintain state consistency, and mitigating execution overhead (C5) of target servers for efficient testing. To address these multi-layered complexities and transport-level hurdles simultaneously, we present Q3Fuzz, a state-aware black-box fuzzing framework designed for the intertwined QUIC-HTTP/3 protocol stack. Q3Fuzz operates in two distinct phases: (1) State Machine Inference and (2) State-Aware Fuzzing. This approach provides several key advantages:

- **Holistic Modeling.** The novelty of our inference process

lies in its multi-layer awareness. Q3Fuzz does not exclusively cover the QUIC transport layer; it simultaneously dissects and models the HTTP/3 application-layer frames (e.g., `HEADERS`, `SETTINGS`) and their explicit mapping to underlying QUIC streams. This methodology constructs a holistic model of the entire stateful interaction, thereby directly addressing the core dual-layer challenge.

- **Heterogeneous Analysis.** Q3Fuzz transcends the limitations associated with generic models by effectively addressing heterogeneous implementations [17], [18]. This capability derives from adapting prior research [4], which enables automated reverse-engineering of stateful behaviors specific to implementations. Consequently, Q3Fuzz is proficient in identifying unique bugs that are frequently overlooked by conventional fuzzers [19].
- **Broad Applicability.** Unlike grey-box approaches restricted by language dependencies [16] or source code availability [20], Q3Fuzz is inherently language-agnostic and capable of testing proprietary targets. This black-box nature enables the framework to scale seamlessly across heterogeneous implementations. The inferred state machine compensates for the lack of internal feedback, guiding the fuzzer to explore the state space with high coverage systematically.

We evaluated Q3Fuzz against 14 most popular open-source QUIC-HTTP/3 protocol stack implementations. The framework proved highly effective, discovering 17 vulnerabilities, 15 of which were previously unknown zero-days. The identified flaws were severe, primarily consisting of server crashes and low-rate DoS vulnerabilities that could be triggered remotely. Our responsible disclosure of these findings has so far led to the assignment of two CVE IDs, which cover five vulnerabilities, validating the practical impact of our state-guided fuzzing approach.

This paper makes the following key contributions:

- We design and implement Q3Fuzz, a black-box framework that automates the reverse-engineering of implementation-specific state machines. While existing protocol fuzzers model and fuzz a single network-layer protocol [21], Q3Fuzz is the first approach to jointly model and fuzz dual network-layer protocols, instantiated for the QUIC-HTTP/3 stack.
- We provide the first concrete outline of real-world HTTP/3 state machines. This analysis empirically validates the security risks posed by protocol ambiguity and implementation divergence, a problem previously discussed in theory [4], [17], [19].
- We confirm the effectiveness of our model-guided approach by discovering 17 critical vulnerabilities (15 zero-days). The findings include bugs both in the transport-layer and application-layer protocol implementations.

II. Background

To provide the necessary context for the approach proposed in this paper, this section details the distinct architectures and responsibilities of the HTTP/3 and QUIC protocols.

A. HTTP/3

At the application layer, HTTP/3 [12] is responsible for request and response semantics and header compression (QPACK) [22], which are mapped onto the underlying QUIC transport layer. The QUIC layer, in turn, manages stateful streams, connection IDs, and transport parameters [1]. Each HTTP request and response is mapped to an independent QUIC stream, enabling true parallelism where multiple requests can be in flight simultaneously without blocking each other. HTTP/3 maintains the stateless semantics of HTTP at the application layer. However, its frame-based encoding introduces stateful dependencies, where frame sequencing and transitions must be carefully managed.

B. QUIC

QUIC is a binary frame-based transport layer protocol [1] that is responsible for encryption, ordered and guaranteed data delivery, connection management, and multiplexing. QUIC connections are identified by the endpoints independently using one or more connection IDs, which allows for seamless connection migration. For ease, we can describe a typical QUIC connection as a sequence of three phases.

Connection establishment. A QUIC connection starts with a handshake, where the client and server establish a shared secret and negotiate the application protocol. For existing connections, the endpoints can optionally send application-layer data during the handshake phase (0-RTT), which brings certain security weaknesses. Once the handshake completes, the endpoints switch to 1-RTT packets.

Data exchange. After the connection establishment, the endpoints can exchange application-layer data carried as `STREAM` frame payload. There are two types of streams depending on the direction of data transmission: unidirectional and bidirectional. QUIC RFC [1] provides state machines for sending and receiving streams, which define basic states and how certain events should affect the streams. However, those SMs are not strict, since the RFC allows implementations to define different SMs.

Connection shutdown. Once the data exchange is done, any endpoint can gracefully shut down the connection. Other ways to tear down the connection include error occurrence or a timeout in case of inactivity.

III. Motivation

This section outlines the threat model for QUIC and HTTP/3 protocols, and emphasizes the importance of multi-layered stateful fuzzing. The focus of this paper is attacks targeting server-side implementations, particularly those that exploit the complex logic within the state machine of the QUIC-HTTP/3 protocol stack.

A. Threat Model

We consider an external adversary capable of transmitting arbitrary sequences of QUIC and HTTP/3 packets to a target

server. The primary objective is to compromise service availability by triggering one of the following conditions:

- **Service termination:** Causing an immediate server crash through fatal logic errors or memory corruption vulnerabilities.
- **Resource exhaustion:** Depleting critical system resources (e.g., CPU, memory, disk) to render the service unresponsive or paralyzed.

B. Multi-Layer Stateful Testing

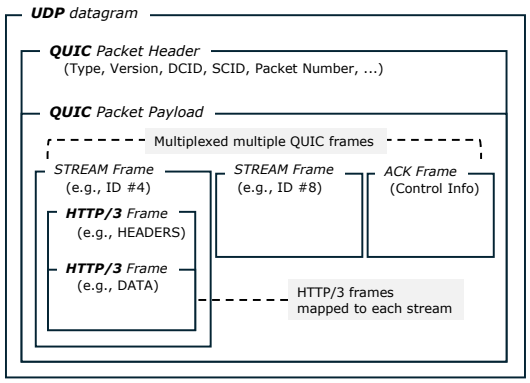


Fig. 1: Encapsulation and multiplexing of HTTP/3

The core motivation for our approach stems from the complex architecture of the QUIC-HTTP/3 protocol stack, visualized in Figure 1. The interaction between HTTP/3 and QUIC is a complex arrangement of encapsulation and multiplexing. A QUIC packet is composed of a header and a payload that is encapsulated as the payload of a UDP datagram. The key innovation lies within the QUIC packet's payload, which can multiplex different frames. For example, a single QUIC packet can bundle independent `STREAM` frames (e.g., for Stream 4 and Stream 8) alongside transport-level control frames (e.g., `ACK`). Finally, each QUIC `STREAM` frame acts as a container for application-layer data, carrying one or more HTTP/3 frames (e.g., `HEADERS`, `DATA`) or QPACK encoder/decoder instructions.

This encapsulation model creates a challenge for stateful fuzzers. For instance, AFLNet [23], [24] infers the protocol state model by analyzing server response codes (e.g., "200 OK"). However, relying on response codes creates an *observability gap*, as transport-layer signals (e.g., a QUIC `ACK` frame) and application-layer controls (e.g., an HTTP/3 `GOAWAY` frame) lack a uniform status indicator and manifest only as distinct binary frame types, causing legacy fuzzers to perceive them as indistinguishable network events [17]. Consequently, this blindness prevents the fuzzer from correlating an unexpected server crash with its specific multi-layered root cause, such as receiving a transport-layer QUIC `STREAM` frame that encapsulates an application-layer HTTP/3 `SETTINGS` frame at a specific state.

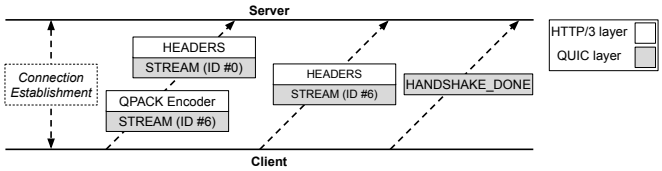


Fig. 2: Attack vector discovered by Q3Fuzz to crash a remote Proxygen server

1) Motivating example: To demonstrate the importance of multi-layer fuzzing, we present a motivating example. Q3Fuzz discovered a memory-safety vulnerability in the Proxygen server (Table V) by transmitting three QUIC packets one after another (Figure 2). According to the patch [25], an attacker can exploit the vulnerability by (1) requesting an application-layer resource and (2) then sending a malformed QUIC frame while the resource is being served. The malformed QUIC frame causes the transactions to abort before the server completes serving the application-layer resource. The vulnerability comes from the intersection of QUIC and HTTP/3 layers. Therefore, the attacker needs to specifically craft both application-layer (first step) and transport-layer (second step) data.

To systematically identify the interconnected flaws, we modify and improve the state inference methodology developed in PRETT2 [4], which effectively reverse-engineered HTTP/2, the prior version of the HTTP series. Our approach builds on this by incorporating in-depth, multi-layer grammar analysis. This allows the creation of a behavioral model that relates the grammatical structures of both QUIC transport streams and the HTTP/3 application frames they encapsulate, facilitating the detection of complex, state-dependent vulnerabilities.

IV. Q3Fuzz Framework

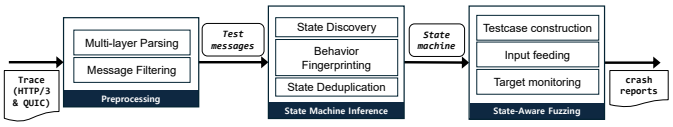


Fig. 3: Overview of Q3Fuzz framework

In this section, we present Q3Fuzz, a framework for systematically fuzzing the multi-layered QUIC-HTTP/3 protocol stack. Figure 3 illustrates the overall architecture, composed of three sequential phases: *preprocessing*, *state machine inference*, and *state-aware fuzzing*.

A. Phase 1: Preprocessing

We start with recording and decrypting sample QUIC-HTTP/3 traffic. In this phase, we extract and transform the observed client messages (also called *test* messages) into a structured format suitable for state inference.

1) Multi-layer parsing: This module dissects raw packets into their constituent protocol layers. Unlike conventional parsers that treat payloads as opaque blobs, Q3Fuzz performs dual-layer dissection. It first parses the QUIC frames (Layer 1) and then recursively decapsulates the contained HTTP/3 application data (Layer 2). This hierarchical parsing preserves the semantic context (e.g., stream ID, frame type), which is essential for identifying multi-layer dependencies (C1).

2) Message filtering: To ensure the efficiency of the inference process, this module filters out noise from the parsed traces. It removes frames that do not have a semantic meaning, such as `PING` and `PADDING`, which simply check server reachability and increase payload size, respectively. Moreover, to convince the server that there is no packet loss, we implemented a custom acknowledgment mechanism, where we send an `ACK` frame for every received server response. Therefore, we also removed the `ACK` frames from the traffic to make sure they do not interfere with our acknowledgment mechanism and confuse the server.

B. Phase 2: State Machine Inference

To construct a high-fidelity behavioral model, we adopt the inference algorithm established in prior literature [4], [26], enhancing it for the multi-layered QUIC-HTTP/3 stack. The process, detailed in Figure 4, iteratively builds an SM through three key steps. Due to the cryptographic complexities, the QUIC handshake stage is not included in our model. We start building the SM after establishing a QUIC connection (`CONNECTED` state).

1) State discovery: Starting from the `CONNECTED` state, Q3Fuzz iterates over each state and dispatches *test* messages to trigger potential transitions. The transmitted message puts the server in a new state, which we call a *candidate* state, because eventually it might or might not be added to the actual SM.

2) Behavior fingerprinting: After moving the server to a *candidate* state, Q3Fuzz generates a fingerprint for the *candidate* state. The fingerprint is generated by sending all *test* messages one-by-one and recording the server's response for each of them. Such a fingerprint, which simply is a list of request-response pairs, identifies the *candidate* state. We need the fingerprint of a state to know whether or not the state has previously been observed. To overcome non-determinism (C4), after transmitting a message, the client waits for a certain duration (0.1 seconds), letting the server finish processing the message.

3) State deduplication: To maintain a minimized model, we compare the generated fingerprint against those of existing states. If the fingerprint matches an existing state, those two states are merged, and a transition to the existing state is added. If the fingerprint is unique, the *candidate* is registered as a *new* state and added to the SM for further analysis. This cycle continues until no new states are discovered.

C. Phase 3: State-Aware Fuzzing

Leveraging the inferred state model, this phase systematically tests for vulnerabilities. We focus only on vulnerabilities

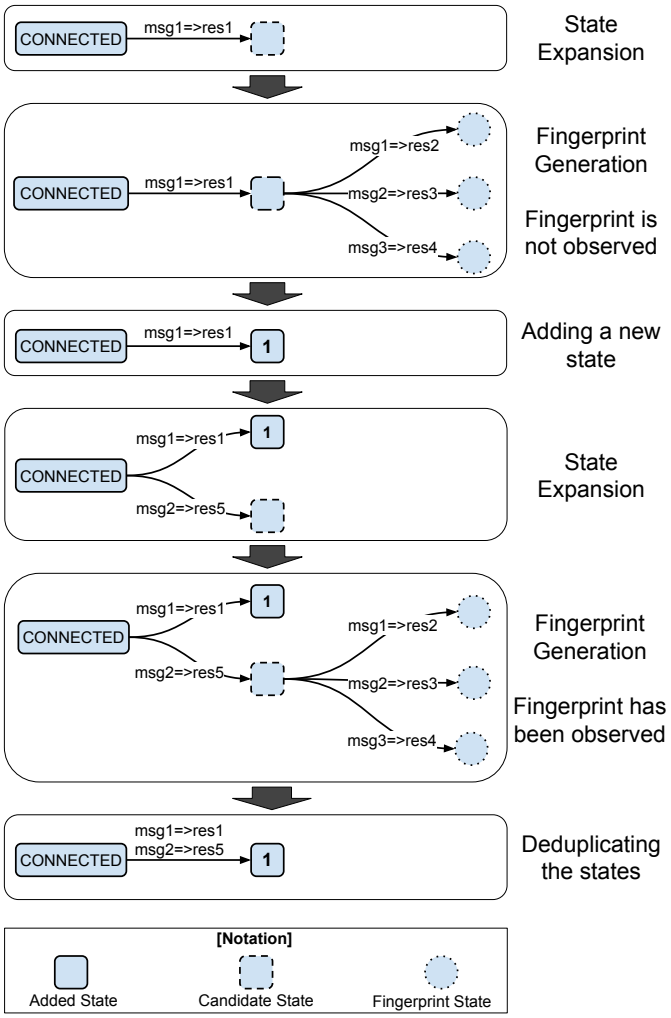


Fig. 4: Overview of the State Machine Inference phase

in the **post-handshake phase** (1-RTT packets), targeting transitions starting from the `CONNECTED` state.

1) Testcase Construction: Q3Fuzz employs two complementary fuzzing strategies to craft test inputs, as shown in Figure 5.

Note that we use the following terms to refer to certain transitions and messages. A *moving message* is a message sent to drive the protocol toward the target state. A *triggering message* is the message that induces the target state transition itself. A *following message* is the message that induces the subsequent state transition immediately after the target state (Figure 5).

Mutation-based fuzzing. After selecting a target state, Q3Fuzz finds and extracts the *triggering message* from the traffic. The message is first dissected into QUIC and HTTP/3 frames, allowing Q3Fuzz to easily modify the message contents before crafting a new packet. Q3Fuzz mutates one frame and field at a time, while keeping the rest of the message intact. This way, the test input mimics the real traffic, so that the manipulated message has a higher chance of being

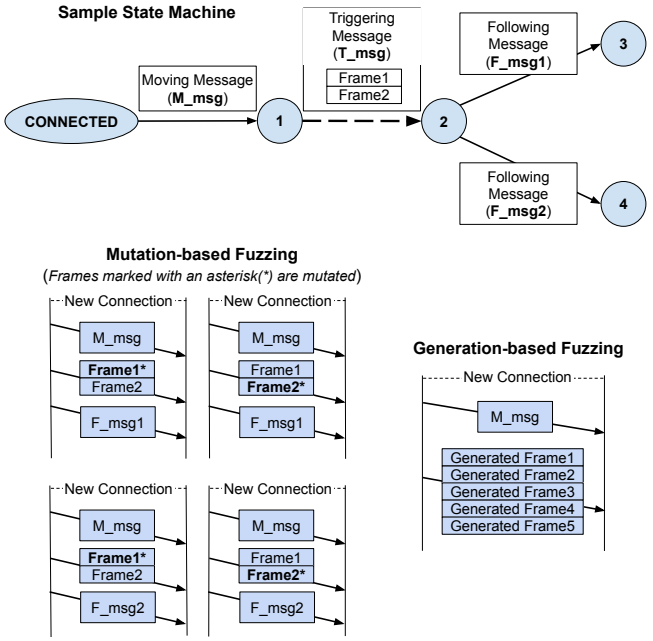


Fig. 5: Overview of mutation-based and generation-based fuzzing strategies

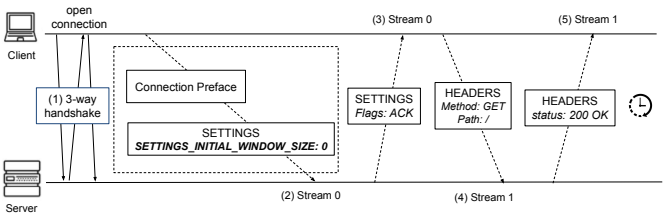


Fig. 6: CVE-2023-43622 vulnerability exploitation [4]

accepted/processed by the server.

After transmitting the mutated *triggering message*, Q3Fuzz traverses the subsequent states by sending the *following messages*, since sending a malicious message alone is not sufficient to exploit certain vulnerabilities. One example is CVE-2023-43622, a Denial-of-Service vulnerability in Apache's HTTP/2 protocol implementation [4]. To exploit the vulnerability, an attacker first sends an HTTP/2 `SETTINGS` frame with `SETTINGS_INITIAL_WINDOW_SIZE` parameter set to `0`. After that, the attacker must send an HTTP/2 `HEADERS` frame (Figure 6), and only then does the server indefinitely lock a thread for the connection.

Generation-based fuzzing. One major drawback of mutation-based fuzzing is that it results in low code coverage (C2). The number of frames that the fuzzer uses to mutate and fuzz a certain transition is limited by the number of frames the *triggering message* has. Mutation-based grey-box fuzzers generally overcome this problem by using the feedback from the target as guidance to maximize their code coverage. However, unlike grey-box fuzzers, we assume our client has

TABLE I: Protocol frames and their attributes that are used to craft test inputs. Bold frames were used both in mutation-based and generation-based fuzzing, while the others were only used in generation-based fuzzing.

#	Frame Type	Fuzzed Attribute	Data Type
	QUIC Layer		
1	PADDING	-	-
2	PING	-	-
3	**ACK**	Largest Acknowledged	Integer
		ACK Delay	Integer
		ACK Range Count	Integer
		First ACK Range	Integer
		ACK Range	Sequence of ranges
4	RESET_STREAM	Stream ID	Integer
		Application Protocol Error Code	Integer
		Final Size	Integer
5	STOP_SENDING	Stream ID	Integer
		Application Protocol Error Code	Integer
6	CRYPTO	Offset	Integer
		Crypto Data	Bytes
7	NEW_TOKEN	Token	Bytes
8	**STREAM**	Stream ID	Integer
		FIN bit	Boolean
		Offset	Integer
		Data	HTTP/3 layer entry
9	MAX_DATA	Maximum Data	Integer
10	MAX_STREAM_DATA	Stream ID	Integer
		Maximum Stream Data	Integer
11	**MAX_STREAMS**	Maximum Streams	Integer
12	DATA_BLOCKED	Maximum Data	Integer
13	STREAM_DATA_BLOCKED	Stream ID	Integer
		Maximum Stream Data	Integer
14	STREAMS_BLOCKED	Is Bidirectional	Boolean
		Maximum Streams	Integer
15	**NEW_CONNECTION_ID**	Sequence Number	Integer
		Retire Prior To	Integer
		Connection ID	Bytes
		Stateless Reset Token	Bytes
16	RETIRE_CONNECTION_ID	Sequence Number	Integer
17	PATH_CHALLENGE	Data	Bytes
18	PATH_RESPONSE	Data	Bytes
19	CONNECTION_CLOSE	Is Transport Layer	Boolean
		Error Code	Integer
		Frame Type	Integer
		Reason Phrase	Bytes
20	HANDSHAKE_DONE	-	-
21	DATAGRAM (*extension*)	Quarter Stream ID	Integer
		Payload	Bytes
	HTTP/3 Layer		
1	**DATA**	Data	Bytes
2	**HEADERS**	Encoded Field Section	Bytes
3	CANCEL_PUSH	Push ID	Integer
4	**SETTINGS**	Max Table Capacity	Integer
		Max Field Section Size	Integer
		Blocked Streams	Integer
		H3 Datagram	Integer
		Webtransport	Integer
5	PUSH_PROMISE	Push ID	Integer
		Field Section	Bytes
6	GOAWAY	Stream ID	Integer
7	ORIGIN (*extension*)	Entries	Sequence of strings
8	MAX_PUSH_ID	Push ID	Integer
9	**PRIORITY_UPDATE (*extension*)**	Element ID	Integer
		Field Value	String

no access to internal information about the target server, such as its source code or runtime environment.

We overcome this challenge by maximizing the range of the test inputs. We additionally employ a generation-based fuzzing strategy, where we construct a sequence of QUIC and HTTP/3 frames from scratch, which enables us to cover all the QUIC and HTTP/3 frames defined by the protocol specifications. Each crafted packet contains 5 randomly selected QUIC frames. In case `STREAM` frame is randomly selected, its application-layer data (HTTP/3 frame or QPACK data) is also chosen randomly. Because `STREAM` frames carry the application-layer data and have relatively higher logical complexity, we give higher priority to `STREAM` frames in random selection.

The generation-based fuzzing significantly improves the

coverage. Protocol specifications define 21 [27] (20 core [1] and 1 extension [28]) QUIC frames, as well as 9 [29] (7 core [12] and 2 extension [11], [13]) HTTP/3 frames. However, we observed only 4 QUIC and 4 HTTP/3 frames (excluding QPACK instructions) in all the generated traffic combined. As a result, mutation-based fuzzing alone covers less than 20% of QUIC and 45% of HTTP/3 frames (Table I).

2) Input Feeding: Q3Fuzz transmits a test input to the target server in three different ways.

1) **Parallel Flooding:** To trigger low-rate DoS [30], we send requests in parallel batches (default: 20 parallel connections) at fixed intervals (1 second) for a sustained duration (30 seconds). These values are similar to the attack parameters proposed in Chatzoglou et al. [10]. We refrain from sending a higher number of parallel requests with a lower interval, as it would lead to flooding attacks, which is out of this project's scope.
2) **Single-Shot Execution:** To efficiently detect immediate crash bugs such as assertion failures or null pointer dereferences, we transmit each test input exactly once. This strategy allows transmitting a higher number of different test inputs in a short period of time. After transmitting each message (*moving*, *triggering*, and *following*), Q3Fuzz waits for a certain duration (0.1 seconds) for the server to process each of them.
3) **Temporal Execution:** Sending a sequence of messages with different intervals can cause the server to process the same input differently [15]. To explore *temporal* [15] state dependencies and race conditions, we transmit the sequence of *moving*, *triggering*, and *following* messages continuously as a burst, without waiting for intermediate server responses. Certain vulnerabilities, such as the aforementioned crash vulnerability in Proxygen (§III-B1), can be discovered only through this strategy. In that vulnerability exploitation, by sending the malformed QUIC frame **immediately**, we do not give the server enough time to finish serving the resource.

3) Target Monitoring: In this work, we target only DoS vulnerabilities. For this purpose, we employ a *liveness check* mechanism. After each fuzzing iteration, Q3Fuzz attempts to establish a benign QUIC connection. A failure to establish a connection indicates a vulnerability. To mitigate false positives from network issues, we confirm vulnerabilities after multiple failed liveness checks.

4) Resetting the environment: One of the challenges (C5) faced by an existing work [16] is overcoming the long boot-up of the servers. QUIC-Fuzz requires resetting the target environment, since it uses a static, hard-coded connection ID for every connection. Q3Fuzz, on the other hand, establishes a new connection with a random connection ID before transmitting every test input, eliminating the need to reset the environment.

However, we still restarted certain servers (Neqo and Aioquic), where we observed a memory leak vulnerability (§V-C1). During fuzzing, their memory usage continuously went up, leading to a different target environment for test inputs.

D. Implementation

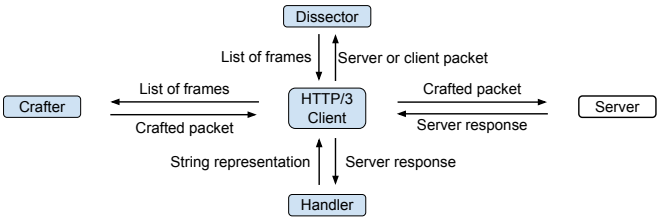


Fig. 7: Overview of Q3Fuzz fuzzing client

Figure 7 shows the implementation of Q3Fuzz. In particular, we implemented an HTTP/3 client by extending the Aioquic [31] library. While Aioquic provides a robust foundation for core frames, it lacks native support for specific extension frames, such as `Origin` [13] and `PRIORITY_UPDATE` [11]. We therefore implemented the extension for these frames.

The framework is composed of four main components:

- *HTTP/3 Client* communicates with the server. It starts the connection by sending an `INITIAL` packet with a QUIC `CRYPTO` frame. After receiving a response from the server, the client sends a `HANDSHAKE` packet, which completes the connection establishment stage. From this point, it transfers only `1-RTT` packets.
- *Dissector* extracts frames from a QUIC packet. Representing a packet as a list of dissected frames makes it easier to manage/modify frames before crafting a new packet.
- *Crafter* builds a QUIC packet based on the given QUIC and HTTP/3 frames.
- *Handler* processes the server response. It manages the server's cryptographic keys and returns the server response as human-readable characters, which we use as the state machine's transition label.

A primary challenge (C3) for mutation-based fuzzers is maintaining packet integrity. Fuzzers that directly modify encrypted seeds generate invalid inputs, which are rejected by the server. Q3Fuzz's architecture solves this by design: the *Dissector* and *Crafter* operate only on unencrypted frame logic using pre-provided SSL keys. The *HTTP/3 Client* then re-encrypts these frames into valid QUIC packets before transmission, ensuring all test inputs are well-formed and processed by the server.

V. Evaluation

A. Experimental Setup

To systematically evaluate Q3Fuzz, we established a comprehensive testbed comprising diverse QUIC-HTTP/3 server implementations and a consistent reference client. This section outlines the criteria for selecting the target servers and client, as well as the fuzzing infrastructure specifications.

1) Target Servers: The IETF QUIC Working Group references 18 open-source QUIC server implementations [32]. We selected 14 most popular open-source server implementations (Table II), excluding the servers with fewer than 500 GitHub

TABLE II: Target QUIC and HTTP/3 servers

#	QUIC Layer	HTTP/3 Layer	GitHub Stars	Main Language	Version ("Commit")
1	Nginx	*(standalone)*	28.1k	C	v1.28.0
2	Quic-go	*(standalone)*	11.0k	Go	v0.50.1
3	Quiche*	*(standalone)*	10.6k	Rust	v0.23.5
4	Quinn	H3	4.6k	Rust	v0.0.9†
5	MsQuic	Kestrel	4.5k	C & C#	v2.4.8
6	Neqo	*(standalone)*	2.0k	Rust	v0.13.1
7	Aioquic	*(standalone)*	1.9k	Python	v1.2.0
8	XQUIC	*(standalone)*	1.8k	C	v1.8.3
9	LSQUIC	OpenLiteSpeed	1.7k	C	v1.8.3.1†
10	Mvfst	Proxygen	1.6k	C++	v2025.04.14†
11	Ngtcp2	nghttp3	1.3k	C	v1.12.0
12	QUICHE**	*(standalone)*	790	C++	"7b2b126"
13	Quicly	H2O	645	C	"f1918a5"†
14	Picoquic	*(standalone)*	644	C	"b19dcf1"

† Indicates HTTP/3 implementation version; others are QUIC.
*developed by Cloudflare, **developed by Google

stars [33] to ensure relevance. The majority of the applications provide not only a QUIC, but also an HTTP/3 layer implementation, which we used to run as a server. However, some applications provide only the QUIC layer implementation. In such cases, we either used a web server that runs the given implementation as its built-in QUIC layer (e.g., OpenLiteSpeed, H2O, Kestrel) or a separate HTTP/3 layer application running on top of the given QUIC implementation (e.g., Quinn+H3, Mvfst+Proxygen, Ngtcp2+Nghttp3). We excluded S2n-quic [34], as it provides only the transport layer implementation, and we are not aware of any HTTP/3 application integrating its API.

2) Selected Client: We utilized Mozilla Firefox (v132.0.2) [35] as the reference client to generate QUIC-HTTP/3 traffic by communicating with the target servers. This selection was driven by two factors: first, unlike basic command-line tools (e.g., `curl`) that produce simplistic patterns, Firefox generates traffic containing complex sequences of QUIC and HTTP/3 frames. This complexity is necessary to exercise the multi-layer state logic of the targets. Second, using a single, consistent client ensures that any observed variations in the inferred SMs are attributed solely to server-side implementation divergence.

TABLE III: Virtual hardware specifications

Resource	Target Server (VM)	Fuzzing Client (VM)
vCPU	1	4
Memory	2 GB	2 GB
Disk	15 GB	20 GB

3) Environment: We created 56 virtual instances. Each server implementation was running on 2 instances in parallel; hence, 28 instances were allocated to run the server implementations. 14 instances were running mutation-based, and the other 14 were running the generation-based fuzzers.

All instances run Ubuntu 22.04 LTS server. For the virtual hardware specifications (Table III), we were inspired by an existing work [36]. The virtual client hardware mimics the server hardware with a slight difference. To eliminate any potential bottleneck on the client side during fuzzing, we allocated 4 virtual vCPUs (virtual CPUs) for the clients. The host server runs Ubuntu 20.04.6 LTS on Intel(R) Xeon(R) E-2278G CPU with 128 GB of memory.

Causing a Denial-of-Service by requesting a huge file is out of this project's scope. Therefore, all servers host a small file, the default `index.html` file of Nginx [2].

TABLE IV: Quantitative Comparison of Inferred SMs.

Server (Implementation)	States ($\|S\|$)	Transitions ($\|T\|$)
Picoquic	7	17
Quic-go	8	26
Neqo	8	26
MsQuic (Kestrel)	11	33
Quicly (H2O)	11	33
Quiche*	11	41
Mvfst + Proxygen	12	55
XQUIC	13	61
QUICHE**	13	61
Quinn + H3	13	50
Aioquic	15	61
Nginx	17	68
Ngtcp2 + Nghttp3	32	175
LSQUIC (OpenLiteSpeed)	54	307

*developed by Cloudflare, **developed by Google

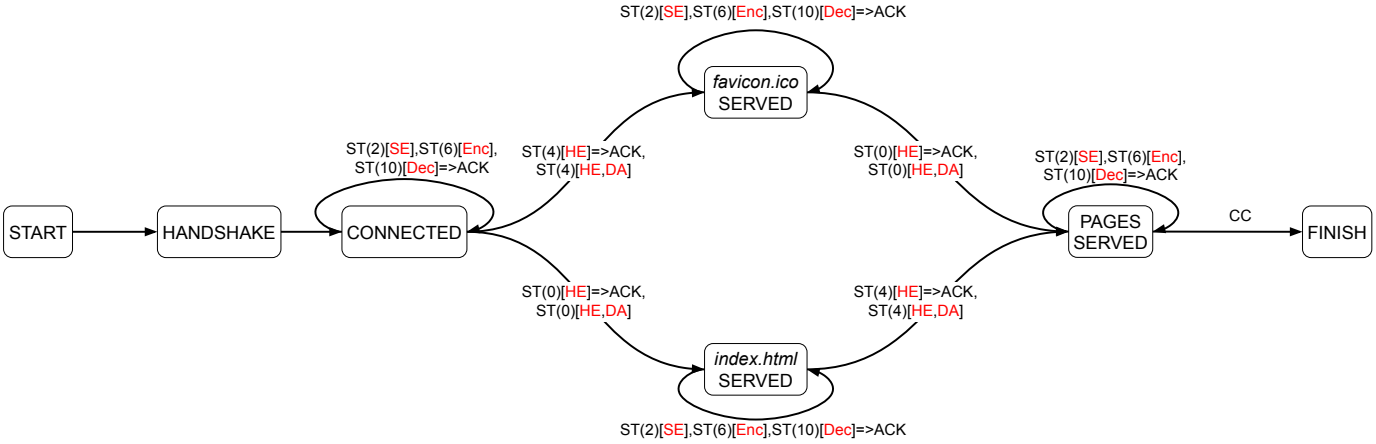
B. State Machine Inference

A core security challenge for the QUIC-HTTP/3 stack stems from its complex design, creating an environment where each server has an implementation-specific attack surface. Q3Fuzz systematically reverse-engineers such logic in an automated manner. In this section, we compare two sample state machines, highlighting the shared and divergent server behaviors.

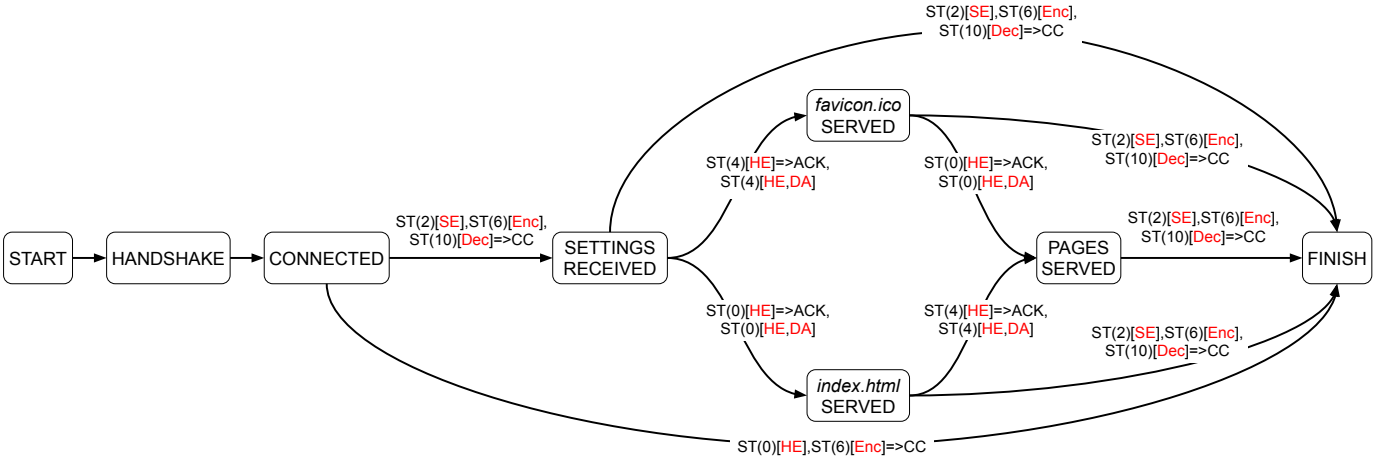
Figure 8 presents a comparative analysis between Quic-go and Mvfst+Proxygen server models. We simplified the SMs to highlight a few shared and distinct behavioral patterns regarding the servers' stream and parameter management. Note that the `HEADERS` frames transmitted over stream ID 0 and 4 request `index.html` and `favicon.ico` files, respectively.

1) Shared Behaviour: As shown in both Figure 8a and 8b, the diamond-shaped transitions merge at the `PAGES SERVED` state. The two different paths leading to that state differ from each other based on the sequence of the messages. In one path, the server first processes `ST(4)[HE]` message and responds with the `favicon.ico` file, and then receives `ST(0)[HE]` message and returns the `index.html` file. However, in the other path, the messages are processed in reverse order, meaning that the server first receives `ST(0)[HE]` and then `ST(4)[HE]`. Both paths merge at the `PAGES SERVED` state, since at that point, the server has completed processing both `ST(4)[HE]` and `ST(0)[HE]` messages, regardless of their reception order.

[2]https://github.com/nginx/nginx/blob/master/docs/html/index.html



(a) Quic-go State Machine (simplified)



(b) Mvfst+Proxygen State Machine (simplified)

[Notation]	[QUIC Frames]	[HTTP/3 Frames] (RED)
`Request =>Response`	`ST: STREAM`	`HE: HEADERS`
`St(N)[X]`: Stream with ID `N` carrying payload `X`	`ACK: ACK`	`DA: DATA`
	`CC: CONNECTION_CLOSE`	`SE: SETTINGS`
		`Enc/Dec`: QPACK Encoder/Decoder

Fig. 8: Comparative analysis of inferred state machines

2) Divergence 1: The SMs (Figure 8) highlight how the servers behave differently in terms of processing connection parameters and application-layer data.

Quic-go allows the client to request an application-layer resource by sending an HTTP/3 `HEADERS` frame right after connection establishment without sending the `SETTINGS` frame (Figure 8a). The server successfully responds with HTTP/3 `DATA` frame (see `CONNECTED->favicon.ico SERVED` and `CONNECTED->index.html SERVED` transitions).

In contrast, Mvfst+Proxygen requires the client to send an HTTP/3 `SETTINGS` frame after connection establishment (Figure 8b). Otherwise, the server does not give the requested resource to the client. When the server receives `HEADERS` frame on stream ID 0 right after connection establishment, it closes the connection (see `CONNECTED->FINISH` transition). When the server receives the `SETTINGS` frame, it moves to a new state (`SETTINGS RECEIVED`), and only in that state the server successfully processes the client's HTTP/3 messages (see `SETTINGS RECEIVED->favicon.ico SERVED` and `SETTINGS RECEIVED->index.html SERVED` transitions).

This divergence stems from the fact that the HTTP/3 specification [12] requires the `SETTINGS` frame to be the first frame on the **control** stream, but does not specifically mention the **request** streams.

RFC 9114 – HTTP/3 (Section 3.2)

After the QUIC connection is established, a `SETTINGS` frame MUST be sent by each endpoint as the initial frame of their respective HTTP control stream.

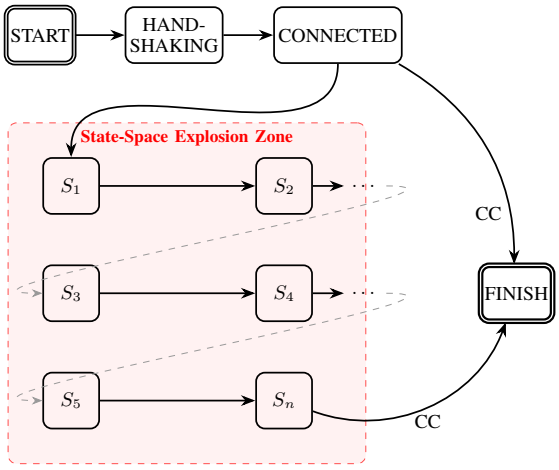


Fig. 9: Abstracted SM depicting an example State Space Explosion due to nondeterministic server behavior. ($S_1 \rightarrow \cdots \rightarrow S_n$).

3) Divergence 2: Another observed divergent behavior is how the servers handle receiving the `SETTINGS` frame multiple times.

In case of Quic-go, when the server receives `SETTINGS` frame multiple times, it simply acknowledges the recipient of the message (responds with `ACK` frame) and stays at the same state (see the self-loop transitions in Figure 8a).

However, the Mvfst+Proxygen server accepts the `SETTINGS` frame only when it is in the `CONNECTED` state. If the server receives the frame again, it closes the connection (see `SETTINGS RECEIVED->FINISH`, `favicon.ico SERVED->FINISH`, `index.html SERVED->FINISH`, `PAGES SERVED->FINISH` transitions).

This divergence shows that Quic-go does not follow the protocol specifications properly, since the HTTP/3 specification [12] states that subsequent `SETTINGS` frames should close the connection.

RFC 9114 – HTTP/3 (Section 7.2.4)

If an endpoint receives a second `SETTINGS` frame on the control stream, the endpoint MUST respond with a connection error of type `H3_FRAME_UNEXPECTED`.

4) Scalability: As demonstrated in Table IV, our framework inferred state machines for 14 distinct server implementations, encompassing a wide range of protocol logic complexity ranging from minimal models such as Picoquic ($|S| = 7$ and $|T| = 17$) to highly complex ones. This extensive applicability proves that Q3Fuzz's multi-layer inference approach generalizes effectively across diverse real-world deployments, independent of specific architectural designs.

State space explosion and nondeterminism.

During SM generation, we observed multiple servers, especially Ngtcp2 and LSQUIC, exhibiting seemingly nondeterministic behavior, which means a server can give different responses to identical client messages. Based on our observation, one of the main mechanisms that leads to nondeterminism is message acknowledgement. Implementations are not required to send an acknowledgment for every received message. Instead, they can acknowledge multiple message receptions with a single `ACK` frame. Therefore, when the client sends a sequence of messages, the server might respond by (1) simply acknowledging that message, (2) coalescing acknowledgment with other frames, or (3) not acknowledging at all. For example, a server response might be (1) `ACK`, (2) `ACK,ST(11)[Dec]`, or (3) `ST(11)[Dec]`, respectively.

Similar behavior can also be observed in the connection closure mechanism. The server might choose to send the client a `CC` frame, or simply close its local socket without giving any response.

Such server behavior forces Q3Fuzz to generate different fingerprints for states that were supposed to be the same, preventing the states from being deduplicated (IV-B3). It inevitably leads to a state space explosion (Figure 9). Q3Fuzz employs a heuristic fallback mechanism to mitigate this issue. The framework halts the discovery of new states on any specific branch once it reaches a predefined threshold of levels (e.g., 10 levels). This approach intentionally omits semantic state abstraction and introduces a clear tradeoff. Vulnerabilities in extremely deep states might remain undiscovered. This strategic bound, however, effectively prevents infinite state discovery. It ensures the inference phase concludes within a practical time budget while retaining the essential structural protocol logic.

5) Implication for Stateful Fuzzing: The identification of these divergent behaviors is necessary for the effectiveness of dynamic testing. Without such inference, a black-box fuzzer remains blind to implementation-specific features. For instance, it would send `DATA` frames prematurely against servers, which require the reception of connection parameters first (§V-B3). It would immediately result in connection closure and, consequently, the fuzzer would fail to reach the code paths vulnerable to deep-state logic bugs. By leveraging the inferred SM, Q3Fuzz resolves such cases by enabling the fuzzer to navigate through mandatory initialization sequences to access and test the deep protocol states.

C. Stateful Fuzzing

In this section, we evaluate Q3Fuzz's effectiveness at discovering vulnerabilities and compare it with existing work.

1) Discovered Bugs: Q3Fuzz discovered 17 DoS vulnerabilities in 7 different QUIC-HTTP/3 servers (Table V), all of which were responsibly reported to the vendors and CERT/CC through VINCE [37]. The vulnerabilities can be grouped into six different Common Weakness Enumeration (CWE) [38] categories. The CWE-119, CWE-617, CWE-248, and CWE-190 group vulnerabilities allow an attacker to crash the remote server, while the CWE-400 and CWE-401 group vulnerabilities cause high resource consumption at the server, which leads to DoS.

Q3Fuzz discovered multiple attack vectors to exploit certain vulnerabilities. Attack vectors differ from each other based

TABLE V: Vulnerabilities discovered by Q3Fuzz during fuzzing

#	Server	CWE	Vulnerability Impact	Zero-day	# Attack Vectors
1	Proxygen	CWE-119	Server crash: `Segmentation fault`	Yes	3+
2	Xquic		Server crash: `Segmentation fault`	Yes	1
3	Quicly	CWE-617	Server crash: `Assertion 'quicly_num_streams(conn) == 0' failed`	Yes (**CVE-2025-61684**)	3+
4	Quicly		Server crash: `Assertion 'v <= 4611686018427387903' failed`	Yes (**CVE-2025-61684**)	2
5	Quicly		Server crash: `Assertion 'iter->p->acked == quicly_sentmap__type_packet' failed`	Yes (**CVE-2025-61684**)	3+
6	Quicly		Server crash: `Assertion '!"invalid CID sequence number"' failed`	Yes (**CVE-2025-61684**)	1
7	Neqo		Server crash: `internal error: entered unreachable code`	No	2
8	Quiche	CWE-248	Server crash: `called 'Result::unwrap()' on an 'Err' value: Done`	Yes	1
9	Neqo		Server crash: `called 'Result::unwrap()' on an 'Err' value: InvalidStreamId`	Yes	1
10	Neqo		Server crash: `called 'Result::unwrap()' on an 'Err' value: TransportError(InvalidStreamId)`	Yes	3+
11	Neqo		Server crash: `called 'Result::unwrap()' on an 'Err' value: Transport(InvalidStreamId)`	Yes	3+
12	Neqo	CWE-190	Server crash: `Varint value too large`	No	2
13	Kestrel	CWE-400	Low-rate DoS due to high CPU usage	Yes (**CVE-2026-25667**)	1
14	Quiche		Low-rate DoS due to high CPU and memory usage	Yes	1
15	Xquic		Low-rate DoS due to high disk and memory usage	Yes	1
16	Aioquic	CWE-401	Low-rate DoS due to memory leak	Yes	1
17	Neqo		Out-of-Memory: Kernel kills server process	Yes	1

on the sequence of frames it takes to exploit the vulnerability. For example, vulnerability #7 (Table V) is a known vulnerability and according to the security advisory [3], it is exploited by sending a `NEW_TOKEN` frame right after the connection establishment. In addition to the aforementioned attack vector, Q3Fuzz found that it is also possible to exploit the vulnerability by sending `PATH_RESPONSE` and `STREAM` frames.

We demonstrate the attack vector to exploit the memory safety vulnerability in the Xquic server (vulnerability #2 in Table V) to highlight the importance of stateful fuzzing. While fuzzing a certain state transition, Q3Fuzz transmitted two *moving messages* (`M_msg1`, `M_msg2`) to reach the source state and then sent a randomly generated message (Figure 10), which caused the server crash. For a successful exploitation, one QUIC frame (`ST[0](HE)`) from `M_msg2` and four QUIC frames (`ACK`, `ST[0](Enc)`, `ST[0](GO)`, `ST[1](GO)`) from the generated test input are crucial. `M_msg1` is not required to exploit the vulnerability; however, interestingly, sending `M_msg2` without `M_msg1` normally causes the server to close the QUIC connection (moves to `FINISH` state in SM), showing how unexpected behavior servers can demonstrate in stateful fuzzing.

2) Baseline Extension: To objectively evaluate Q3Fuzz's performance, we established a comparative baseline. Given the absence of state-aware fuzzers dedicated to HTTP/3, for extension, we selected QUIC-Fuzz [16], the state-of-the-art grey-box fuzzer optimized for the QUIC transport layer.

[3]https://github.com/mozilla/neqo/security/advisories/GHSA-56c6-rfrf-rh4r

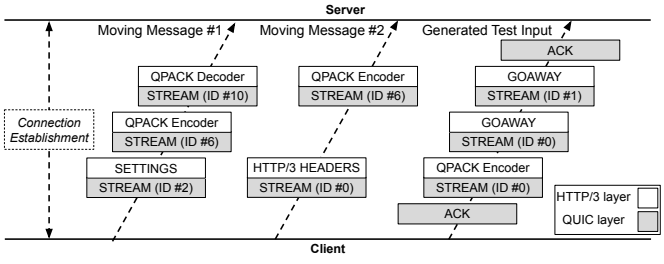


Fig. 10: Attack vector to exploit the crash vulnerability in a remote Xquic server

QUIC-Fuzz. We invested significant engineering effort to adapt QUIC-Fuzz for the application layer. This involved generating valid HTTP/3 Initial seeds containing the `h3` ALPN extension [39] and reconfiguring the harness to support valid TLS certificate chains. We verified via manual replay that the target server correctly accepts these inputs and initiates the QUIC handshake in a native environment.

However, our experiments revealed a fundamental structural limitation. Although the server responded with valid QUIC packets, the fuzzer failed to infer the states. The root cause is a limitation in QUIC-Fuzz's shallow dissection, which relies on extracting explicit response codes to infer states. In the QUIC-HTTP/3 stack, critical application-layer signals (e.g., HTTP status codes) are encapsulated within encrypted QUIC `STREAM` frames. Lacking the capability for recursive dissection, QUIC-Fuzz perceives distinct HTTP/3 responses merely as opaque transport-layer data. Consequently, it failed to detect state transitions, resulting in persistent execution timeouts and rendering state exploration infeasible.

Protocol-agnostic fuzzers. We also tried to compare Q3Fuzz with the existing protocol-agnostic black box fuzzers, such as Bleem [40], BooFuzz [41], Peach [42], and Snipuzz [43]. Bleem includes Mvfst, a QUIC server, in its evaluation and demonstrates how it achieves a higher branch coverage than the other aforementioned black box fuzzers. However, to the best of our knowledge, the source code of Bleem is not publicly available. Moreover, it is not clear how Bleem instructed the other black box fuzzers to target a QUIC server, since none of these fuzzers support the QUIC protocol out of the box due to the protocol's mandatory encryption requirement. For example, BooFuzz provides socket-based connections for only TCP, UDP, and SSL protocols [4]. Similar to how QUIC-Fuzz extends AFLNet [23], these fuzzers also need an extension to support the QUIC protocol.

[4]https://boofuzz.readthedocs.io/en/stable/user/connections.html

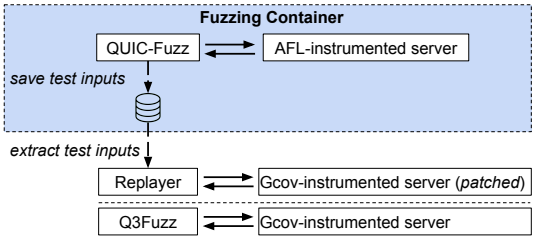


Fig. 11: Process of the coverage measurement

3) Code Coverage: Despite the baseline's structural limitations, we conducted a code coverage analysis to demonstrate differences in state reachability.

Setup. To ensure a fair comparison, we targeted different instances of the same server instrumented with **gcov** [44] for both fuzzers. Unlike Q3Fuzz, QUIC-Fuzz targets an AFL-instrumented server, from which it extracts runtime information for guidance. We executed QUIC-Fuzz in the Docker containers provided by the authors. During fuzzing, QUIC-Fuzz stores the test inputs in a folder inside the container. With a custom-written script (*replayer*), we extracted whenever a new test input was generated and replayed to the standard target server (Figure 11). The details of the evaluation are also available on the Q3Fuzz code repository. Before fuzzing with QUIC-Fuzz, we applied the patches provided by the authors to the common target server. The patches make several modifications to the server source code, such as hardcoding the connection ID and cryptographic secrets. Before sending each test input of QUIC-Fuzz, the server was restarted.

Furthermore, we evaluated how Q3Fuzz performs solely on the QUIC layer. For this purpose, we modified the *Crafter* component of Q3Fuzz (IV-D) so that all `STREAM` frames carried an empty application-layer payload during fuzzing.

Results. Figure 12 compares the average branch coverage achieved by all fuzzers during 24-hour fuzzing. To make the results statistically significant [45], we repeated the experiment 10 times. As a result, Q3Fuzz achieved up to 76.1% higher average branch coverage than QUIC-Fuzz.

While QUIC-Fuzz saturates quickly, its coverage is confined to the transport layer logic. In contrast, Q3Fuzz penetrates deep into the application layer logic. This gap empirically confirms that Q3Fuzz's multi-layered architecture is essential for reaching and testing the complex state space of the QUIC-HTTP/3 stack.

Targeting only the QUIC layer dropped the overall efficiency of Q3Fuzz. Despite reduced efficiency, Q3Fuzz still outperformed QUIC-Fuzz on all servers, except Picoquic, where it achieved branch coverage similar to QUIC-Fuzz. Interestingly, we observed very little performance degradation on the Ngtcp2 server.

VI. Discussion

This section presents the limitations of this research and the direction of future work.

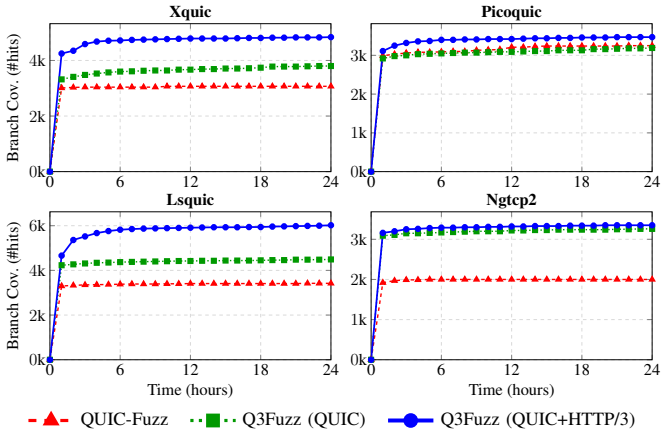


Fig. 12: The average branch coverage achieved by Q3Fuzz and QUIC-Fuzz over 24 hours.

A. Limitation: Invalid QPACK Instructions

QPACK is a stateful header compression protocol. It allows the endpoints to reduce the size of transmitted HTTP headers. In brief, both endpoints maintain synchronized dynamic tables. If an endpoint wants to send a new header, it first sends an insert instruction to the dynamic table, the receiver synchronizes its local dynamic table, and after that, the sender can reference that header. This mechanism adds an additional stateful behavior on top of HTTP/3. It is challenging to ensure the synchronization of the endpoints of dynamic tables during fuzzing. Therefore, when we generate QPACK instructions, we simply generate arbitrary byte sequences.

B. Future Work

1) Application to Other Protocols: While the methodology to model and fuzz dual-layered protocols is versatile and can be applied to other protocols, it introduces engineering challenges. In this work, we implemented the methodology for the QUIC-HTTP/3 dual-layer protocol stack. In future work, the methodology can be applied to other protocols, such as TCP-HTTP/2.

2) Model Usage: The SMs that have been generated based on observed server behavior can be used for cross-comparison to detect differences in server behaviors. Such divergences might reveal bugs. Moreover, since protocol specifications [1], [12] do not define a connection-wide SM, such observation-based models can potentially be expressed in a formal specification language and then model-checked for protocol correctness early in the development time.

VII. Related Work

Since the proposal of the QUIC and HTTP/3 protocols, research into their security has largely fallen into two primary categories: manual analysis and automated fuzzing frameworks.

A. Manual Analysis

The security analysis in both QUIC and HTTP/3 protocols was initially led by manual inspection and feature-specific testing. As highlighted in (§I), these foundational efforts confirmed that critical attack surfaces exist. For instance, studies confirmed that known HTTP/2 attack vectors can be manually migrated [10], and that core protocol logics such as connection migration can be abused to cause DoS [8].

Even semi-automated approaches utilizing formal methods, such as Network-centric Compositional Testing (NCT) [46], require significant manual effort to define the specification, despite successfully finding deep flaws. Targeted discoveries such as QUIC-Leak [47] further demonstrate that subtle, state-dependent bugs remain prevalent.

Although these manual and feature-specific approaches are foundational for identifying vulnerabilities, they possess inherent scalability limitations. They rely on expert-level knowledge and extensive labor to test specific components, an approach that is insufficient to systematically navigate the vast, multi-layered state space. This limitation necessitates an automated framework that infers stateful behaviors of implementations and utilizes stateful fuzzing to discover such complex vulnerabilities systematically.

B. Automated Fuzzing Frameworks

Ang et al. propose two QUIC protocol fuzzers: QUICTester [15] and QUIC-Fuzz [16]. QUICTester is a blackbox noncompliance checker that models the handshake phase of QUIC communication using TTT-based [48] active automata learning and the Wp-method [49] conformance testing algorithm. Similar to Q3Fuzz, they model the state space of QUIC server implementations by sending QUIC packets and observing how the server responds. They employ pair-wise testing, where they compare the behavior of a server with that of the other servers. If a server behaves differently from the others, it indicates a potential bug in the implementation.

The main difference between QUICTester and Q3Fuzz is their scope. While QUICTester solely focuses on the *handshake* phase of QUIC communication, Q3Fuzz focuses on its *post-handshake* phase, as well as the HTTP/3 protocol. The work has similarity with Q3Fuzz in terms of modeling the QUIC servers in a black box environment. However, unlike QUICTester, we automatically extract messages from generated traffic instead of defining symbols beforehand.

QUIC-Fuzz [16], on the other hand, is a grey-box mutation-based QUIC fuzzer which is an extension of AFLNet [23]. QUIC-Fuzz extracts messages from generated traffic, applies mutation, and re-encrypts before sending to the server. It uses the state feedback from the server to guide the fuzzer. Additionally, it also requires applying modifications to the server source code for better performance.

While the scopes of QUIC-Fuzz and Q3Fuzz are similar (*post-handshake* phase), there are substantial differences. Firstly, Q3Fuzz works in a blackbox environment, where it does not need access to the server's runtime environment. Hence, with Q3Fuzz, we analyzed 14 servers written in 6 different programming languages, while QUIC-Fuzz is evaluated on 6 server implementations, all written in C/C++ programming languages. Secondly, QUIC-Fuzz supports only mutation-based fuzzing, while Q3Fuzz additionally employs generation-based fuzzing methodology.

There also exist fuzzers that are intended for specific QUIC implementations, such as quic-fuzz [50] for Quiwi and Fuzi_q [51] for Picoquic.

McMillan et al. [52] and Crochet et al. [53] take a different approach by testing QUIC implementations based on formal specifications. Both work define specifications in the Ivy [54] language and check whether the protocol implementations follow the defined rules. Any noncompliance indicates a potential bug in the implementation.

Reen et al. [14] and Rath et al. [55] target different aspects of the QUIC protocol. The former proposes DPIFuzzer, a differential fuzzer to detect strategies to bypass Deep Packet Inspection (DPI) in middleboxes. The latter, on the other hand, employs a symbolic execution-based methodology to discover interoperability issues in QUIC implementations.

VIII. Conclusion

In this paper, we presented Q3Fuzz, a framework to model dual-layered protocols and fuzz for vulnerability discovery. Q3Fuzz models the state space of QUIC-HTTP/3 protocol stack implementations by transmitting recorded messages and observing the server's response. It supports both mutation-based and generation-based fuzzing. For mutation-based fuzzing, it fuzzes each state transition by applying modifications to the recorded message that cause the state transition. For better coverage, it also fuzzes each state transition by generating messages based on protocol specifications.

We applied the Q3Fuzz methodology to the QUIC-HTTP/3 dual-layer protocol stack. Modeling and fuzzing two protocol layers simultaneously led to considerably higher code coverage than the existing work. Our evaluation includes modeling and fuzzing 14 most popular open-source QUIC-HTTP/3 stack implementations, which resulted in the discovery of 17 exploitable vulnerabilities.

IX. Acknowledgement

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP), funded by the Korea MSIT (No. RS-2024-00440780 Development of Automated SBOM and VEX Verification Technologies), ICT Creative Consilience Program (IITP-2025-RS-2020-II201819, 10%), and the Research Foundation of the City University of New York (RFCUNY).

References

[1] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[2] P. Megyesi, Z. Krämer, and S. Molnár, "How quick is QUIC?" in *2016 IEEE International Conference on Communications (ICC)*, 2016, pp. 1–6.

[3] C. Sander, I. Kunze, and K. Wehrle, "Analyzing the Influence of Resource Prioritization on HTTP/3 HOL Blocking and Performance." in *Network Traffic Measurement and Analysis Conference*, 2022.

[4] C. Lee, I. Jafarov, S. Dietrich, and H. Lee, "PRETT2: Discovering HTTP/2 DoS Vulnerabilities via Protocol Reverse Engineering," in *Computer Security – ESORICS 2024: 29th European Symposium on Research in Computer Security*. Springer-Verlag, 2024, p. 3–23. [Online]. Available: https://doi.org/10.1007/978-3-031-70890-9_1

[5] R. Marx, T. De Decker, P. Quax, and W. Lamotte, "Resource multiplexing and prioritization in HTTP/2 over TCP versus HTTP/3 over QUIC," in *Web Information Systems and Technologies*, A. Bozzon, F. J. Domínguez Mayo, and J. Filipe, Eds. Springer International Publishing, 2020, pp. 96–126.

[6] W3Techs, "Usage statistics of HTTP/3 for websites," W3Techs, 2025, Accessed: 2026-04-27. [Online]. Available: https://w3techs.com/technologies/details/ce-http3

[7] D. Belson and L. Pardue, "Examining HTTP/3 usage one year on," Cloudflare Blog, June 2023, Accessed: 2026-04-27. [Online]. Available: https://blog.cloudflare.com/http3-usage-one-year-on/

[8] H. H. Sudhan S and S. G. Kulkarni, "Security and Service Vulnerabilities with HTTP/3," in *2024 16th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2024, pp. 55–60.

[9] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher, *Internet Denial of Service: Attack and Defense Mechanisms*. Pearson, Dec 2004.

[10] E. Chatzoglou, V. Kouliaridis, G. Kambourakis, G. Karopoulos, and S. Gritzalis, "A hands-on gaze on HTTP/3 security through the lens of HTTP/2 and a public dataset," *Computers and Security*, vol. 125, no. C, Feb. 2023. [Online]. Available: https://doi.org/10.1016/j.cose.2022.103051

[11] K. Oku and L. Pardue, "Extensible Prioritization Scheme for HTTP," RFC 9218, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9218

[12] M. Bishop, "HTTP/3," RFC 9114, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9114

[13] M. Bishop, "The ORIGIN Extension in HTTP/3," RFC 9412, Jun. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9412

[14] G. S. Reen and C. Rossow, "DPIFuzz: A differential fuzzing framework to detect DPI elusion strategies for QUIC," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20. Association for Computing Machinery, 2020, p. 332–344. [Online]. Available: https://doi.org/10.1145/3427228.3427662

[15] K. K. Ang, G. Farrelly, C. Pope, and D. C. Ranasinghe, "An automated blackbox noncompliance checker for QUIC server implementations," in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '25. Association for Computing Machinery, 2025, p. 1459–1475. [Online]. Available: https://doi.org/10.1145/3708821.3736222

[16] K. K. Ang and D. 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*. Springer-Verlag, 2025, p. 1–22. [Online]. Available: https://doi.org/10.1007/978-3-032-07894-0_1

[17] R. Natella, "StateAFL: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, Dec. 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10233-3

[18] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 2, p. 292–333, Mar. 1997. [Online]. Available: https://doi.org/10.1145/244795.244801

[19] E. Poll, J. de Ruiter, and A. Schubert, "Protocol state machines and session languages: specification, implementation, and security flaws," in *2015 IEEE Security and Privacy Workshops*, 2015, pp. 125–133.

[20] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, "Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols," *Commun. ACM*, vol. 62, no. 7, p. 86–94, Jun. 2019. [Online]. Available: https://doi.org/10.1145/3330336

[21] X. Zhang, C. Zhang, X. Li, Z. Du, B. Mao, Y. Li, Y. Zheng, Y. Li, L. Pan, Y. Liu, and R. Deng, "A Survey of Protocol Fuzzing," *ACM Computing Surveys*, vol. 57, no. 2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3696788

[22] C. B. Krasic, M. Bishop, and A. Frindell, "QPACK: Field Compression for HTTP/3," RFC 9204, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9204

[23] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A Greybox Fuzzer for Network Protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 460–465.

[24] R. Meng, V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet Five Years Later: On Coverage-Guided Protocol Fuzzing," *IEEE Transactions on Software Engineering*, vol. 51, no. 4, p. 960–974, Apr. 2025. [Online]. Available: https://doi.org/10.1109/TSE.2025.3535925

[25] H. Damlaj, "fix uaf in StaticFileHandler," GitHub commit, Facebook, 2025, commit 91608fcb3825f532e006096bdd88ffdc8953bb1b. [Online]. Available: https://github.com/facebook/proxygen/commit/91608fcb3825f532e006096bdd88ffdc8953bb1b

[26] C. Lee, J. Bae, and H. Lee, "PRETT: Protocol Reverse Engineering Using Binary Tokens and Network Traces," in *ICT Systems Security and Privacy Protection*, L. J. Janczewski and M. Kutyłowski, Eds. Springer International Publishing, 2018, pp. 141–155.

[27] Internet Assigned Numbers Authority, "QUIC," 2021, Accessed: 2026-04-27. [Online]. Available: https://www.iana.org/assignments/quic/quic.xhtml

[28] T. Pauly, E. Kinnear, and D. Schinazi, "An Unreliable Datagram Extension to QUIC," RFC 9221, Mar. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9221

[29] Internet Assigned Numbers Authority, "Hypertext Transfer Protocol version 3 (HTTP/3)," 2021, Accessed: 2026-04-27. [Online]. Available: https://www.iana.org/assignments/http3-parameters/http3-parameters.xhtml

[30] V. D. M. Rios, P. R. M. Inácio, D. Magoni, and M. M. Freire, "Detection and Mitigation of Low-Rate Denial-of-Service Attacks: A Survey," *IEEE Access*, vol. 10, pp. 76 648–76 668, 2022.

[31] Aiortc, "Aioquic," Accessed: 2026-04-27. [Online]. Available: https://github.com/aiortc/aioquic

[32] QUIC Working Group, "QUIC implementations," Accessed: 2026-04-27. [Online]. Available: https://github.com/quicwg/quicwg.github.io/blob/main/implementations.md

[33] GitHub, "Saving repositories with stars," Accessed: 2026-04-27. [Online]. Available: https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars

[34] Amazon Web Services, "S2n-quic," Accessed: 2026-04-27. [Online]. Available: https://github.com/aws/s2n-quic

[35] Mozilla, "Firefox," Accessed: 2026-04-27. [Online]. Available: https://www.firefox.com/

[36] E. Chatzoglou, V. Kouliaridis, G. Karopoulos, and G. Kambourakis, "Revisiting QUIC attacks: a comprehensive review on QUIC security and a hands-on study," *International Journal of Information Security*, vol. 22, no. 2, p. 347–365, Dec. 2022. [Online]. Available: https://doi.org/10.1007/s10207-022-00630-6

[37] CERT Coordination Center, "VINCE: Vulnerability Information and Coordination Environment," Accessed: 2026-04-27. [Online]. Available: https://www.kb.cert.org/vince/

[38] MITRE Corporation, "Common Weakness Enumeration," MITRE Corporation, Tech. Rep., Accessed: 2026-04-27. [Online]. Available: https://cwe.mitre.org/

[39] S. Friedl, A. Popov, A. Langley, and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension," RFC 7301, Jul. 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7301

[40] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations," in *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Aug. 2023, pp. 4481–4498. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/luo-zhengxiong

[41] J. Pereyda, "boofuzz: Network Protocol Fuzzing for Humans," Accessed: 2026-04-27. [Online]. Available: https://github.com/jtpereyda/boofuzz

[42] Mozilla Fuzzing Security, "Peach," Accessed: 2026-04-27. [Online]. Available: https://github.com/MozillaSecurity/peach

[43] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. Association for Computing Machinery, 2021, p. 337–350. [Online]. Available: https://doi.org/10.1145/3460120.3484543

[44] GNU Project, *gcov — a Test Coverage Program*, Accessed: 2026-04-27. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[45] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[46] K. L. McMillan and L. D. Zuck, "Compositional testing of internet protocols," in *2019 IEEE Cybersecurity Development (SecDev)*, 2019, pp. 161–174.

[47] National Institute of Standards and Technology (NIST), "CVE-2025-54939," Aug. 2025, Accessed: 2026-04-27. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2025-54939

[48] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds. Springer International Publishing, 2014, pp. 307–322.

[49] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 591–603, Jun. 1991. [Online]. Available: https://doi.org/10.1109/32.87284

[50] G. Burrow, "quic-fuzz," Accessed: 2026-04-27. [Online]. Available: https://github.com/goburrow/quic-fuzz

[51] C. Huitema, "fuzi_q," Accessed: 2026-04-27. [Online]. Available: https://github.com/private-octopus/fuzi_q

[52] K. L. McMillan and L. D. Zuck, "Formal specification and testing of QUIC," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. Association for Computing Machinery, 2019, p. 227–240. [Online]. Available: https://doi.org/10.1145/3341302.3342087

[53] C. Crochet, J. Aoga, and A. Legay, "Formally Discovering and Reproducing Network Protocols Vulnerabilities," in *Secure IT Systems: 29th Nordic Conference (NordSec)*. Springer-Verlag, 2025, p. 424–443. [Online]. Available: https://doi.org/10.1007/978-3-031-79007-2_22

[54] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: Safety Verification by Interactive Generalization," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. Association for Computing Machinery, 2016, p. 614–630. [Online]. Available: https://doi.org/10.1145/2908080.2908118

[55] F. Rath, D. Schemmel, and K. Wehrle, "Interoperability-Guided Testing of QUIC Implementations using Symbolic Execution," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ'18. Association for Computing Machinery, 2018, p. 15–21. [Online]. Available: https://doi.org/10.1145/3284850.3284853